Analytical queries on semantic trajectories using graph databases

Leticia I. Gómez[1] | **Bart Kuijpers**[2] | **Alejandro A. Vaisman**[1]

[1]Department of Information Engineering, Instituto Tecnológico de Buenos Aires, Ciudad Autónoma de Buenos Aires, Argentina

[2]Databases and Theoretical Computer Science Research Group, UHasselt - Hasselt University and transnational University Limburg, Agoralaan, Gebouw D, Diepenbeek, B-3590, Belgium

Correspondence
Alejandro Vaisman, Department of Information Engineering, Instituto Tecnológico de Buenos Aires, Lavardén 389, Ciudad Autónoma de Buenos Aires C1437FBG, Argentina.
Email: avaisman@itba.edu.ar

Abstract

This article studies the analysis of moving object data collected by location-aware devices, such as GPS, using graph databases. Such raw trajectories can be transformed into so-called semantic trajectories, which are sequences of stops that occur at "places of interest." Trajectory data analysis can be enriched if spatial and non-spatial contextual data associated with the moving objects are taken into account, and aggregation of trajectory data can reveal hidden patterns within such data. When trajectory data are stored in relational databases, there is an "impedance mismatch" between the representation and storage models. Graphs in which the nodes and edges are annotated with properties are gaining increasing interest to model a variety of networks. Therefore, this article proposes the use of graph databases (Neo4j in this case) to represent and store trajectory data, which can thus be analyzed at different aggregation levels using graph query languages (Cypher, for Neo4j). Through a real-world public data case study, the article shows that trajectory queries are expressed more naturally on the graph-based representation than over the relational alternative, and perform better in many typical cases.

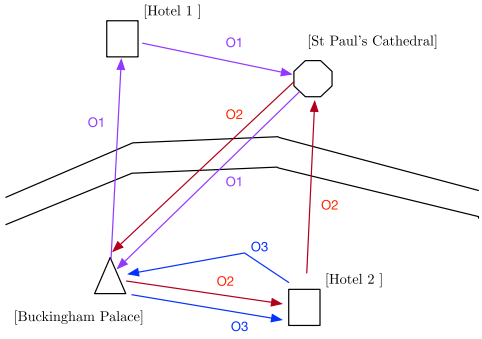
1 | INTRODUCTION AND MOTIVATION

Moving object data (MOD) applications (Güting & Schneider, 2005) have long been a relevant topic for the GIS community. The behavior of moving objects can be traced using location-aware devices (e.g., GPS, RFID). This produces trajectory data, which can be analyzed in order to obtain interesting mobility patterns (Renso, Spaccapietra,

& Zimányi, 2013). The trajectory of a moving object is given by samples consisting of a finite number of ⟨*Oid*,*t*,*x*,*y*⟩-tuples, meaning that at a moment in time *t*, the object with identifier *Oid* is located at coordinates (*x*,*y*). Different kinds of analyses can be performed on such data (Giannotti, Nanni, & Pedreschi, 2006; Giannotti, Nanni, Pinelli, & Pedreschi, 2007, Karli & Saygin, 2009). Trajectory analysis can not only be performed on the original (raw) trajectories, but also on a database constructed based on the ideas introduced by Spaccapietra and co-workers (Parent et al., 2013; Spaccapietra et al., 2008), where it is assumed that objects move over a background map consisting of disjoint geometrical figures to which semantically meaningful attributes are associated. These geometrical figures are referred to as *places of interest* (PoIs) *of the application*. Typically, they depend on the application area. In a tourist application, usual examples of PoIs are restaurants, historical buildings and hotels, while for traffic analysis PoIs could be defined as interesting road junctions or large parking lots. A PoI is considered as a *stop* when a moving object remains in it for a length of time above some threshold, in which case all (*x*,*y*) points of a trajectory that are located inside the PoI are transformed to the spatial object that represents this stop. As such, each object's trajectory, being a sequence of points, can be transformed into a sequence of stops. Thus, trajectory analysis can be applied to these transformed trajectories, which are called *semantic trajectories*, given that they can provide more information than that provided by the (*t*,*x*,*y*) points alone.

The intuition with regard to the problem addressed in this article is given next. Figure 1 (left) shows a simplified version of part of a map of London, which shows two hotels, labeled as Hotel 1 and Hotel 2 in the figure and denoted by H_1 and H_2 in the table on the right. The map also shows St Paul's Cathedral and Buckingham Palace. Furthermore, the map shows the movement of the objects O_1,O_2,O_3. Object O_1 moves from Hotel 1 to the Cathedral, then to the Palace, where it remains for some time, and then returns to its hotel. Object O_2 moves from Hotel 2 to the Cathedral, next to the Palace (where it spends a few hours) and finally returns to its hotel. Object O_3 leaves Hotel 2, visits the Palace, and returns to Hotel 2. Figure 1 (center) shows a portion of a table containing the raw trajectories (i.e., expressed for each object as ⟨*t*,*x*,*y*⟩-tuples). Figure 1 (right) gives the table with the PoIs corresponding to the application (details of how these tables are obtained are beyond the scope of this article). The points belonging to the same trajectory are temporally ordered, and identified by an object identifier. In this setting, a data scientist may pose queries like "How many persons went from Hotel 1 to St Paul's Cathedral, and then to Buckingham Palace (stopping to visit both places) during the same day?" An analyst may also want to identify interesting patterns in the trajectory data, or be interested in queries such as "Give the percentage of trajectories visiting two restaurants in the same day."

A typical way of performing trajectory analysis is to store trajectory data in relational databases, for example, in repositories called *trajectory data warehouses* (Leonardi et al., 2014; Vaisman & Zimányi, 2013), over which online analytical processing (OLAP) is performed. OLAP refers to a collection of operations for exploiting multidimensional databases. In a multidimensional database, data are perceived as data cubes, such that the axes in the cubes are called dimensions, and cells in the cubes contain one or more measures that quantify



O_{id}	t	x	y
O_1	1	x_1	y_1
O_1	2	x_2	y_2
O_1	3	x_3	y_3
O_1	4	x_4	y_4
...	...	...	...
O_2	5	x_5	y_5
O_2	6	x_6	y_6
O_2	7	x_7	y_7
...	...	...	...
O_3	4	x_5	y_5
O_3	5	x_8	y_8
O_3	6	x_9	y_9
...	...	...	...

O_{id}	g_{id}	t_s	t_f
O_1	H_1	1	10
O_1	C	20	30
O_1	P	50	70
O_1	H_1	100	140
O_2	H_2	5	20
O_2	C	25	40
O_2	P	50	80
O_2	H_2	120	140
O_3	H_2	4	10
O_3	P	15	40
O_3	H_2	60	140

FIGURE 1 Introductory example (left), raw trajectories (center), and PoI-based trajectory (right)

facts. Dimensions are further organized in aggregation hierarchies, thus measures can be aggregated along them. Queries in OLAP consist of sequences of operations that manipulate the data cube. The most usual of these operations allow aggregating and disaggregating measure values in the cube cells along the dimensions (Roll−Up and Drill−Down operations, respectively); selecting a portion of the cube (Dice); or projecting the data cube over some of its dimensions (Slice). A problem with this approach, particularly with the huge volumes of data now available, is the "impedance mismatch" between the way in which data are modeled and stored. Given that trajectories can be typically considered as graphs, storing trajectory data as relations may seem unnatural, since current database technology provides solutions that allow graphs to be stored in native form, as explained next. Thus, this article discusses how this OLAP-style semantic trajectory analysis can be performed on graph databases.

Property graphs are graphs whose nodes and edges are annotated with properties (Robinson, Webber, & Eifrém, 2013). They are typically used to model networks (e.g., social networks, sensor networks) to perform data analysis. The property graph data model is an abstraction that can be used to represent trajectories, in either their raw or semantic form. In this model, for example, the point coordinates or PoIs can be represented as nodes, and there is an edge from one point (or PoI) to the next one in the sequence of points (or PoIs) in the trajectory. In addition, spatio-temporal coordinates can be included as properties, as well as other characteristics of the places visited. Also, hierarchical contextual data can be defined, which would allow the trajectory graph to be represented at different granularities, leading to the notion of *trajectory aggregation*. With these tools, a data scientist may also perform OLAP-like analysis on trajectory graphs. Several different graph data models for this (henceforth called graph OLAP) can be found in the literature, and are discussed in Section 2. Modeling trajectories using graphs allows them to be stored in native form (i.e., as graphs) using graph databases (Angles, 2012, 2018), rather than in relational databases, thus preventing the "impedance mismatch" problem mentioned above. In particular, in this article, the graph database software Neo4j is used (http://www.neo4j.com). The Neo4j community has developed several libraries of functions; one of these is a spatial library (https://neo4j-contrib.github.io/spatial/) which allows different spatial layers to be defined that can enhance the possibilities for analysis.

Surprisingly, a careful analysis of the literature reveals that there is almost no work on this subject, in light of which this article aims to answer the following questions. Can graph databases be successfully used to model, store, and query semantic trajectory data? If so, what kinds of queries could benefit most from this approach? In particular, the article focuses on analytical queries, that typically require aggregating the trajectory graph up to different granularity levels. In addition, the article aims to show that using graph databases has several advantages over the typical solution of storing semantic trajectory data in relational databases, based on the following assumptions and facts. First, as already mentioned, graphs are a natural way to represent trajectories. Second, storing trajectories in a graph database provides a powerful set of algorithms to exploit these data, at no extra cost. Ad hoc alternatives require using different tools for different problems (e.g., querying, finding patterns). Third, graph databases such as Neo4j include a high-level graph query language (in the case of Neo4j, Cypher), together with a large collection of functions included in plugins that are easily added to the database, while the relational solution requires the use of different languages for the different tasks. Finally, expressing queries using graph query languages is far more intuitive for a non-expert user, than doing so through, for example, complex SQL queries.

A real-world running example is used for this study, based on the Foursquare New York data set (https://www.kaggle.com/chetanism/foursquare-nyc-and-tokyo-checkin-data set). This data set includes about 10 months of check-in data in New York City, collected from the Foursquare social network. In addition, a Time dimension hierarchy, and a Stop dimension hierarchy are defined as contextual information. To make the analysis more interesting, the data set is enriched with geographic New York City data.[1] It is worth noting that Foursquare is a location-based social network, therefore check-ins in this data set do not conform to typical GPS-based trajectories.

However, taking into account the aims of this research, these sequences of check-ins can be considered as semantic trajectories where stops have no duration, and there is no move between these stops (or, equivalently, it can be assumed that a "move" occurs between two zero-duration stops).

In summary, to address the research questions above, the following tasks are performed:

1. A property graph data model for representing semantic trajectories is set up. This model is based on the family of graph OLAP modeling techniques to favor aggregation of trajectory data represented as graphs.
2. The model is applied to the Foursquare New York data set, enriched with dimensional contextual information and spatial data, and implemented on a Neo4j database. In addition, the trajectory database is implemented on a PostgresSQL relational database.
3. A collection of 12 analytical queries, classified into five classes, is defined. These queries are written in the Cypher query language for the Neo4j implementation and in SQL for the relational implementation.
4. The queries are run and the results discussed.

As will be explained, the study shows that most of the hypotheses are verified. Queries in the graph-based trajectory model are not only more natural to express than in the relational alternative, but also show better performance (ranging from 1.2 to 7 times faster) than the latter in three of the five classes of queries studied, particularly when sequential patterns are looked for (e.g., in queries like "Find trajectories that go directly from a home to a station and then to an airport"), and when aggregation is involved (e.g., "Compute the total length of each trajectory, as the sum of the distance between each pair of consecutive stops").

The remainder of this article is organized as follows. In Section 2 related work is discussed. Section 3 very briefly presents some basic notions on trajectories and OLAP on graphs. Section 4 introduces the data model, while in Section 5 a case study is discussed. Section 6 concludes the article and addresses future work and open problems.

2 | RELATED WORK

The field of MODs has been extensively studied by the GIS community. The interested reader is referred to Güting and Schneider (2005) for a survey of this large area. Several techniques for the semantic annotation of trajectory data have been proposed and studied. du Mouza and Rigaux (2005) propose a model in which raw trajectory data are transformed in a sequence of moves (zones represented by labels or IDs). They also define a regular expression-based query language that allows queries for mobility patterns. Giannotti et al. (2007) introduce temporally annotated sequences as a basis for trajectory pattern mining. A trajectory pattern is defined as a collection of trajectories that visit the same places in sequence, with similar time gaps between each of these places. The concept of a region of interest (RoI) dynamically computed from the trajectories is defined. With a similar idea, Spaccapietra et al. (2008) define "stops and moves" to semantically enrich trajectories. Alvares et al. (2007) study trajectory analysis based on the concepts of stops and moves. The concept of stop here differs from the notion of RoI: the former is application-dependent, defined in advance and really relevant to a trajectory, while the latter is detected dynamically. Finally, Gómez and Vaisman (2013) presented RE-Spam, a language for discovering sequential patterns in semantic trajectories, based on regular expressions. Parent et al. (2013) provide a comprehensive description of the notions of trajectory and semantic trajectory.

Regarding graph databases, two database models are used in practice: (a) models based on RDF (https://www.w3.org/RDF/), 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 arbitrary objects to be described in terms of their attributes and their relationships to other objects. Informally, a collection of RDF triples is an RDF graph. In models of type (b) (Angles et al., 2017), nodes and edges are labeled with a sequence of attribute-value pairs. This is an extension of classical graph database models, frequently used for implementations in practical applications. The main reason for storing attributes in nodes and edges is to speed up the retrieval of the data directly related to a certain node. For an extensive and comprehensive bibliography on graph database models, the interested reader is referred to Angles and Gutierrez (2008) and Angles (2018). Although the models in (a) have a general scope, the structure of RDF makes them not as efficient as the other models, which are aimed at reaching a local scope. An important feature of RDF-base graph models, however, is that they follow a standard, which is not yet the case for the other graph databases, therefore they are typically used for metadata representation. Therefore, many works have proposed RDF to annotate trajectories with semantic information (da Silva, Times, de Macêdo, & Renso, 2015; Fileto et al., 2015; Ruback, Casanova, Raffaetà, Renso, & Vidal, 2016). Hartig (2014) 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 (https://www.w3.org/TR/rdf-sparql-query/), the standard query language for the Semantic Web. The model used in the next sections to represent and query trajectory data is based on the concept of property graphs.

Several data models to perform OLAP (Kimball, 1996) on graphs have been proposed. GraphOLAP (Chen, Yan, Zhu, Han, & Yu, 2009), conceptually, is a framework for OLAP on a set of homogeneous graphs, based on splitting the graph into a collection of snapshots that are aggregated in two ways, called informational and topological OLAP aggregations. GraphCube (Zhao, Li, Xin, & Han, 2011) provides a framework for computation and analysis on OLAP cubes using the different levels of aggregation of a graph. This framework introduced the notion of cuboids. A recent proposal, called Graph OLAP (Gómez, Kuijpers, & Vaisman, 2017), models the problem as basic graph data (at the finest granularity defined for the application), background information in the form of dimension hierarchies, and a collection of so-called graphoids (the basic graph aggregated at different granularity levels defined by the background dimensions). Analogously to the models commented above, the classic OLAP operations (Roll–*up*, Slice, Dice, Drill–*down*) are also defined in terms of the components of the model.

Regarding the use of graph databases for analyzing trajectory data, Gryllakis, Pelekis, Doulkeridis, Sideridis, and Theodoridis (2018) implemented in Neo4j the Hermes MOD originally developed in Oracle (Pelekis, Sideridis, & Theodoridis, 2015). This is basically a datatype system for representing semantic trajectories. The authors also extended the Neo4j spatial plugin to facilitate operations on semantic trajectories. In particular, the authors address the problem of answering what they call spatio-temporal keyword pattern (STKP) queries (Gryllakis, Pelekis, Doulkeridis, Sideridis, & Theodoridis, 2017). Queries of this kind ask for episodes satisfying a pattern which may include keywords, spatial and temporal conditions. For example, an STKP query may ask for trajectories starting with an episode (basically a stop) whose geometry is contained in a certain bounding box, followed by an indefinite sequence of episodes.

As far as the authors of this work are aware, the only work discussed above that somehow compares with that presented here is the work by Gryllakis et al. (2018). However, such work is aimed at extending Hermes with datatypes, and the Neo4j spatial plugin, to answer a specific kind of queries. The work in the present article focuses on modeling and storing semantic trajectories as graphs, in order to support analytical (OLAP) queries of different kinds, topics not addressed elsewhere. This approach can therefore be applied to any graph database, although in this article Neo4j is used, with no addition whatsoever.

3 | BACKGROUND

Some background on the topics addressed in this article is now given, to make the article self-contained. Basic notions on trajectories are introduced first. The second part of this section briefly presents the graph OLAP data model that will be used in the reminder of the article.

3.1 | Trajectories and semantic trajectories

The following definitions formalize the intuitive notions given in the example of Section 1. The notion of "trajectory" is defined first.

> **Definition 3.1.** A *trajectory* is a sequence $\langle(t_0, x_0, y_0), (t_1, x_1, y_1),...,(t_N, x_N, y_N)\rangle$ of spatio-temporal points, where, for $i = 0,...,N$, $t_i, x_i, y_i \in \mathbf{R}$. Their order $t_0<t_1<\cdots<t_N$ induces a natural order on the time-space points in the trajectory. The *time domain* of the trajectory is the interval $[t_0,t_N]$.

A table like the one in Figure 1 (center) is called a moving object table (MOT). In practice, MOTs can contain huge amounts of data. Thus, querying raw trajectory data may be extremely time-consuming. Furthermore, data scientists are often not concerned with such level of geometric detail, but are looking for more aggregated information. Also, answering queries may require semantic information that is not present in the MOT. As a solution, the literature in the field proposes using the notion of *stops and moves* to reduce the size of the MOT. Thus, a trajectory can be represented in terms of so-called places of interest for a particular application, characterized as what are referred to as *stops*. This concise table (see Figure 1 (right)) cannot encode the complete information contained in the MOT. However, it allows information of interest to be quickly accessed without having to consult the complete data set. For this, the notion of "place of interest of an application" must be defined first (Alvares et al., 2007).

> **Definition 3.2.** A *place of interest C* is represented by a tuple (R_C, Δ_C). The geometrical figure R_C is a (topologically closed) polygon, polyline or point in the plane $\mathbf{R}^2$, called the *geometry* of *C*, and Δ_C is called the *minimum duration* of *C*.

Given an application $\mathcal{A}$, the *places of interest of* $\mathcal{A}$, denoted by $\mathcal{P}_\mathcal{A}$, are a finite collection of PoIs (relevant to this application) whose geometries do not mutually intersect.

> **Definition 3.3.** Let $T=\langle(t_0, x_0, y_0), (t_1, x_1, y_1),...,(t_n, x_n, y_n)\rangle$ be a trajectory and let $\mathcal{P}_\mathcal{A} = \{C_1 = (R_{C_1},\Delta_{C_1}), \ldots, C_N = (R_{C_N},\Delta_{C_N})\}$ be the PoIs of an application $\mathcal{A}$. A *stop of T with respect to* $\mathcal{P}_\mathcal{A}$ is a contiguous subtrajectory $\langle(t_i, x_i, y_i), (t_{i+1}, x_{i+1}, y_{i+1}),...,(t_{i+\ell}, x_{i+\ell}, y_{i+\ell})\rangle$ of *T* of maximal size such that for some $k \in \{1,...,N\}$ the following hold: (a) $(x_{i+j},y_{i+j}) \in R_{C_k}$, for $j=0,1,...,\ell$; and (b) $t_{i+\ell} - t_i > \Delta_{C_k}$. That means, if the user stays more than the Δ threshold, the place is considered a stop. A *move of T with respect to* $\mathcal{P}_\mathcal{A}$ is a maximal contiguous subtrajectory of *T*:

1. in between two temporally consecutive stops of *T*;
2. between the starting point and the first stop of *T*;
3. between the last stop of *T* and ending point of *T*;

or the trajectory *T* itself, if *T* has no stops.

There are many possible variations of the definition of stops and moves of a trajectory, depending, for instance, on the interpolation technique used on the trajectory samples, or in the tolerance used to consider whether an object is inside or outside a PoI (Parent et al., 2013, Spaccapietra, Parent, & Spinsanti, 2013). Discussing these alternatives is beyond the scope of this article. Intuitively, semantic trajectories are produced by replacing a sequence of $\langle t,x,y\rangle$-tuples by a sequence of stops, taken from the collection of PoIs. More formally, we have the following definition.

> **Definition 3.4.** A *semantic trajectory* is a trajectory (see Definition 3.1) with added semantic annotations. Formally, a semantic trajectory is a structure of the form $\mathcal{T}_s(O_{id},\mathcal{S})$, where O_{id} is a moving object identifier, and $\mathcal{S}$ is a sequence of pairs of the form (s_i,L_i), where s_i and L_i are defined as follows: s_i is a stop (Definition 3.3) traversed by O_{id}; L_i is a list of pairs (*metadata*,*value*), where *metadata* is an attribute representing a characteristic of s_i and *value* is the value of such attribute for s_i. Moreover, $\mathcal{S}$ is time-ordered, which means that the s_i are listed in the order they were traversed by the object with O_{id} in $\mathcal{T}_s$.

3.2 | Graph OLAP

The model adopted in this article for representing trajectories as graphs is composed of three main types of elements. The first type is a collection of *OLAP dimension hierarchies* that represent contextual (or background) information for the graph (trajectory) data. Dimensions have schema and instances, as usual in databases. A dimension schema is a lattice with a unique top and a unique bottom. Each node in a dimension schema is called a dimension level, which, in turn, is associated with level instances, containing elements of a certain domain. Dimension instances are collections of hierarchies. The second element in the model is the *base graphoid*. Assuming a collection of dimensions $D_1,...,D_d$ in a certain application domain, the nodes and edges of the base graphoid are defined at the bottom levels of the background dimensions (i.e., $D_1,...,D_d$). The nodes and edges in a graphoid have a type, associated with the corresponding background dimensions. The third type of element in the model are the *graphoids* defined at different levels of granularity (these levels are defined in the background dimensions).

Figure 2 illustrates the above for the running example (the complete example is detailed in Section 4 and depicted in Figure 3). The base graphoid, representing the trajectory data, is shown in the upper part of the figure. Here, the Stop nodes are linked by edges labeled #trajstep. Background dimension hierarchies for the Stop and Time dimensions are defined as Stop→Venue→Categories→Category and Minute→Hour→Day→Month→Year hierarchies, respectively. The base graphoid is defined at the Stop and Instant granularity levels of these dimensions. Other trajectory graphoids can be defined at different dimension levels, climbing along the Stop and Time hierarchies. For example, a trajectory graphoid called a (Stops.Category, Time.Datehour)-graphoid is the base graphoid aggregated at the [#Datehour] and [#Category] levels, and it is depicted in the bottom part of Figure 2.

For this model, a collection of operations are defined, analogously to the classic OLAP operations that are performed over multidimensional data cubes. This way, the Climb operation replaces each node in a graphoid with the corresponding node according to the associated dimension level. For example, if an attribute of a node representing trajectory stops (as in Figure 2) is instant, a climbing operation to the Hour level along dimension Time will produce the (Time.Datehour)-graphoid. Other operations are defined for the graph model resembling

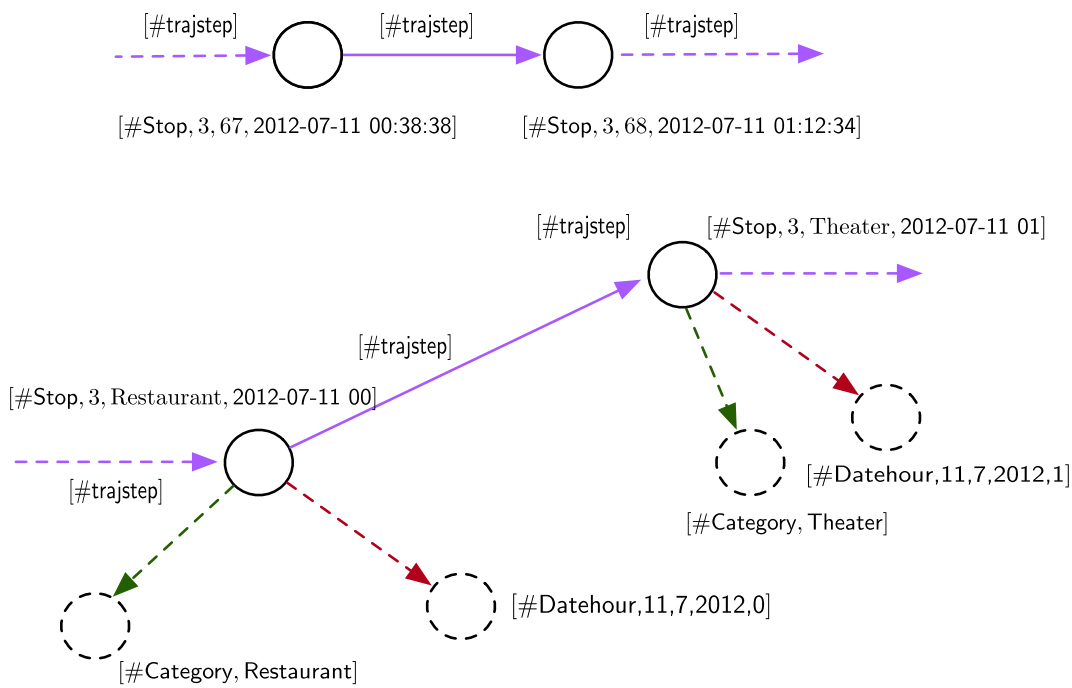


FIGURE 2 Base graphoid and (Stops.Category,Time.Datehour)-graphoid for the trajectory graph in the running example

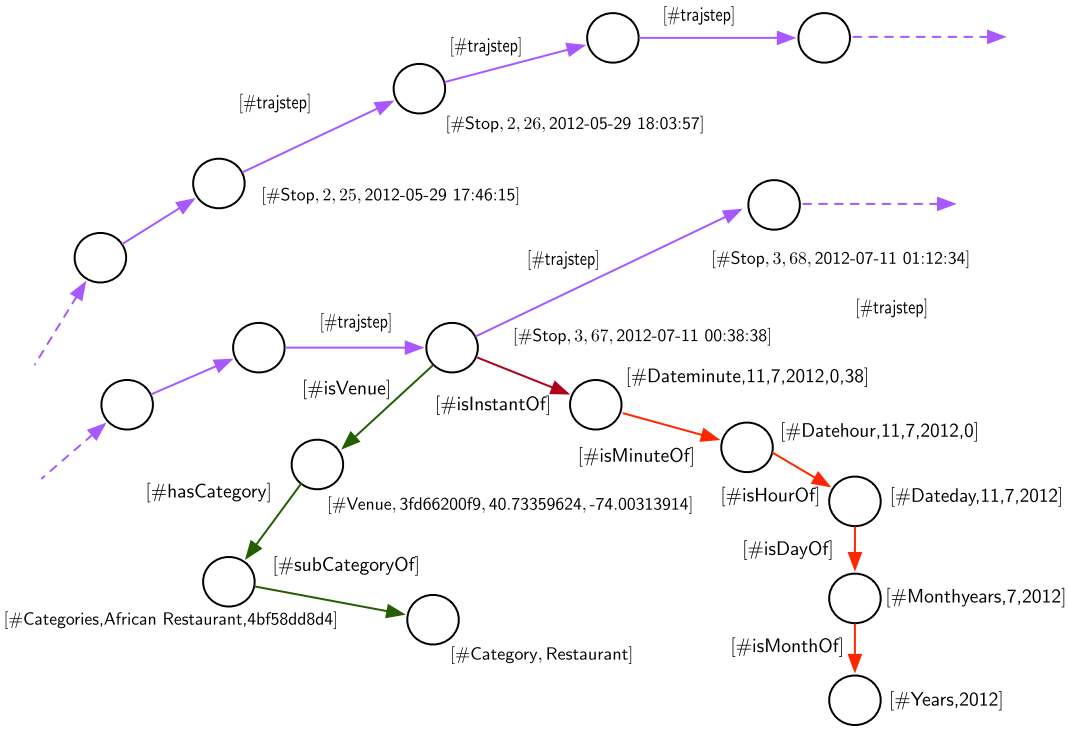


FIGURE 3 Portion of the trajectory graph instance

the corresponding operations on cubes. The Roll−*Up* operation takes a graphoid to a coarser granularity level along a dimension D_d up to a level ℓ_{up}, and performs an aggregation over a collection of measures. The Drill−Down operation does the opposite of Roll−Up, taking a graphoid to a finer granularity level, along a dimension D_d, down to a level ℓ_{down}. The Dice operation produces a subgraphoid of another one, whose nodes satisfy a Boolean condition φ defined over a elements in the graphoid model. All these operations will be further explained in the running example in Section 5.

4 | REPRESENTING AND STORING A TRAJECTORY AS A GRAPH

This section shows, through an example, how a trajectory can be represented using the property graph data model and loaded into a graph database. First, the running example to be used in the remainder of the article is presented. Then this example is modeled as a trajectory graph.

4.1 | Running example

As mentioned in Section 1, the running example considers data from the Foursquare New York data set. This data set includes about 10 months of check-in data in New York City, with a total of 227,428 check-ins collected from the Foursquare social network. When someone checks in somewhere (e.g., an airport, a restaurant, her private home), a point is recorded. Data from more than a thousand users are stored in this data set. The data set is used to analyze how people move in the city. Further, the data set is enriched by adding spatial and non-spatial contextual information to the PoIs. Such information is organized as aggregation hierarchies, so semantic trajectories can be represented at different granularity levels. For example, in the data set, a point (x,y) is represented as a

(zero-duration) stop by means of a (latitude, longitude) pair. This stop can be a train station, which in turn is categorized as "Station". The next stop may be a Thai restaurant, which, at a coarser level, is categorized as "Restaurant". Thus, this subtrajectory becomes a sequence of the form ⟨... Station,Restaurant,... ⟩. A temporal hierarchy is also used for analysis. For example, a check-in at a PoI at 3 p.m. can be aggregated as an "afternoon" stop if the analyst is not interested in a finer level of detail. Note that the original data set contains the check-in data, and the data enrichment has been performed to enhance this use case, as is explained below.

Many interesting queries of different types can be posed in the scenario described above. For example, since check-ins in this data set also include public transport, queries like the following can be expressed:

- "List the users moving between places of interest using a taxi."
- "Which users travel to an airport by taxi during the night?"

The second query implies an aggregation along the temporal dimension. As another type of problem, interesting patterns may be investigated, such as:

- "Return the trajectories in which users go from their homes to an airport after 5 p.m."
- "Give the percentage of trajectories in which users go from a restaurant to a sport event and end at a coffee shop."

The addition of spatial data layers allows queries such as:

- "Compute the number of users moving between two or more boroughs in the same day."
- "Compute the average distance traveled per user and per day"

Accounting for all of the above, the graph trajectory model is presented next.

4.2 | Modeling the trajectory graph

The Foursquare New York data set contains check-in data of 1,083 users at different places, or "venues." Thus, there are 1,083 trajectories of different lengths. For this study, long trajectories are not split into smaller ones in a "preparatory phase." In any case, this could be done through queries, asking, for example, for places visited by a user on a certain day. Each row in the data set contains the following information: the user identifier, denoted userId in the table below; the identifier of the place where the user checked-in, called venueId; the categorization of the venue, with the identifier and the category name, called, respectively, venueCatId and venueCat (these describe the kind of venue, such as private home, Thai restaurant); the geographic coordinates of the stops, denoted latitude and longitude; the timezone offset, called t–*zone* in the table; the time-stamp for the stop, called timestamp (representing the check-in time for the user at the stop). These data were loaded into a Postgres relational database and an additional field was added, indicating the relative position of the stop in the trajectory, called pos. The resulting relational schema and an example tuple are depicted next.

The Trajectories table has the form:

userId	venueId	venueCatId	venueCat	lat.	long.	t-zone	timestamp	pos
1	4abc1....	4bf5...	SeafoodRest.	40.78..	–73.97.	–240	2012–04–0423:31:31	1
...	...	...	...	...	...	...	...	...

There is also a Categories table, not in the data set. This table has been included in order to make the case study more interesting, and to add semantics to the trajectory data. The idea is to further categorize venues.

For example, in the Trajectories table, the venueCat attribute in the first row details the kind of venue. Thus, a "Restaurant" category type can be defined, to aggregate data at a coarser level of detail. Actually, this represents a level in an aggregation hierarchy. This new aggregation level is materialized by the catType attribute in the Categories table below. This categorization been produced manually by analyzing all kinds of venues one by one, and assigning to them a category type. All in all, 33 category types have been defined. The data set defines 251 venue categories, and each one of them has been assigned to one category type. In summary, 38,333 venues are classified into 251 categories, which in turn are classified into 33 category types.

venueId	venueCat	venueCatId	catType
4aa06479f964a5..	PizzaPlace	4bf58dd...	Restaurant
...	...	...	...

To model semantic trajectory data as a graph, with graph aggregation in mind, the article adopts the graph OLAP model described in Section 3.2. The base graphoid is composed of the trajectories themselves. Each node in the base graphoid represents a stop in the trajectory, and has properties (attributes) userid (i.e., the trajectory id), instant (the time instant when reaching the stop), and position (the relative position of the stop in the trajectory). Note that, as in any conceptual design, many modeling options can be considered (e.g., regarding the representation of an object as a node or as an attribute, or placing an attribute in a node or in a relationship). The one chosen for this use case is only one of them.

In order to analyze trajectories along contextual dimensions (called background dimensions in graph OLAP), and at different levels of granularity, these dimensions must be constructed and associated with the base graphoid (in what follows, the "trajectory graph"). In this case, hierarchies for Stop and Time dimensions are built. For the former, fields in the original data set are used, together with the categories defined for each kind of venue. The latter is constructed using software libraries (as explained below).

The Time hierarchy aggregates data from the instant represented in the stops, up to the year level as follows (this is a conceptual representation, the actual one is described below):

$$\text{Minute} \rightarrow \text{Hour} \rightarrow \text{Day} \rightarrow \text{Month} \rightarrow \text{Year}.$$

The background dimension for stops is defined as follows. The bottom level of the hierarchy, that is, Stops (represented in the trajectory graph), is associated with the level Venue, which in turn is associated with level Categories through a relationship denoted by hasCategory (the category of the venue is included in the data set, and denoted by venueCat in the table above). The hierarchy is completed with the level Category such that there is an *m*-to-one relationship, denoted by isSubcategoryOf, from Categories to Category. The instance of the level Category corresponds to the attribute catType in the table Categories. For example, the element AfghanRestaurant at the Categories level is associated with the element Restaurant at the level Category. Thus, the Stop dimension hierarchy is (conceptually) of the form:

$$\text{Stop} \rightarrow \text{Venue} \rightarrow \text{Categories} \rightarrow \text{Category}.$$

Note that although the name Categories for the level may seem confusing, the decision to keep it (instead of, for example, subCategories), is led by the intention to remain faithful to the original data.

Figure 4 depicts the schema of the trajectory database expanded with the background dimensions, using the graph OLAP notation. Stops in each trajectory are represented by a node of type #Stops, with its corresponding attributes. Between each pair of stops there is an edge labeled trajstep. The figure also shows the schema of the

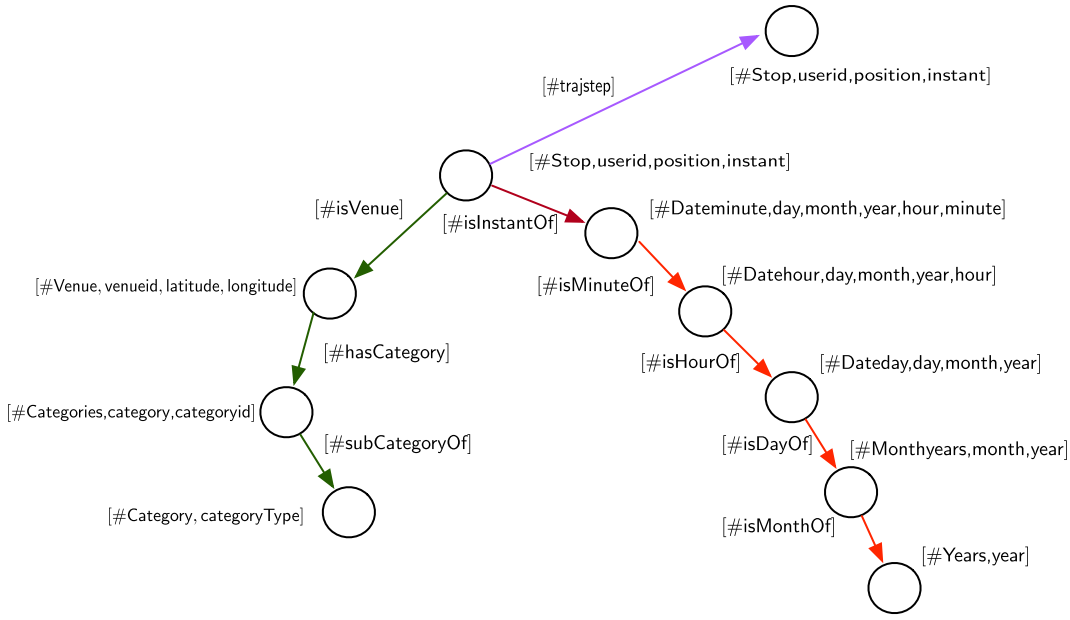


FIGURE 4 Schema of the trajectory database

background dimensions. Each aggregation level in the dimension hierarchies is shown as a tuple containing the node type and its corresponding level attributes. For example, in the Stop dimension, the level Venues is represented as the tuple [#Venue,venueid,latitude,longitude]. The relationships representing the hierarchy for the venues are indicated by the edges labeled [#isVenue], [#hasCategory], and [#subcategoryOf]. The Time dimension is represented analogously. Note that, although in this case the relationships do not contain attributes, in general this will not be the case.

Figure 5 shows a portion of an instance of the trajectory graph, represented as a property graph using the graph OLAP notation. Two trajectories are shown, namely for user identifiers 2 and 3. At the instance level, there is an edge between two stops s_1 and s_2 if s_2 is the stop occurring immediately after s_1. For the latter trajectory, the complete instance of the hierarchy for the stop in position 67 is shown. It can be seen that, for each node attribute in the schema, there is a value in the instance. Also, note that although is this case the data are quite structured, one feature of a graph data model is that the schema is totally flexible and unstructured, meaning that, for example, it is not a requirement that all nodes contain the same attributes.

4.3 | Storing the trajectory graph

The last step of this process involves loading the trajectory graph into the Neo4j graph, which is quite straightforward, and therefore details are omitted here for the sake of brevity. To build the Time dimension, functions in the APOC library for Neo4j are used (https://neo4j.com/developer/apoc/). A portion of the resulting graph is shown in Figure 3 (the attributes are not shown, because of the graph interface). The edges between stops are highlighted in bold. The Time hierarchy is not shown for the sake of clarity. It can be seen, for example, that node 86 corresponds to the venue with id 11943, which in turn is a vegetarian restaurant, further classified as a restaurant. The path between nodes 84 through 87 is also shown, with edges labeled trajstep.

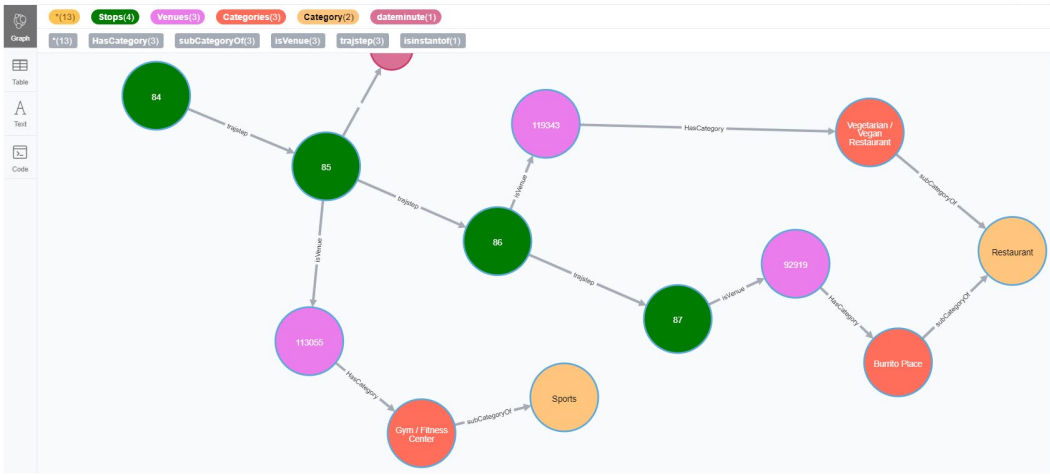


FIGURE 5 The trajectory graph in a Neo4j database

In addition, three spatial layers are defined (not shown in the figure), namely nycdistricts, new_york_highway, and nyorkpois, representing, respectively, the districts, highways and places of interest in New York City. These layers were imported from public data, and loaded into the Neo4j database.

The trajectory graph is now ready to be exploited with the Neo4j tools, and queried using the high-level language Cypher. This is addressed in the next section.

5 | CASE STUDY

The trajectory graph described in Section 4 can be analyzed in many ways. The first part of this section presents a collection of queries classified into five query types. Note that the model is aimed at addressing analytical queries, which take advantage of the contextual dimensions of the graph OLAP data model used here, in order to study semantic trajectories at different granularity levels. Thus, most queries include climbing and aggregating data along the dimension hierarchies, and the operations *à la* OLAP involved are explained for each query. Queries are expressed in the Cypher query language, the high-level language that comes with Neo4j. Only the Cypher expressions which may be intuitively understood even by non-expert readers have been included. The remaining queries are detailed in the appendix. The second part of the section discusses the performance of the queries, and compares it (when possible) against the analogous relational ones.

5.1 | Analytical semantic trajectory queries

This section addresses queries aimed at helping in the decision-making process, for example, for transport policy-makers, companies considering starting a new business and the like. Queries are organized into classes that account for their main characteristics.

5.1.1 | Queries computing non-recursive patterns

These are typical queries in trajectory analysis. Below, an example of a query asking for a simple pattern is presented. Complex patterns are discussed later on, when studying transitive closure queries. The graph OLAP model allows patterns at different granularity levels to be found. In this case, note that the semantic trajectory graph is defined at the granularity of the stops, and the query below works at the Category level.

Query 5.1 Find the trajectories that go from a private home to a station and then to an airport, without intermediate stops.

```
MATCH (cat1:Category{categoryType:'Home'})<-[*3..3]-(s1:Stops)-
      [r:trajstep]->(s2:Stops)-[:trajstep]->(s3:Stops)-[*3..3]->
      (cat3:Category{categoryType:'Airport'})
WHERE s3.position=s1.position+2
MATCH (s2)-[*3..3]->(cat2:Category{categoryType:'Station'})
WITH s1 order by s1.position
RETURN s1.userid, collect(distinct s1.position) order by s1.userid
```

The first `MATCH` clause describes a pattern matching a sequence of three consecutive stops; the first and last ones include OLAP Climb operations along the Stops dimension up to the Category dimension level (the coarser level in the hierarchy). The second `MATCH` describes the same climbing pattern for the intermediate stop. At the Category level, a Dice operation is performed, keeping the desired category, such that only the required three-stop patterns are kept. The `WITH` clause before the last line acts like a "pipe," which passes variables from one portion of the code to the next one (in this case, it passes `s1`). The result is given as pairs of the form (userId, ⟨LOP⟩), where userId identifies the trajectory, and LOP is a list of the initial positions of each pattern, within the trajectory. Note that in the climbings a shorthand is used, since there is only one possible path up to the Category level, and no variables are needed over any intermediate level. If the climbing path were needed, the second `MATCH` clause would read:

```
(s2)-->(:Venues)-[:HasCategory]->(:Categories)-[:subCategoryOf]->
(cat2:Category{categoryType:'Station'})
```

The aim of the next query is to compare performance for a longer pattern.

Query 5.2 Find the trajectories that go from a bar (or similar) to a restaurant, again to a bar (or similar) and end at a restaurant, without intermediate stops.

The Cypher expression for this query is given in the Appendix and is further discussed in Section 5.3.

5.1.2 | Queries computing traveled distance

Queries in this class compute the distance between points, but they do not use spatial layers or special libraries. The functions are part of the Cypher language.

Query 5.3 For each trajectory, compute the distance traveled between each pair of consecutive stops.

```
MATCH (s1:Stops)-[:trajstep]->(s2:Stops)
WITH point({longitude: s1.longitude, latitude:s1.latitude})
     AS p1, point({longitude:s2.longitude,
     latitude: s2.latitude }) AS p2, s1, s2, s1.userid AS user
RETURN user, s1.position, s2.position, round(distance(p1, p2))
     AS travelDistance order by s1.userid asc, travelDistance desc
```

Here, all consecutive pairs of stops are computed first (by pattern matching, rather than joins, which would be the case in the relational model). Then, the (latitude, longitude) pairs of each stop are obtained. Finally, for each trajectory, the distance between two consecutive stops is computed.

The next query shows how to easily analyze distances traveled between different kinds of places, combining characteristics of Queries 5.1 and 5.3 and including aggregation along background dimensions.

Query 5.4 For all trajectories that go directly from a private home to an airport, list the user identifier, together with the distance traveled between these two places each time that this pattern occurs.

```
MATCH (cat:Category{categoryType:'Home'})<-[*3..3]-(s1:Stops)-
      [r:trajstep]->(s2:Stops)-[*3..3]->
      (cat1:Category{categoryType:'Airport'})
WITH s1, s2, cat, cat1, point({longitude: tofloat(s1.longitude),
     latitude: tofloat(s1.latitude)}) AS p1,  point({longitude:
     tofloat(s2.longitude), latitude: tofloat(s2.latitude) }) AS p2
RETURN s1.userid, s1.position, s2.position, round(distance(p1, p2))
     AS travelDistance  order by travelDistance desc
```

It can be seen that this query first performs two climbings up to the Category level to select (Dice) only stops corresponding to homes and airports. Then the aggregation is computed at the bottom level of the dimension (i.e., it first climbs, and then computes the distance at the Stop level).

5.1.3 | Spatial trajectory queries

These queries make use of the spatial information contained in layers other than the one where the trajectories are represented. The query below uses the layers containing the places of interest ("nyorkpois"), and calls the function `withinDistance` contained in the Neo4j spatial library.

Query 5.5 Find the trajectories passing less than 100 m from a public school.

```
MATCH (s:Stops)
CALL spatial.withinDistance('nyorkpois', {latitude: tofloat(s.latitude),
     longitude:tofloat(s.longitude) },0.1) YIELD node as c, distance as a

WITH s, c, a WHERE c.NAME CONTAINS 'Public School'
RETURN s.userid, s.position, a
```

This query uses the NAME feature included in the spatial layer. The `spatial.withinDistance` function computes the distance between stops in the trajectories and the PoIs in the spatial layer. The query returns for each user (trajectory) the user identifier, the position of the stop in the trajectory, and the distance between the stop and the school.

The following query is more general, since it does not ask for a particular kind of PoI.

Query 5.6 List the trajectories starting at less than 300 m from a place of interest of the city, returning the trajectory identifier (i.e., userId), and the actual distance, for all the PoIs in the answer.

The Cypher expression is given in the Appendix.

5.1.4 | Aggregation queries

Queries in this class include different forms of aggregations, typical in data analytics.

Query 5.7 Compute the number of different categories of venues visited by month.

```
MATCH p=(s1:Stops)-[*3..3]->(scat:Category)
WITH p,s1,scat
MATCH (s1)-[*4..4]->(m1:MonthsYears)
RETURN m1.month as month, scat.categoryType, count(*) as qty
       order by  qty desc
```

Here, two climbings are required: one along the Stop dimension hierarchy up to the Category level, and another one along the Time dimension up to the MonthsYears level. Finally, the aggregation is performed. The climbing and the aggregation conform to a Roll–up operation.

Two more examples of aggregate queries are given next.

Query 5.8 Compute the number of stops per day per user, along with the starting position of each subtrajectory for each day.

Query 5.9 For each trajectory, compute its total length as the sum of the distances between each pair of stops.

The Cypher expressions for these queries are given in the Appendix. The queries are also discussed in Section 5.3.

5.1.5 | Transitive closure queries

Queries of this type are rather more complex than the previous ones, since they not only include graph OLAP operations or pattern to match, but also require computing the transitive closure of the trajectory graph. In the sequel, the main idea is explained. However, given that it may be hard for readers not familiar with Cypher to understand the details, most technical details are given in the Appendix.

Query 5.10 For each trajectory, find the paths that go from a private home to an airport in the same day.

This query requires some explanation. Several Climb operations are required along both background dimensions: (1) up to the Category level along the Stops dimension, to find stops corresponding to homes and airports; (2) up to the Day level along the Time dimension. Dice operations are finally used to filter out the subtrajectories not occurring during the same day, and to keep only the trajectories going from a home to an airport. The transitive closure of the resulting subgraph is finally computed.

```
MATCH (cat1:Category{categoryType:'Home'})<-[*3..3]-(s1 :Stops)
MATCH (cat2:Category{categoryType:'Airport'})<-[*3..3]-(s2 :Stops)
WHERE s1.userid = s2.userid AND s1.position < s2.position and
      apoc.date.fields(s1.instant, 'yyyy-MM-dd HH:mm:ss').years =
      apoc.date.fields(s2.instant, 'yyyy-MM-dd HH:mm:ss').years and
      apoc.date.fields(s1.instant, 'yyyy-MM-dd HH:mm:ss').months=
      apoc.date.fields(s2.instant, 'yyyy-MM-dd HH:mm:ss').months and
      apoc.date.fields(s2.instant, 'yyyy-MM-dd HH:mm:ss').days=
      apoc.date.fields(s1.instant, 'yyyy-MM-dd HH:mm:ss').days
WITH s1, apoc.coll.sort(collect(s2.position)) as firstAirports
WITH s1, head(firstAirports) as s2pos
MATCH  path= (s1)-[:trajstep*]->(s2 :Stops {position: s2pos})
RETURN s1, path
```

The climbings and dicings are computed in the first two parts of the query (the two `MATCH` clauses). Note that this is very intuitive, even for the non-expert user. There are two tricky parts, however. First, the climbing along the Time dimension is performed through a conjunction of Boolean conditions over the instant property of the nodes of type Stop. The APOC library is used to operate with dates. The reason is that, in this case, this turns out to be more efficient than performing two matchings along the Time hierarchy as follows:

```
MATCH (s1)-[:isinstantof]->()-[:isminuteof]->()-[:ishourof]->(d:dateday),
      (s2)-[:isinstantof]->()-[:isminuteof]->()-[:ishourof]->(d:dateday)
```

The second tricky part involved in this query is caused by the fact that the sequence of stops in the graph, at the Category aggregation level, can be of the form {Home, Home, Airport, Airport, Airport...}, and thus all combinations are included in the transitive closure. However, the query must only capture the first airport in that sequence. The `WITH s1, head(firstAirports)` clause does the job.

The next query partitions a trajectory into its subtrajectories using the date as a partition function, by means of keeping only the longest subtrajectory occurring within a day. The Cypher expression is given in the Appendix.

Query 5.11 For each day, and for each trajectory, find the longest subtrajectory.

As a final example, the next query makes use of the trajectory graph, together with map information, to find the districts from which people travel to an airport, together with the part of the day in which this travel starts.

Query 5.12 Give the districts in which people leave from their home to go to the airport before 3 p.m.

The Cypher expression is also given in the Appendix.

5.2 | Running the queries on Neo4j and PostgresSQL

Although query performance is not the core aim of this article, the queries in Section 5.1 are run on the Neo4j database designed and populated as described in Section 4. Further, in order to compare performance against a relational alternative, the queries are written in the SQL language, and executed on a PostgresSQL database. The exception are the queries that involve spatial functions, which it would be unfair to compare, since performance heavily depends on the spatial libraries used which, in the case of Neo4j, are still at an early stage of development. Both databases are indexed in order to obtain the best possible query performance.

For the Neo4j database, the numbers of nodes and edges are given in Table 1. For the PostgresSQL database, there are two tables: Trajectories, with 226,252 tuples; and Categories, with 38,333 tuples.

Queries are run on a machine with a i7-6700 processor, 12 GB of RAM and a 250 GB disk (actually a virtual node in a cluster). The execution times reported are the averages of five runs of each experiment.

5.3 | Discussion

Although comparing performance of a graph database like Neo4j against relational databases with more than 20 years in the marketplace may at first sight seem unfair, it can give an idea of the potential of the former, even for the current state-of-the-art software. The discussion that follows is organized in terms of the query classes defined in Section 5.1. Results are summarized in Table 2. In the discussion, some of the queries will be expressed in SQL, although not all of them, since it is assumed that the reader has at least a basic knowledge of SQL.

TABLE 1 Number of nodes and edges in the graph database

Type	Name	Size (#)
Node	Stops	2,26,345
Node	Venues	38,333
Node	Categories	400
Node	Category	33
Node	Dateminute	1,28,412
Node	Datehour	4,523
Node	Dateday	252
Node	MonthYears	11
Node	Years	2
Edge	trajstep	2,25,262
Egde	isVenue	2,26,345
Edge	HasCategoryOf	38,333
Edge	subCategoryOf	400
Egde	isInstantOf	226,345
Edge	isMinuteOf	1,28,412
Edge	isHourOf	4,523
Edge	isDayOf	252
Edge	isMonthOf	11
Total	#Objects	7,96,552

TABLE 2 Execution times for the example queries

1	2 #	3	4	5
Type of Query	**Query #**	**Neo4j (s)**	**Postgres (s)**	**3/4**
Non-recursive pattern	5.1	**0.18**	0.3	0.6
Non-recursive pattern	5.2	**0.41**	0.5	0.82
Travelled distance	5.3	**2.2**	31 (w/o distance)	0.07
Distance & pattern	5.4	**0.1**	0.25	0.4
Spatial	5.5	720	N/A	N/A
Spatial	5.6	22	N/A	N/A
Aggregation	5.7	**0.8**	1	0.8
Aggregation	5.8	**1.8**	4	0.45
Aggregation & distance	5.9	2.4	N/A	N/A
Transitive closure	5.10 (alt. 1)	**7.8**	60	0.13
Transitive closure	5.10 (alt. 2)	**27**	60	0.45
Transitive closure	5.11	99	**59**	1.67
Spatial & transitive closure	5.12	970	N/A	N/A

For *non-recursive pattern queries*, writing the Cypher query implies just writing the pattern the user wants to check, in a very simple and intuitive way, provided the user has a basic knowledge. In addition to this, these kinds of queries are very efficient on a graph database, which can be observed in Table 2: Queries 5.1 and 5.2 are

executed in Neo4j in 0.18 and 0.41 s, respectively, while the SQL equivalents take 0.3 and 0.5 s to execute. The reason is that the SQL queries require several joins and reads to be performed, as shown below for Query 5.1, while joins are solved by direct path navigation in Neo4j.

```
SELECT t1.userid, t1.tpos,t2.tpos,t3.tpos
FROM trajectories t1 join trajectories t2 on (t1.userid=t2.userid
     and t1.tpos+1=t2.tpos)
     join trajectories t3 on (t2.userid=t3.userid and t2.tpos+1=t3.tpos)
     join categories c1 on (t1.venueid=c1.venueid)
     join categories c2 on (t2.venueid=c2.venueid)
     join categories c3 on (t3.venueid=c3.venueid)
WHERE c1.cattype= 'Home' and c2.cattype='Station'
     and c3.cattype='Airport'
```

For *queries computing traveled distance*, Query 5.3 also performs better on the graph alternative than on the relational one. This Neo4j query directly looks for all pairs of stops, and then computes the distance between each pair. This is performed, on average, in 2.2 s, while just the join in PostgresSQL takes 31 s (as explained above, the geographic part has not been evaluated in the Postgres version). That is, when the SQL query requires joining and/or sorting the complete database, performance tends to benefit the graph alternative. Query 5.4, which combines *patterns and distance computation*, is also a good example of the former. In this case, the pattern just involves checking two kinds of stops (going from home to the airport, in this case). The navigation along the Stop dimension is performed very fast, as well as the distance computation. The query takes 0.1 s to complete, outperforming SQL which took (without distance computation), 0.25 s.

For *spatial queries*, only the Neo4j alternative is evaluated, as explained above. For the case of Neo4j, results are just given for completeness since, clearly, the spatial plugin provided for Neo4j still has to be developed. Spatial queries are included here to give an idea of the potential for enhancing graph queries of any kind (e.g., transitive closure queries, pattern queries), with spatial functions. Spatial queries that require going through all the stops take, naturally, a long time to execute, given the large number of stops in the data set. For example, Query 5.6 takes just 22 s to execute, since it only queries the starting position of each trajectory. On the other hand, Query 5.5, which needs to go through all the stops in all trajectories, takes over 700 s.

Aggregation queries also require going through the whole database. Thus, the same observations above are also valid in this case. Query 5.7 takes 0.8 s to complete on Neo4j, while the SQL alternative takes 1 second. Both queries return 359 records. The reason is that the pattern matching performing the join in Neo4j (to climb up to the Category level) is more efficient than the SQL join. This is somehow compensated by the fact that SQL performs no additional join to climb up to the Month level, while the Cypher query actually performs such climbing. When the size of the result increases, the advantage for Neo4j increases, as can be seen in the result for the other aggregation queries.

At this point, the reader may argue that there would be better design alternatives for the relational database. This is true for some queries (such as Query 5.3), while a different design would be worse for other ones. And the same may hold for the graph model. The relational design chosen is a generic one, with denormalized dimensions, which favors navigation along the hierarchies, preventing joining dimension levels—this is called a "star schema" design in data warehousing jargon (Kimball, 1996).

For *transitive closure queries*, it is worth remarking that Cypher has not yet included many functions that would be needed to compute paths when the transitive closure of the graph is involved. This is why, as explained, extra statements are needed to solve some queries (as in Query 5.10 to filter out unnecessary subpaths). Query 5.10 takes 7.8 s to execute on the graph database, while the SQL alternative (see the Appendix), which requires recursion techniques, takes 1 min. This suggests that, for transitive closure queries, the graph database alternative can be competitive and in some cases even better than the recursive SQL solution. Of course, this is not conclusive, since many other factors impact on the results. For example, Query 5.10 asks for paths between home and airports. If

this changes for more common stops such as restaurants and banks, since there are more redundant paths to eliminate (due to the Cypher limitations commented above), performance decreases to 27 s in Neo4j (this is alternative 2 in Table 2), while the SQL query is not significantly affected. In both cases, however, Neo4j performs better than Postgres, and these results repeat for many different combinations of filters. This is due to the fact that the transitive closure is not computed over the complete graph. There is a situation, however, in which the relational alternative is still better than the graph one. This is the case where transitive closure computation is performed over the whole graph, as in Query 5.11. When this computation is required, parallel graph computation is needed (Neo4j does not scale horizontally). Note that trajectory graphs are excellent candidates for parallelization, since the transitive closure of each trajectory can be computed independently of the others. On the other hand, it can be seen that the SQL query is very efficient for this task, and the query times are stable. These results suggest, then, that when the transitive closure is computed over a relatively small portion of the trajectory graph, the graph database alternative works better than the relational ones, but when it must be taken over the whole graph, the relational database performs better. However, for the same reasons, graph parallelization solutions are great candidates for this problem, particularly for trajectory graphs. A study of these alternatives is beyond the scope of this article.

Table 2 summarizes the test results. The last column on the right gives the ratio between the execution times on Neo4j and PostgresSQL. The best execution times for each query have been highlighted in bold. In conclusion, it can be said that for most kinds of queries the graph database outperforms the relational one, the exception being the transitive closure queries requiring computation of the closure of the *whole* trajectory graph. In this case, the relational alternative is clearly better. However, when the closure is taken over a small portion of the database, the results are competitive, or even favor the graph option.

6 | OPEN PROBLEMS AND FUTURE WORK

This article discussed the problem of querying a collection of semantic trajectories modeled as a property graph and stored in a graph database, focusing on analytical queries, which imply aggregating the trajectories up to different granularity levels. Typically, trajectories are stored in a relational database. Given that trajectories can be seen as a graph, modeling and storing them as graphs instead of relations sounds natural, and merits studying the plausibility of this solution. The outcome of this study suggests, from a qualitative point of view, that analytical trajectory queries are more naturally expressed as graphs, using a graph query language (in this case, Cypher, the query language for Neo4j), than via a relational representation. Moreover, from a quantitative point of view, for three out of the five classes of queries studied (non-recursive patterns, distance, spatial, aggregation, and transitive closure queries), the graph database queries run from 1.2 to 7 times faster than the relational ones. Only transitive closure queries that must go through the complete graph delivered better performance on the relational database. However, when queries require computing the transitive closure of only a relatively small portion of the database, the graph database is competitive, and even outperforms the relational database.

The results reported here, however promising, leave plenty of room for more research work. As a first indication of the road to follow, note that even for the queries where results are not positive (basically transitive closure queries), such results are very likely to change if parallel execution is employed. There are many parallel processing graph databases (e.g., GraphFrames (https://graphframes.github.io/), Janusgraph (http://janusgraph.org/) that may take advantage of the characteristics of trajectory graphs in order to, for example, compute the transitive closure of the trajectory graph in parallel, and aggregate the results after this computation. This may speed up computation by orders of magnitude. Therefore, the work presented will be followed up with an exploration the benefits of using those kinds of graph databases.

ORCID

Alejandro A. Vaisman https://orcid.org/0000-0002-3945-4187

ENDNOTE

[1] Maps where downloaded from http://www.mapcruzin.com

REFERENCES

Alvares, L. O., Bogorny, V., Kuijpers, B., Fernandes de Macedo, J. A., Moelans, B., & Vaisman, A. (2007). A model for enriching trajectories with semantic geographical information. In *Proceedings of the 15th ACM International Symposium on Advances in Geographic Information Systems* (pp. 162–169). New York, NY: ACM.

Angles, R. (2012). A comparison of current graph database models. In *Proceedings of ICDE Workshops* (pp. 171–177). Arlington, VA.

Angles, R. (2018). The property graph database model. *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia*.

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., & Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Computing Surveys*, *50*(5), 68:1–68:40.

Angles, R., & Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, *40*(1), 1:1–1:39.

Chen, C., Yan, X., Zhu, F., Han, J., & Yu, P. S. (2009). Graph OLAP: A multi-dimensional framework for graph data analysis. *Knowledge and Information Systems*, *21*(1), 41–63.

da Silva, M. C. T., Times, V. C., de Macêdo, J. A., & Renso, C. (2015). SWOT: A conceptual data warehouse model for semantic trajectories. In *Proceedings of the 18th ACM International Workshop on Data Warehousing and OLAP* (pp. 11–14). New York, NY: ACM.

du Mouza, C., & Rigaux, P. (2005). Mobility patterns. *GeoInformatica*, *9*(4), 297–319.

Fileto, R., May, C., Renso, C., Pelekis, N., Klein, D., & Theodoridis, Y. (2015). The Baquara2 knowledge-based framework for semantic enrichment and analysis of movement data. *Data & Knowledge Engineering*, *98*, 104–122.

Giannotti, F., Nanni, M., & Pedreschi, D. (2006). Efficient mining of temporally annotated sequences. In *Proceedings of the Sixth SIAM International Conference on Data Mining*. Philadelphia, PA: SIAM.

Giannotti, F., Nanni, M., Pinelli, F., & Pedreschi, D. (2007). Trajectory pattern mining. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 330–339). New York, NY: ACM.

Gómez, L. I., Kuijpers, B., & Vaisman, A. A. (2017). Performing OLAP over graph data: Query language, implementation, and a case study. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics* (pp. 6.1–6.8). New York, NY: ACM.

Gómez, L. I., & Vaisman, A. A. (2013). Mining semantic trajectories. *Intelligent Data Analysis*, *17*(5), 857–898.

Gryllakis, F., Pelekis, N., Doulkeridis, C., Sideridis, S., & Theodoridis, Y. (2017). Searching for spatio-temporal-keyword patterns in semantic trajectories. In N. Adams, A. Tucker, & D. Weston (Eds.), *Advances in intelligent data analysis XVI* (pp. 112–124). Cham, Switzerland: Springer.

Gryllakis, F., Pelekis, N., Doulkeridis, C., Sideridis, S., & Theodoridis, Y. (2018). Spatio-temporal-keyword pattern queries over semantic trajectories with Hermes@Neo4j. *Proceedings of the 21th International Conference on Extending Database Technology, 2018, Vienna, Austria*, 678–681.

Güting, R. H., & Schneider, M. (2005). *Moving objects databases*. San Francisco, CA: Morgan Kaufmann.

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

Karli, S., & Saygin, Y. (2009). Mining periodic patterns in spatio-temporal sequences at different time granularities. *Intelligent Data Analysis*, *13*(2), 301–335.

Kimball, R. (1996). *The data warehouse toolkit*. New York, NY: John Wiley & Sons.

Leonardi, L., Orlando, S., Raffaetà, A., Roncato, A., Silvestri, G., Andrienko, G., & Andrienko, N. (2014). A general framework for trajectory data warehousing and visual OLAP. *GeoInformatica*, *18*(2), 273–312.

Parent, C., Spaccapietra, S., Renso, C. Andrienko, G., Andrienko, N., Bogorny, V., ... Yan, Z. (2013). Semantic trajectories modeling and analysis. *ACM Computing Surveys*, *45*(4), 42:1–42:32.

Pelekis, N., Sideridis, S., & Theodoridis, Y. (2015). Hermessem: A semantic-aware framework for the management and analysis of our LifeSteps. In *Proceedings of the 2015 IEEE International Conference on Data Science and Advanced Analytics*. Piscataway, NJ: IEEE.

Renso, C., Spaccapietra, S., & Zimányi, E. (Eds.). (2013). *Mobility data: Modeling, management, and understanding*. Cambridge, UK: Cambridge University Press.

Robinson, I., Webber, J., & Eifrém, E. (2013). *Graph databases*. Sebastopol, CA: O'Reilly Media.

Ruback, L., Casanova, M. A., Raffaetà, A., Renso, C., & Vidal, V. (2016). Enriching mobility data with linked open data. In *Proceedings of the 20th International Database Engineering & Applications Symposium* (pp. 173–182). New York, NY: ACM.

Spaccapietra, S., Parent, C. Damiani, M. L., de Macedo, J. A., Porto, F., & Vangenot, C. (2008). A conceptual view on trajectories. *Data & Knowledge Engineering*, *65*(1), 126–146.

Spaccapietra, S., Parent, C., & Spinsanti, L. (2013). Trajectories and their representations. In C. Renso, S. Spaccapietra, & E. Zimányi (Eds.), *Mobility data: Modeling, management, and understanding* (pp. 3–22). Cambridge, UK: Cambridge University Press.

Vaisman, A. A., & Zimányi, E. (2013). Trajectory data warehouses. In C. Renso, S. Spaccapietra, & E. Zimányi (Eds.), *Mobility data: Modeling, management, and understanding* (pp. 62–82). Cambridge, UK: Cambridge University Press.

Zhao, P., Li, X., Xin, D., & Han, J. (2011). Graph cube: On warehousing and OLAP multidimensional networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (pp. 853–864). New York, NY: ACM.

How to cite this article: Gómez LI, Kuijpers B, Vaisman AA. Analytical queries on semantic trajectories using graph databases. *Transactions in GIS*. 2019;00:1–24. https://doi.org/10.1111/tgis.12556

APPENDIX CYPHER AND SQL EXPRESSIONS FOR QUERIES IN SECTION 5.1

This Appendix includes the Cypher or SQL expressions not included in the main body of the paper.

Query 5.2 Find the trajectories that go from a bar (or similar) to a restaurant, again to a bar (or similar) and end at a restaurant, without intermediate stops.

The aim of this query is to compare performance for long patterns.

```
MATCH (cat1:Category{categoryType: 'Bar-Coffee-Tea'}) <-[*3..3]-
      (s1:Stops)-[:trajstep]->(s2:Stops)-[:trajstep]->
      (s3:Stops)-[:trajstep]->(s4:Stops)-[*3..3]->
      (cat4:Category{categoryType:'Restaurant'})
WHERE toint(s4.position) = toint(s1.position)+3
MATCH (s2) -[*3..3]->(cat2:Category{categoryType:'Restaurant'})
MATCH (s3) -[*3..3]->(cat3:Category{categoryType:'Bar-Coffee-Tea'})
WITH s1 order by s1.position
RETURN s1.userid, collect(distinct s1.position) order by s1.userid
```

Query 5.6 List the trajectories starting at less than 300 m from a place of interest of the city, returning the trajectory identifier (i.e., userId), and the actual distance, for all the PoIs in the answer.

```
MATCH (s:Stops)
WHERE s.position='1'
CALL spatial.withinDistance('nyorkpois', {latitude: tofloat(s.latitude),
     longitude:tofloat(s.longitude)},0.3) YIELD node as c, distance as a
RETURN s.userid, s.position, a
```

Query 5.8 Compute the number of stops per day per user, along with the starting position of each subtrajectory for each day.

```
MATCH (s:Stops)
WITH s
MATCH (s)-[:isinstantof]->()-[:isminuteof]->()-[:ishourof]->(d1:dateday)
RETURN s.userid as usr, d1.day as day,d1.month as month,
       min(s.position), (toint(max(s.position))-toint(min(s.position)))
       as dif order by usr asc, month asc,day asc
```

Query 5.9 For each trajectory, compute its total length as the sum of the distances between each pair of stops.

```
MATCH (s1:Stops)-[:trajstep]-(s2:Stops)
WITH point({longitude:tofloat(s1.longitude),
     latitude:tofloat(s1.latitude)}) AS p1,
     point({longitude: tofloat(s2.longitude),
     latitude:tofloat(s2.latitude)}) AS p2,
     s1, s2, s1.userid AS user
RETURN user, sum(round(distance(p1, p2)))
     AS totalLength order by user asc, totalLength desc
```

Query 5.10 For each trajectory, find the paths that go from a private home to an airport in the same day.

The Cypher expression is given in the main body of the paper. The SQL expression for this query reads as follows.

```
 WITH RECURSIVE path(tpos_src, venueid_src, venuecategory_src,
     cattype_src,tpos_dst, venueid_dst, venuecategory_dst,
     cattype_dst,userid,date,stopscattype, stopsvenueid)
     as
    (SELECT  i1.tpos, i1.venueid, i1.venuecategory, i1.cattype,
        i2.tpos,i2.venueid, i2.venuecategory,i2.cattype,
        i2.userid, date_trunc('day',i2.utctimestamp),
        Array[i1.cattype] || i2.cattype, Array[i1.venueid]
        || i2.venueid
      FROM trajectories i1, trajectories i2
      WHERE  i1.userid = i2.userid and i1.tpos +1= i2.tpos and
         date_trunc('day', i1.utctimestamp)= date_trunc('day',
        i2.utctimestamp) and i1.cattype <> 'Airport'

 UNION ALL

    SELECT path.tpos_src, path.venueid_src, path.venuecategory_src,
       path.cattype_src, i2.tpos, i2.venueid, i2.venuecategory,
       i2.cattype, i2.userid, date_trunc('day', i2.utctimestamp),
       stopscattype || Array[i2.cattype], stopsvenueid ||
       Array[i2.venueid]
    FROM path, trajectories i2
    WHERE  path.userid = i2.userid and path.tpos_dst+1= i2.tpos  and
      date_trunc('day',path.date)=date_trunc('day', i2.utctimestamp)
      and 'Airport' <> path.cattype_dst
     )

SELECT tpos_src, venueid_src, venuecategory_src, cattype_src,
     tpos_dst, venueid_dst,venuecategory_dst, cattype_dst, userid,
     date, stopscattype,stopsvenueid
FROM   path
WHERE cattype_src= 'Home' and cattype_dst='Airport'
GROUP BY tpos_src, venueid_src, venuecategory_src, cattype_src,
    tpos_dst, venueid_dst, venuecategory_dst, cattype_dst, userid,
    date, stopscattype, stopsvenueid
ORDER BY userid, tpos_src, tpos_dst;
```

Query 5.11 For each day, and for each trajectory, find the longest subtrajectory.

The corresponding Cypher query reads:

```
MATCH (s :Stops)-[*3..3]->(d: dateday)
WITH s.userid AS sid, d,  min(s.position) as initialpos,
     max(s.position) as lastpos  WHERE initialpos <> lastpos
MATCH path=(s1:Stops{userid: sid, position:initialpos})-[:trajstep*]->
     (s2:Stops{userid: sid, position: lastpos})
RETURN s1, d, path
```

In SQL the query reads:

```
WITH RECURSIVE path( srcpos, currentpos, userid, date,
     stopsvenueid)
     AS (SELECT i1.tpos, i1.tpos, i1.userid,
     date_trunc('day', i1.utctimestamp), Array[i1.venueid]
     FROM trajectories i1
UNION ALL
     SELECT srcpos, i2.tpos, i2.userid, date_trunc('day',
     i2.utctimestamp), stopsvenueid ||  Array[i2.venueid]
     FROM path, trajectories i2
     WHERE path.userid = i2.userid and path.currentpos + 1 = i2.tpos
     and date_trunc('day', path.date) =
     date_trunc('day', i2.utctimestamp)),
     maximal(userid, date, maxlength) AS
           (SELECT userid, date, max(array_length(stopsvenueid,1))
     FROM path
     GROUP BY userid, date
     HAVING max(array_length(stopsvenueid, 1) ) > 1)
SELECT userid, date, stopsvenueid
FROM path
WHERE EXISTS (SELECT * FROM maximal
             WHERE path.userid=maximal.userid AND path.date=maximal.date
             and array_length(path.stopsvenueid,1) = maximal.maxlength)
ORDER BY userid, date
```

Query 5.12 Give the districts in which people leave from their home to go to the airport before 3 p.m.

The query makes use of additional layers, in this case, the layer containing the districts. The query in Cypher is expressed as follows.

```
MATCH (cat1:Category{categoryType:'Home'})<-[*3..3]-(s1:Stops)-
   [:trajstep*]->(s2:Stops)-[*3..3]->
   (cat2:Category{categoryType:'Airport'})
WITH min(s1.longitude) as izq, max(s1.longitude) as der,
   min(s1.latitude) as aba, max(s1.latitude) as arri
CALL  spatial.bbox('nycdistricts', {lat:tofloat(aba), lon:tofloat(izq)},
  {lat:  tofloat(arri),  lon:  tofloat(der)})  yield node as c
WITH  c
MATCH (cat1:Category{categoryType:'Home'})<-[*3..3]-(:Categories)-
   (s1:Stops)-[:trajstep*]->(s2:Stops)-[*3..3]->
   (cat2:Category{categoryType:'Airport'})
WHERE apoc.date.fields(s1.instant, 'yyyy-MM-dd HH:mm:ss').years =
  apoc.date.fields(s2.instant, 'yyyy-MM-dd HH:mm:ss').years and
  apoc.date.fields(s1.instant, 'yyyy-MM-dd HH:mm:ss').months=
  apoc.date.fields(s2.instant, 'yyyy-MM-dd HH:mm:ss').months and
  apoc.date.fields(s2.instant, 'yyyy-MM-dd HH:mm:ss').days=
  apoc.date.fields(s1.instant, 'yyyy-MM-dd HH:mm:ss').days
  AND apoc.date.fields(s1.instant,'yyyy-MM-dd HH:mm:ss').hours < 15
  AND tofloat(s1.longitude) > tofloat(c.bbox[0]) and
  tofloat(s1.latitude) > tofloat(c.bbox[1]) and
  tofloat(s1.longitude) < tofloat(c.bbox[2])
  and tofloat(s1.latitude) <  tofloat(c.bbox[3])
RETURN  s1, s2, c.NAME
```

The Neo4j spatial plugin still does not provide a function that can compute if a point is contained inside a polygon. Therefore, this query is solved using the bounding boxes of the districts, as can be seen in the last part of the query. The first part computes the bounding box containing all stops in the data set. Then, the `spatial.bbox` function returns all the district in this bounding box. Then all trajectories leaving from a home to go to an airport before 3pm are computed. Finally, the districts corresponding at these homes are computed (based on the bounding box of the district geometry).