Journal of Computer Languages 51 (2019) 97–111

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.editorialmanager.com/cola/default.aspx

GraphTQL: A visual query system for graph databases

María Constanza Pabón*,[a], Marta Millán[b], Claudia Roncancio[c], César A. Collazos[d]

[a] Facultad de Ingeniería, Pontificia Universidad Javeriana, Cali, Valle del Cauca, Colombia
[b] Ciudad Universitaria Meléndez, Edificio 331, Cali, Valle del Cauca, Colombia
[c] Laboratoire LIG, Bâtiment IMAG, 700 avenue Centrale Domaine Universitaire de Saint-Martin-d'Hères, CS 40700-38058 Grenoble cedex 9, Grenoble, Isere, France
[d] Campus Universitario de Tulcán, Facultad de Ingeniería Electrónica y Telecomunicaciones, Popayán, Cauca, Colombia

ARTICLE INFO

Keywords:
Visual query systems
Graph query languages
User-centered design

ABSTRACT

End-users who are experts in an application domain need to retrieve data from databases in order to answer particular inquiries. Visual Query Systems (VQSs) facilitate this task; but, formulating complex queries is still challenging for end-users. In particular, VQSs for graph databases can be broadly classified into two groups: One group is based on data exploration and allow for direct manipulation of data graphs, but with limited query expressiveness. The other group, oriented to build visual sentences that represent the query, provide greater expressiveness, but they transfer complex language concepts and operations to the visual language. Thus, this research developed GraphTQL a visual query system that provides greater expressiveness and ease in the query formulation. GraphTQL uses a graph data model to depict the domain of interest at the conceptual level. A set of operators provides the means to transform schema and instance graphs, reducing them progressively to obtain the data of interest. The operators include implicit management of incomplete data; as well as clarification dialogues to guide users in order to express complex filter conditions and to select the paths which connect entities in the graph. The development of GraphTQL followed a user-centered design approach which focused on the medical domain. Thus, potential users' contribution and feedback played a significant role in this work. A comparison with a graphical query interface for SPARQL showed better results for GraphTQL regarding the number of queries correctly formulated by users, the number of errors and the time spent for query formulation.

1. Introduction

Providing end-users with appropriate ways to access the system's data information is still a subject for research and development. Visual Query Systems (VQSs) are one way to simplify the query formulation process by enabling end-users to interact with a friendly environment. However, any one of the VQSs faces the tension between the ease of use and the expressive power [8,22]; since, usually, improvements in one causes detriment to the other. In particular, regarding graph databases, this tension has led to two broad kinds of tools that allow end-users to query a database. The first family of tools [14,34,39,40] focuses on the exploration and analysis of data graphs, the visualization of large graphs, data clustering, and the calculation of graph properties (e.g. centrality, density). Most of these tools offer direct manipulation of the graphs elements, facilitating their use for end-users. Usually, they allow for the filtering of data by hiding edges or nodes which attributes do not fulfill given conditions or by finding k-neighbors of previously selected nodes. Thus, these tools have limitations on query expressiveness making it impossible to select objects of a particular class based on the characteristics of other objects which are related to it. However, this feature is required, for example, when it is necessary to retrieve patient data regarding previous medical diagnoses and procedures represented by different nodes in the data graph.

The second family of tools [1,15,26,32] are based on query languages and provide for greater expressiveness. In these tools, the user express the query as a visual sentence formed by a set of elements. Commonly, through these elements complex language concepts and operations are translated to the visual notation; for example: the operations for the management of incomplete data. Besides, most tools are based on query patterns that end-users have to build starting from scratch, using objects and property lists; however, this could result cumbersome for them.

This research developed GraphTQL, a visual query language for graph data focused on end-users. GraphTQL offers greater expressive power than graph exploration tools do; and, at the same time, it includes some features which facilitate the expression of ad-hoc medium

* Corresponding author.
E-mail addresses: mcpabon@javerianacali.edu.co (M.C. Pabón), millan@correounivalle.edu.co (M. Millán), claudia.roncancio@imag.fr (C. Roncancio), ccollazo@unicauca.edu.co (C.A. Collazos).

https://doi.org/10.1016/j.cola.2018.12.006
Received 12 June 2017; Received in revised form 10 August 2018; Accepted 27 December 2018

complexity queries. As defined in this research, medium complexity queries, regarding formulation processes, are those which allow for the retrieval of incomplete data while, at the same time, include filters composed of several conditions combined with logical connectives. By ad-hoc queries, we mean those which are not previously known (or defined). Furthermore, this research focuses on providing query tools to professional users, experts in the application domain. According to Jagadish et al. [25], these users can have query requirements with complex semantics; and they expect precise and complete answers as well as a structured result. Moreover, the focus is on domains that usually have incomplete data. Therefore, since this is a recurring feature found in data repositories, it has an impact on query formulation, particularly on graph pattern based queries which require operations in order to handle that specific data (i.e. optional in SPARQL). The clinical data domain, among others, meets the features mentioned above. Physicians, nurses, and other health professionals often require specific patient data for various purposes, and that data is frequently incomplete. This domain is used as the frame of reference in this work.

The main contributions of this work are the following:

- GraphTQL: a visual query system allowing end-users to formulate a query by transforming the diagrammatic representation of the schema graph.
  The transformations are applied successively, reducing both the schema and instance graphs progressively, until the required data is selected. Direct manipulation is the base of GraphTQL interaction. GraphTQL offers a high level of abstraction while allowing for feedback during the query formulation. Additionally, it includes a reduced amount of query language concepts that the user must learn and apply.
  The development of GraphTQL followed a user-centered design approach utilizing the medical application domain. Exercising this approach, we performed five usability tests at various stages of the project development with the participation of health professionals and students. The test results ratified some of the language design decisions and led to changing others.
- A clarification dialogue based mechanism: guiding users in the definition of complex logical expressions.
  The dialogues are generated based both on an *interest class*, chosen by the user, as well as a set of *bifurcation nodes*, where the user can define a logical connective that combines filter conditions. Clarification dialogues also guide users in selecting paths between nodes when the graph has cycles.
- Visual interaction and logical operator sets: transforming and reducing the scheme and instance graphs for the selection of user data required.
  Visual interaction operators are available to the users of GraphTQL, while the logical operators are used to execute the query. The logical level expression, composed of logical operators, depends not only on the operators applied during the interaction, but also on the information captured by the clarification dialogues. The logical operators are formally defined, and a proof of concept of their implementation based on path traversals over a graph database engine was made.
- A functional prototype of GraphTQL: implementing the logical operators using path traversals over a graph database.
  This prototype was used to perform both a comparative usability test and a comparative execution time experiment. The comparative usability test aimed to compare ease of use and satisfaction between GraphTQL and an SPARQL graphic query tool. It was done with the participation of health professionals and students. The comparative execution time experiment compared the execution time of rewritten GraphTQL queries with the execution time of equivalent SPARQL queries over a triple store. In both cases, based on the usability and time execution comparisons, GraphTQL rewritten queries using the logical operators implemented using path traversals, obtained better results than the other tool.

The remainder of this paper is organized as follows: Section 2 discusses existing work related to the approach used; Section 3 describes the visual query language developed, GraphTQL: the interaction mechanisms, an informal description of the graph transformation operators, and an example of their use; in Section 4 is the formal definition of GraphTQL's transformation operators, logical level operators, and clarification dialogues; Section 5 describes the user-centered design techniques applied during the development of GraphTQL; Section 6 presents implementation aspects; results of the usability and execution evaluations are in Section 7; finally, Section 8 presents the conclusions and future work.

2. Related work

Catarci [8] defines Visual Query Systems (VQSs) as "*systems for querying databases that use a visual representation to depict the domain of interest and express related requests*." Usually, the visual representations are icons, diagrams or a combination. In this study, VQSs for graph databases have been classified, according to how they use the visual notation, into two broad types: 1) The ones that provide means to explore or analyze the graph by browsing or by the application of operations and filters on nodes and edges [7,14,34,39,40]. They support the execution of queries by applying several steps. 2) The ones that use visual expressions to formulate queries. They include visual query languages [13,15,17,24,26,33] and visual query editors [1,4,6,12,20–22,32]. Although both of them propose a visual notation for language elements that combined form a query expression, in visual query editors those elements belong to a previously existing textual language.

Graph exploration and analysis tools focus on visualization and navigation of large data graphs, and usually offer graph property calculations (e.g. centrality), k-neighbors, and clustering [7]. They provide innovative, aesthetic and useful ways to lay out the graph. They use colors, sizes, and shapes to highlight different features in the data (e.g. [14]); as well as employing diverse techniques to facilitate graph manipulation (e.g. zoom and lenses [38]). They usually offer direct manipulation of the data graph and give feedback to users by applying one of the following ways for exploration:

- Beginning with the selection of one or a few nodes which are then displayed in a diagram or a table, users can add connected nodes and edges. In these cases, by direct manipulation, users mark the nodes to explore and add, usually utilizing menus or neighbor nodes.
- Beginning with all nodes and edges of the database deployed in a diagram, users set filters on nodes and edges to hide some of the data. In these cases, two forms of interaction were identified: (a) Direct manipulation of the graph, marking the filters on the nodes (e.g. [7,34,39]); and (b) Use of lists or tables, presented in additional windows, where users select attributes and operations to apply (e.g. [40]).

In both cases, users can employ filters to hide nodes and edges which have certain characteristics. We call this kind of filter, the *local filter* which takes into account the node features, but not their relationship with other nodes. Some tools offer a way to define more complex conditions through the use of DOI (Degree Of Interest) functions [2,41]. Abello et al. [2] support DOI functions that could take into account not only the attributes of nodes and edges, but its connectivity on the graph (ex. the degree of the node), and use the DOI results to assign visual features to each node or edge (ex. position, shape, size, color).

Finding the desired information can be difficult in this kind of tool, when working with large and incomplete data graphs. Incomplete data

complicates the search since objects of the same entity do not exhibit the same set of attributes. In order to handle this issue, Shamir and Stolpnik [34] and Cayli et al. [14] presented summary graphs that grouped nodes having the same attributes and relationships. Furthermore, these summary graphs could be highly dispersed since objects of the same entity could belong to different groups due to incomplete data. However, objects of different entities could be in the same group if attributes and relationships that make the difference are not stored.

Query language focused tools offer greater expressiveness than graph exploration and analysis tools. The former usually provide a visual notation for the elements of the language. Thus, users need to understand the, usually complex, semantics of those elements and to combine them to build the query, generally from scratch, which is also a complex task. Visual query editors map the elements of a textual language to a visual notation, thus the complexity of the analysis required to formulate the queries is the same as in the underlying textual language that, in most cases, operate at the logical level.

Commonly, users formulate queries by building graph patterns, as in [1,6,11,15,17,24,36], including filter conditions with logical and other types of operations such as those for managing incomplete data (e.g. optional in SPARQL-based languages), or set operations (e.g. union) [1,15,26,36]. Contrary to local filters, it is possible to select a node or edge based on the features of another node or edge, with which the first one has a relationship. However, these systems usually do not provide direct manipulation of the graph. Patterns must be built from scratch based on object and property lists, and users should explore the data graph in order to identify data organization. This exploration could be particularly cumbersome if data is incomplete. Moreover, as Jagadish et al. [25] points out, users do not have the time and knowledge of technology needed in order to undertake such exploration. Additionally, small differences in the query formulation could generate significant variations on the result, for example: the specification of the subgraphs for some clauses (e.g. SPARQL's *optional* and *union*); the manner in which these subgraphs relate with query graph patterns; and the manner in which filter conditions are defined within the subgraphs. On the other hand, some tools guide the user in the process of selecting the clases and attributes to include in the query, by presenting them in trees [12] or menues [1].

It was noted that in both kinds of the tools mentioned above, end users do not generally participate in the language design and development process. Only half of the related work which was reviewed employed some usability techniques; however, most of these applied usability tests at the end of the development cycle (e.g. [14,34]). Only Catarci et al. [13] report the application of a UCD methodology involving users beginning with the first development stages. Using UDC techniques on this kind of project can lead to better strategies regarding ease of use.

Furthermore, in spite of the fact that the use of a logical data model as the base of interaction with end users can reduce the ease of use and the user's satisfaction with a query tool, this research has found that only systems which permit query ontologies use a conceptual representation of the application domain.

As a result of the literature revision, a set of features to include in GraphTQL was defined. Table 1 list those features and shows, based on these characteristics, the differences between GraphTQL and a significant selection of the revised VQSs. The first row in the table indicates the kind of VQS (EA: exploration and analysis tool, VL: visual language, VE: visual query editor). GraphTQL is classified as a visual query editor, however is worth noting that the undelying language was proposed after the visual interaction was defined, with the aim to facilite the query formulation.

3. GraphTQL

A GraphTQL query consists of a composition of operations. Each operation takes both a schema and instance graph along with other input parameters; producing transformed schema and instance graphs. End-users mark the operation parameters on the schema graph and, when necessary, refine them through clarification dialogues. This section presents the GraphTQL interaction mechanism, an informal definition of GraphTQL's transformation operators, and an example of their use in the formulation of a query. The operators' formal definition is in the next section.

3.1. GraphTQL interaction mechanisms

The basis for GraphTQL user interaction is direct manipulation [35] of the schema graph. Through the application of successive operations, end-users transform schema and instance graphs; thereby, reducing them until they contain the data needed by the user.

Fig. 1 shows the graphical interface of GraphTQL. On the left, there is a conceptual view of the domain, represented by a GDM schema graph. The right side offers four frames. The topmost one shows a reference view of the schema graph in order to facilitate the scrolling. The operator frame contains a button for each graph transformation operation. The buttons included in the control frame have the following functions: the blue question mark is used to ask for help; the blue arrow is used to clean the marks on the graph or to undo the last operation; the red arrow undoes all the operations applied; the + and - magnifiers zoom the graph; and the buttons used to execute the query and clean all the marks. Finally, at the bottom, the history frame shows the sequence of operations applied on the current query.

In order to support query formulation, GraphTQL displays a simplified diagrammatic representation of the schema graph that depicts a conceptual data model of a domain. It is simplified since it removes information which is useful for database design but irrelevant for query formulation. For example, the diagram does not include cardinality of relationships or edge directions, and does not distinguish multivalued or mandatory attributes.

On the schema graph, each node type is represented by a different shape: big rectangles for object nodes, ovals for basic value nodes, small rectangles for labeled composite nodes, and small circles for unlabeled composite nodes (See in Section 4.1 the type of nodes definition). Moreover, the tool supports the addition of icon images to object nodes.

Using left and right clicks, end-users mark nodes and edges as well as the conditions that selected data must fulfill. As end-users add marks on the nodes, the tool marks the paths among them, easing the marking task. When there is more than one path between the last marked node and the previously marked ones, a *path clarification dialogue* appears; the dialogue presents all possible paths and allows users to select which ones to mark. Marked nodes and edges change their color, according to a visual notation. They can be unmarked by clicking them again.

Once the input for an operation is marked on the graph, the user could apply the operation. If there are filter conditions specified over several nodes, a *filter clarification dialogue* appears to let the user define how to compose these conditions. In this way, GraphTQL increases expressive power without including additional concepts and visual notation.

Each time the user applies an operation, the resulting schema graph is displayed; giving feedback to the user.

3.2. GraphTQL's graph transformation operators

Currently, GraphTQL offers four transformation operations: *Select a portion, Filter, Filter + additional data*, and *Put two objects closer*. Each operation applies a transformation to both schema and instance graphs. An outstanding feature of these operators is the inclusion of implicit incomplete data management. Below is an informal description of these operations.

3.2.1. Select a Portion

By marking a subset of the schema graph, composed of connected

Table 1
Comparison of GraphTQL with other VQSs.

	[34]	[40]	CGV [39]	Gephi [7]	Glyphlink [14]	MashQL [26]	[33]	OntoVQL [17]	Graphite [15]	[22]	Qgraph [11]	Gruff [1]	SPARQL Filter Flow [21]	GraphTQL
Kind of VQS	EA	EA	EA	EA	EA	VE	VE	VE	VL	VE	VL	VE	VE	VE
Use conceptual model					✓		✓	✓						✓
Filter expressions with logical operators			✓	✓		✓		✓				✓	✓	✓
Not only local filters						✓	✓	✓	✓	✓	✓	✓		✓
Implicit management of incomplete data														✓
Avoids learning query language concepts (optional, variables, output variables)	✓	✓	✓	✓	✓				✓	✓			✓	✓
Guides filter condition specification														✓
Guides path selection						✓								✓
UCD design was applied														✓
Reports test with users or user feedback	✓		✓		✓	✓	✓						✓	✓
Avoids schema exploration						✓								✓
Direct manipulation of schema or data	✓	✓	✓	✓	✓									✓
Uses a graph diagram as base for user interaction	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓		✓
Avoids building patterns from scratch or from lists	✓	✓	✓	✓	✓									✓
Formally defined operators	✓	✓				✓		✓						✓
Use simple graphs	✓	✓				✓	✓	✓	✓	✓	✓	✓	✓	✓

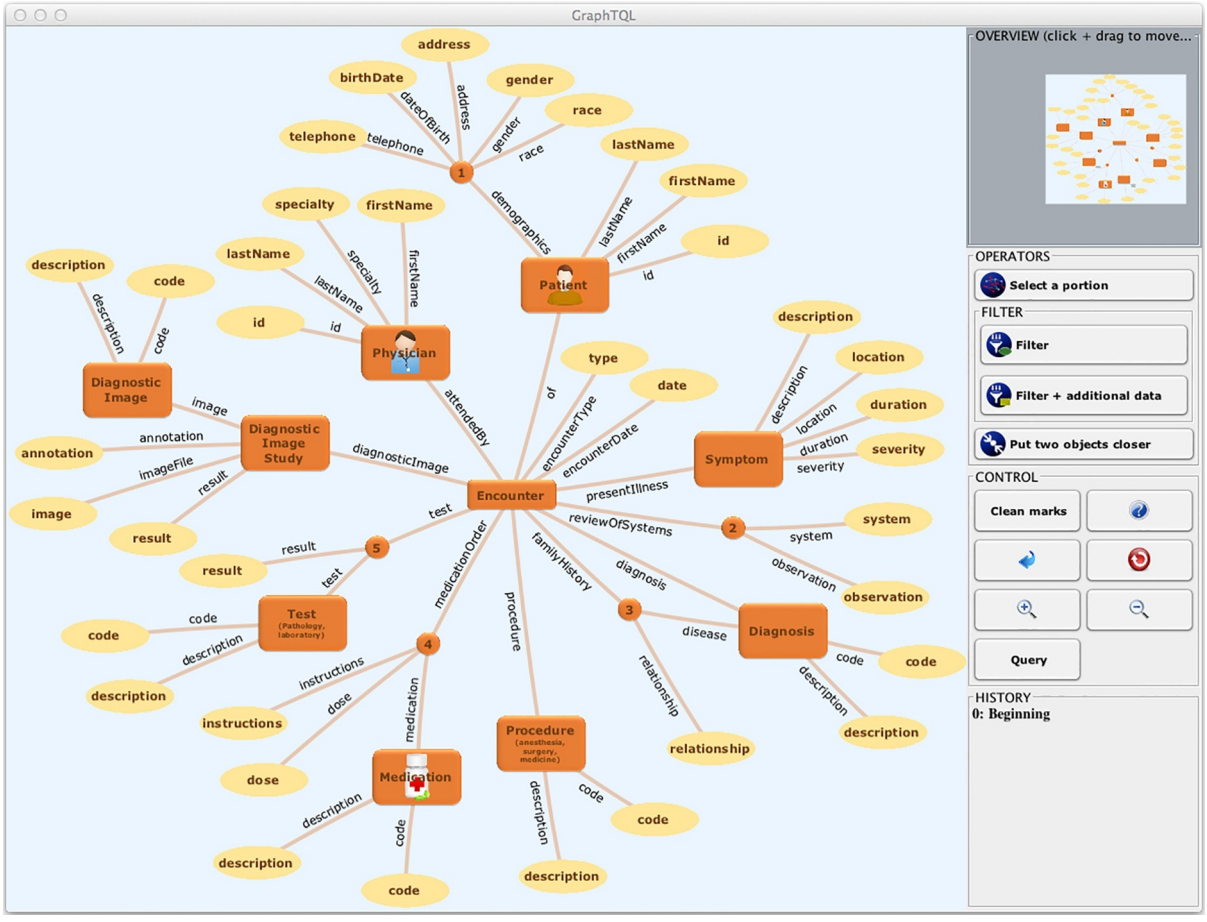


Fig. 1. GraphTQL interface.

nodes and edges, this operation retrieves all nodes and edges that belong to the classes of the nodes and edges marked on the schema.

For example, if, in the schema of Fig. 1, a user marks the patient's *ID, name, lastName* and the *Patient* node, and then applies *Select a Portion*, all patients will be retrieved along with their IDs, names, and last names wherever they are registered. If a patient record has incomplete data, that patient will be still retrieved along with its available data. Thus, *Select a Portion* exhibits a different behavior from that of *pattern matching queries*, which look for strict coincidence with the marked pattern.

3.2.2. Filter and Filter + Additional Data

These operators allow for the selection of objects belonging to a particular class, marked as the *interest class*, along with other data

connected to them. The condition to select these objects is specified over some nodes which are connected to the interest class, these are called *conditioned nodes*. The *filter pattern* is the subgraph formed by the interest class, the conditioned nodes, and the paths that connect them. Additionally, other nodes marked (different to the interest class and the conditioned nodes) are part of the *optional component*.

The objects of the interest class are selected when they fullfill the operation conditions; that is to say, they are connected by a path marked in the schema to objects of conditioned classes that fulfill their own conditions. Once the objects of the interest class are selected, it is added the data of the optional component. *Filter* and *Filter + Additional Data* differ on how they form the optional component, as explained with the following examples.

Let's say we have marked *Patient* as the interest class, *description* of *Diagnosis* with the filter *description = "DIABETES"*, *description* of *Procedure*, and *lastName* of *Patient*. A *Filter* with this input will take as *optional component* two paths: the one that connects *Patient* with *description* of *Procedure* and the one that connects *Patient* with the *lastName*. Thus, once selected the *Patients* connected to a *description* of *Diagnosis* with value "DIABETES", the *description* of all the procedures connected to those *Patients* will be added to the query result, as well as the *lastNames*. On the other hand, a *Filter + Additional Data* with the same input will also add to the result the *description* of all the diagnoses connected to those *Patients*. Then, the *Filter* retrieves the patients who have been diagnosed with diabetes, along with their last name and all the procedures that have been performed on them. The *Filter + Additional Data* retrieves the patients who have been diagnosed with diabetes, along with their last name, all the procedures that have been performed on them, and all other diagnosis given to them. In this case, the conditioned nodes serve to characterize the patients to select and also to retrieve data associated to those patients.

To simplify users' interaction, both filter operation only require marking the interest class, the nodes with a filter condition, and other nodes relevant to the query. However, two problems arise from this approach. First, filter expressiveness is reduced since users cannot express a condition by combining logical operations (i.e. conjunction and disjunction). Second, if more than one node have a filter condition, the graphical representation of the query could be ambiguous given the fact that many logical expressions could represent it; each one with a different combination of conjunctions and disjunctions. Furthermore, the specification of filter expressions can be confusing for end users. Thus, this work proposes *Filter Clarification Dialogues* as a mechanism to disambiguate the meaning of filter conditions marked on several nodes.

The filter clarification dialogue consist of a serie of question made to the user, each one with a set of posible answers, so the user can select one of them. The basis for the generation of the questions is the *filter pattern*. Currently, the generation supports two kinds of filter patterns: (a) one without cycles, and (b) one with only one conditioned node and cycles formed by several paths between that node and another marked node. In the first kind, without cycles, the filter pattern is a tree with the interest class as root, as the one in Fig. 2(a) The dialogue generator performs a top-down traversal of that tree in order to identify *bifurcation nodes*, nodes that are the root of a subtree that includes more than one conditioned node. A bifurcation node provides the opportunity to refine the logical expression by combining, with conjunctions or disjunctions, the conditions on the subtree. Furthermore, when the filter pattern has more than one condition, the interest class is always considered a bifurcation node. To manage the second kind of filter pattern in the same way, a tree is derived, by separating the paths between the bifurcation node and the conditioned nodes. Figs. 2(b) and (c) show a filter pattern with a cycle and its derived tree. Since there is only one conditioned node but several paths, the possibble answers take into account the paths. The form of the questions and answers is described in Section 4 and an example is given in Section 3.3.

3.2.3. Put two objects closer

This operator replaces the inner nodes and edges of a given schema path with a relationship connecting the path's initial and terminal nodes; bringing them closer together on the diagram. Corresponding paths on the instance graph are replaced too. When users apply this operator, they have no interest in intermediate nodes of the path that connects the end nodes. Therefore, they disappear from the resulting schema and instance graphs, being replaced by a new path, shortest than the original. In this way, relationships that were implicit in the original graphs become more evident.

For example, *Put Two Objects Closer* is useful when establishing a direct relationship between the patient and the diagnosis he/she received.

3.3. Example of query formulation

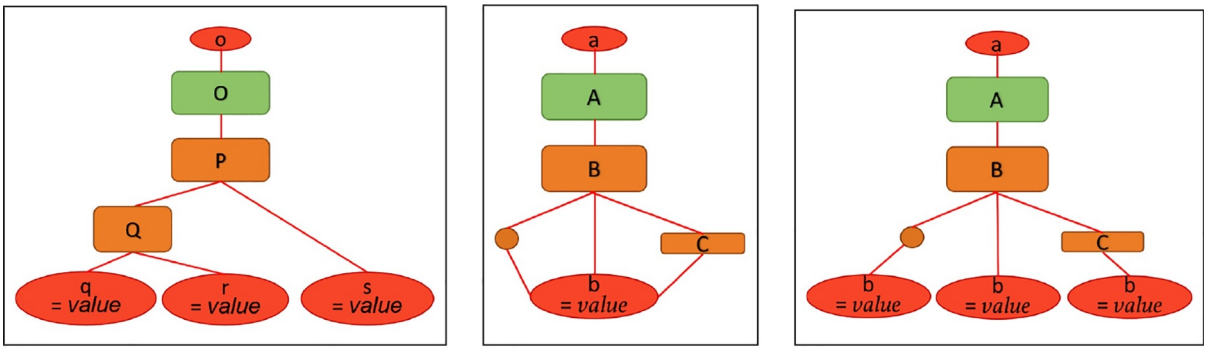
Fig. 1 shows the schema graph used as the reference in this work. It represents a conceptual view of clinical data that includes a representative subset of the RIM (HL7 Reference Information Model)[1] and the OpenEHR information models [9]. This view includes data such as: patient's ID, date of birth, gender and address; physician's ID, first name and specialty; encounters between patients and physicians, type of encounters and dates. This view also includes the data registered during those encounters. Among others, these data include: symptoms (description, location...); diagnosis (code, description); and medication orders (medications, dose, instructions...). Fig. 3 shows an instance graph that corresponds to the schema of Fig. 1 (a number, which is not part of the model, has been aggregated to the object labels in order to facilitate its differentiation). *Patient 1* appears on the upper left-hand portion, his identification number is *001* and the patient's name is *John*. John had an ambulatory encounter (*Encounter 4*) in which a test, called *Colonoscopy*, was administered, and a diagnosis (*Diagnosis 1*) with code *C18.7* was registered.

For example, to find the patient's *ID* and *diagnosis*, for those patients whose history registers both a *PLEURAL EFFUSION* diagnosis and an *APPENDECTOMY* test, the user marks the *Id*, the interest class, *Patient* (as shown in Fig. 4(a)), and puts the condition over the *description* of *Diagnosis* (*description = "PLEURAL EFFUSION"*). After that, the corresponding *path clarification dialogue* appears (Fig. 4(b)), it contains an option for each possible path connecting the interest class and the conditioned node. To evince all possible paths, these are marked on the graph while the dialogue is displayed. Once the user accepts, selected paths remain marked.

Continuing with the query formulation, Fig. 4(c) exhibits the marks and filter conditions for a *Filter + additional data* operator. This operator will retrieve the patient's nodes that fulfill the condition as well as other associated data when available; this data correspond to the nodes and edges marked on the schema graph (IDs, procedures and diagnoses, each with their given description). Figs. 4(c) and 4(d) show the clarification dialogues required to refine the filter condition; they allow to choose among the conjunction or the disjunction of the conditions. This decision is made for each bifurcation node, in the example, *encounter* and *Patient*. Particularly, this example requires *Patients* that fulfill both filter conditions (referring to diagnosis and test), but the fact is that different *Encounters* may have them registered. Thus, at the *Patient* node a conjunction is needed; but at the *Encounter* node, a disjunction is needed. Clarification dialogues guide users to refine the filter expression, combining conjunction and disjunction, and providing more expressive power, without adding concepts or visual notation. Fig. 4(e) presents the schema graph resulting from applying *Filter + additional data*.

Fig. 4(f) show the marks and parameters for the *Put Two Objects Closer* operator. This operator replaces the marked path (*ID-Patient-*

[1] http://www.hl7.org.



(a) Filter pattern without cycles (b) Filter pattern with cycles (c) Tree derived from the filter pattern with cycles

Fig. 2. Two kind of *filter pattern*.

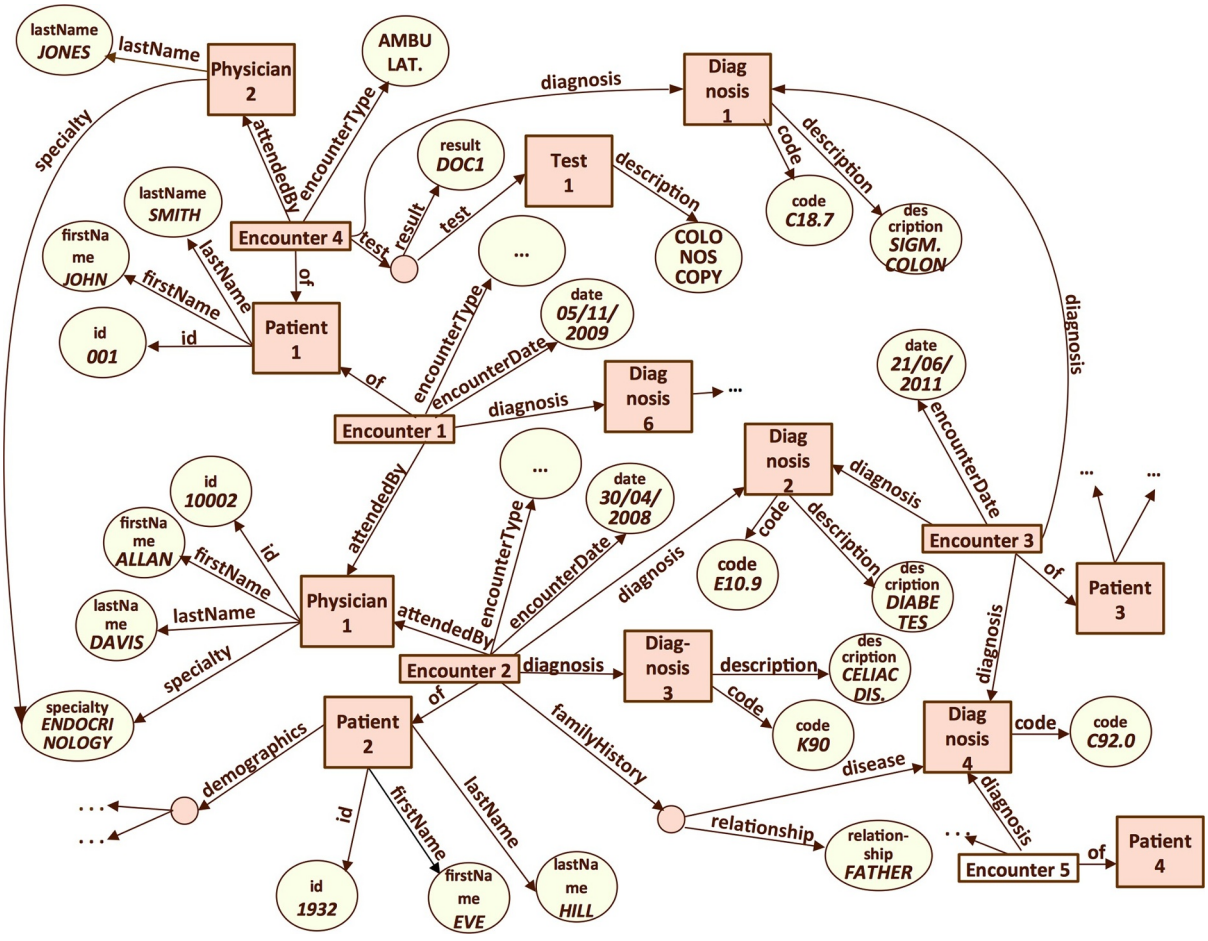


Fig. 3. Instance graph example.

Encounter-Diagnosis-description) with a new relationship (creating the path *ID-diagnosed-description*). Finally, 4(g) shows the schema graph of the query result.

Each time an operation is applied, it appears in the *History* framework. Once the query is defined, it can be executed by clicking the *Query* button (Fig. 1).

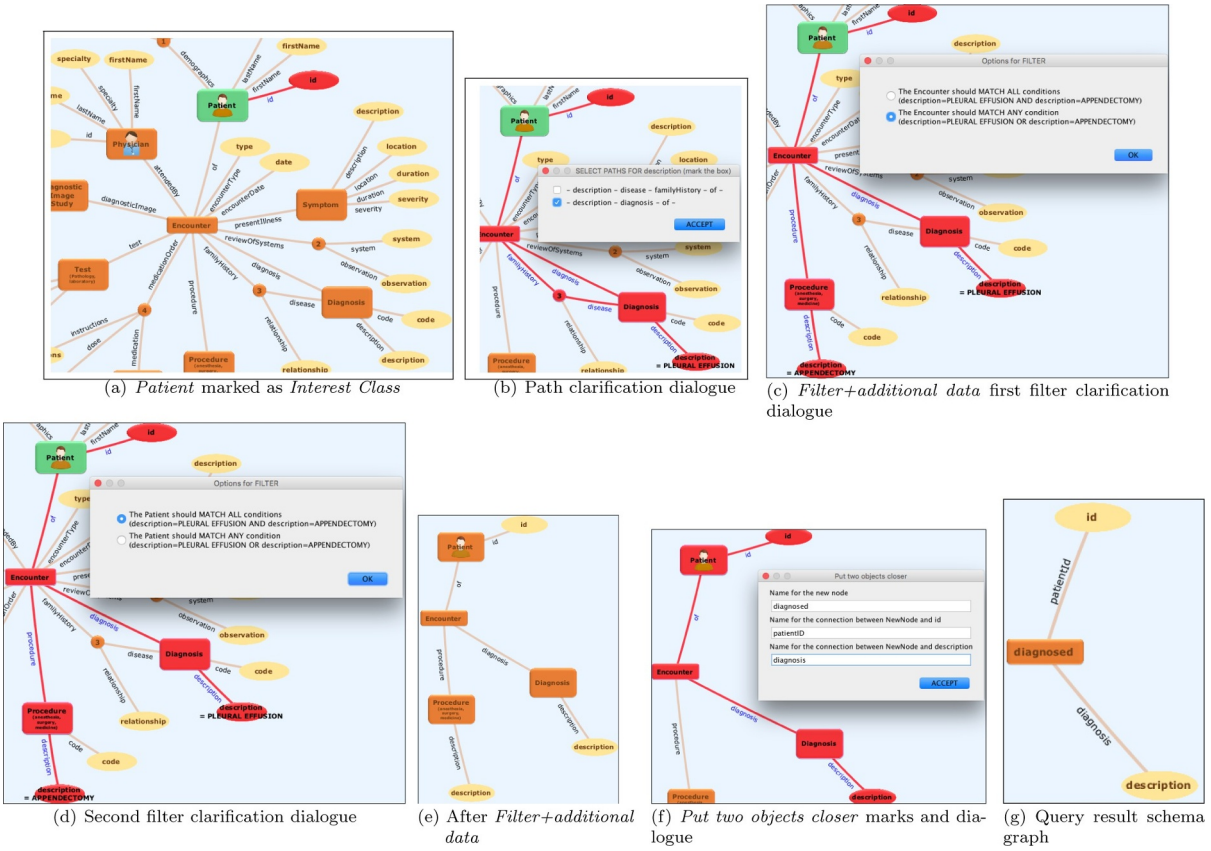
4. Transformation and logic level operators definition

This section includes the formal definition of GDM data model, GraphTQL operators and filter clarification dialogues. Each transformation operator is defined in terms of the logical level operators proposed.

4.1. The graph data model

GDM [23], the data model underlying GraphTQL, differentiates between the schema and instance graphs. Both of them are simple graphs, composed of simple nodes and edges. Nodes and edges are simple since they have only two features: a label and a class to which each belongs. This simplicity facilitates the depicting of the domain of interest by using a conceptual data model as represented by the schema graph. It also facilitates the mapping of the GDM with other data models.

We use the formal definition of GDM graphs provided by Hidders [23]. To do this, the following sets are introduced. Let C be the set of



(a) *Patient* marked as *Interest Class* (b) Path clarification dialogue (c) *Filter+additional data* first filter clarification dialogue

(d) Second filter clarification dialogue (e) After *Filter+additional data* (f) *Put two objects closer* marks and dialogue (g) Query result schema graph

Fig. 4. Example of query formulation.

class names; let $\mathcal{A}$ be the set of attribute names (not containing *isa* or *is*); let $\mathcal{B}$ be the set of basic-type names (not containing *com* or *obj*); let $\mathcal{D}$ be the set of representations of basic values; and let δ: $\mathcal{B} \rightarrow \mathcal{P}(\mathcal{D})$ be a function that assigns a domain to every basic-type name, where $\mathcal{P}(\mathcal{D})$ denotes the power set of $\mathcal{D}$. Finally, $\mathcal{P}_{fin}(C)$ denotes the set of finite subsets of C.

The schema graph consists of nodes and edges representing classes and attributes respectively. A schema graph is a tuple $S = \langle N_S, E_S, \lambda_S, \sigma_S \rangle$, where $\langle N_S, E_S, \lambda_S \rangle$ is a finite, partially labeled graph; with N_S being the set of nodes, E_S the set of edges ($E_S \in N_S \times \mathcal{A} \times N_S$), and the labeling function λ_S: $N_S \cup E_S \rightarrow (C \cup \mathcal{A})$; $\lambda_S(n) \in C$ for $n \in N_S$ and $\lambda_S(\langle n_1, \alpha, n_2 \rangle) \in \mathcal{A}$ for $\langle n_1, \alpha, n_2 \rangle \in E_S$. GDM offers three *sort* of nodes: *object class nodes* (*obj*), *composite-value class nodes* (*com*), and *basic-value class nodes*. The function σ: $N_S \rightarrow \{obj, com\} \cup \mathcal{B}$ assigns a sort to each node. Labeled composite-value nodes represent *n*-ary relations between classes; while the unlabeled ones group a set of nodes [23]. For the sake of simplicity, labels are used to refer to nodes and edges, and β_i (for some integer *i*) to refer to unlabeled composite-value nodes.

In the instance graph, the nodes and edges represent entities and attributes, respectively. An instance graph is a tuple $I = \langle N_I, E_I, \lambda_I, \sigma_I, \rho_I \rangle$, where $\langle N_I, E_I, \lambda_I \rangle$ is a finite labeled graph with N_I being the set of nodes, E_I the set of edges ($E_I \in N_I \times \mathcal{A} \times N_I$) and the labeling function λ_I: $N_I \cup E_I \rightarrow (\mathcal{P}_{fin}(C) \cup \mathcal{A})$; $\lambda_I(n) \in \mathcal{P}_{fin}(C)$ for $n \in N_I$ and $\lambda_I(\langle n_1, \alpha, n_2 \rangle) \in \mathcal{A}$ for $\langle n_1, \alpha, n_2 \rangle \in E_I$. Therefore, a node is labeled with zero or more class names, whereas an edge is labeled with an attribute name. The function σ_I: $N_I \rightarrow \{com, obj\} \cup \mathcal{B}$ assigns a sort to each node. Therefore, according to its sort, a node can be an *object node*, a *composite-value node*, or a *basic-value node*. The function ρ_I: $\{n \in N_I | \sigma_I(n) \in \mathcal{B}\} \rightarrow \mathcal{D}$ assigns a basic-value to each basic-value node.

An *extension relation* $\xi = \{(m, n) | m \in N_S \text{ and } n \in N_I\}$ indicates the membership relationship among entities from an instance graph *I* and classes from a schema graph *S*. $\xi(m, n)$ denotes $\langle m, n \rangle \in \xi$. An instance graph *I belongs to* a schema graph *S* [23] when the minimal extension relation ξ from *S* to *I* is class name correct, sort correct and covers I.

4.2. GraphTQL's transformation operators definition

4.2.1. Select a Portion

By marking a subset *S′* of the schema graph, composed of connected nodes and edges, this operation allows for the selection of the subgraph of the instance graph covered by *S′*. *Select a Portion* takes as input a schema *S*, an instance of *S, I*, and a set of edges marked on the schema $\{e_{S1}, e_{S2}, ...e_{Sk}\}$ which form a connected subgraph. It is defined in terms of the *Subgraph Extraction* logical level operator (See Section 4.3) as follows:

$$Extract(\langle S, I \rangle, \{e_{S1}, e_{S2}, ..., e_{Sk}\}) \rightarrow \langle S', I' \rangle$$

4.2.2. Filter and Filter + Additional Data

Filter and *Filter + Additional Data* take as input a schema *S*, an instance of *S, I*, and schema's marked nodes and edges, which must include: the interest class m_I; a set $Cc = \{c_1, c_2, ..., c_k\}$ of nodes with a condition; a set $Co = \{o_1, o_2, ..., o_l\}$ disjoint to $Cc \cup \{m_I\}$; and a set $Em = \{e_{S1}, e_{S2}, ..., e_{St}\}$ of marked edges that connect the marked nodes. All of these must form a connected subgraph. These input data form two groups: the *filter pattern* and the *optional component*. The *filter pattern* as the subgraph which is formed by the edges in *Em* that connect m_I with any node in *Cc*, is used as a pattern to find the objects belonging to the interest class which fulfill the condition in the same sense of subgraph matching. The *optional component* has a different definition for each filter operator: the *Filter*'s optional component is the set of marked paths that connect the interest class with a node in *Co*; the *Filter + Additional*

The $\langle\lambda_S(b_i)\rangle$ should MATCH ALL conditions
($\langle\lambda_S(c_1)\rangle\langle f_1\rangle$ AND...AND $\langle\lambda_S(c_j)\rangle\langle f_j\rangle$ AND is connected to a $\langle\lambda_S(b_{c1})\rangle$ that fulfills its conditions AND...AND is connected to a $\langle\lambda_S(b_{cq})\rangle$ that fulfills its conditions AND $\langle$*the conjunction of the conditions of* $b_{d1}\rangle$ AND...AND $\langle$*the conjunction of the conditions of* $b_{dk}\rangle$

The $\langle\lambda_S(b)\rangle$ should MATCH ANY conditions
($\langle\lambda_S(c_1)\rangle\langle f_1\rangle$ OR...OR $\langle\lambda_S(c_j)\rangle\langle f_j\rangle$ OR is connected to a $\langle\lambda_S(b_{c1})\rangle$ that fulfills its conditions OR...OR is connected to a $\langle\lambda_S(b_{cq})\rangle$ that fulfills its conditions OR $\langle$*the disjunction of the conditions of* $b_{d1}\rangle$ OR...OR $\langle$*the disjunction of the conditions of* $b_{dk}\rangle$

Fig. 5. Clarification dialogue for an intermediate bifurcation node.

Data's optional component is the set of marked paths that connect the interest class with a node in $Co \cup Cc$. Data corresponding to the optional component are retrieved if available for the selected interest class objects.

Filter and *Filter + Additional Data* are specified in terms of the *Logical level filter* operator as well as the *Selective union* logical level operator (defined in Section 4.3). The *Logical level filter* operator proceses the *filter pattern* by selecting the objects belonging m_I that fulfill the condition, and the *Selective union* operator adds the data corresponding to the optional component. The logical level expression corresponding to *Filter* and *Filter + Additional Data* varies depending on the results of the filter clarification dialogues.

The ***filter clarification dialogue*** asks several questions to the user, at most one for each bifurcation node. This questions are generated by a bottom-up traversal of the tree formed by the marked paths connecting the interest with the conditioned nodes. For each bifurcation node b, the question gives two options, the first one for the conjunction of the conditions on the subtree rooted by b, and the second one for its disjunction. The questions are formed as defined below.

Let b be a bifurcation node connected directly (i.e. by paths not including other bifurcation nodes) with: the bifurcation nodes $\{b_{c1}, ..., b_{cq}, b_{d1}, ..., b_{dk}\}$; and the conditioned nodes $\{c_1, ..., c_j\}$. During clarification dialogue of each bifurcation node, the user selected the conjunction for the $\{b_{c1}, ..., b_{cq}\}$ nodes and, the disjunction for the $\{b_{d1}, ..., b_{dk}\}$ nodes. Fig. 5 shows the form of the options of the dialogue for b. The first option's expression, corresponding to the conjunction of the conditions, is formed as follows: Conditions $\{c_1, ..., c_j\}$ appear directly (i.e. $\langle\lambda_S(c_1)\rangle\langle f_1\rangle AND ... AND \langle\lambda_S(c_j)\rangle\langle f_j\rangle$, being $\lambda_S(c_i)$ the label of the node c_i, for $1 \le i \le j$; and f_i the condition marked for the node c_i, with form $op_h\ o_h$ where op_h is an operator and o_h an operand). The expression of the $b_c \in \{b_{c1}, ..., b_{cq}\}$ nodes have the following form: "*is connected to a* $\langle\lambda_S(b_c)\rangle$ *that fulfills its conditions*". This form is necessary since b_c must fulfill all its conditions; thus the expression of the b's can not separate them. Finally, the expressions of the $b_d \in \{b_{d1}, ..., b_{dk}\}$ nodes include the conjunction of the b_d conditions; since b_d holds a disjunction, the restriction of b will be the one imposed on b_d's conditions. The second option of the dialogue is produced in the same way, using the OR instead of the AND operator.

When the filter pattern includes cycles and a node c holding the condition, the dialogue expression for the first bifurcation node b (the one nearest to c) changes its form for each path p_i connecting c and b by using the following formula: "*is connected with* $\langle\lambda_S(c)\rangle$ *by* $\langle p_i\rangle$".

Once the clarification dialogue is completed, a *Logical Level Filter* expression is generated for each bifurcation node with a conjunction applied on it. This expression has the following form:

$$LogicLFilter(\langle S, I\rangle, b, expression) \rightarrow \langle S^b, I^b\rangle$$

being b the bifurcation node, and *expression*, a logical expression formed as follows.

Let b be a bifurcation node with the conjunction selected, connected directly with the bifurcation nodes, $\{b_{c1}, ..., b_{cq}, b_{d1}, ..., b_{dk}\}$ and the conditioned nodes $\{c_1, ..., c_j\}$, the expression has the form:

$$(\langle c_{1\{p_1\}}\rangle\langle f_1\rangle\ AND\ ...\ AND\ \langle c_{j\{p_j\}}\rangle\langle f_j\rangle AND$$
$$\langle b_{c1\{p_{bc1}\}}\rangle INSUBGRAPH\langle S^{b_{c1}}, I^{b_{c1}}\rangle\ AND\ ...$$
$$AND\ \langle b_{cq\{p_{bcq}\}}\rangle INSUBGRAPH\langle S^{b_{cq}}, I^{b_{cq}}\rangle$$
$$\langle the\ conjunction\ of\ the\ conditions\ of\ b_{d1}\rangle\ AND\ ...$$
$$AND\langle the\ conjunction\ of\ the\ conditions\ of\ b_{dk}\rangle)$$

where p_h $(1 \le h \le j)$ is the path connecting b and c_h, f_h is the condition hold by c_h; p_{bci} $(1 \le i \le q)$ is the path connecting b and b_{ci}, $\langle S^{b_i}, I^{b_i}\rangle$ is the graph resulting from the filter applied on b_{ci}. The conjunction of conditions of nodes in $\{b_{d1}, ..., b_{dk}\}$ expand the paths that reach each conditioned node by concatenating p_{dt} at the beginning, being p_{dt} the path connecting b to b_{dt} for $1 \le t \le k$. Thus, for the bifurcation nodes with a disjunction, the expression is formed by connecting the conditions directly with b, since those bifurcation nodes do not add any restriction to the filter.

The interest class is a particular case for which a *Logical Level Filter* expression is generated even if it has a disjunction selected. When the interest class has a disjunction selected, the filter expression for the interest class is generated as previously described but using the OR operator. This is the only case when a bifurcation node with a disjunction selected generates a logical level filter expression.

Finally, the optional component of the GraphTQL filters produces *Selective union* logical level expressions as follows. Let *Co* be the set of nodes and *Eo* be the set of edges of the optional component of a *Filter*. Let P be $\{p_1, p_2, ..., p_t\}$ such that $p \in P$ is a simple path connecting a node in *Co* with a node in the filter pattern and p includes only one node of the filter pattern. Being $Cx = \{x_1, x_2, ..., x_k\}$ the set of nodes that are in the filter pattern as well as in at least one path of P. A *Selective Union* expression is generated for each node in *Cx*:

$$SelUnion(\langle S, I\rangle, \langle S', I'\rangle, x_1, P_{x1}) \rightarrow \langle S', I'\rangle$$
$$\dots$$
$$SelUnion(\langle S, I\rangle, \langle S', I'\rangle, x_k, P_{xk}) \rightarrow \langle S', I'\rangle$$

where P_{xi} $(1 \le i \le k)$ is the set of paths having x_i as the final node. Note that *Selective Union* adds nodes and edges to S' and I'; thus these graphs accumulate the result.

On the other hand, the *Filter + additional data* operator generates only one *Selective union* expression with a set of paths that connect the interest class with some node in $Co \bigcup Cc$ such that $p \in P$ if p is not part of another path q, $q \in P$ and $p \neq q$.

4.2.3. Put Two Objects Closer

The *Put Two Objects Closer* operator takes as input a schema S, an instance of S, I, a simple undirected path marked on the schema, $\{m_i, e_1, m_1, ..., e_k, m_f\}$, and the names to create the new relationship (a class name c and the edge labels α_i and α_f). The operator is defined by the composition of both the *Path contraction* logical level operator and the *Selective union* logical level operator (defined in Section 4.3). The *Path contraction* operator replaces the path $\{m_i, e_1, m_1, ..., e_k, m_f\}$ and the *Selective union* operator adds data connected to m_i and m_f. The logical level expression for *Put Two Objects Closer* is as follows:

$PathCt(\langle S, I\rangle, \{m_i, e_1, m_1, ..., e_k, m_f\}, c, \alpha_i, \alpha_f) \rightarrow \langle S', I'\rangle$
$SelUnion(\langle S, I\rangle, \langle S', I'\rangle, m_i, \{p_{i1}, p_{i2}, ..., p_{ik}\}) \rightarrow \langle S', I'\rangle$
$SelUnion(\langle S, I\rangle, \langle S', I'\rangle, m_f, \{p_{f1}, p_{f2}, ..., p_{fl}\}) \rightarrow \langle S', I'\rangle$

where $Pi = \{p_{i1}, p_{i2}, ..., p_{ik}\}$ is the set of all undirected simple paths in S such that any $p_{ij} \in Pi$ $(1 \le j \le k)$ begins with m_i; and $Pf = \{p_{f1}, p_{f2}, ..., p_{fl}\}$ is the set of all undirected simple paths in S' such that any $p_{fq} \in Pf$ $(1 \le q \le l)$ begins with m_f.

4.3. GraphTQL logical level operators

GraphTQL logical level operators are defined in terms of the transformations they apply on the schema and instance graphs. For this definition the GDM data model specification is used; however, since the schema is mandatory in GraphTQL, this restriction was included. Thus, any instance graph I must be an instance of a schema graph S.

For this section, it is necessary to introduce the following definitions:

- *Nodes function: Nodes*(q) denotes the set of nodes in q with q being an edge or a path.
- *Edges function: Edges*(p) denotes the set of edges in p with p being a path.
- *Path correspondence*: We say that a path in the instance graph *corresponds* to a path in the schema graph to express subgraph matching.
- *Subgraph coverage*: Given an instance I belonging to a schema S and S' a subgraph of S, we say that the nodes and edges in I related to any node or edge in S' is *the subgraph of I covered by S'*. Formally, let $I = \langle N_I, E_I, \lambda_I, \sigma_I, \rho_I\rangle$ and let ξ be the extension relation from S to I. We say that $I' = \langle N_{I'}, E_{I'}, \lambda_{I'}, \sigma_{I'}, \rho_{I'}\rangle$ is the *subgraph of I covered by* $S' = \langle N_{S'}, E_{S'}, \lambda_{S'}, \sigma_{S'}\rangle$, $S' \subseteq S$; provided that the following five assertions hold:
  a) $N_{I'} = \{n \in N_I \mid \exists\ m, m \in N_{S'}$ and $\xi(m, n)\}$;
  b) $E_{I'} = \{(n_i, \alpha, n_j) \in E_I \mid \exists\ (m_k, \alpha, m_l), (m_k, \alpha, m_l) \in E_{S'}, \xi(m_k, n_i), \xi(m_l, n_j)\}$
  c) for each $n \in N_{I'}$, $\lambda_{I'}(n) = \{c \in \lambda_I(n) \mid c = \lambda_{S'}(m)$ and $m \in N_{S'}\}$ and $\sigma_{I'}(n) = \sigma_I(n)$;
  d) for each $n \in N_{I'}$ such that $\sigma_{I'}(n) \in \mathcal{B}$, $\rho_{I'}(n) = \rho_I(n)$;
  e) for each $e \in E_{I'}$, $\lambda_{I'}(e) = \lambda_I(e)$.

4.3.1. Subgraph Extraction

Subgraph Extraction and *Select a Portion* share the same semantics. They both select data covered by a picked schema connected subgraph. This subgraph does not represent a pattern, thus allowing for the retrieval of incomplete data. *Subgraph Extraction* takes as input a schema graph S, an instance of S, I, and a set of schema edges $\{e_{S1}, e_{S2}, ..., e_{Sk}\}$, and returns the subgraphs S' and I':

$Extract(\langle S, I\rangle, \{e_{S1}, e_{S2}, ..., e_{Sk}\}) \rightarrow \langle S', I'\rangle$

where $S' = \langle N_{S'}, E_{S'}, \lambda_{S'}, \sigma_{S'}\rangle$, $E_{S'} = \{e_{S1}, e_{S2}, ..., e_{Sk}\}$, $N_{S'} = \bigcup_{e_S \in E_{S'}} Nodes(e_S)$, and for each $m \in N_{S'}$ and $e_S \in E_{S'}$: $\lambda_{S'}(m) = \lambda_S(m)$, $\lambda_{S'}(e_S) = \lambda_S(e_S)$, and $\sigma_{S'}(m) = \sigma_S(m)$. I' is the subgraph of I covered by S' (see Section 4.1 for *covered* definition).

Given the fact that Subgraph Extraction returns the union of the selected edges, cycles, if any, will be on the resulting S' and I' subgraphs. It is worth noting that cycles formed by the chosen edges do not have any effect on the operator semantics.

4.3.2. Logical Level Filter

Logical Level Filter processes the filter pattern of GraphTQL *Filter* and *Filter + additional data* operators. It retrieves nodes belonging to the interest class connected by a set of selected undirected paths with other nodes that fulfill a given condition.

Logical Level Filter takes as input a schema graph S, an instance of S, I, an interest class m_I, and a logical expression, *exp; exp* is formed by the conjunction (or disjunction) of basic expressions $bexp_1\ bexp_2\ ...\ bexp_k$ each one having the form $m_{j\{p_j\}}\ op_j\ o_j$ $(1 \le j \le k)$, where m_j is a bifurcation or a conditioned node $(m_j \in N_S)$, p_j is an undirected path between m_I and m_j, op_j is an operator, and o_j an operand. GraphTQL currently supports *inSubgraph, exists, in, between*, and comparison operators. Particularly, *inSubgraph* operator allows to have composition of several filter operations. A node n_j belonging to m_j class, fulfills the basic expression $m_{j\{p_j\}}$ *inSubgraph* $\langle S', I'\rangle$ when $n_j \in N_{I'}$. *Logical Level Filter* returns the subgraphs S' and I'.

$LogicLFilter(\langle S, I\rangle, m_I, \text{exp}) \rightarrow \langle S', I'\rangle$

where $S' = \langle N_{S'}, E_{S'}, \lambda_{S'}, \sigma_{S'}\rangle$, $N_{S'} = \bigcup_{j=1}^{k} Nodes(p_j)$, $E_{S'} = \bigcup_{j=1}^{k} Edges(p_j)$, and for each $m \in N_{S'}$ and $e_S \in E_{S'}$: $\lambda_{S'}(m) = \lambda_S(m)$, $\lambda_{S'}(e_S) = \lambda_S(e_S)$, and $\sigma_{S'}(m) = \sigma_S(m)$. Now, let C_I be the set of entities belonging to m_I, $C_I = \{n \mid n \in N_I \wedge \exists\ \xi(m_I, n)\}$; for each $n \in C_I$, n satisfies the basic expression $bexp_j$ if $P_{nj} \neq \emptyset$ where P_{nj} is the set of paths beginning in n and satisfying the basic expression $bexp_j$; i.e. $P_{nj} = \{p \mid p$ *corresponds with* p_j, n *is the initial node of* p, n_f *is the final node of* p *and* $eval(n_f, op_j, o_j) = true\}$ for $1 \le j \le k$, with *eval* being a function that evaluates the basic expression $n_f\, op_j\, o_j$. Now, let $O = \{n \mid n \in C_I \wedge n$ *satisfies* exp$\}$ where *exp* is evaluated for each $n \in C_I$ by using n's basic expressions (*bexp*) results. Then, $I' = \langle N_{I'}, E_{I'}, \lambda_{I'}, \sigma_{I'}, \rho_{I'}\rangle$ where, let P_I be $\{p \mid p \in P_{nj} \wedge n \in O$ *for* $0 \le j \le k\}$, $N_{I'} = \bigcup_{p \in P_I} Nodes(p)$, $E_{I'} = \bigcup_{p \in P_I} Edges(p)$, and for each $n \in N_{I'}$ and $e_I \in E_{I'}$: $\lambda_{I'}(n) = \{c \in \lambda_I(n) \mid c = \lambda_{S'}(m)$ and $m \in N_{S'}\}$, $\lambda_{I'}(e_I) = \lambda_I(e_I)$, $\sigma_{I'}(n) = \sigma_I(n)$ and $\rho_{I'}(n) = \rho_I(n)$.

Logical Level Filter assumes that each basic expression is independent from other basic expressions. This is ensured by the filter clarification dialogues.

4.3.3. Selective Union

Selective Union is a binary operator that processes the optional component of GraphTQL *Filter* operator and *Put Two Objects Closer* operator by adding to graphs $\langle S', I'\rangle$ data from graphs $\langle S, I\rangle$. *Selective Union* retrieves entities belonging to m_I $(m_I \in N_S \wedge m_I \in N_{S'})$, then adds to I' data in I connected to those entities by a simple undirected path that corresponds with one in P_S. P_S is a given set of schema paths, each path beginning with m_I. In general terms:

$SelUnion(\langle S, I\rangle, \langle S', I'\rangle, m_I, P_S) \rightarrow \langle S', I'\rangle$

Given $P_S = \{p_{S1}, p_{S2}, ..., p_{Sk}\}$ and $S' = \langle N_{S'}, E_{S'}, \lambda_{S'}, \sigma_{S'}\rangle$, resulting S' is defined by

$N_{S'} = \left(\bigcup_{p \in P_S} Nodes(p)\right) \bigcup N_{S'}$, $E_{S'} = \left(\bigcup_{p \in P_S} Edges(p)\right) \bigcup E_{S'}$, and for each $m \in N_{S'}$ and $e_S \in E_{S'}$: if $m \in S$ then $\lambda_{S'}(m) = \lambda_S(m)$ and $\sigma_{S'}(m) = \sigma_S(m)$, if $e_S \in S$ then $\lambda_{S'}(e_S) = \lambda_S(e_S)$. Now, given $O = \{n \mid n \in N_I \wedge n \in N_{I'} \wedge \xi(m_I, n)\}$ and $P_I = \{p \mid p$ *is a path in* I' *beginning with* n, $n \in O$, *and corresponding with a path in* $P_S\}$, the resulting $I' = \langle N_{I'}, E_{I'}, \lambda_{I'}, \sigma_{I'}, \rho_{I'}\rangle$ is composed of $N_{I'} = \left(\bigcup_{p \in P_I} Nodes(p)\right) \bigcup N_{I'}$, $E_{I'} = \left(\bigcup_{p \in P_I} Edges(p)\right) \bigcup E_{I'}$, and for each $n \in N_{I'}$ and $e_I \in E_{I'}$: $\lambda_{I'}(n) = \{c \in \lambda_I(n) \mid c = \lambda_{S'}(m)$ and $m \in N_{S'}\} \bigcup \lambda_{I'}(n)$, if $e_I \in I$ then $\lambda_{I'}(e_I) = \lambda_I(e_I)$, if $n \in I$ then $\sigma_{I'}(n) = \sigma_I(n)$ and $\rho_{I'}(n) = \rho_I(n)$.

4.3.4. Path Contraction

Path Contraction replaces nodes and edges corresponding to a simple undirected path

$\{m_i, e_1, m_1, ..., e_k, m_f\}$ with a new relationship formed by a composite value node m connected to the initial and final nodes of the path. The path could be a simple cycle (i.e. $m_i = m_f$). In general terms, the *Path Contraction* expression is as follows:

$PathCt(\langle S, I\rangle, \{m_i, e_1, m_1, ..., e_k, m_f\}, c, \alpha_i, \alpha_f) \rightarrow \langle S', I'\rangle$

where, c is a class name for m $(\lambda(m) = c)$, α_i and α_f are, respectively, attribute names for the edges connecting m with m_i and m_f; m_i and m_f are object or basic value nodes.

The resulting schema is $S' = \langle N_{S'}, E_{S'}, \lambda_{S'}, \sigma_{S'}\rangle$, where $N_{S'} = \{m_i, m_f, m\}$ with m, a new composite value node, such that

$\sigma(m) = com$ and $\lambda(m) = c$. $E_{S'} = \{(m, \alpha_i, m_i), (m, \alpha_f, m_f)\}$, (m, α_i, m_i) and (m, α_f, m_f) are new edges. Additionally, $\lambda_{S'}(m_i) = \lambda_S(m_i)$, $\lambda_{S'}(m_f) = \lambda_S(m_f)$, $\lambda_{S'}(m, \alpha_i, m_i) = \alpha_i$, $\lambda_{S'}(m, \alpha_f, m_f) = \alpha_f$, $\sigma_{S'}(m_i) = \sigma_S(m_i)$, $\sigma_{S'}(m_f) = \sigma_S(m_f)$.

Now, let $P_I = \{p | p \text{ is a path in } I \text{ corresponding to } \{m_i, e_1, m_1, ..., e_k, m_f\}\}$. Then, the resulting instance is $I' = \langle N_{I'}, E_{I'}, \lambda_{I'}, \sigma_{I'}, \rho_{I'} \rangle$, where $E_{I'} = \{(n_c, \alpha_i, n_i) \bigcup (n_c, \alpha_f, n_f) | \text{ for some } p \in P_I, n_i \text{ is the initial node of } p, n_f \text{ is the final node of } p, \xi(m_i, n_i), \xi(m_f, n_f), n_c \text{is a new node such as } \sigma_{I'}(n_c) = com, \lambda_{I'}(n_c) = c, \xi(m, n_c)\}$; $N_{I'} = \bigcup_{e_{I'} \in E_{I'}} Nodes(e_{I'})$, for each $n \in N_{I'}$ such that $n \neq n_c$: $\lambda_{I'}(n) = \{d \in \lambda_I(n) | \exists\, m, m \in N_{S'}, d = \lambda_{S'}(m)$ and $\xi(m, n)\}$, $\sigma_{I'}(n) = \sigma_I(n)$ and $\rho_{I'}(n) = \rho_I(n)$; and for each $e_{I'} = (o, \alpha, q) \in E_{I'}$, $\lambda_{I'}(e_{I'}) = \alpha$.

5. UCD development approach

User-Centered Design (UCD) includes a set of methods and techniques to design products and systems that meet usability features, taking into account the user experience [19,31]. It meets three basic principles: (1) focus on the user from the earliest stages of the development process, (2) evaluation of the product usability, and (3) iterative design. Thus, the main characteristic of UCD design is the user involvement during the product development.

Since a central interest of this study was to provide end-users (experts in an application domain) a query tool that eases formulating *ad-hoc* medium complexity queries, the application of UCD techniques was appropriate. UCD focuses on improving the user experience beyond functionality and aesthetics. When a task is simple, a successful user experience is built into the definition of the product. On the other hand, on complex tasks the user experience is independent of the definition of the product [19]: thus requiring particular design efforts to achieve ease of use.

Petrelli [28] proposes three phases of usability evaluation for the development of a software product. The first phase seeks to understand the task performed by the user. In the second phase, design ideas are corroborated using a low fidelity prototype (e.g. a paper prototype). The third phase includes an assessment by means of a high fidelity prototype (e.g. a functional prototype). Following Petrelli's proposal, GraphTQL development included usability tests (with the participation of health professionals) for each of these phases. We favored usability testing over inquiry and inspection methods that do not involve users directly, because we wanted to know if the language would make it easier when carrying out complex tasks. It is possible that usability experts would not be able to assess the ease of use accurately due to their technical background which places them on a different level and context from end-users. Thus, experts might overlook features that could lead to difficulties for users. Conversely, they might consider difficult other features that would not pose a problem for end users, due to the users knowledge of the domain.

5.1. Preliminary interview

The preliminary interview's objective was to identify information needs of health professionals regarding patients' clinical histories and scenarios where needed. Four physicians participated in a free-form interview. These physicians were specialized in different areas: gastroenterology, pediatrics, nutriology, epidemiology, gynecology, and otolaryngology. We obtained 31 query examples of several scenarios, such as: consultation, clinical research, teaching and learning, health services management, prevention planing, and epidemiological surveillance. These queries were classified into three categories: queries to retrieve data registered in the records of specific patients, queries to find sets of patients that share some features, and queries that require grouping and aggregate functions. Currently, GraphTQL focuses on the first two categories.

5.2. Exploratory test

The purpose or this test was to confirm that the conceptual graph, the proposed operators, the icons and the explanatory texts were understandable for the participants. A paper prototype was used. Participants were asked to find particular data on the graphs and to identify the operation applied in the four cases which showed the graphs before and after the application of one operator.

We widened the participants profile to include other professionals with interest in the clinical history information. Thus, the test was carried out with a public health doctor, an epidemiology medical specialist, and a nurse. The main results were: (1) participants identified and understood the information represented by the schema and instance graphs (participants took longer to comprehend the instance graph than to comprehend the schema graph); (2) the shape of labeled composite value nodes made them difficult to notice; (3) 91.6% of the answers about the identification of operations applied were correct (in 16.6% of them the test moderator intervened to point out some part of the graphs); (4) participants affirmed that the operators allowed them to propose relevant queries useful for clinical practice and research; (5) participants also expressed that their first impression was that the graph representation was complex; however, once they used them to answer the questions, they found the graphs easy to read and understand.

5.3. Evaluation tests

There were two evaluation tests. The objective of the first was to evaluate the GUI organization, icons, and colors by using two functional prototypes. The test involved: an expert in user interface design; three physicians; and six students, each with a different profile (medicine, dentistry and computer science). Their observations allowed for a better selection of: a window organization, set of colors, icons and names of operators. Participants' opinions were weighted differently according to their profile; the opinions of the expert and the physicians were the most weighted. It is worth noting that the aspects relevant for each participant varied in according with her/his profile. Physicians and medical and dentistry students focused on the utility of the tool to support research and decision-making; the user interface expert focused on interface design aspects; while, computer science students centered their attention on the language operators and the use of the graph as a data representation model.

In the second evaluation test, four participants (two professionals and two students of health-related areas) evaluated how easy it is to use the language, based on a functional prototype. This test and its results are presented in Section 7.

5.4. Comparative test

The last usability test was exploratory and had a between-subjects design. Its objective was to validate the ease of query formulation by comparing two visual query tools using graph databases. The two tools were: Gruff [1], a graphical query tool for SPARQL queries; and GraphTQL, the proposed language. The evaluation focused on the formulation of medium complexity queries.

We had into account several features to select the tool for the comparison test, among them: a) the support of medium complexity queries, including complex non-local filters and incomplete data management; b) the availability to download and use the tool; c) maturity and completeness of the software developed; d) formal definition of the language supported.

The limitations in the expressiveness of most exploration and analysis tools make them inappropriate for comparison since they do not support medium complexity queries. Therefore, we had to choose among the visual query languages and the graphical query tools. Of these, only a few were found available to be used, being Gruff one of the most mature and complete, since it offers additional functionality

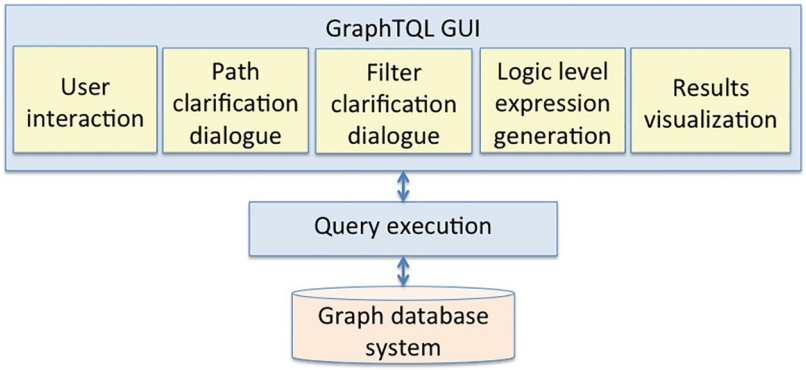


Fig. 6. GraphTQL functional structure.

beyond the construction and execution of queries. For example, it provides a mean for exploring the data graph that was useful to the participants on the test. Finally, we found more appropriate to use a graphical query tool for SPARQL queries because SPARQL has a formal and standard definition provided by the W3C, whereas many visual query languages do not provide a formal definition. A formal definition of the language avoids possible ambiguities resulting from interpretation of the queries.

This test and its results are presented in Section 7.

6. GraphTQL implementation

GraphTQL is proposed as a layer of interaction with the end user in an architecture that allows integrating data from several information systems in a specific application domain. The GraphTQL functional structure, shown in Fig. 6, has three components: 1) the GraphTQL GUI that supports the query formulation process, which was implemented in Java[2] with the Jung framework[3]; 2) the query executor that includes the logic level operators, implemented in Java; and 3) the graph database system for which Neo4j[4] [29] is used.

The GraphTQL GUI includes modules that implement the following: a) loading the schema graph from a graphML file b) users interaction during query formulation; c) clarification dialogues for path disambiguation; d) clarification dialogues for filter condition disambiguation; e) generation of the query expression with logical level operators; and e) result visualization (not currently implemented).

The implementation of GraphTQL logical level operators can be done with languages based on pattern matching or path traversal. Pattern matching queries retrieve from the database all occurrences that coincide with a given pattern. A pattern is a graph whose nodes and edges could be labeled with variables or literals. Pattern matching is, generally, defined in terms of subgraph isomorphism [18]. On the other hand, in the path traversal approach [30], nodes are reached by navigating through edges, beginning in a particular set of nodes and following a given path. The low-level computational cost of path traversal [16,30] is an advantage for executing some types of queries [3].

Since the input for *Logical Level Filter, Selective Union* and *Subgraph Extraction* are defined in terms of the paths that connect an interest class with other nodes in the graph, it is possible to find the corresponding data following each path and then to consolidate partial results. Thus, these operators were implemented using the Neo4j Traversal API [37], with the aim to take advantage of path traversal's low-level computational cost.

Additionally, we developed a generator of synthetic data for the data model depicted in Fig. 1, based on portions of structured vocabularies or ontologies, such as ICD10[5], LOINC[6], CPT[7], and SNOMED[8]. The datasets it generates include a set of clinical cases that described symptoms, tests, diagnostic images, procedures, medical orders, and specialties. These data were used for the usability tests queries in order to have some grade of coherence in the data. To enlarge the size of the datasets, the generator added other clinical cases resulting from the random combination of symptoms, tests, diagnostic images, procedures, medicines and medical specialties.

7. Usability and execution evaluation

7.1. Usability evaluation

The test plan for both the second evaluation and the comparative tests consisted of four stages: the first was the introduction in which a moderator described the purpose and plan of the test; the second was the presentation of the tool in order to train the user in the formulation of queries (using examples of queries over a movie and TV series database); the third stage required that the participant carry out the formulation of four queries, listed in Table 2, over a synthetic database of clinical histories which schema is shown if Fig. 1; finally, in the fourth stage, the participant filled in a SUS [5] questionnaire and gives her/his opinions.

We would use different databases for the second and third stages since we wanted to evaluate the ease of use of the schema graph and to note what happened when participants used a new data model. The graphML definition of schema graph for both databases was constructed manually to feed GraphTQL.

During the query formulation activity, users would be able to ask for help; therefore, the requests would be classified into three categories according to how the assistance impacted the effectiveness of the query formulation (understood as the ability to find a correct solution, i.e. combination of operators). Category A requests would have a significant impact (e.g. the moderator gives clues to formulate the query). Category B requests would have a small impact (e.g. the moderator explains something related to the language functionality). Finally, Category C requests would have no impact (e.g. explanations regarding the use of mouse clicks and graph drag).

Third stage queries are of medium complexity in the sense that they include patterns formed by several triples with combinations of conjunction and disjunction of filter conditions as well as management of incomplete data. The filter conditions are not local since they impact nodes which are different to the conditioned nodes (i.e. used to select objects related to the ones that fulfill the condition). Four queries were

[2] https://www.oracle.com/java/.
[3] http://jung.sourceforge.net.
[4] http://neo4j.com.
[5] http://apps.who.int/classifications/icd10/browse/2015/en.
[6] http://loinc.org.
[7] http://www.ama-assn.org/go/cpt.
[8] https://www.nlm.nih.gov/research/umls/Snomed/snomed_main.html.

Table 2
Queries used in usability evaluation.

Q1	Find the description of the diagnosis registered in the family history of the patient with ID 723628; include family relationship if available
Q2	Find the description of the diagnosis given to the patient with ID 723628 by medical specialists in "ENDOCRINOLOGY"; and, if available, encounter dates and the doses and description of medicines prescribed on those encounter appointments
Q3	Find patient ID and description of diagnosis and procedures registered to patients that have received an "OSTEOMYELITIS" diagnosis since the year 2000. Include all diagnoses given, even the ones different to "OSTEOMYELITIS".
Q4	Find patients that have a diagnostic image study with a "GALLSTONES" annotation and a "COLECTOMY" procedure registered. Give, if available, birthdate, gender, and race of those patients. Note that both the image study and the procedure could be registered on different encounters

chosen according to the complexity level and based on the results of the preliminary interviews. Complexity levels similar to Bell and Rowe [10] and Owei et al. [27] proposals were established taking into account particular graph query features and the hypothesis of this study. Level one queries (e.g. *Q1*) included pattern matching with **one filter condition**, with or without optional data. Level two queries (e.g. *Q2*) required pattern matching with **more than one filter condition**, with or without optional data. Level three queries (e.g. *Q3*) included pattern matching with several filter conditions, **requiring all data connected** to the objects that fulfilled the conditions. Finally, level four queries (e.g. *Q4*) included pattern matching with several filter conditions in addition to some **other operation** (e.g. the SPARQL union). The test focused on query formulation; it did not cover the presentation of the results.

Three performance metrics and a SUS (Software Usability Scale) questionnaire [5] were used to evaluate ease of use and satisfaction, respectively. The performance metrics were: effectiveness, efficiency, and the number of help requests for each category. Effectiveness included the number of queries returning the data required and the number of errors in the queries which had not been correctly formulated (only different types of error per query were counted, e.g. if the query needed three optional elements of data not included by the participant, this counted as one error). Efficiency was measured according to the time required to formulate the query. Help requests for each category were counted.

The test proposal (hypothesis, plan, queries, metrics) was subject to an expert judge. His observations allowed for some adjustments on the definition of the metrics. The expert judge also suggested having more than one query per complexity level; however, this was not possible due to the participants' limited time availability.

Table 3 shows the results of the second evaluation test regarding the number of the correctly formulated queries and the number of errors made during the formulation of the non-correctly formulated ones. Even though the rate of correctly formulated queries was low (18,8%), the test evidenced repetitive problems that allowed us to implement some improvements. Among them, the inclusion of the path clarification dialogue, the change of the name of filter operators (previously called *Portion with filter* and *Object with characteristic*), a change in the definition of the *Filter + additional data* operator (previously *Object with characteristic*), and the validation of some operator parameters (e.g. marks that are not part of the operator's parameters can indicate a mistake in the choice of the operator).

As an additional result, participants affirmed that the language is easy to use, requires a short training phase, the graph representation is understandable; and furthermore, it would be professionally useful.

Four health professionals and four medical students participated in the last test: the comparative test. Half evaluated the GraphTQL and the other half, the SPARQL graphical interface, Gruff.

Table 3
Second evaluation test results.

	Q1	*Q2*	*Q3*	*Q4*	All queries
Correctly formulated	0%	50%	25%	0%	18,8%
Number of errors per query (avg)	1,25	1,2	2,0	1,5	1,5

Table 4 shows: the average time, the percentage of the activities correctly formulated, the average number of errors, and the average number of help requests for each query. The last row shows the score of the usability scale (SUS) for each tool (according to the SUS design, each score ranges in an interval between 0 and 100, 100 being the best result). Results show significant differences in the time of formulation for both tools; the average time for the SPARQL graphical interface is more than three times that of GraphTQL. Perhaps this is because the participants had to build the query pattern from scratch; and, given the fact that the lists of entities and properties were not enough in order to decide what nodes and edges they needed for the query pattern, the participants had to continually check data graph examples. The number of correct answers (zero) and errors (3.1 on average) using the SPARQL graphical interface suggests that this tool required more training time, possibly due to the complexity of the concepts involved. However, participants carried out the activities with confidence, and most of the help they required was related to interface management aspects (e.g. where to find or how to get to certain options). It is worth noting that, in some cases, the usability evaluation for the SPARQL graphical tool was high in spite of the incorrect formulation of the queries. This was perhaps due to the perception of usefulness of the tool for each of their professional fields. The test results also suggested a prone to introduce errors when participants needed to explore the data graph in order to construct the query pattern. These errors included: wrong edge direction, use of wrong paths to represent relationships, or the mistaking of attribute names. Additionally, the majority of the participants did not take into account the concept of optional data; others confused the nodes representing a variable value with the nodes representing a literal value; and others had difficulties specifying patterns for queries that required more than one variable for the referencing of the same entity. Finally, it is worth noting that the average time necessary to explain the tool functionality was 50 minutes for the SPARQL graphical interface and 35 minutes for the GraphTQL.

When comparing GraphTQL results of the second evaluation (Table 3) with the ones of the comparative test (Table 4) two notable differences were encountered: a significant increase of the percentage of correctly formulated queries (18,8% vs. 62,5%); as well as an important reduction in the average number of errors (1.5 vs. 0.7). These variations indicated that the changes incorporated into GraphTQL had a positive impact, showing the importance of conducting usability tests during development phases. However, the comparative test results revealed the necessity to improve the clarification dialogues in cases where the user needs to apply a combination of logical operators (conjunction and disjunction).

7.2. Execution evaluation

GraphTQL logic level operator implementation was evaluated by comparing the execution time of five queries running on our system versus the same queries written on SPARQL and executed by using AllegroGraph[9]. For both cases, a Java application, which connects to the respective database, opened a file that contained the query and executed it. Cold execution time (i.e. the first time the query runs after

[9] http://franz.com/agraph/allegrograph/.

Table 4
Comparative test - Average of the performance results.

	SPARQL graphical interface				GraphTQL			
Query	Time (mm:ss)	Correct queries	Number of errors	Number of help requests	Time (mm:ss)	Correct queries	Number of errors	Number of help requests
Q1	11:12	0%	2,25	B:0.5, C:1	4:11	100%	0	B:0.25, C:0.5
Q2	13:08	0%	2,75	A:0.25, B:0.25, C:0.25	3:17	50%	0,5	C:0.25
Q3	10:48	0%	5	None	4:25	100%	0	C:0.75
Q4	10:31	0%	2,25	A:0.25	2:35	0%	2,5	C:0.25
Total	**11:24**	**0%**	**3,1**	**A:0.12, B:0.19, C:0.31**	**3:31**	**62,5%**	**0,7**	**B:0.06, C:0.44**
SUS Score			**53.0**				**71.9**	

starting the database service) and warm execution time (i.e. subsequent executions that benefit from caches) were measured. The measure of time for the SPARQL queries began before sending the query to the server and ended after displaying the results. For GraphTQL, it began before opening the file that held the query and ended after displaying the results. The evaluation included the queries listed on Table 2 plus an additional one:

Q5: Find patients with a "CAROTID PULSE TRACING WITH ECG LEAD" diagnostic image and patients with an "INSERTION OF PALATAL IMPLANT" procedure registered. Give the patient's ID if available.

The comparison was carried out in two different settings. On each one ten measures were taken for each metric (cold and warm times). The first setting had a database of a million triples running on a virtual machine (64bits) with Ubuntu 14.04LTS and 1024MB of RAM memory. The host has an Intel Core I3-4010Y processor (64bit), 4G of RAM memory and Windows 8. AllegroGraph 4.14.1, Java JRE 1.7, Neo4j 2.1.3 and VirtualBox 4.3.28 were used. The second setting had a database of five million triples running on a cluster. It consisted of a front-end with 32 AMD processors (2.3 GHz), 32G of RAM and Ubuntu Server 14.04; and four compute nodes, each one with 64 cores (AMD, 2.3 GHz), 64G de RAM and Ubuntu Server 14.04. A 10Gbit Ethernet connected nodes for processing and 1Gbit Ethernet was used for both administration and file system. AllegroGraph 5.1, Java JRE 1.8, and Neo4j version 2.1.3 were used.

Fig. 7 shows the size of the query results based on the number of triples, the average cold and warm execution times (milliseconds); shown on both virtual machine and cluster environments. Results obtained in both settings show similar trends and taking into account that part of the process of the GraphTQL implementation does not include optimization techniques; it is worth noting that 80% of the average query processing times for GraphTQL is lower than the one for SPARQL queries. The other 20% does not exhibit any particular characteristic that explains the difference in their behavior. This percentage corresponds to different queries (1, 2, and 3), for cold and warm execution times, and for both execution environments.

8. Conclusion and future work

In this paper a visual query language which facilitates the formulation of complex ad-hoc queries has been presented. The language, called GraphTQL, has several distinctive features which include: (a) using a graph data model that differentiates schema and instance graphs; (b) taking advantage of the schema graph to represent a conceptual model of the data domain and to be used as the base of the direct interaction manipulation mechanism; (c) querying data by transforming the schema graph through successive application of transformation operations that reduce the set of selected data progressively; (d) liberating end-users of the need to explicitly manage incomplete data; (e) guiding end-users to formulate complex filters.

All of these features are the result of a user-centered design (UCD) approach whose application domain was the medical domain. This approach included five tests carried out at different stages of the project with the participation of professionals and students of health-related programs. The interaction with the test participants allowed for the identification of query needs which were useful for their professional activities. This interaction also allowed for the making of design decisions using a broader criteria than that applied in a traditional development process; and for the introduction of changes on the GraphTQL design. Moreover, these usability tests validated the merit of our proposed language.

The UCD approach had a high impact on the design of GraphTQL. It led beyond the task realization, taking into account the effect of each language design decision based on the ease of use. It resulted in establishing a difference between GraphTQL and the systems that define a visual notation for language elements; given that an important part of the query formulation complexity is based both on the concepts underlying those elements as well as the ones resulting from their combination. Accordingly, the implementation of a strategy for the reduction of the user's cognitive load, resulted in both the reduction of the number of concepts to be learned as well as the number of factors users should consider when reasoning on how to obtain some required data. Other features included in the GraphTQL design are: (1) keeping the user interface simple and centered on the manipulation of the schema graph; (2) using dialogues to guide the user in specifying complex query elements, such as composite condition expressions or the selection of paths that are part of a cycle in the graph; (3) defining an aesthetic design that combines appropriate graph layout (regarding symmetry and clarity), frame layout, colors, and icons; and (4) naming the entities and attributes of the schema graph in accordance with the terms used by final users.

The tests allowed for the corroboration of the design decisions made during different stages of the proposed language development process. As a result, changes were made to: the name and definition of the operators; the way paths are marked; the schema graph layout; and the validations of the inputs required for each operator. It was noted that small technical changes can generate great impact on the usability of the product. However, one must be careful with the suggestions that the tool provides by default since it was noted that users tend to accept these suggestions unquestioningly, sometimes leading to errors. It was also noted that participants, with seemingly greater effort, took more time to interpret the instance graph than the schema; thus validating the choice of using the schema graph to support the query formulation. Additionally, it was noted that the usage of unusual vocabulary could generate difficulties when trying to understand and use the data model. Although, the first impression was that the schema graph produced an information overload for users, once the participants used the tool, they found the graph easy to peruse, read and understand. This could be due to their expertise in the domain area.

All these aspects reflect the importance of having interaction with users during the development process. On the other hand, in all five studies, participants emphasized the usefulness of this kind of tool for the development of their activities.

The comparative test showed that participants got better results

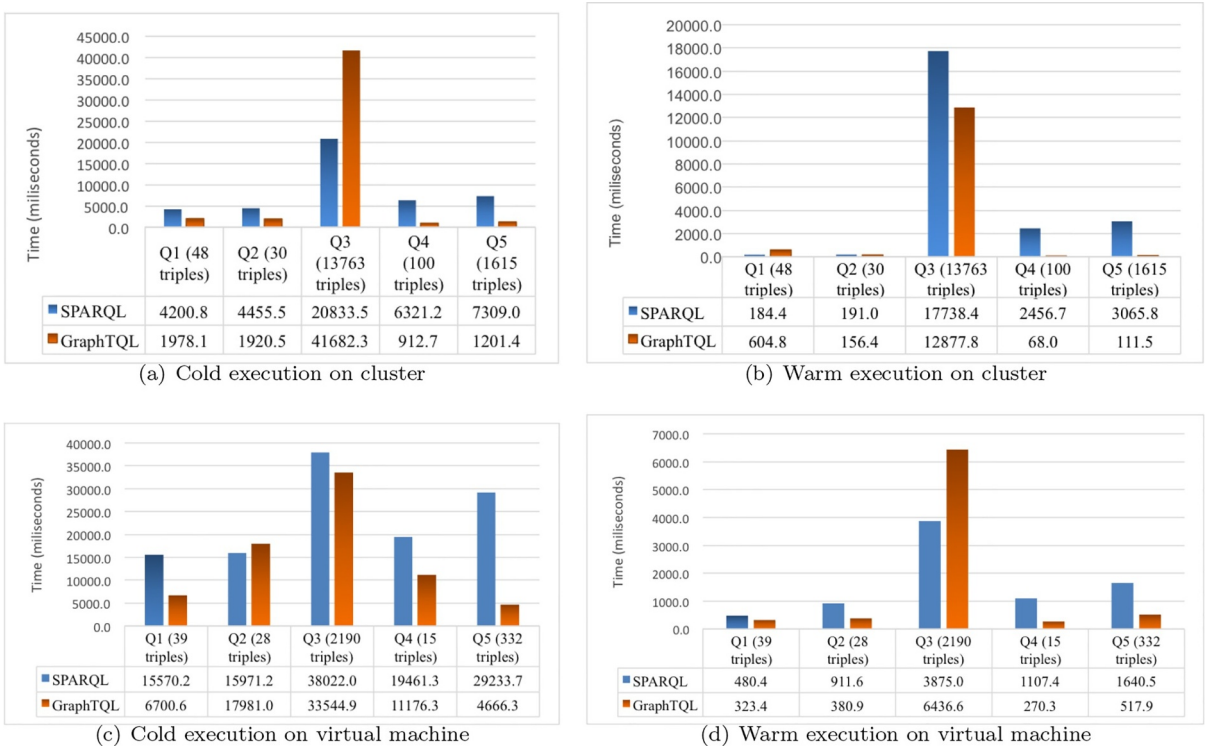


Fig. 7. Average cold and warm execution times comparison.

when formulating queries on GraphTQL, than when formulating queries using the graphical interface for SPARQL. These results are made evident by the number of correctly formulated queries, the number of errors committed during query formulation, the time needed to formulate the query, and the kind of help requests made. According to the tests, GraphTQL allows for the reduction of errors anticipated, such as: the wrong direction of edges; wrong paths; mishandling of incomplete data; absence of data required. In addition, there was a noted reduction of the difficulty involved in the specification of complex filters and the exploration of data. Unforeseen errors were also reduced, e.g. confusing the name of the variable with the filter condition; or placing a condition on an object instead of placing it on the attributes. Furthermore, building the pattern required more time because users need to browse the data to find out how that data is modeled; it also introduced errors. The comparative test also showed that changes introduced in GraphTQL as result of the evaluation test improved the query formulation task.

It is interesting to note that, in some cases, the usability evaluation for the SPARQL graphic interface was high despite the fact that the queries were not correct. This fact may indicate that these participants liked the tool because of its potential usefulness for their professional field.

As well, GraphTQL's logical level operators were formally defined using the GDM graph data model and were implemented based on path traversal. The implementation evaluation showed that path traversal is a good option for the implementation of GraphTQL. Future work should include optimization techniques employed during the implementation of the operators when outside the graph database engine.

There will always be a trade-off between user interface complexity and language expressiveness [8,22], causing a gap between the queries in a text-based language and those of a visual query language. The main limitations identified on GraphTQL are: (a) the impossibility of combining logical operators on bifurcation nodes (although the logical level expressions support that combination, the clarification dialogues do not provide this option); (b) the impossibility of comparing values of different attributes; (c) the lack of aggregate functions. These limitations should be dealt with in the future.

Future work will also include: the use of visualization techniques for the schema graph in order to facilitate its navigation; the automatic (or semi-automatic) generation of the schema graph based on an instance one; the application of query expansion techniques, taking advantage of the ontologies and structured vocabularies defined on the medical domain; the use of other forms in order to represent query results (e.g., a table representation could facilitate the application of aggregate functions over previous results); and the improvement of dialogues: using semantic metadata added to the data model, using a visual representation of some options, and using natural language to formulate explanations.

Acknowledgement

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

[1] J. Aasman, K. Cheetham, RDF browser for data discovery and visual query building, Procs. of the Workshop on Visual Interfaces to the Social and Semantic Web, (2011).
[2] J. Abello, S. Hadlak, H. Schumann, H.J. Schulz, A modular degree-of-interest specification for the visual analysis of large dynamic networks, IEEE Trans. Visual. Comput. Graph. 20 (3) (2014) 337–350, https://doi.org/10.1109/TVCG.2013.109.
[3] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, J.-L. Larriba-Pey, Benchmarking database systems for social network applications, First International Workshop on Graph Data Management Experiences and Systems - GRADES '13, ACM Press, New York, New York, USA, 2013, pp. 1–7, https://doi.org/10.1145/2484425.2484440.
[4] N. Athanasis, V. Christophides, D. Kotzinos, Generating on the fly queries for the semantic web: The ics-forth graphical rql interface (grql), in: S.A. McIlraith, D. Plexousakis, F. van Harmelen (Eds.), The Semantic Web – ISWC 2004, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 486–501.
[5] A. Bangor, P. Kortum, J. Miller, An empirical evaluation of the system usability scale, Int. J. Human-Comput. Interact. 24 (6) (2008).
[6] G. Barzdins, E. Liepins, M. Veilande, M. Zviedris, Ontology enabled graphical database query tool for end-users, in: H.-M. Haav, A. Kalja (Eds.), Databases and Information Systems V: Selected Papers from the Eighth Intl. Baltic Conf., DB&IS 2008, IOS Press, 2009.
[7] M. Bastian, S. Heymann, M. Jacomy, Gephi: an open source software for exploring and manipulating networks, Intl. AAAI Conf. on Weblogs and Social Media, (2009).
[8] C. Batini, T. Catarci, M.F. Costabile, S. Levialdi, Visual query systems: a taxonomy,

Procs. of the IFIP TC2/WG 2.6 Second Working Conf. on Visual Database Systems II, North-Holland Publishing Co., 1991.
[9] T. Beale, S. Heard, D. Kalra, D. Lloyd (Eds.), EHR Information Model, The openEHR Foundation, 2007.
[10] J. Bell, L. Rowe, An exploratory study of ad hoc query languages to databases, Eight Intl. Conf. on Data Engineering, IEEE Computer Society Press, 1992.
[11] H. Blau, N. Immerman, D. Jensen, A Visual Language for Querying and Updating Graphs, Technical Report, University of Massachusetts, Amherst, 2002.
[12] J. Borsje, H. Embregts, Graphical Query Composition and Natural Language Processing in an RDF Visualization Interface, Master's thesis, Erasmus School of Economics and Business Economics, 2006.
[13] T. Catarci, T. Di Mascio, P. Dongilli, E. Franconi, G. Santucci, S. Tessaris, Usability evaluation in the SEWASIE (SEmantic Webs and AgentS in Integrated Economies) project, 11th Intl. Conf. on Human- Computer Interaction (HCII), (2005).
[14] M. Cayli, M.C. Cobanoglu, S. Balcisoy, Glyphlink: an interactive visualization approach for semantic graphs, J. Visual Lang. Comput. 24 (6) (2013), https://doi.org/10.1016/j.jvlc.2013.09.002.
[15] D.H. Chau, C. Faloutsos, H. Tong, J.I. Hong, B. Gallagher, T. Eliassi-Rad, GRAPHITE: a visual query system for large graphs, 2008 IEEE International Conference on Data Mining Workshops, IEEE, 2008, pp. 963–966, https://doi.org/10.1109/ICDMW.2008.99.
[16] M. Dayarathna, T. Suzumura, Graph database benchmarking on cloud environments with XGDBench, Autom. Softw. Eng. 21 (4) (2013) 509–533, https://doi.org/10.1007/s10515-013-0138-7.
[17] A. Fadhil, V. Haarslev, OntoVQL: A Graphical Query Language for OWL Ontologies, Proceedings of the 2007 international workshop on description logics DL'07, Bozen, Bolzano University Press, 2008.
[18] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, Y. Wu, Graph pattern matching, Procs. of the VLDB Endowment 3 (1–2) (2010), https://doi.org/10.14778/1920841.1920878.
[19] J.J. Garrett, Elements of User Experience,The: User-Centered Design for the Web and Beyond, Pearson Education, 2010.
[20] J. Groppe, S. Groppe, A. Schleifer, Visual query system for analyzing social semantic web, Proc. of the 20th Intl. Conf. Companion on World Wide Web, WWW '11, ACM, 2011, https://doi.org/10.1145/1963192.1963293.
[21] F. Haag, S. Lohmann, T. Ertl, SparqlFilterFlow: SPARQL query composition for everyone, in: V. Presutti, E. Blomqvist, R. Troncy, H. Sack, I. Papadakis, A. Tordai (Eds.), The Semantic Web: ESWC 2014 Satellite Events, LNCS, 8798 Springer International Publishing, 2014, , https://doi.org/10.1007/978-3-319-11955-7_49.
[22] A. Harth, S.R. Kruk, S. Decker, Graphical representation of RDF queries, Procs. of the 15th intl. conf. on World Wide Web - WWW '06, ACM Press, New York, USA, 2006, p. 859, https://doi.org/10.1145/1135777.1135914.
[23] J. Hidders, Typing graph-manipulation operations, Proc. of the 9th Intl. Conf. on Database Theory, ICDT, Springer-Verlag, 2002.
[24] F. Hogenboom, V. Milea, F. Frasincar, K. Uzay, RDF-GL: A SPARQL-Based graphical query language for RDF, in: R. Chbeir, Y. Badr, A. Abraham, A.-E. Hassanien (Eds.), Emergent Web Intelligence: Advanced Information Retrieval, Advanced Information and Knowledge Processing, Springer London, 2010, pp. 87–116, , https://doi.org/10.1007/978-1-84996-074-8.
[25] H.V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, C. Yu, Making database systems usable, Procs. of the 2007 ACM SIGMOD Intl. Conf. on Management of data - SIGMOD '07, ACM Press, New York, USA, 2007, p. 13, https://doi.org/10.1145/1247480.1247483.
[26] M. Jarrar, M.D. Dikaiakos, A query formulation language for the data web, IEEE Trans. Knowl. Data Eng. 24 (5) (2012) 783–798, https://doi.org/10.1109/TKDE.2011.41.
[27] V. Owei, S.B. Navathe, H.-S. Rhee, An abbreviated concept-based query language and its exploratory evaluation, J. Syst. Softw. 63 (1) (2002) 45–67, https://doi.org/10.1016/S0164-1212(01)00139-X.
[28] D. Petrelli, On the role of user-centred evaluation in the advancement of interactive information retrieval, Inf. Process. Manage. 44 (1) (2008) 22–38, https://doi.org/10.1016/j.ipm.2007.01.024.
[29] I. Robinson, J. Webber, E. Eifrém, Graph Databases, O'Reilly Media, 2013.
[30] M.A. Rodriguez, P. Neubauer, The graph traversal pattern, in: S. Sakr, E. Pardede (Eds.), Graph Data Management: Techniques and Applications, IGI Global, 2011.
[31] J. Rubin, D. Chisnell, J. Spool, Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests, 2nd edition, Wiley Publishing, Inc., 2008.
[32] A. Russell, P. Smart, D. Braines, N. Shadbolt, NITELIGHT: a graphical tool for semantic query construction, Workshop, Semantic Web User Interaction (SWUI 2008), Florence, (2008).
[33] A.A. Sadanandan, K.W. Onn, D. Lukose, Ontology based graphical query language supporting recursion, Proc. of the 14th Intl. Conf. on Knowledge-based and Intelligent Information and Eng. Systems: Part I, LNCS 6276 Springer, 2010.
[34] A. Shamir, A. Stolpnik, Interactive visual queries for multivariate graphs exploration, Comput. Graph. 36 (4) (2012), https://doi.org/10.1016/j.cag.2012.02.006.
[35] B. Shneiderman, Direct manipulation: A Step beyond programming languages, Computer 16 (8) (1983), https://doi.org/10.1109/MC.1983.1654471.
[36] P.R. Smart, A. Russell, D. Braines, Y. Kalfoglou, J. Bao, N.R. Shadbolt, A visual approach to semantic query design using a web-based graphical query designer, Proc. of the 16th Intl. Conf. on Knowledge Engineering: Practice and Patterns, LNCS 5268 Springer, Berlin, Heidelberg, 2008, https://doi.org/10.1007/978-3-540-87696-0.
[37] The Neo4j Team, The Neo4j Manual v2.2.1, NeoTechnologies, 2015.
[38] C. Tominski, J. Abello, F. van Ham, H. Schumann, Fisheye tree views and lenses for graph visualization, Tenth International Conference on Information Visualisation (IV'06), IEEE, 2006, pp. 17–24, https://doi.org/10.1109/IV.2006.54.
[39] C. Tominski, J. Abello, H. Schumann, CGV An interactive graph visualization system, Comput. Graph. 33 (6) (2009) 660–678, https://doi.org/10.1016/j.cag.2009.06.002.
[40] S. van den Elzen, J.J. van Wijk, Multivariate network exploration and presentation: from detail to overview via selections and aggregations, IEEE Trans. Visual. Comput. Graph. 20 (12) (2014) 2310–2319, https://doi.org/10.1109/TVCG.2014.2346441.
[41] F. van Ham, A. Perer, Search, show context, expand on demand: supporting large graph exploration with degree-of-interest, IEEE Trans. Visual. Comput. Graph. 15 (6) (2009) 953–960.