Extended UML Class Diagram Constructs for Visual SPARQL Queries in ViziQuer/web

Kārlis Čerāns[1,*], Juris Bārzdiņš[2], Agris Šostaks[1], Jūlija Ovčiņņikova[1,*], Lelde Lāce[1,*], Mikus Grasmanis[1], Artūrs Sproģis[1]

[1] Institute of Mathematics and Computer Science, University of Latvia
{karlis.cerans, agris.sostaks, julija.ovcinnikova, lelde.lace, mikus.grasmanis, arturs.sprogis}@lumii.lv
[2] Department of Medicine, University of Latvia
juris.barzdins@gmail.com

Abstract. Visual notations based on customized entity relationship and basic UML class diagram paradigm of classes, associations and attributes are successfully used in data modeling and data querying alike, both in relational database and in semantic/conceptual model setting. We propose a number of constructs for extending the basic diagrammatic framework in the setting of visual specification of SPARQL select queries for enabling visual specification of hierarchic data instance and aggregated queries. Our proposal includes: (i) a visual notation for subqueries, (ii) separate lists of aggregated and grouping attributes in aggregated queries, (iii) query control nodes (unit and union) not corresponding to data instances, and (iv) integration with textual SPARQL fragments. We report on an initial user study validating the understandability of the subquery notion by users without specific IT training, as well as announce implementation of the proposed notation constructs in a web-based diagrammatic (multi-modal) query tool.

Keywords: Visual queries, ad-hoc queries, data analysis, RDF data endpoints

1 Introduction

The ontology-based data access (OBDA) paradigm (cf. [7]) opens a new perspective of data access on the basis of high-level domain ontology, rather than technical database schema structure. An end-user notation for query formulation is important to enable direct access to data by its end users that may not be IT professionals [11], it can be aimed towards easing the professionals' work with queries, as well.

The visual/diagrammatic notations, along with approaches based on keyword search, forms (e.g. [12]) and natural language (e.g. [5]), are a major paradigm considered for end-user accessing of data organized in the form of RDF [9] data model, natively accessible by the textual SPARQL [10] query language.

* Supported, in part, by Latvian State Research program NexIT project No.1 "Technologies of ontologies, semantic web and security".

Most of the existing diagrammatic notations for RDF data access, including Optique VQs [11], Query VOWL [6], or early versions of ViziQuer [13], although efficient for visual formulation of wide range of queries, still do not include queries with data aggregation and statistics options, so important and central e.g. for business intelligence area (cf. [1],[8]). The work on visual queries for OBDA in [11] explicitly limits the visual query notation and end user involvement to non-aggregated queries with simple data attribute selection, justifying these to be most important in their usage context.

The notations of [11] and [13] rely on UML class diagram constructs for query construction; they use nodes for data instance class specification, edges for data instance links and attributes for both the selection attributes and their links to the node data instance; there are also conditions for additional filters on data instance attribute values. A similar query paradigm is also used in query systems for major relational databases.

We propose here to extend the basic UML-style query formulation paradigm with means for visual formulation aggregated and nested queries, so to enable the users preferring the visual query formulation style, to work also with this kind of queries.

The previous work by authors on aggregate queries within the UML-style ViziQuer tool [3, 4] introduces aggregate queries simply by including aggregate fields within the query node field lists. This suffices for simply structured queries that e.g. select data from a number of linked classes, while leaving open the issue of more complex queries, available e.g. in SPARQL. Here we make a more radical extension to the basic UML-style query formulation notation, involving the following principal novel points:

(i) Fully visual notation for subqueries;
(ii) Separate aggregate and grouping field lists in query nodes;
(iii) Unit and union nodes for query structuring;
(iv) Integration of textual SPARQL fragments into the visual notation.

We report also on a pilot user study testing the understandability of simple constructs from the introduced notation; the study results show the understanding ability of the subquery notion also by domain experts that are not IT professionals.

The visual query notation proposed here is supported by a web-based diagrammatic (multi-modal) query tool, available at viziquer.lumii.lv.

In what follows, Section 2 reviews the revised basic visual notation. Section 3, the central one of the paper, presents the extended query constructs. Section 4 discusses the SPARQL construct coverage; Section 5 describes the user study for notation evaluation. Tool environment is outlined in Section 6. Section 7 concludes the paper.

2 Basic Visual Notation

The visual/diagrammatic query definition is based on data model containing the vocabulary of entities, each identified by a local name and optional name prefix and providing the full entity URI, and the schema information stating the applicability, ordering and cardinalities of properties in the context of the model classes. We shall consider here example queries over a simple mini-hospital data schema, developed originally in [2] to describe a fragment of a realistic hospital information system and presented here in Fig. 1. The names of properties connecting the classes, if not specified,

coincide with the target class name with lowercase first letter[1]. There is default maximum and minimum default cardinality 1 assumption for properties, as well.

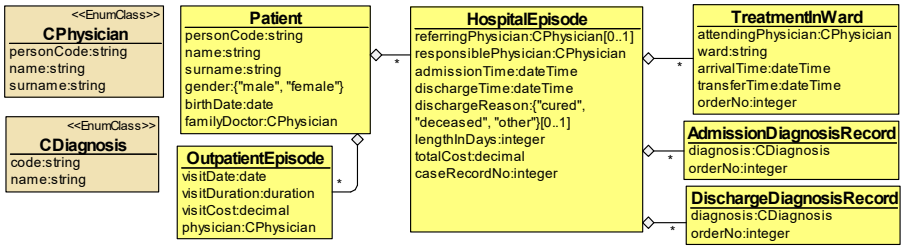

Figure 1. Hospital domain ontology fragment

The ontology is organized around the patient class, with possibly both hospital and outpatient episodes for each patient. A hospital episode can run over several treatments in hospital wards, each identified by an order number. A hospital episode has records of admission and discharge diagnoses, pointing to the *CDiagnosis* classifier. Hospital and outpatient episodes and treatments in wards may have associated physicians in different roles, as well.

A basic visual query (cf. [13,4]) is a UML class diagram style graph with nodes describing data instances, the edges describing their connections and the attributes forming the query selection list from the node instance attributes and their expressions; every node can specify both the instance class and additional conditions on the instance. One of the graph nodes is the main query node (shown as orange round rectangle in the diagram); the structural edges (all edges except the condition ones, as described below) within the graph form its spanning tree with the main query node being its root.

Figure 2 shows an example visual query and its translation into SPARQL. The query is based on three related classes, a mandatory link (*HospitalEpisode -> Patient*) and an optional link to *CPhysician*. The attributes are by default assumed to be optional, not to bypass entire solution rows because of missing attribute values (this contrasts [13,4]).

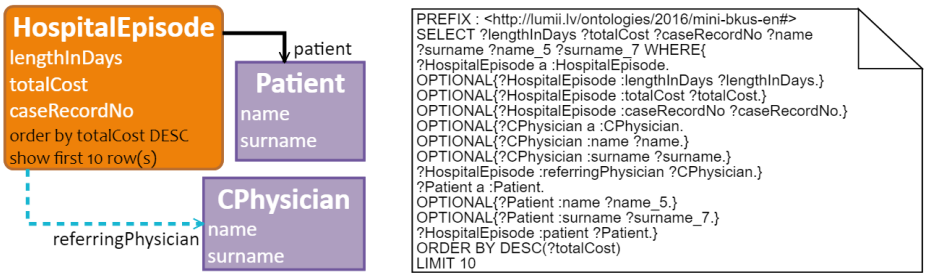

Figure 2. Select top 10 most expensive hospital episodes, show episode length in days, total cost and case record number, together with patient name and surname, as well as name and surname of referring physician, if specified.

[1] The data schema is presented in UML class diagram style with an attribute or outgoing association role ascribed to a class denoting the availability of the property within the context of the class, without domain assertion claims typically used in ontology visualizations.

The required attribute specification (corresponding to the class instance, attribute property and attribute value triple in RDF/SPARQL sense) can be achieved by a single check box click for 'Required Values' in the query tool user interface, and is marked by the '{+}' mark besides the attribute (expression) specification, as in Fig. 3 (a).

Figure 3 (b) illustrates the notation for conditions and the '*'- notation for selecting all attributes defined in the data model for the query class (the optional attribute semantics is essential here not to have counter-intuitive results). Fig. 3 (c) shows a negated link to a condition class and the option to hide the link name from the query presentation, if it is unanimously clear from the class name specifications in the link start and end nodes (the hiding could have been applied also to the *patient* link in Fig. 2).

Figure 3. Further basic query notation illustrations

3 Extended Query Structuring Facilities

In this section we explain advanced query structuring facilities that go beyond the simple class – attribute – link – condition paradigm for query specification.

3.1 Aggregate and Grouping Field Separation

Figure 4. (a) count the hospital episodes lasting for at least 10 days (a single-number query), (b) count the treatment in ward cases for each ward (a simple statistics query) (c) count the hospital episodes, grouped by discharge reason and patient's gender.

Figure 4 shows examples of aggregated query definition in the visual notation. An aggregated field in the query is specified in a compartment above the query node class name. This explicitly separates the aggregation fields from the instance-level fields in the usual node field compartment below the node class name that are acting as the grouping fields within the aggregate query. The separation offers means of visual capturing of the fundamentally different roles that the aggregation and grouping fields play within an aggregate query. This separation is especially important for enabling smooth aggregate inclusion in queries with further advanced subquery and control structures.

The aggregation is computed over the raw data set returned by the query obtained from the original one by replacing the aggregate fields by their respective aggregate function arguments. We restrict the aggregate field specifications to the main query node and the head nodes of subqueries (cf. Section 3.2) only.

Figure 5 shows an example of simultaneous multiple aggregate specification.

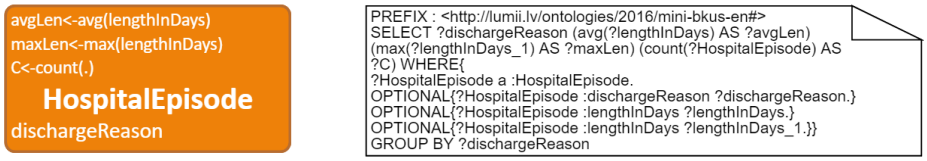

Figure 5. A query with multiple aggregations

The multiple aggregation possibility shall be exploited with care not to induce the query raw result set blow-up due to additional dimensions introduced by possibly multiple-valued various attributes and expressions that are to be aggregated (the additional data dimensions can be introduced by the "usual" field values, as well).

The *count_distinct(.)* aggregation option allows counting only the distinct instances from the raw result set. The subquery and control node constructs, explained in the following sections, can be used to create queries without multiplicative effects, as well.

3.2 Explicit Subquery Notation

The SPARQL 1.1 notation allows for subquery specification. In combination with aggregation, the subquery construct allows *inter alia* selecting data instances together with their aggregate characteristics within the data environment (e.g. the count of related instances). There is no common similar construct in UML-style query notations, neither within the semantic technology domain (cf. [11], [13], [4]), nor for the relational database visual query builders.

Since the primary target of the subquery notation is to describe subqueries over instances related to a certain main query node instance, it is natural to associate the main query to subquery relation with an edge in the query. Both plain (non-subquery) and subquery edges typically correspond to a link in the data model; the options for query links not matching exactly the data links are described in Section 3.3.

The subquery links in the visual notation are starting by black bullets. The example in Fig. 6 example (left) contains a subquery link based on the single *treatmentInWard* link between *HospitalEpisode* and *TreatmentInWard* classes in the data model.

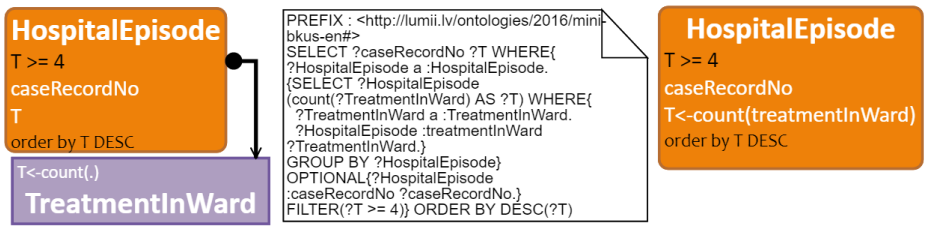

Figure 6. Select all hospital episodes with at least 4 treatment wards, show episode case record number and number of treatment wards; order descending by the treatment ward count

The scope of the subquery involves the subquery edge itself and the part of the query graph below it in the graph's tree shape structure. The selection list of the subquery contains the explicitly specified query outputs, as well as all references to query nodes and fields outside the subquery scope (typically including the instance of the subquery

hosting node). The subquery results are projected into the hosting query; they can be used in conditions, further field definitions and ordering lists in the same way, as instance model attributes. To include a subquery output in the selection list for the main query, it has to be explicitly included in the subquery hosting node field list.

The Fig. 6 (right) single-node query shows the option to define a single-number subquery over property-linked instances in a textual form; in this case the value *T* is by default included in the main query selection list, as well as it is computed also for episodes without any treatments in wards, should there be ones.

Figure 7 shows a simple example of nested subqueries.

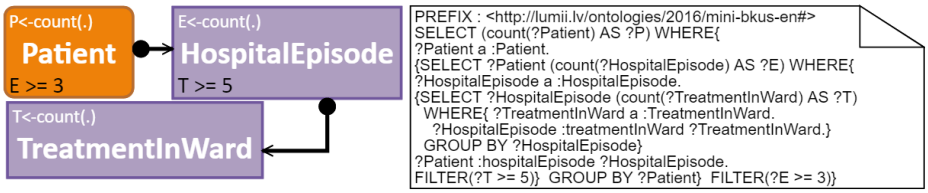

Figure 7. Count patients with at least 3 hospital episodes, running over at least 5 wards each

The subquery mechanism is not bound to just local aggregate computation. In the case, if a subquery does not return any result except the host node instance for which it has been created, it works as an existential filter, as in Fig. 8, left. The right single-node query models the same behavior, using explicit predicate *exists* and property paths.

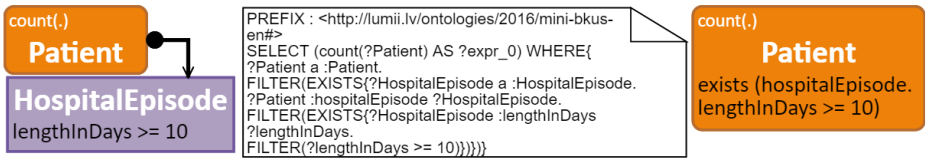

Figure 8. Subquery as a graphical existential filter

It has been an explicit design decision to stay with simple join semantics in the case of queries with nodes linked by plain required or optional edges. Should the link in Fig. 8 not have been marked as a subquery, the raw data set consisting of patients and all their hospital episodes would have been created leading to counting each patient as many times as it has the hospital episodes lasting for at least 10 days. In the case of counting the instances one could use *count_distinct(.)*, however using the subquery mechanism leaves a clean un-duplicated result set of subquery host node instances, ready for output, as well as participation in further aggregate operations.

The count of patients having a hospital episode and not having an outpatient episode can be specified by either of query diagrams in Fig. 9 since the non-existence of a related instance would not cause any data graph blow-up.

Further subquery notation usage possibilities are discussed in Section 3.3.

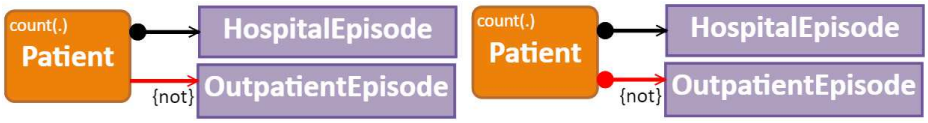

Figure 9. Condition of existence and non-existence of a link

3.3 Extensions to Model Tree Shape

The visual notation discussed so far is suitable for visual query specification, if the query has a tree form that is matching a data model fragment. The query structure can be extended either by ***visual condition links*** (visualized by a thinner line with diamonds on both ends, cf. Fig. 10a) that are added on top of the query tree structure, or by explicit (other) ***node references*** in the node attribute expressions (e.g. H and P in Fig. 10b). The non-model links (marked by the label ++), allow to have structural query links that do not correspond to links in the data model.

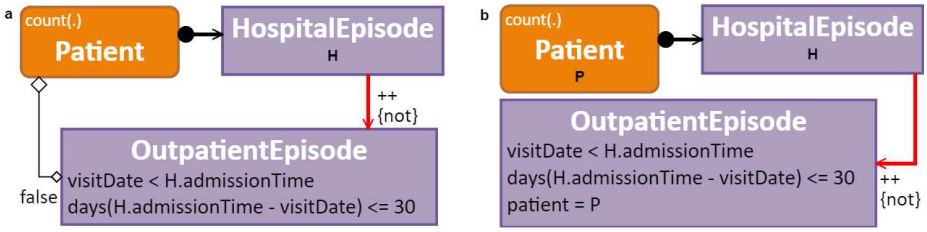

Figure 10. Count patients having at least three hospital episodes without a matching outpatient episode for the same patient within the 30 day range before the hospital episode.

In Fig. 10 the non-existence of an outpatient episode is considered not for a patient but for a hospital episode. Therefore, the query structure link is drawn from the *HospitalEpisode* to *OutpatientEpisode* (in the example it happens to be a negated link) while the necessary model link between the *Patient* and *OutpatientEpisode* is encoded either as a condition link (a), or as an explicit condition within the *OutpatientEpisode* node (b).

As a more radical structure extension, the ***control nodes***: ***unit*** (denoted by *[ ]*) and ***union*** (denoted by *[+]*), not describing any data instance, can be introduced to the queries. The unit node can typically be used as an outer query structuring layer, able to collect the results from a single or multiple subqueries, combine, project, filter them, as well as apply distinct or aggregation clauses over them. The example queries in Fig. 11 use the statistics by attribute notation from Section 3.1 within a subquery, embraced within an outer main query consisting of a union node.

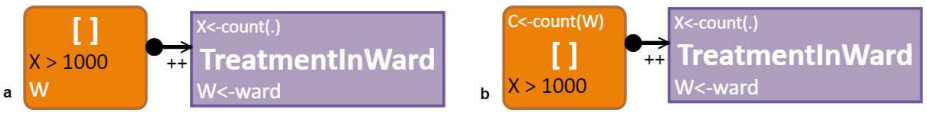

Figure 11. List (a) and count (b) all wards (attribute *ward* values of *TreatmentInWard* class instances) with more than 1000 treatment cases.

Figure 11a shows the possibility of modeling the SPARQL *having* clause over the aggregated result set. The technique of unit nodes offered here, however, allows for much richer aggregate result handling than just filtering, as can be seen e.g. in Fig. 11b.

The ***union node*** introduces a disjunction of its sub-trees into the query (the link to the subtree from the union node is perceived as the link from the parent of the union node), as illustrated in Fig. 12. There are options to use subquery links both above and below the union node, as well.

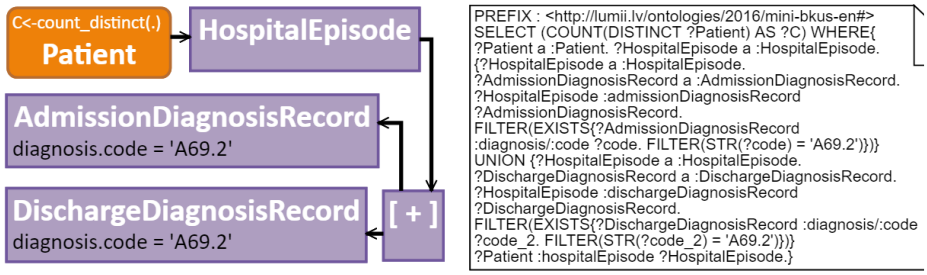

Figure 12. Query with union node: Count all (distinct) patients that are related to diagnosis 'A69.2' (Lyme disease) either as admission or as discharge diagnosis of a hospital episode.

The idea of the subquery concept, as explained in Section 3.2, is to offer means of computing local characteristics of a data node within a query. Since the semantics of the query language constructs is defined on the basis of their translation into SPARQL, the local characteristics computing subqueries cannot be allowed to contain the slicing limit and offset modifiers (in SPARQL the subqueries are computed globally). For the situations where the ordering and slicing modifiers are important within a subquery the ***global subquery notation*** (a white bullet at the edge start) has been introduced.

As an example, consider the query in Fig 13. The ***'==' notation*** marks the query link to connect a resource (or a data value) to itself. It is necessary to take 5 most expensive hospital episodes before their joining to the treatment in ward data.

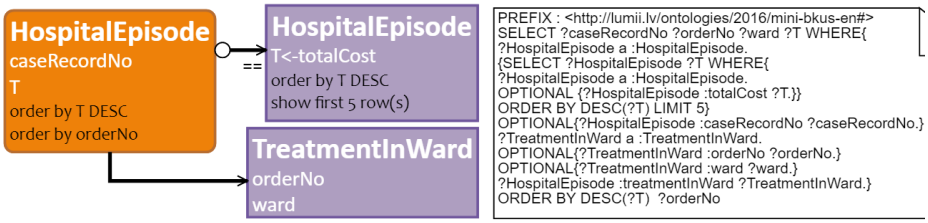

Figure 13. Limit-bound "global" subquery example: Select top 5 most expensive hospital episodes (show the episode case record number and total cost), list them together with all their treatment wards (show the order number and the ward).

As far as the translation into SPARQL semantics is considered, there is no difference between local and global subqueries (except that slicing is not allowed in local subqueries). The main difference between local and global subqueries is in the mindset of reading the queries (local vs. global context). Should other visual query language implementations become available (e.g. by translating the visual queries directly into SQL), the slicing could be introduced also for local subqueries, with a principally different semantics of computing the slice (e.g. the topmost or the top n records) from the subquery for each data instance tuple forming the subquery context.

3.4 Exploratory Queries and Textual SPARQL Fragments

The ViziQuer/web notation allows also for explicit query variable usage in the place of class and property names. Using an explicit query variable in place of a class or

property name creates the SPARQL query variable with the same name, uses it within the query just as the class or property name, then adds it to the query output. Fig. 14 shows examples of exploratory queries over a SPARQL endpoint in the visual notation, re-worked and extended from [4]: (a) select all classes together with their instance count, (b) select all properties together with all class pairs connected by them (select distinct specification is necessary since the raw data graph is computed on data instance level); (c,d) two forms of selecting all triples from the data set (in (d) 'a' denotes the "current instance", p – the property, 'b' – the "target instance"; {+} marks a required triple presence); (e,f) select all properties together with their usage count (in (f) the triple has to be marked as required by {+} as well as by {internal} not to have its resulting value included in the selection set); (g) if an explicit query variable is introduced within a subquery, a reference to it within an enclosing query is to be made by the variable name without the preceding variable mark '?'.

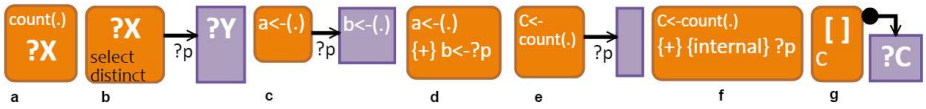

Figure 14. Exploratory query examples.

The visual notation has not been designed specifically to ease the formulation of the exploratory queries, Fig. 14 just shows the possibility and options for exploratory queries in the visual notation, outlining also the usage variability of the notation constructs (e.g. using query nodes with empty class names or with no content at all).

For the situations where the visual notation constructs either do not support the query definition, or are not convenient for it, there is an option (clearly meant for expert users) to introduce explicit textual SPARQL fragments into the visual symbols. Both the syntax of SPARQL group graph patterns and SPARQL select queries are supported. Fig. 15 shows two options of a simple example of selecting all data set triples. The variables selected out of the direct SPARQL clause can in the enclosing query be referred to by their name (without the preceding variable mark '?').

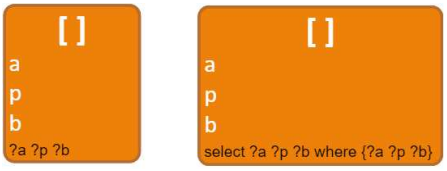

Figure 15. Direct SPARQL examples.

4 Expressivity and Limitations

Apart from the possibility to include direct SPARQL graph patterns and full SPARQL select sub-queries in the query node compartments, there are explicit counterparts for most of SPARQL query constructs in the visual query notation itself.

So, a basic graph pattern can be modeled by a property-labelled edge between two nodes (the class name specification in a visual query node is optional). A group graph

pattern can be a set of graph pattern fragments, interconnected by free edges; a variable from several patterns can go into a single query node. The filters correspond to conditions at query nodes; the variable binding and selection – to field lists at query nodes.

The projection can be modeled by adding a new unit node to the query and connecting the original query to it as a subquery; the unit node provides the context of selecting the fields corresponding to the projection variables; further operations on initial query results in the unit node are possible, as well. If the initial query is aggregated one, a filter in the unit head node can model SPARQL *having* clause of the initial query.

The optional blocks correspond to optional edges, existence filters to subqueries with empty result sets, non-existence filters to negated edges. There are also direct counterparts in the visual notation to aggregation, subqueries, *select distinct* and *union* constructs. Although not described here, the SPARQL *minus* construct can be handled by negated global subqueries.

The main limitations of the current visual notation are the requirement of queries to be placed over single graph, non-covered advanced property path expressions, select * (in SPARQL sense) and *reduced*. None of these constructs is of principal difficulty for the visual notation with SPARQL-based implementation, however, they are not so clear for eventual other implementations, e.g. by translating queries directly into SQL.

5 Evaluation: Visual Queries over Hospital Data

The hospital data schema shown in Figure 1 corresponds to a fragment of data structure of Children's Hospital in Riga, Latvia. The resource point viziquer.lumii.lv/med illustrates a number of queries that have been important during the analysis of the real hospital data. An expert assessment of the query creation process allows to judge that it should be possible for an IT-trained expert, who has learned the notation, to successfully create the queries.

An early pilot study on visual notation understandability was performed on a group of Master degree medical students at University of Latvia and their teacher, together 7 participants, neither of whom have a specific IT training.

The participants were given an ontology (a fragment of Fig. 1 ontology, with vocabulary expressed in Latvian), and a simple data instance graph with two patients, altogether with 2 hospital episodes and 1 outpatient episode, and related diagnoses. The participants were asked to interpret 10 simple queries, shown in Fig. 16, over the given data set. The introductory time for notation presentation was 40 minutes, followed by 40 minutes query answering time. The results are collected in Fig. 17, where each row corresponds to replies by one participant (Row 1 are the teacher's replies).

The results indicate that 6 of 7 participants were able to interpret at least 70% of the queries, with similar results both for non-aggregated and aggregated queries. The subquery notation used in examples (6), (7), (8) can be observed as not causing a particular difficulty. For the queries (4) and (5) the difficulty was to interpret a condition placed outside the main query class. The unexpected difficulty with query (9) can indicate the need for appropriate examples in query notation presentation since an analogous statistics query (10) presented over several data nodes, was answered quite well.

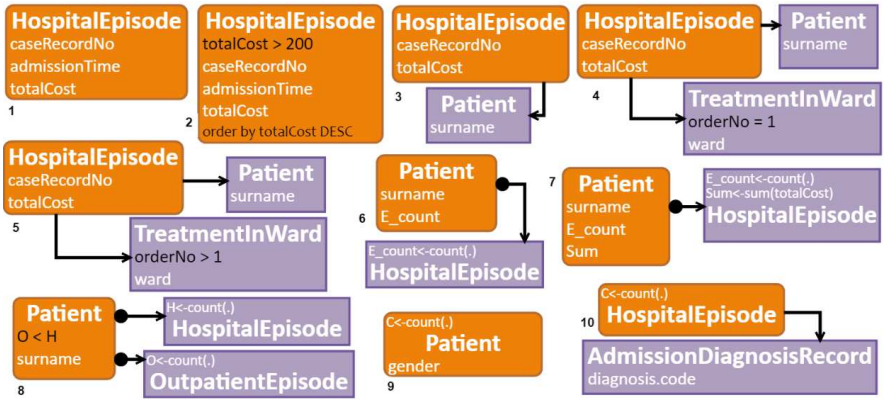

Figure 16. User study queries.

	1	2	3	4	5	6	7	8	9	10
1	++	++	++	++	/	++	++	++	++	++
2	++	++	++				-		-	-
3	++	++	++	/		++	++	++	-	++
4	++	++	++	/		++	++	++	-	++
5	++	++	++	++	++	++	++	++	-	++
6	++	++	+	++	++	++	++	+	-	++
7	++	++	++	++	/	++	++	+		++

++: correct interpretation
+: almost correct
/: part of constructs understood correctly (difficulty with conditions outside the main class)
-: incorrect interpretation
empty cell: no answer

Figure 17. Users' response evaluation

6 Query Tool Environment

A web-based prototype tool supporting the visual notation presented in this paper is available from http://viziquer.lumii.lv/. After the registration and logging into the environment the user has an option to create and manage projects. Each project has to be configured by uploading the data schema (used e.g. for SPARQL generation from visual representations of entity local names, as well as for code completion) and setting up project parameters (e.g. connection to the SPARQL endpoint for direct execution of generated SPARQL queries). The containers for queries within a project are diagrams; each diagram can typically contain multiple visual queries; the user each time makes a visual selection of the query to be executed within the query diagram.

The tool architecture allows both for a centralized server with each user uploading a custom data schema, as well as configurations that are dedicated for work with specific SPARQL endpoints. We plan to release the tool code as open source.

7 Conclusions

The presented notation and examples demonstrate the possibility of extending the UML-style diagrammatic query creation paradigm of classes, associations, attributes and conditions by simple visual means for specification of advanced query building constructs, including subqueries, aggregations and additional query control structures.

The performed pilot user study has shown that it is possible to introduce the visual notation, including the aggregation and subquery constructs to domain experts without specific IT training. The visual format of the query presentation in the style of UML class diagrams, together with the notation expressivity, may allow the notation to have a potential to ease the query formulation work of a semantic technology and/or database professional, as well.

The visual notation can be extended, for instance, by a practically important construct of sliced local subqueries, not supported directly in SPARQL 1.1, if alternative query language implementations to translation into SPARQL are considered.

We expect that the availability of a web-based visual query environment infrastructure would initiate also use cases of the technology outside the group of its developers.

References

1. Adamson, C.: Mastering data warehouse aggregates: solutions for star schema performance. John Wiley & Sons, 2012.
2. Barzdins, J., Grasmanis, M., Rencis, E., Sostaks, A., Barzdins, J.: Ad-Hoc Querying of Semistar Data Ontologies Using Controlled Natural Language. // In: Frontiers of AI and Applications, Vol. 291, Databases and Information Systems IX, IOS Press, pp. 3-16, 2016, http://ebooks.iospress.com/volumearticle/45695
3. Čerāns, K., Ovčiņņikova, J., Zviedris, M.: SPARQL Aggregate Queries Made Easy with Diagrammatic Query Language ViziQuer. In: Proceedings of the ISWC 2015 Posters & Demonstrations Track, CEUR Vol. 1486, (2015), http://ceur-ws.org/Vol-1486/paper_68.pdf
4. Čerāns, K., Ovčiņņikova, J.: ViziQuer: Notation and Tool for Data Analysis SPARQL Queries. In Proc. of the Second International Workshop on Visualization and Interaction for Ontologies and Linked Data (VOILA '16), Kobe, Japan. CEUR Workshop Proceedings, vol. 1704, CEUR-WS.org, 2016, pp.151-159.
5. Ferré, S.: SPARKLIS: a SPARQL Endpoint Explorer for Expressive Question Answering. In: In: Proceedings of the ISWC 2014 Posters & Demonstrations Track, CEUR Vol. 1272, (2014), http://ceur-ws.org/Vol-1272/paper_39.pdf
6. Haag, F., Lohmann, S., Siek, S., Ertl, T.: QueryVOWL: Visual Composition of SPARQL Queries. In: The Semantic Web: ESWC 2015 Satellite Events. LNCS, Vol.9341, pp. 62-66. Springer, (2015), http://vowl.visualdataweb.org/queryvowl/
7. Optique. Scalable End-User Access to Big Data, http://optique-project.eu
8. Ponniah, P.: Data warehousing fundamentals for IT professionals. John Wiley & Sons, 2011.
9. Resource Description Framework (RDF), http://www.w3.org/RDF/
10. SPARQL 1.1 Query Language. W3C Recommendation 21 March 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/
11. Soylu, A., Giese, M., Jimenez-Ruiz, E., Vega-Gorgojo, G.., Horrocks, I.: Experiencing OptiqueVQS: A Multi-paradigm and Ontology-based Visual Query System for End Users. Universal Access in the Information Society, March 2016, Volume 15, Issue 1, pp 129–152.
12. Vega-Gorgojo, G., Giese, M., Heggestøyl, S., Soylu, A., Waaler, A. (2016). PepeSearch: Semantic Data for the Masses. PLoS ONE 11(3): e0151573. https://doi.org/10.1371/journal.pone.0151573.
13. Zviedris, M., Barzdins, G.: ViziQuer: A Tool to Explore and Query SPARQL Endpoints. In: The Semantic Web: Research and Applications, LNCS, Volume 6644, pp. 441-445, (2011)