Towards Temporal Graph Databases

Alejandro Vaisman
Instituto Tecnológico de Buenos Aires
155 PUBLICATIONS **3,049** CITATIONS

Towards Temporal Graph Databases

Alexander Campos, Jorge Mozzino, and Alejandro Vaisman

Instituto Tecnologico de Buenos Aires
`jcamposi,jmozzino,avaisman@itba.edu.ar`

Abstract. In spite of the extensive literature on graph databases (GDBs), temporal GDBs have not received too much attention so far. Temporal GBDs can capture, for example, the evolution of social networks across time, a relevant topic in data analysis nowadays. In this paper we propose a data model and query language (denoted TEG-QL) for temporal GDBs, based on the notion of attribute graphs. This allows a straightforward translation to Neo4J, a well-known GBD. We present extensive examples of the use of TEG-QL, and comment our implementation.

Keywords: Neo4J, Graph Database, Temporal Graphs

1 Introduction

Graphs, and, particularly, attributed graphs [12], are becoming increasingly popular to model different kinds of networks (e.g., social networks, sensor networks, and the kind) for analysis in a classical way, and also for performing Online Analytical Processing (OLAP) on graphs [5, 12]. Also, these kinds of graphs underlie the data model of Neo4J,[1] probably the most popular graph database for social network analysis at the time of writing this paper [1], together with AllegroGraph.[2] In spite of the fact that social networks are heavily changing structures, not much attention has yet been paid to temporal graph databases (see Section 2). In this paper we introduce our approach to this topic, based on attribute graph data models. We first present a temporal data model, consisting in a data structure (an attribute graph), and a set of constraints. Then, we sketch a temporal query language, called TEG-QL (for Temporal Graph Query Language), an SQL/SPARQL-like style language, with the idea of facilitating the translation to Cypher, the language that comes with Neo4J. We provide extensive examples of the possible use of TEG-QL. We also describe a prototype we have implemented, allowing time navigation of temporal graphs (and of the result of a TEG-QL query). Throughout the paper we will be using a running example consisting in a network containing two kinds of nodes, representing persons and buildings. Edges in this network are of two kinds: One, representing friendship relationships between people across time; the other one telling the buildings where people had lived through time. Nodes contain information about

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

the name of the people, type of building, number of bedrooms in the apartment, etc. In addition, nodes and edges have a temporal attribute, which is a temporal element indicating the periods of validity of the node and/or the edge.

The rest of the paper is structured as follows: Section 2 discusses related work. Then, Section 3 introduces the data model we propose. Section 4 presents TEG-QL by example. Section 5 briefly explains the basics of our implementation and translation from TEG-QL to Neo4J. We conclude in Section 6.

2 Related Work

There is an extensive bibliography on graph database models. This is comprehensively studied in [2]. Surprisingly, given the clear need for temporal graph modeling, querying, and analysis, the number of works on the subject is not that large. We survey some work next, and compare against our proposal.

The model we describe in the next section, fits in what are called *Attributed Graphs*, which are appropriate for Online Analytical Processing on graphs [12, 5]. For example, attributes in this model allow us to aggregate nodes and edges, provided we define an attribute hierarchy over them. This is one of the reasons of our modeling choice.

A temporal graph model has been presented in [3, 4], where temporal data are organized in so-called frames, namely the finest unit of temporal aggregation. A frame is associated with a time interval and allows to retrieve the status of the social network during such interval. One limitation of that model is that it does not allow registering changes in attributes of the nodes and that frame nodes become too cluttered with edges (one for each actor and for each relationship that existed in that frame). Redundant data are also a problem since each frame is connected to all the existing data, so a frequently changing graph becomes full of redundant connections. Opposite to this model, to keep track of the changes in the value of node attributes, we define *Attribute* and *Value* nodes.

Khurana and Deshpande [7, 8] 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. Also, the authors work on a data model that is based on versioning. Basically, they store the current graph, plus a series of deltas, which contain the graph variation over time. Our model, on the contrary, is based on timestamps, where the complete history is stored in the same graph. This is a typical trade-off problem in temporal databases, discussed also in the temporal XML area [10]. With goals different than ours, Han et al. [6] presented an engine for temporal graph mining, and Kostakos [9] show the use of temporal graphs to represent dynamic events.

3 Data Model

We now define the temporal data model supporting our proposal. We base the model in the notion of attribute graphs, that is, graphs whose nodes and edges are annotated with attributes, describing their characteristics.

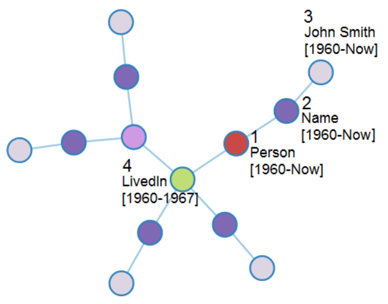


Fig. 1: A temporal graph and its different kinds of nodes

Definition 1 (Temporal graph). *A temporal graph is a structure $G(No, Ne, Na, Nv, E)$ where G is the name of the graph, E is a set of edges, and No, Ne, Na, and Nv are sets of nodes, denoted object nodes, edge nodes, attribute nodes,and value nodes, respectively. Every node in the graph is associated with a tuple (`name`,`interval`). The `name` represents the content of the node, and the `interval` represent the period(s) in which the node is (was) valid and it is a temporal element. As usual in temporal databases, a special value Now is used to represent that the node is currently valid.* □

In the definition above, object nodes represent entities (e.g., Person), edge nodes 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). The underlying idea is to allow not only to query the graph, but also to perform OLAP analysis. This is why, instead of placing edges between two object nodes, we define *edge nodes*, which will make aggregation over edges easier. We do not address OLAP analysis of graphs here.

Example 1 (Data model). Figure 1 depicts a portion of a graph showing the `name` and `interval` properties of the different node types, using our running example. The properties labeling the nodes are, from top to bottom, `id`, `name` and `interval`. This way, the node with `id`=4 (in light green) is an *Edge node*, the node with `id`=1 (in red) is an *Object node*, the node with `id`=2 (purple) is an *Attribute node*, and the one with `id`=3 (grey) is a *Value node*. □

Before introducing this graph's constraints, we introduce some notation. We denote edge nodes as $ne\{na, nb\}$, meaning that ne is an edge node connected to object nodes na and nb. An edge will be represented by $e\{na, nb\}$ where na and nb are nodes connected by the edge e. An attribute node will be represented as $na\{n\}$ where n is the object or edge node connected to na. Finally, we denote a value node as $nv\{na\}$ where na is the attribute node connected to nv.

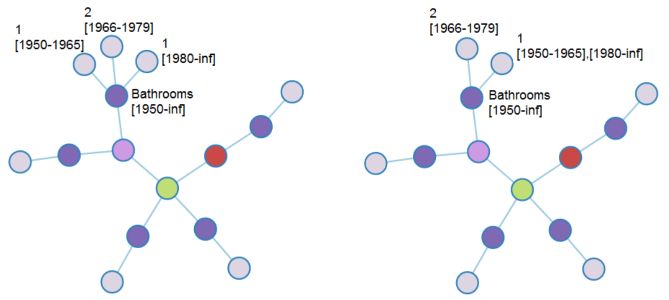
Definition 2 (Constraints). *For the graph in Definition 1, the following constraints hold:*

1. $\forall n, n' \in No,\ n = n' \vee n.id \neq n'.id$
2. $\forall n, n' \in Ne,\ n = n' \vee n.id \neq n'.id$
3. $\forall n, n' \in Na,\ n = n' \vee n.id \neq n'.id$
4. $\forall n, n' \in Nv,\ n = n' \vee n.id \neq n'.id$
5. $\forall nv\{na\}, nv'\{na\} \in Nv,\ nv = nv' \vee nv.value \neq nv'.value$
6. $\forall n \in No,\ e\{n, n'\} \in E \Rightarrow n' \in Ne \bigcup Na$
7. $\forall n \in Ne,\ e\{n, n'\} \in E \Rightarrow n' \in No \bigcup Na$
8. $\forall n \in Na,\ e\{n, n'\} \in E \Rightarrow n' \in No \bigcup Ne \bigcup Nv$
9. $\forall n \in Nv,\ e\{n, n'\} \in E \Rightarrow n' \in Nv$
10. $\forall ne \in Ne,\ if\ \exists\ e\{no, ne\} \wedge \exists e'\{ne, no'\} \Rightarrow \nexists e'' \in E, \nexists no'' \in No \wedge no'' \neq no \wedge no'' \neq no' \wedge e''\{no'', ne\} \wedge e''\{ne, no''\}$
11. $\forall n \in Na(\exists no \in No \exists e \in E(e(no, n) \vee \exists ne \in Ne \wedge e\{ne, n\} \wedge (\nexists n' \in (Na \bigcup Ne \bigcup Nv \bigcup No) \wedge e' \in E \wedge e'\{n', n\})$
12. $\forall n \in Nv \wedge e\{n', n\} \wedge n \in Na \Rightarrow \nexists ! n'' \in (Na \bigcup Ne \bigcup Nv \bigcup No) \wedge (e''\{n'', n\} \in E \vee e''\{n, n''\} \in E$
13. $\exists e\{n, n'\}, e'\{n, n'\} \in E \Rightarrow e = e'$
14. $\forall ne\{n, n'\} \in Ne, ne.interval \subset n.interval \cap n'.interval$
15. $\forall na\{n\} \in Na, na.interval \subset n.interval$
16. $\forall nv\{na\} \in Nv, nv.interval \subset nv.interval$
17. $\forall nv\{na\}, nv'\{na\}, nv \neq nv', nv.interval \cap nv'.interval = \emptyset$

Constraints 1 through 4 state that no two nodes can have the same id. Constraint 5 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. Figure 2 explains this. Constraints 6 through 9 state how the nodes must be connected, namely: (a) Object nodes can only be connected to edge nodes or attribute nodes; (b) Edge nodes can only be connected to object nodes or attribute nodes; (c) Attribute nodes can be connected to non-attribute nodes; and (d) Value nodes can only be connected to attribute nodes. The cardinalities of these connections is stated by Constraints 10 through 13. Edge nodes must be connected to exactly two different object nodes through exactly one edge, attribute nodes must be connected by only one edge to either an object node or an edge node, and value nodes must only be connected to one attribute node with one edge. Constraint 13 states that there cannot be more than one edge between any given pair of nodes. Constraints 14 to 17 restrict the values of the `interval` property. Finally, constraint 17 forces value nodes connected to the same attribute node to have non-overlapping intervals. □

4 TEG-QL: A Query Language for Graphs

We now sketch through examples our query language. The syntax of the language resembles the one of SQL, having the typical `SELECT-FROM-WHERE` form. Queries, as usual in graphs, are based on pattern matching. Thus, the `FROM` clause contains one or more paths (of fixed or variable length), over which a selection is performed. The `SELECT` clause may either mention just attributes or



(a) a

Fig. 2: A graph not compliant with constraint 5 (left); A graph compliant with constraint 5 (right)

paths. The temporal semantics in embedded in the language, as is normally the case in temporal databases [11]. That is, the answer to the query is a temporal graph, although the query may not mention temporal attributes. However, this can be changed by the `SNAPSHOT` modifier, which allows to retrieve the state of the graph at a certain point in time, or the `IN` modifier, which allows retrieving the status of the graph in a certain interval. There is also the possibility of indicating if we want all the components of the graph, or only the object nodes, and so on. We show these in the examples below.

We start with a query retrieving object nodes. Consider the query *Buildings where John Smith lives or lived*. This query just returns the nodes in the graph representing the buildings, and it is expressed in TEG-QL as:

```
SELECT Building
FROM Person-LivedIn->Building
WHERE Person.Name = 'John Smith'
```

We now move to queries returning paths in the `SELECT` clause. These paths are very similar as the paths in Cypher (the query language for Neo4J) paths. Consider the query *People and buildings such that a person named John Smith has lived in such buildings.* The predicate we are looking for is LivedIn. The query is expressed in TEG-QL as:

```
SELECT Person-LivedIn->Building
FROM Person-LivedIn->Building
WHERE Person.Name = 'John Smith'
```

That is, we take the paths matching the `FROM` clause, and filter them using the condition in the WHERE clause. Figure 3 (left) shows the result. The center node (in orange) is the `Person` node that represents John Smith, middle nodes

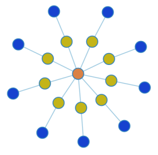
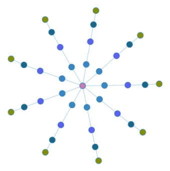


Fig. 3: Query selecting a path (left); Query returning attributes (right)

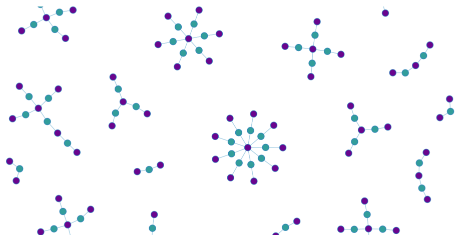


Fig. 4: Friends of John Smith

(yellow) nodes are the edge nodes representing the Lived In relationships; and outer nodes (blue) are the Building nodes.

If we just want to return certain attributes, we specify them in a comma-separated list, between parenthesis, in the SELECT clause (like in SQL, also a * can be used to represent selecting every attribute of the node). The next query shows this: we just want the street of the buildings satisfying the condition in the previous query. This is expressed as follows.

```
SELECT Person-LivedIn->Building(Street)
FROM Person-LivedIn->Building
WHERE Person.Name = 'John Smith'
```

We only return the attribute Street of the nodes of type Building. Figure 3 (right) shows the result. From inside out, the nodes are of type Person (in pink, representing John Smith), LivedIn, Building, Street, and the value node with the street name (in green).

A typical query in social network analysis asks for friendship relationships. Let us start with *Friends of someone called John Smith*. Below we show the TEG-QL expression, and Figure 4 depicts the result, showing clusters of people who know someone with the name "John Smith".

```
SELECT Person-Friend->P2
FROM Person-Friend->Person as P2
WHERE Person.Name = 'John Smith'
```

We mentioned that TEG-QL supports multiple paths in the SELECT and FROM clauses, allowing also a kind of join between paths. Consider the query *Places where John Smith lived, along with John's friends.* The query reads in TEG-QL:

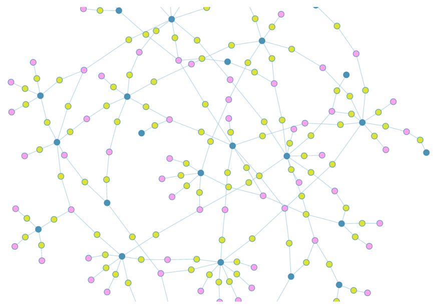


Fig. 5: Using the SNAPSHOT clause

```
SELECT *
FROM Person-LivedIn->Building,
     Person as P2-Friend->Person as P3
WHERE Person.Name = 'John Smith'
      and P2.Name= 'John Smith'
```

This query is equivalent to:

```
SELECT *
FROM Person as P2<-Friend-Person-LivedIn->Building
WHERE Person.Name = 'John Smith'
```

Here, the path in the `FROM` clause reminds Cypher syntax. Note that in Neo4J the nodes are of the same type, thus, the path `Person-Friend-Person` represents, in an abstract way, a traversal of two nodes, while, physically, the traversal of three nodes.

Variable length paths are also supported, allowing for example, to ask for the *friends of John Smith's friends*, also a typical social network query.

```
SELECT *
FROM Person-Friend[1..3]->Person
WHERE Person.Name = 'John Smith'
```

We conclude the section showing the use of the `SNAPSHOT` and `IN` modifiers. The query below returns all the people named John Smith, and the buildings where they live, during 1990.

```
SELECT Person-LivedIn->Building
FROM Person-LivedIn->Building
WHERE Person.Name = 'John Smith'
SNAPSHOT 1990
```

Note that we assume a temporal granularity at the *year* level here. We do not get into the details of how to manipulate granularities here. Figure 5 shows the result. Pink nodes represent Building nodes, blue nodes represent People nodes, and yellow nodes represent the Lived in relationship.

The IN predicate allows selecting an interval where nodes and edges were valid. The next query is similar to the one above, just selecting those paths existing between 1986 and 1989.

```
SELECT *
FROM Person-LivedIn->Building
WHERE Person.Name = ’John Smith’
IN [1986-1989]
```

5 Implementation

In this section we explain how TEG-QL queries are translated into Cypher (the Neo4J language), and describe the visual interface, that makes use of the temporal model to navigate the graphs across time. To allow understanding the process, let us comment on the Cypher syntax. Cypher queries match paths and then return parts of that paths. A simple query which returns every node, reads:

```
MATCH (n)  RETURN n
```

Node labels can be specified by modifying the node matcher, like the following query, which returns all the nodes with the `Object` label.

```
MATCH (n:Object) RETURN n
```

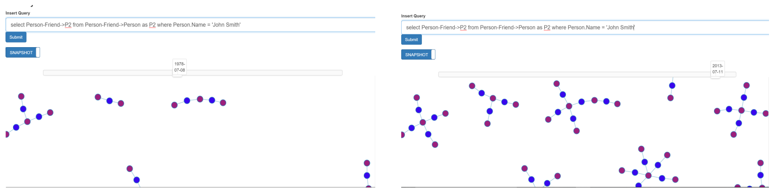
In Neo4J, edges are also labeled, so a path can be queried based on these labels. The following query shows how we match a path of length one between 2 nodes of type person and the relationship representing friendship.

```
MATCH (o:Object {title: ’Person’})-->(e:Edge {title: ’Friend’})
     --> (o1:Object:’Person’)
RETURN o,e,o1
```

Query Translation A TEG-QL query is translated and executed as follows. First, we translate each path in the `FROM` clause, checking if each element of each path has an alias, and building the actual Cypher path. An Object node is translated as `element.alias:OBJECT title:element.name`; an Edge node, is translated as `element.name:EDGE title:element.name`. After this, we have our first sentence that, for every path in the `FROM` looks like:

```
MATCH (element1.alias:OBJECT {title:element1.name})-->
    (element2.name:EDGE {title:element2.name})-->
    (element3.alias:OBJECT {title:element3.name}) ...
```

We then expand the `SELECT` clause with the corresponding attributes. If the attributes are empty we do nothing. If the attribute ”*” is present, then we want all the attributes. If the attributes are explicit, we translate them one by one. For this, we build a Cypher path from the Object or Edge nodes, adding the Attribute Node and the Value node(s). For example if we have `Person(Age, Gender)` the translated query will look like this:



```
select Person-Friend->P2 from Person-Friend->Person as P2 where Person.Name = 'John Smith'
```

```
select Person-Friend->P2 from Person-Friend->Person as P2 where Person.Name = 'John Smith'
```

Fig. 6: Visual interface showing two states of the graph

```
MATCH (Person:OBJECT {title:Person})-->(x:ATTRIBUTE {title:Age})
        --> (y:VALUE),
MATCH (Person:OBJECT {title:Person})-->(w:ATTRIBUTE {title:Gender})
        --> (z:VALUE)
```

Finally we address the `WHERE` statement. We split `AND`s and `OR`s and translate each part as follows. If an *id* is present, we produce the path to that id, if it is not already present; if is a constant we translate it as it is. For example if in the `WHERE` condition we have `Person.age = 12 AND Person.gender = "Male"`, the translated query will look like this:

```
MATCH (Person:OBJECT {title:Person})-->
       (x:ATTRIBUTE {title:Age})--> (y:VALUE),
MATCH (Person:OBJECT {title:Person})-->
       (w:ATTRIBUTE {title:Gender})--> (z:VALUE)
WHERE y = 12 AND z = "Male"
```

Once the query has been executed and the result retrieved, we treat the temporal conditions `SNAPSHOT` and `IN`. This is done by a Java application that interprets the query and filters the result retrieved by Neo4J. Of course, this is a naïve way of treating this part of the query. In future work we will address the problem of query processing.

Finally, we have implemented a prototype to process queries, and a visual interface[3]. This interface allows the user to see the graph (i.e., the result of a query) at a certain point in time (a snapshot), or navigate it across time making use of a sliding bar. Figure 6 depicts two states of a graph , as displayed in the interface, showing the friends of "our" John Smith at two points in time.

We ran very preliminary tests over our naïve implementation, using a T2.Micro on the Amazon cloud, with a High Frequency Intel Xeon Processors with Turbo up to 3.3GHz CPU, and 1GB RAM. We built a graph containing 1000 people nodes, 100 building nodes, 2500 friendship relationships, and 500 lived-in relationships. For the RAM we had available, we could not build larger Neo4J graphs, since Neo4J cannot handle them. Translation times are negligible, as expected; just to give an idea of execution times, for queries like the ones showed in the example, those times go from 4 to 6 seconds. The reader can try these queries at the URL mentioned above.

[3] This interface is available at http://52.37.51.136:8080/

6 Future Work

We have described our approach to model and query temporal graph, which we believe is a relevant problem, for example, in social network analysis, given the dynamic nature of such networks. Future work will focus on expanding the capabilities of TEG-QL, and, most of all, on addressing the problem of query optimization, for which, we believe that efficient indexing techniques must be developed, opening an interesting research field.

References

1. Angles, R.: A Comparison of Current Graph Database Models. In: Proceedings of ICDE Workshops. pp. 171–177. Arlington, VA, USA (2012)
2. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Computing Surveys (CSUR) 40(1), 1–39 (2008)
3. Cattuto, C., Panisson, A., Quaggiotto, M.: Representing time dependent graphs in Neo4j. https://github.com/SocioPatterns/neo4j-dynagraph/wiki/Representing-time-dependent-graphs-in-Neo4j (2013)
4. Cattuto, C., Quaggiotto, M., Panisson, A., Averbuch, A.: Time-varying social networks in a graph database. In: Proceedings of GRADES 2013. p. 11. NY, USA (2013)
5. Ghrab, A., Romero, O., Skhiri, S., Vaisman, A., Zimányi, E.: GRAD: Modeling and Querying Data Warehouses. In: Proceedings of ADBIS. pp. 92–105. Poitiers, France (2015)
6. Han, W., Miao, Y., K.Li, Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., Chen, E.: Chronos: A Graph Engine por Temporal Graph Analysis. In: Eurosys. pp. 1–14 (2014)
7. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. CoRR arxiv:1207.5777 (2012)
8. Khurana, U., Deshpande, A.: HiNGE: Enabling Temporal Analytics at Scale. In: Proceedings of SIGMOD. NY, USA (2013)
9. Kostakos, V.: Temporal graphs. CoRR arxiv:0807.2357 (2008)
10. Rizzolo, F., Vaisman, A.: Temporal XML: Modeling, indexing, and query processing. VLDB Journal 1179–1212(5), 39–65 (2008)
11. Tansel, A., Clifford, J., Gadia (eds.), S.: Temporal Databases: Theory, Design and Implementation. Benjamin/Cummings (1993)
12. Wang, Z., Fan, Q., Wang, H., Tan, K., Aggrawal, D., Abbadi, A.E.: PArallel GRaph OLap Over Large Scale Attributed Graphs. In: Proceedings of ICDE. Chicago, USA (2014)