ViziQuer: A Visual Notation for RDF Data Analysis Queries

Kārlis Čerāns[1,2(✉)], Agris Šostaks[1,2], Uldis Bojārs[1,2], Juris Bārzdiņš[3], Jūlija Ovčiņņikova[1,2], Lelde Lāce[1,2], Mikus Grasmanis[1], and Artūrs Sproģis[1]

[1] Institute of Mathematics and Computer Science, University of Latvia, Riga, Latvia
karlis.cerans@lumii.lv
[2] Department of Computing, University of Latvia, Riga, Latvia
[3] Department of Medicine, University of Latvia, Riga, Latvia

Abstract. Visual SPARQL query notations aim at easing the RDF data querying task. At the current state of the art there is still no generally accepted visual graph-based notation suitable to describe RDF data analysis queries that involve aggregation and subqueries. In this paper we present a visual diagram-centered notation for SPARQL select query formulation, capable to handle aggregate/statistics queries and hierarchic queries with subquery structure. The notation is supported by a web-based prototype tool. We present the notation examples, describe its syntax and semantics and describe studies with possible end users, involving both IT and medicine students.

Keywords: Visual notation · Diagrammatic queries · RDF data · SPARQL · Ad-hoc queries · Data analysis queries

1 Introduction

SPARQL, as defined by a W3C standard [1], is the main query language over data structured in accordance to the RDF [2] data model. This includes most of the Semantic Web data, as well as data brought into the semantic-web formats by various mapping approaches, as ontology-based data access (OBDA), cf. [3]. Although the semantic RDF/SPARQL technologies offer a higher-level view on data than the classical relational databases with the SQL query language, the formal textual notation of SPARQL queries still complicates its usage by domain experts and IT professionals alike.

A number of approaches exist to ease the SPARQL query formulation. These include form-based interfaces as e.g. PepeSearch [4] and WYSIWYQ [5]. SPARKLIS [6] offers faceted SPARQL query composition from natural language based snippets. The visual/diagrammatic formalisms for SPARQL query creation apply/extend the visual querying principles studied extensively for relational databases (cf. e.g. [7, 8]). Most of the existing visual/diagrammatic SPARQL query builders, as SEWASIE [9], Optique VQs [10], QueryVOWL [11], LinDA [12] or early versions of ViziQuer [13], although efficient for visual formulation of substantial range of queries, however, are not designed to support queries with data aggregation available in SPARQL 1.1. [1].

E. Garoufallou et al. (Eds.): MTSR 2018, CCIS 846, pp. 50–62, 2019.
https://doi.org/10.1007/978-3-030-14401-2_5

The UML class diagram style visual specification of SPARQL select queries with aggregation has been introduced in [14, 15] and re-formulated in a way to allow for subquery specification in [16]. In this paper we present for the first time an overall ViziQuer abstract syntax and semantics (by means of visual query translation into SPARQL), as well as describe a comparative user study showing an advantage of query over RDF data composition in ViziQuer vs. query composition in textual SPARQL notation. Should a well-established tool support be available, we would aim the ViziQuer visual notation to be useful for query over RDF data creation, presentation and sharing both for persons without specific IT training, as well. Currently there is a web-based prototype implementation available for the visual notation [17].

Onwards the Sect. 2 introduces the visual notation, Sect. 3 describes the query model and Sect. 4 outlines its semantics. Section 5 describes the user study and Sect. 6 concludes the paper. The site http://viziquer.lumii.lv/mtsr2018 contains supplementary materials for the paper, including the materials for user study reproduction.

2 Notation Examples

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 example queries over a simple mini-hospital data schema, shown in Fig. 1 and adapted from [18], where it has been presented as a fragment of a realistic hospital information system. The names of properties connecting the classes, if not specified, coincide with the target class name with lowercase first letter. There is default minimum and maximum cardinality 1 assumption for properties.

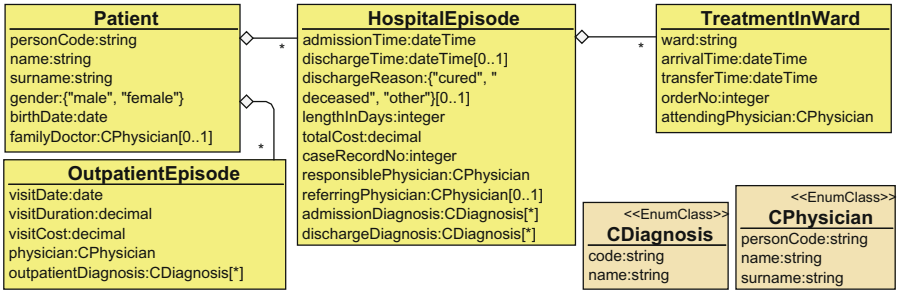
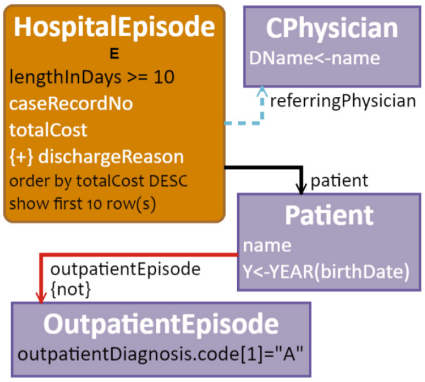


Fig. 1. Example hospital domain ontology fragment

2.1 Basic Visual Queries

A basic visual query (cf. [13, 16]) is a UML class diagram style graph with nodes describing data instances, the edges describing their connections and the fields forming the query selection list from the node instance model 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 concrete syntax); the structural edges (all edges except the condition ones, cf. Sect. 2.4) within the graph form its spanning tree with the main query node being its root.




```
PREFIX : <http://lumii.lv/ontologies/2016/mini-bkus-en#>
SELECT ?caseRecordNo ?totalCost ?dischargeReason ?name
?Y ?DName WHERE{ ?E a :HospitalEpisode.
OPTIONAL{?E :caseRecordNo ?caseRecordNo.}
OPTIONAL{?E :totalCost ?totalCost.}
?E :dischargeReason ?dischargeReason.
?Patient a :Patient. OPTIONAL{?Patient :name ?name.}
OPTIONAL{?Patient a :Patient. ?Patient :birthDate ?birthDate_1.
BIND(YEAR(?birthDate_1) AS ?Y)}
FILTER NOT EXISTS{?OutpatientEpisode a :OutpatientEpisode.
FILTER(EXISTS{?OutpatientEpisode  :outpatientDiagnosis/:code
?code. FILTER(SUBSTR(?code, 1, 1) = "A")})
?Patient :outpatientEpisode ?OutpatientEpisode.}
OPTIONAL{?CPhysician a :CPhysician.
OPTIONAL{?CPhysician :name ?DName.}
?E :referringPhysician ?CPhysician.}
?E :patient ?Patient.
FILTER(EXISTS{?E :lengthInDays ?lengthInDays.
FILTER(?lengthInDays >= 10)})}
ORDER BY DESC(?totalCost) LIMIT 10
```

Fig. 2. An example basic visual query and its translation into SPARQL

Figure 1 shows an example basic visual query: *find 10 most expensive hospital episodes among those lasting at least for 10 days, having a discharge reason specified and having a patient that does not have any outpatient episode with an infectious disease diagnosis; list episode case record number, total cost and discharge reason, the patient name and birth year, and the name of the referring physician, if specified.*

The basic visual query links are labelled by properties (or sequences thereof) from the data model. The query in Fig. 2 illustrates the links that are required (*patient*), optional (*referringPhysician*) and negated (*outpatientEpisode*).

Each query node contains an ordered list of (instance) fields denoting the properties of instances corresponding to the node that are to be included in the query output; each field has a data expression (in the simplest case just an instance model attribute name) and an optional alias (e.g. *E, Y* and *DName* in Fig. 2 query). Additionally, conditions over field and other instance attribute values can be placed in the query nodes.

The presence of a node field value in the query output is optional to not bypass entire solution rows because of some missing attribute values. The *{+}* mark is used to mark a field as required (cf. *{+} dischargeReason* in Fig. 2).

The *YEAR()* function calculates the birth year from the patient's birth date that is available in the data model. The operations and functions used to construct expressions in SPARQL [1] are allowed also in ViziQuer; further operation shortcuts, such as *x[1]* standing for the first symbol in *x* are available to ease the query definition.

The *outpatientDiagnosis.code* notation in the example illustrates property chaining that is allowed both in field value expressions and in conditions.

2.2 Aggregated and Grouping Attributes

Figure 3 shows three example queries specifying the aggregated attribute computation:

(a) *Count the hospital episodes lasting for at least 10 days;*
(b) *Count the treatment cases for each ward, and*
(c) *Count the hospital instances and find their average length in days, grouped by the patient's gender and patient's age at admission time*

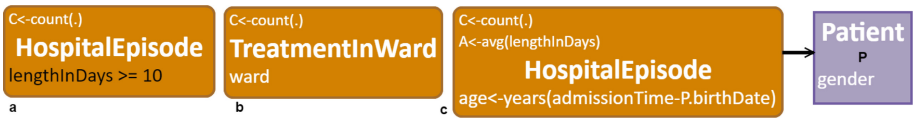


Fig. 3. Aggregate query examples

The principal design idea for aggregate attribute inclusion in the query, introduced in [13], is to place them in a special compartment, situated above the node class name. Should there be instance-level attributes in an aggregated query, as e.g. in Fig. 3(b) and (c) examples, these are to be regarded as grouping attributes for the aggregations.

The function *years* in Fig. 3(c) is a custom ViziQuer notation for expressing the *date* or *datetime* value difference in years (similar functions for months, days, hours, minutes and seconds are available, as well). Figure 3(c) uses also the concrete syntax option to hide the default label for links connecting the data model classes (the link name remains still present in the query abstract syntax discussed onwards in Sect. 3).

2.3 Visual Subquery Notation

The ability to create subqueries is important both for SQL queries over relational databases, and for SPARQL queries over RDF databases. Still, there is no generally accepted visual notation for definition of data queries that involve subqueries. Our proposal [13] for including subqueries in the visual query notation consists in letting certain edges in the query structure tree to be marked as subquery ones, so considering the edge together with the part of the query tree behind it as a subquery. The visual notation for the subquery edge is proposed to be a black bullet at the hosting (non-subquery) end of the edge. Figure 4 shows example queries that can be phrased, as follows:

(a) *Select all hospital episodes with at least 4 treatments in wards, show the episode case record number, treatment in ward count and list of admission diagnosis codes, order descending by treatment in ward count;*
(b) *For every physician responsible for at least one hospital episode, select the surname, as well as count and average treatment in ward count for episodes with this responsible physician.*

Intuitively, each subquery is computed in the context of a single hosting node class instance (e.g. a hospital episode, in the Fig. 4(a)). The subquery link together with its

reference to the hosting node instance is considered to be a part of the subquery. The subquery results (the selection variables, as well as the references to the hosting query nodes) are projected into the hosting query, where they can be handled in a similar way as the hosting node attributes themselves (e.g. included in filters, computations, order lists and further aggregates). The subqueries can be nested, as shown in Fig. 4(b).

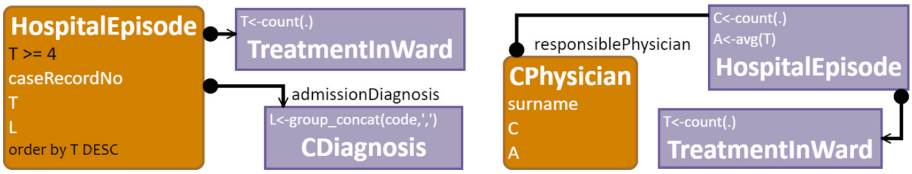


Fig. 4. Visual subquery examples

In the case, if a subquery does not return any result except its host node instance, it works as an existence filter, as in Fig. 5(a). In Fig. 5(c) a single-node query models the same behavior, using explicit predicate *exists* and property paths. These examples should be contrasted with a simple join query in Fig. 5(b), where the count of patients is taken over joined patient and hospital episode records.

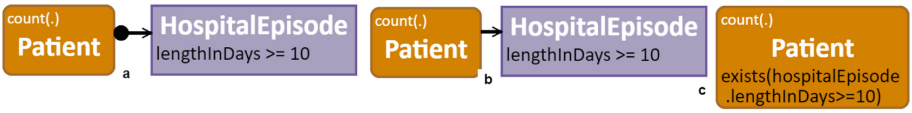


Fig. 5. Subquery as an existential quantifier

2.4 Query Structure Extensions

The visual notation considered so far is suitable for visual query specification, if the query has a tree form that is matching a data model fragment. The following more advanced notations (cf. [16]) raise the query language expressive power beyond the query tree shape and model structure matching.

Figure 6 shows two visual options for the query *"Count patients with at least 3 hospital episodes without a matching outpatient episode within the 30 day range before it"*. The query structure requires non-existence of an outpatient episode for a hospital episode, however, there is no direct link in the data model connecting these classes. To build the query structure in this case, a non-model edge (a "free" edge), marked by '++' is used (in the example it happens to be a negation link). The data connections can then be established either by extra condition links, as in Fig. 6(a), or by explicit node references (Fig. 6(b)). The condition links, drawn using a thinner line with white diamond ends, are not structure links; they are added on top of the query tree shape structure.

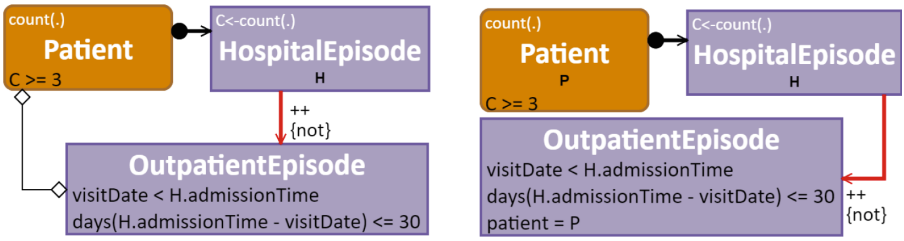


Fig. 6. Condition and non-model links

Further visual query examples, including the ones that use unit node [ ] and union node [+] for query structuring (these nodes do not correspond to data instances in the query) can be found in [13]. The ViziQuer visual notation contains counterparts of most SPARQL 1.1 select query constructs. The currently not covered constructs are named graphs, advanced property path expressions, SELECT * (in SPARQL sense) and *reduced* (cf. [16]).

3 Abstract Query Model

Figure 7 summarizes the abstract syntax of the visual queries in a UML-style model[1] that shows the query structure and is the basis of further query semantics definition.

The hierarchy of nodes (*Node*) and structure edges (*StructureEdge*) describes the query graph *G* spanning tree *T(G)*, rooted at the main query node (an edge is a *structure edge*, if it is not a condition one). Let an edge be *plain*, *condition*, *local subquery* or *global subquery* one by its *edgeType*, and *required*, *optional* or *negated* one by its *relationType*. A structure edge is a *union edge*, if its target is a union node (*UnionNode*), a *sub-union edge*, if its source is a union node, and a *union-free edge* otherwise. Let a structure edge characteristics apply also to the edge target node (so, there are e.g. optional local subquery nodes).

A *query fragment* is a (maximal) set of nodes in *T(G)* connected by plain required union-free edges only, together with structure edges *incoming* into fragment nodes and condition edges *outgoing* from fragment nodes. Let the fragment *head node* be the node that is above all other fragment nodes in *T(G)* and let any fragment head node attributes (e.g. optional, local subquery, union-free) extend to the fragment, as well.

We call a query or its subquery fragment *aggregated*, if it has at least one aggregated field or the distinct option (*distinct* = *true*) within its head node specified.

The *aggregated field* list and *distinctness* specification is allowed in the main query node and the (local and global) subquery nodes only. The ordering, limit and offset specifications are for the main query node and global subquery nodes only.

We shall assume also that there are no name coincidences among the explicit node instance names and field aliases (except for alternative union branches).

[1] The composition notation (a little diamond at an edge end) indicates the query item structure. For all generalization groups in the diagram the superclass is a disjoint union of subclasses.

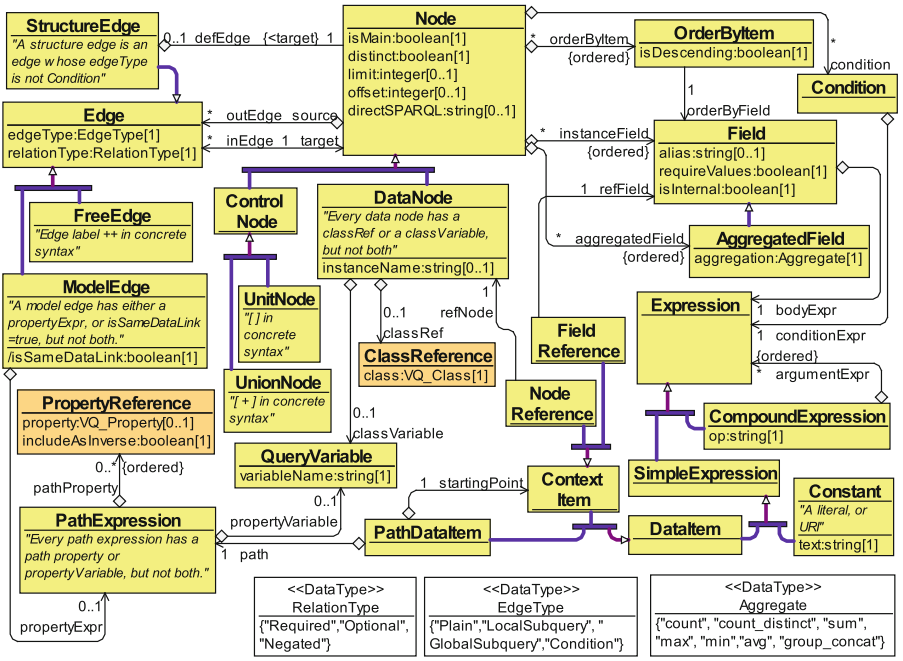


Fig. 7. Core abstract syntax of queries

The ***expressions***, central both for field and condition specification, are defined on the basis of compound and simple expressions, where the operators from SPARQL expression notation [1] can be used to obtain expressions from simple expressions.

A simple expression can be a *constant* or a *data item*. A data item is either a *context item* itself (a *reference* to a node or field defined elsewhere in the query), or, most typically, it would be a *path data item* consisting of a *path* (e.g. an attribute name) starting from a context item (the context item specification is omitted in the concrete syntax in the most typical case, if it coincides with the node containing the expression's container field or condition, allowing it to become just a property name or a path expression).

For fragments X and Y let $X \rightarrow Y$ if Y is a direct child fragment of X (i.e., Y head node is a child of some X node in $T(G)$). For a fragment X we let $UC(X)$, the *upwards context* of X, to consist of all fragments reachable from X upwards in $T(G)$ (including X itself).

We allow the condition edges from a fragment X nodes to go only to nodes in $UC(X)$.

Let the *optional-closure* $o(F)$ of a fragment F be the union of F and all fragments downwards reachable from it by edges that are (i) union-free and plain, and (ii) required or optional. For a fragment F let its *selection set* $Sel(F)$ consist of all non-internal fields, all query variables, as well as all "upwards" references to nodes in $UC(F)$, found within $o(F)$. Let $SubSel(F)$ be the union of $Sel(H)$ for all subquery fragments H hosted in $o(F)$.

The node and field references in the field and condition expressions within a fragment F node n shall refer to only:

- *SubSel(F)*, the results projected out of sub-fragments;
- Nodes from *UC(F)*, the context information available for the fragment (not allowed in aggregate field expressions);
- Nodes and instance fields from *o(F)* (the field references from an instance field f body expression can go only to a n field above f; the node/field references to *o(F)*/ F shall not start a path expression).

The references to *UC(F)* shall allow locating the reference in the data model within F (an equality assertion with a value computed within *o(F)*, or participation in a path data item would be sufficient to allow the reference usage).

4 Query Semantics

We define the query semantics via translation into SPARQL 1.1 [1], done in three steps:

(1) providing the SPARQL query variable names for query model elements;
(2) defining local query model translations into SPARQL, and
(3) computing the SPARQL query inductively over the query fragment structure.

Let *SS* stand for all selection variables in query node direct SPARQL fragments of select query form and *SG* – for all outer scope variables in node direct SPARQL fragments of group graph pattern form.

The SPARQL query variables in a query G shall be ascribed to the following variable points: $VP(G) = DataNode \cup Field \cup DataItem \cup QueryVariable \cup SS \cup SG$. Let the SPARQL query variable name assignment $m:VP(G) \rightarrow Var$ be such that:

(1) for $x \in SS \cup SG$, $m(x) = x$ (i.e. a direct SPARQL variable is mapped onto itself);
(2) for $x \in QueryVariable$, $m(x) = x.variableName$;
(3) for $x \in DataNode$, $m(x) = x.instanceName$, if x has *instanceName* specified;
(4) for $x \in Field$, $m(x) = x.alias$, if x has *alias* specified;
(5) if $x \in DataNode$ is a node/field reference and $y \in DataNode \cup Field$ is the corresponding node/field, then $m(x) = m(y)$;
(6) $m(x) = m(y)$, if $x \in Field$ and $y = x.bodyExpression \in DataItem$ and x does not have *alias* specified (same variable for the field and the data item within it);
(7) $m(x) = m(y)$ if the nodes $x, y \in DataNode$ are connected by a same data edge;
(8) $m(x) \neq m(y)$, if the equality $m(x) = m(y)$ does not follow by the rules (1)–(7).

It is clear that an appropriate mapping m can be generated for every query G. To define $m(x)$ for $x \in PathDataItem$, a rule of thumb is to use the local name of the last property in its path component with appropriate suffix to avoid name clashes.

For a data model reference $d \in ClassReference \cup PropertyReference$ let $t(d)$ be the full IRI of the referred model entity. We extend t also to map property expressions that are sequences of property references and their inverses to SPARQL property paths concatenating the IRIs (and their inverses, as necessary) of the referred properties; let

for brevity $t(p) = m(v)$ for a path expression p and query variable v, if $v = p.propertyVariable$.

Table 1 shows the definition of the "local" SPARQL-fragments $S(x)$ for x a node, an edge, a data item and a field, and filters $FL(c)$ for c a condition.

Table 1. Local SPARQL translations of query model elements

$n \in Node$	$S(x) = BGP(m(n)\ rdf{:}type\ t(n.classRef.class))$, if $n.classRef$ is defined $S(x) = BGP(m(n)\ rdf{:}type\ m(n.classVariable))$, if $n.classVariable$ is defined; otherwise $S(x)$ is empty
$e \in Edge$	$S(e) = BGP(m(e.rsc)\ t(e.propertyExpr)\ m(e.trg))$, if $e.propertyExpr$ is defined, otherwise $S(e)$ is empty
$d \in DataItem$	$S(e) = BGP(m(d.startingPoint)\ t(d.path)\ m(d))$, if d is a path data item, otherwise $S(d)$ is empty
$x \in Expression$	Let $e_1, \ldots, e_n$ be all (possibly none) data items contained in the expression $x = x(e_1, \ldots, e_n)$ Let for x its ***direct translation*** be $T(x) = x(m(e_1), \ldots, m(e_n))$ and ***support pattern*** $Q(x)$ be the join (the concatenation) of all $S(e_i)$
$f \in Field$	Let x be f body expression Let $S_0(f) = Q(x)$, if $x \in DataItem$ and $m(x) = m(f)$, otherwise let $S_0(f)$ be $Q(x)$ extended by $BIND(T(x)\ AS\ m(f))$ Let $S(f)$ be $S_0(f)$, if $f.requireValues$, and $OPTIONAL\ \{S_0(f)\}$ otherwise
$c \in Condition$	Let x be c condition expression If $Q(x)$ is empty (there are no property references within x), let $FL(c) = T(x)$, otherwise let $FL(c) = EXISTS\{Q(x)\ FILTER\ (T(x))\}$

For $v \in VP(G)$ let its container $c(v) \in Node \cup Edge$ be the graph node or edge where v is located. Let for a fragment F the set of F variables $Vars(F)$ and the set of F external variables $Ext(F)$ be defined inductively over the fragment structure, as follows:

- $Vars(F) = \{m(x) \mid x \in VP(G) \wedge c(x) \in F\} \cup \bigcup \{Ext(F') \mid F \rightarrow F'\}$
- $Ext(F) = Vars(F)$, if F is (i) plain and (ii) either optional, union or sub-union fragment; otherwise $Ext(F) = \{m(x) \mid x \in Sel(F)\}$.

The SPARQL group graph pattern $P(F)$, its non-filtered form $P^X(F)$ and external filter $EFL(F)$ for a query fragment F is defined recursively over the query sub-fragment structure, as follows (we use the SPARQL algebra notation, as defined in [1]):

(1) If F is a ***union fragment*** (consisting of a single union node), let $P(F) = Union(P(F_1), \ldots, P(F_n))$ for $F \rightarrow \{F_1, \ldots, F_n\}$; for all other cases use steps (2)–(13).
(2) Consider the ***raw fragment*** F_0 obtained from F by replacing all node aggregate field function calls by their arguments (if F is non-aggregate, then $F_0 = F$).
(3) Join the ***local SPARQL fragments*** $S(x)$ for data nodes, edges and group graph pattern direct SPARQL clauses within F_0 to obtain the ***initial pattern*** $\boldsymbol{P_0}$.
(4) Join to $\boldsymbol{P_0}$ the patterns $P(H)$ of all ***required subquery fragments*** hosted by F_0 nodes, as well as full select direct SPARQL clauses to obtain $\boldsymbol{P_1}$.

(5) Left join (add optional SPARQL subqueries) the patterns $P(H)$ to $\boldsymbol{P_1}$ of all ***optional subquery fragments*** hosted by F_0 nodes, to obtain $\boldsymbol{P_2}$.
(6) Extend $\boldsymbol{P_2}$ with local SPARQL fragments for ***fields*** in F_0, obtaining $\boldsymbol{P_3}$. The extension ordering has to respect the instance field ordering in all F_0/F nodes, as well as the fields not aggregated in F have to come before F aggregated fields. These conditions ensure that a node instance field body expression can refer to an earlier instance field of the same node, as well as that aggregated field body expressions can refer to instance fields within the nodes of the same fragment. The placement of subquery fragments before the fields enable the field body expressions to refer to the results projected out of the subqueries.
(7) Left join to $\boldsymbol{P_3}$ the non-filtered patterns $P^X(H)$ of ***plain optional fragments*** H hosted by F_0 nodes; each fragment $P^X(H)$ is joined, taking into account the corresponding external filter expression $EFL(H)$, denote the result $\boldsymbol{P_4}$.
(8) Subtract (using *Minus* clause) from P_4 the patterns $P(H)$ for all ***negated global subquery fragments*** hosted by F_0, denote the result $\boldsymbol{P^*}$.
(9) Collect the F_0 filter expressions specified in fragment node conditions into $\boldsymbol{FL_0}$.
(10) Add $fn{:}not(exists(P(F_r)))$ to $\boldsymbol{FL_0}$ for all ***negated plain and local subquery fragments*** F_r, hosted by F_0, (the semantics of negated plain and local subquery fragments coincide). Denote the result $\boldsymbol{FL_1}$.
(11) The raw SPARQL pattern corresponding to F is $\boldsymbol{R(F) = Filter(FL_1, P^*)}$.
(12) If F is an aggregated fragment, add over $R(F)$ the aggregation for all aggregate fields in F head node, with all non-aggregated variables in $Ext(F)$ forming the grouping set; denote the result $\boldsymbol{R'(F)}$. For a non-aggregated F let $\boldsymbol{R'(F) = R(F)}$.
(13) Let $R^*(F)$ be obtained from $R'(F)$ by applying the order by, offset and limit operations (the offset and limit operations are allowed only for the main query and for global subquery fragments). Let $\boldsymbol{P(F) = Project(R^*(F), Ext(F))}$.

For a plain optional union-free F let $P^X(F) = P^*$ and $EFL(F) = FL_1$ (since F is non-aggregated, $Ext(F) = Vars(F)$). In all other cases let $P^X(F) = P(F)$ and $EFL(F) = true$.

This completes the visual query semantics description. The algorithm described here has been implemented in the ViziQuer tool, including a few adaptations required to successfully run the queries over concrete vendor-specific SPARQL endpoints (e.g. OpenLink Virtuoso).

5 User Studies

A pilot user study on visual query *readability* by domain experts without IT training has been reported in [16], indicating that most of the participants (6 out of 7) were able to correctly interpret at least 70% of visually presented queries; there has been a similar interpretation success rate both for queries that involve aggregation and those that do not, also observing that the subquery notation is not causing a particular difficulty (the success rates for 3 queries involving subqueries were 6/7, 6/7 and 4/7) [16].

A new user study was conducted to show that there is a range of data analysis queries (involving aggregation and subqueries) over RDF data that are for IT-trained users (without specific background in SPARQL) easier to compose in the visual notation and

tool than to write in the textual SPARQL notation. The user study was held in conjunction with a presentation on SPARQL and RDF data querying within the Knowledge Engineering course for the Master's degree computing students at the University of Latvia. The study started with the presentation on SPARQL (including a hands-on session over the hospital data endpoint) for 90 min with half of it devoted to the general presentation of SPARQL and the other half to the usage of SPARQL in the context of the hospital data. The ViziQuer notation and tool then was presented for 25 min. The 30 students attending the class were randomly split into two groups of 15 with the groups doing SPARQL queries and the visual notation queries respectively. The students were informed about the test purpose, methodology and voluntary participation. Although the student's results shall be counted towards the credit in a course homework, a similar credit can be obtained by solving the same tasks outside the user study participation. There were two students from the SPARQL group who did not participate in the user study and chose to do visual queries instead of the assigned SPARQL queries; their results are not included in the query result analysis, so leaving 13 participants writing SPARQL queries and 15 doing queries in the visual notation.

There were 10 basic tasks given to the users corresponding to different query patterns: (1) class-attribute-condition, (2) class-attribute-links-conditions, (3) count+condition, (4) count+link+conditions, (5) statistics by attribute (with link and condition), (6) subquery (condition on linked item count), (7) count over condition on subquery results, (8) existential link and sum aggregate, (9) nested subqueries and (10) negated links. There was 70 min time limit for the query completion. Table 2 provides the success of the participants on the queries (+: success, /: notable partial success (not counted in statistics), -: not solved, a: attempted query with no submitted solution). The columns U indicate the user ID's, the columns 1–10 correspond to the tasks; the table left part shows SPARQL text writers and the right part – the visual notation users.

Table 2. Raw test results

U	1	2	3	4	5	6	7	8	9	10	U	1	2	3	4	5	6	7	8	9	10
2	+	a	/	+							1	+	/	+	+	+	−	−	−		
3	+	/	+	+	+	/	/	a			4	+	+	+	+	+	+	+	/	−	
8	+	+									5	+	+	+	+	+	+	+	−		
9	+	−	+	−	a						6	+	+	+	+	+	+	+	a		
11	+		+	+	−	−	a				7	+	a	+	+	a					
12	+	+	+	+	/	a					10	+	+	+	+	+	a				
14	+										13	+	+	+	+	+	a				
17	+	+									15	+	+	+	+	/	+	+	+	+	+
19	+	+	+	+	a						16	+	+	+	+	+	a				
22	+	+	+	+	a	/					18	+	+	+	+	+	−	−	−		
23	−	+	+	a							20	+	+	+		−	+				
24	+	+	+	+	+	/	+	+			21	+	+	+	+	a	+	a			
25	+										26	+	+	+	+	/	+	+	+	a	
											29	+	/	+	/	/	/				
											30	+	+	+	+	+	a		−		−

The rather low overall mean test results (on average 2.92 fully successful queries per participant in SPARQL query writing and 5.27 queries in visual notation) can be partly explained by the limited time the users had for a large number of query creation in a new notation. A comparative analysis of both used notations still can be performed by creating a joint rank of participants by the number of their fully successful queries, finding the rank sums A = 257.5 for SPARQL writing users and B = 148.5 for visual notation users, and calculating the test statistics Z = 3.1785 and p-value 0.0007 < 0.05 threshold for the null hypothesis that writing queries for the IT-trained users in SPARQL is easier or about as easy than composing the queries in the visual notation. The calculation details, the user tasks and the materials needed for the user study reproduction are available on paper's supplementary material site. The visual notation user difficulties with Task 8 (just 2 successful results out of 8 attempts) indicate the need for explicit existential query design pattern in the query notation presentation.

6 Conclusions

We have presented a notation for RDF data analysis query specification in the style of extended UML class diagrams; the notation is able to cover most of SPARQL 1.1 select query constructs, including basic graph patterns, as well as optional and negated blocks, aggregation, grouping and subqueries, without query nesting structure restrictions.

The query notation is meant to be used both by general IT experts who may find using the notation convenient in parallel with textual SPARQL query writing and by non-IT trained domain experts for whom the direct SPARQL query reading and writing is generally expected to be too difficult (cf. e.g. [10]).

The user studies performed so far have shown that non-IT end users shall be able mostly to read and understand the basic visual query notation (including subqueries) [16], as well as that already the current generic query creation environment shall provide an advantage over textual SPARQL query writing in RDF data query creation for general IT experts that do not have previous specific training in SPARQL.

There is web-based tool support for the introduced notation, described in [17].

References

1. SPARQL 1.1 Query Language. W3C Recommendation, 21 March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/
2. Resource Description Framework (RDF). http://www.w3.org/RDF/
3. Optique. Scalable End-User Access to Big Data. http://optique-project.eu
4. Vega-Gorgojo, G., Giese, M., Heggestøyl, S., Soylu, A., Waaler, A.: PepeSearch: semantic data for the masses. PLoS ONE **11**(3), e0151573 (2016). https://doi.org/10.1371/journal.pone.0151573
5. Khalili, A., Merono-Penuela, A.: WYSIWYQ–What You See Is What You Query. In: Voila 2017, vol. 1947, pp. 123–130. CEUR Workshop Proceedings (2017). http://ceur-ws.org/Vol-1947/paper11.pdf

6. Ferré, S.: SPARKLIS: a SPARQL endpoint explorer for expressive question answering. In: Proceedings of the ISWC 2014 Posters & Demonstrations Track, vol. 1272. CEUR (2014). http://ceur-ws.org/Vol-1272/paper_39.pdf
7. Zloof, M.M.: Query by example. In: Proceedings of the National Computer Conference and Exposition, 19–22 May 1975, pp. 431–438. ACM (1975)
8. Catarci, T., Costabile, M.F., Levialdi, S., Batini, C.: Visual query systems for databases: a survey. J. Vis. Lang. Comput. **8**(2), 215–260 (1997)
9. Catarci, T., Dongilli, P., Mascio, T.D., Franconi, E., Santucci, G., Tessaris, S.: An ontology based visual tool for query formulation support. In: Proceedings of the 16th European Conference on Artificial Intelligence, pp. 308–312. IOS Press (2004)
10. 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. Univ. Access Inf. Soc. **15**(1), 129–152 (2016)
11. Haag, F., Lohmann, S., Siek, S., Ertl, T.: QueryVOWL: visual composition of SPARQL queries. In: Gandon, F., Guéret, C., Villata, S., Breslin, J., Faron-Zucker, C., Zimmermann, A. (eds.) ESWC 2015. LNCS, vol. 9341, pp. 62–66. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25639-9_12. http://vowl.visualdataweb.org/queryvowl/
12. Kapourani, B., Fotopoulou, E., Papaspyros, D., Zafeiropoulos, A., Mouzakitis, S., Koussouris, S.: Propelling SMEs business intelligence through linked data production and consumption. In: Ciuciu, I., et al. (eds.) OTM 2015. LNCS, vol. 9416, pp. 107–116. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26138-6_14
13. Zviedris, M., Barzdins, G.: ViziQuer: a tool to explore and query SPARQL endpoints. In: Antoniou, G., et al. (eds.) ESWC 2011. LNCS, vol. 6644, pp. 441–445. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21064-8_31
14. 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, vol. 1486. CEUR (2015). http://ceur-ws.org/Vol1486/paper_68.pdf
15. Čerāns, K., Ovčiņņikova, J.: ViziQuer: notation and tool for data analysis SPARQL queries. In: VOILA 2016, vol. 1704, pp. 151–159. CEUR Workshop Proceedings (2016)
16. Čerāns, K., et al.: Extended UML class diagram constructs for visual SPARQL queries in ViziQuer/web. In: Voila 2017, vol. 1947, pp. 87–98. CEUR Workshop Proceedings (2017)
17. Čerāns, K., et al.: ViziQuer: a web-based tool for visual diagrammatic queries over RDF data. In: Gangemi, A., et al. (eds.) ESWC 2018. LNCS, vol. 11155, pp. 158–163. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98192-5_30
18. Bārzdiņš, J., Grasmanis, M., Rencis, E., Šostaks, A., Bārzdiņš, J.: Ad-hoc querying of semistar data ontologies using controlled natural language. In: Arnicans, G., Arnicane, V., Borzovs, J., Niedrite, L. (eds.) Frontiers of AI and Applications. Databases and Information Systems IX, vol. 291, pp. 3–16. IOS Press, Amsterdam (2016)