ViziQuer: A Visual Notation and Tool for RDF Data Analysis Queries


Institute of Mathematics and Computer Science, University of Latvia
karlis.cerans@lumii.lv

Abstract. Visual SPARQL query formulation notations aim at easing the RDF data querying task, still the existing approaches fall short of providing a generally accepted visual notation suitable for data analysis and statistics queries. In this paper we present a visual diagram-centered notation and tool for SPARQL select query formulation, capable to cover aggregate/statistics queries and hierarchic queries with subquery structure. We present the notation usage examples, describe its syntax structure and provide the semantics by defining the visual query translation into SPARQL. We report on early pilot studies indicating the potential applicability of the visual notation to formulating SPARQL queries, as well as briefly describe the web-based open source implementation of the tool.

Keywords: Visual notation, Diagrammatic queries, RDF data endpoints, SPARQL, Ad-hoc queries, Data analysis, Multi-modal query tool

1 Introduction

SPARQL, as defined by a W3C standard [1], is the main query language over data that are available 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, such 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 IT professionals and domain experts.

A number of approaches exist to ease the query formulation in SPARQL. These include form-based interfaces (e.g. PepeSearch [4]), natural language based approaches (e.g. SPARKLIS [5]) and various kinds of visual/diagrammatic notations. In [6] it has been shown that a graphical diagram-based end-user query notation is a viable alternative to keyword search and natural language query interfaces. The work on Optique VQs in [6] as well as other previous visual query formalisms Query VOWL [7] and early versions of ViziQuer [8], although efficient for visual formulation of certain range

* Supported, in part, by Latvian State Research program NexIT project No.1 “Technologies of ontologies, semantic web and security”.
of queries, do not include queries with data aggregation and statistics facilities, so important and central e.g. for business intelligence area (cf. [9],[10]).

In this paper we present a multi-modal diagram-centered notation for creation of SPARQL select queries that is capable to cover both aggregate/statistics queries and hierarchic queries with subquery structure. Like the approaches of [6] and [8], the presented query notation is based on extended UML class diagrams, however presenting a novel visual notation both for aggregation and subqueries. This work wraps up authors’ earlier work of [11,12], where the initial concept of data aggregation in simple visual UML-style queries has been presented, and [13], where a system of constructs for extending UML class diagrams allowing both aggregate and hierarchic query formulation has been introduced. We recall a preliminary user study from [13] indicating a potential readability of the visual query constructs by domain experts, as well as present a new user study, indicating the advantages of query writing in the ViziQuer notation over the textual SPARQL notation by generally IT literate persons. This gives at least an initial confirmation of the potential usability of visual UML style notation in aggregate query formulation over the RDF data.

We also outline here for the first time the ViziQuer language abstract syntax and semantics (involving aggregates and subqueries), and describe the implementation of the query tool that has been made available as open source.

The aggregate and hierarchic SPARQL query formulation functionality is available also in natural-language based SPARKLIS framework [5], one could expect that, as in the case of non-aggregated queries, the natural language based and visual/diagrammatic approaches would be most beneficial for different user communities and query types.

Onwards the Section 2 introduces the visual notation by examples, Section 3 describes the query model and Section 4 outlines its semantics. Section 5 then provides evaluation and analysis of the notation, Section 6 briefly describes the tool implementation and Section 7 concludes the paper.

The supplementary materials for the paper are available at http://viziquer.lumii.lv/eswc2018_submission.

2 Notation By 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, developed originally in [14] as a fragment of a realistic hospital information system and presented here in Figure 1\(^1\). 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.

\(^1\) 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.
2.1 Basic Visual Queries

A basic visual query (cf. [8,13]) 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. Section 2.4) within the graph form its spanning tree with the main query node being its root.

Figure 1 shows an example basic visual query that can be phrased textually, as follows; 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; for each such episode list its case record number, total cost, discharge reason and unique episode identifier (URI), the patient name and birth year as well as the name of the referring physician, if specified.

The basic visual query links are labelled by properties from the data model (the property paths as sequences of model properties and their inverses are allowed in link labels,
as well). 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. H, 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 notation (.) in the attribute expression stands for the data instance internal identifier (URI). 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; there are also some further operation shortcuts available to ease the query definition such as x[1] standing for the first symbol in the string x (in general, x[a] denotes the symbol at the ath position in x).

The outpatientDiagnosisRecord.diagnosis.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.

![Figure 3. Aggregate query examples](image)

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 example in Fig. 3 (c) also illustrates the explicit node instance name (P at the node with Patient class) and the reference to it from an expression in the HospitalEpisode node. The function years 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). The option to hide the default label for links connecting the data model classes is applied here, as well.
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) example). 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 nodes outside the subquery) 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) example.

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 (should there be a desire to avoid the subquery construct, a count distinct could have been calculated on patients; the solution with subqueries, however, allow for wider result usage, including e.g. other aggregates over patients).

Figure 4. Visual subquery examples

Figure 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 (most of them presented initially in [13]) widen the expressivity of the visual language beyond the query tree shape and model structure matching.

Figure 6 shows two visual options for the query “Count patients having 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.

Figure 6. Condition and non-model links

A further query structure extension is by introducing the control nodes that do not describe any data instances: unit (denoted by /) and union (denoted by /+). The unit node can typically be used as an outer structuring layer of the query, 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. 7 (a) and (b) can be phrased as “List (a) and count (b) attribute ward values with more than 1000 treatment in ward cases”, they use the statistics by attribute notation from Section 2.2 within a subquery, embraced by a host query of a single unit node. The Fig.7 (a) notation corresponds to the usage of SPARQL having clause. The unit nodes introduced here, however, allow for much richer handling of aggregate results than just filtering.

Fig. 7 (c) demonstrates another advanced option of a query node describing a data instance without specifying the class information. In this case the node describes literal values (placed in ward attribute of TreatmentInWard class instances), not the resources.

Figure 7 (d) shows the query “Count all (distinct) patients that are related to diagnosis ‘A69.2’ (Lyme disease) either as admission or as discharge diagnosis of a hospital episode” where union node introduces a disjunction of its sub-trees (the link to the subtree from the union node is perceived as the link from the parent of the union node). There are also options to use subquery links both above and below the union node.

Figure 7 (e) shows the query “Select top 5 most expensive hospital episodes (show the episode case record number and total cost), list them together with all their treat-
ments in wards (show the order number and the ward)”. It illustrates two further important query notations: an edge (labelled by ‘==’ and called same data edge) connecting a data item to itself, and a global subquery (denoted by a white bullet at the edge start). The (local) subquery notion, as explained in Section 2.3, has the intuition of computing local characteristics corresponding to the subquery host node data instance.

The subquery in Fig.7 (e), however, is computed without the a priori host instance context, finding the 5 most expensive hospital episodes, and projecting these, together with their costs, into the main query.

The mechanism of separate subquery computation and projection into the main query that is used in SPARQL is compatible with the local computation intuition, as long as the slicing modifiers are not used within the subquery. Therefore, the ordering and slicing operations in ViziQuer are not allowed in local subqueries; there are no other semantic differences between the local and global subqueries. It could be a matter of a future work to introduce slicing into the local subquery setting, as well, however, with a 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.

Figure 7. Further query examples

It may appear convenient to create a field definition at a query node and to use it further in the definition of other field values, while the original field may be not included in the query output. Such a field not included into the query output is called internal, and marked by the text {internal}. Figure 7 (f) and (g) provides two equivalent query presentations (select first 100 hospital episodes by the case record number and show a ward in which the treatment starts), with and without the internal field usage.

As an advanced feature, ViziQuer supports also data model exploratory queries by allowing an explicit variable (prefixed by ‘?’), as in SPARQL, in the place of a class or
property name (both in role and attribute notations). Figure 7 provides the following examples: 7 (h): List all class names together with their instance count; 7 (i): Find all URIs of Patient class instances with their property names and values.

The examples (j) and (k) in Fig. 7 show the embedding of direct SPARQL fragments into the visual queries; the fragments can be expressed as group graph patterns (j), or as full SPARQL select queries (k); in each case the defined variables or the selection set of the direct SPARQL fragment are integrated into the namespace of the visual query (a SPARQL fragment variable \( ?x \) corresponds to visual node field with alias \( x \)).

3 Abstract Query Model

Figure 8 summarizes the abstract syntax of the visual queries in a UML-style model\(^2\) that shall be 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.

Structure edges of different types (cf. both edgeType and relationType attributes) define a query fragment structure that is essential for query semantics definition. Let an edge be plain, condition, local subquery or global subquery one based on its edgeType, and required, optional or negated one depending on its relationType. An edge is a structure edge, if it is not a condition one. We shall call also a structure edge 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. We shall use a structure edge characteristics to 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 connected in \( T(G) \) by plain required union-free edges only, together with structure edges incoming into fragment nodes and condition edges outgoing from fragment nodes (this reflects the understanding of structure edges semantically “belonging” to their target nodes and condition edges “belonging” to their source 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, subquery, union-free) extend to the fragment, as well.

We shall call a query or its subquery fragment is an aggregated one, if it has at least one aggregated field or the distinct option (\( \text{distinct}=\text{true} \)) within its head node specified.

It is assumed that a data node in a query can have either a class reference, or a query variable, but not both. A path expression, however, shall necessarily have either a list of property references, or a query variable, still not both.

The aggregated field list and/or 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.

For a model edge we assume that its property expression labeling is explicitly present in the abstract syntax, irrespectively of showing or hiding it within the visual notation.

\(^2\) The composition notation (a little diamond at an edge end) indicates the structure of all items in the query. The \( \{\text{ordered}\} \) notation indicates the ordering of model items linked by the edge to a single source model item. For all generalization groups in the diagram it is assumed that the superclass is the disjoint union of subclasses.
The derived \texttt{isSameDataLink} property of a model edge (‘==’ in the concrete syntax) is true if and only if there is no property expression attached to it. Assume, without loss of generality, that there are no two nodes within the query, labelled by different explicit instance names and connected by edges with \texttt{isSameDataLink=true}.

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

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 \cite{1} can be used to obtain expressions from simple expressions.

A simple expression can be a constant (e.g. a literal, or URI) 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 consist of a path expression (e.g. an attribute name) placed in a node/field reference context; in the most typical case the context item specification would have been omitted in the concrete syntax letting it to be the node containing the field or condition with the expression.

To support the SPARQL semantics definition in Section 4 we let $X \rightarrow Y$ denote that $Y$ 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

Figure 8. Core abstract syntax of queries
or optional. For a fragment $F$ let its selection set $\text{Sel}(F)$ consist of all non-internal fields, all query variables, as well as all “upwards” references to nodes in $\text{UC}(F)$, found within $o(F)$. Let $\text{SubSel}(F)$ be the union of $\text{Sel}(H)$ 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:

- $\text{SubSel}(F)$, the results projected out of sub-fragments;
- Nodes from $\text{UC}(F)$, 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 be used as a path data item context).

The references to $\text{UC}(F)$ 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 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 ascried to the following variable points: $VP(G) = \text{DataNode} \cup \text{Field} \cup \text{DataItem} \cup \text{QueryVariable} \cup SS \cup SG$. Let the SPARQL query variable name assignment $m: VP(G) \rightarrow \text{Var}$ 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 $x \in \text{QueryVariable}$, $m(x)=x.\text{variableName}$;
3) for $x \in \text{DataNode}$, $m(x)=x.\text{instanceName}$, if $x$ has $\text{instanceName}$ specified;
4) for $x \in \text{Field}$, $m(x)=x.\text{alias}$, if $x$ has $\text{alias}$ specified;
5) if $x \in \text{DataNode}$ is a node/field reference and $y \in \text{DataNode} \cup \text{Field}$ is the corresponding node/field, then $m(x)=m(y)$;
6) $m(x)=m(y)$, if $x \in \text{Field}$ and $y=x.\text{bodyExpression} \in \text{DataItem}$ and $x$ does not have $\text{alias}$ specified (same variable for the field and the data item within it);
7) $m(x)=m(y)$ if the nodes $x,y \in \text{DataNode}$ are connected by a same data edge;
8) $m(x)=m(y)$, 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 $x \in \text{PathDataItem}$, 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 $d \in \text{ClassReference} \cup \text{PropertyReference}$ 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.

| \( n \in \) Node | \( S(n) = \text{BGP}(m(n) \ \text{rdf:type} \ t(n.classRef.class)) \), if \( n.classRef \) is defined \( S(n) = \text{BGP}(m(n) \ \text{rdf:type} \ m(n.classVariable)) \), if \( n.classVariable \) is defined; otherwise \( S(n) \) is empty. |
| \( e \in \) Edge | \( S(e) = \text{BGP}(m(e.rsc) \ t(e.propertyExpr) m(e.trg)) \), if \( e.propertyExpr \) is defined, otherwise \( S(e) \) is empty. |
| \( d \in \) DataItem | \( S(d) = \text{BGP}(m(d.startingPoint) \ t(d.path) m(d)) \), 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 \( x = (m(e_1), \ldots, m(e_n)) \) and its \textit{direct translation} be \( T(x) = m(e_1), \ldots, m(e_n) \) and \textit{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 \( x \in \text{DataItem} \) and \( m(x) = m(f) \), otherwise let \( S_0(f) \) be \( Q(x) \) extended by \( \text{BIND}(T(x) \ \text{AS} \ m(f)) \). Let \( S(f) \) be \( S_0(f) \), if \( f \).requireValues \), and \textit{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 \( FL(c) = \text{EXISTS} \{ Q(x) \ \text{FILTER} (T(x)) \} \). |

Table 1. Local SPARQL translations of query model elements

For \( v \in \text{VP}(G) \) let its container \( c(v) \in \text{Node} \cup \text{Edge} \) 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:

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

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 \textit{union fragment} (consisting of a single union node), let \( P(F) = \text{Union}(P(F_i), \ldots, P(F_n)) \) for \( F \rightarrow \{ F_i, \ldots, F_n \} \); for all other cases use steps (2)-(13).
2) Consider the \textit{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 \textit{local SPARQL fragments} \( S(x) \) for data nodes, edges and graph pattern direct SPARQL clauses within \( F_0 \) to obtain the \textit{initial pattern} \( P_0 \).
4) Join to \( P_0 \) the patterns \( P(H) \) of all \textit{required subquery fragments} hosted by \( F_0 \) nodes, as well as full select direct SPARQL clauses to obtain \( P_1 \).
5) Left join (add optional SPARQL subqueries) the patterns \( P(H) \) to \( P_1 \) of all **optional subquery fragments** hosted by \( F_0 \) nodes, to obtain \( P_2 \).

6) Extend \( P_2 \) with local SPARQL fragments for **fields** in \( F_0 \), obtaining \( P_3 \). 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 \( P_3 \) the non-filtered patterns \( P^0(H) \) of **plain optional fragments** \( H \) hosted by \( F_0 \) nodes; each fragment \( P^0(H) \) is joined, taking into account the corresponding external filter expression \( EFL(H) \), denote the result \( P_4 \).

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 \( P^* \).

9) Collect the \( F_0 \) filter expressions specified in fragment node conditions into \( FL_0 \).

10) Add \( fn:not(exists(P(F))) \) to \( FL_0 \) 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 \( FL_1 \).

11) The raw SPARQL pattern corresponding to \( F \) is \( R(F) = Filter(FL_1, P^*) \).

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 \( R^*(F) \). For a non-aggregated \( F \) let \( R^*(F) = R(F) \).

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 \( P(F) = Project(R^*(F), Ext(F)) \).

For a plain optional union-free \( F \) let \( P^0(F) = P^* \) and \( EFL(F) = FL_1 \) (since \( F \) is non-aggregated, \( Ext(F) = Vars(F) \)). In all other cases let \( P^0(F) = P(F) \) 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 **Evaluation and Analysis**

As observed in [13], ViziQuer visual query notation contains counterparts of most SPARQL 1.1 select query language constructs. The main constructs currently not covered by the notation are named graphs, advanced property path expressions, SELECT * (in SPARQL sense) and reduced. None of these constructs would be of principal difficulty for a visual notation with SPARQL-based implementation, however, they would be not so clear should the ViziQuer notation be alternatively implemented, e.g. by translating the queries directly into SQL (what would make sense e.g. in OBDA context). The possibility to integrate direct SPARQL query fragments (clearly meant
for professional users) would allow a partially visual formulation of queries that contain fragments that are not expressible or are difficult to express in the visual notation.

The hospital data schema shown in Figure 1 corresponds to a fragment of the data structure of Children’s Hospital in Riga, Latvia. The ViziQuer notation has been sufficient for formulation of almost all queries that have been important for the ad-hoc query creation analysis experiment [14] of real hospital data (the only exception were queries requiring the result data to be of a tree shape, not a table form). Further examples of queries used during the analysis of real-life hospital data are available online from the paper’s supplementary material site.

There have been two pilot user studies conducted to explore the potential usability of the notation and its implementation both in query reading comprehension and query writing. The first pilot study on ViziQuer visual query construct understandability [13] involved a group of Master's degree medicine students at the University of Latvia and their teacher, together 7 participants, neither of whom had 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, two hospital episodes, one outpatient episode, and related diagnoses. The participants were asked to interpret 10 simple queries (covering the basic join query constructs, as well as aggregation and subqueries) over the given data set, 5 of the queries did include aggregation and 5 did not. The study results indicated that most of the participants (6 out of 7) were able to correctly interpret at least 70% of queries; with similar interpretation success rate both for queries that involve aggregation and those that do not.

To evaluate the ViziQuer query writing ability we conducted an initial experiment with a group of final year undergraduate Computer Science students at the University of Latvia. The students can be assumed to be generally IT literate, with at least academic background of SQL, however, without specific training in RDF/SPARQL. The 14 participants were given brief introductions about the hospital data model, RDF, SPARQL and ViziQuer (each about 10 minutes). The participants were split into two groups of 7 and given 10 query writing tasks. One group was asked to create queries using the visual ViziQuer notation and tool, and the other – using textual SPARQL notation (using a text-based SPARQL interface set up in the ViziQuer tool). The students of both groups were provided with an extensive set of query examples to train on and were able to execute the example queries over a pre-configured SPARQL endpoint. After the training period students had 60 minutes to work on the assigned tasks. The numbers of successfully completed tasks by the students in the visual notation group were 10, 5, 5, 4, 4, 3, 2 (in average 4.7 tasks), while in the SPARQL group there were 5, 5, 3, 3, 2, 2, 1 completed tasks (in average 3 tasks). The ViziQuer group outperformed the SPARQL group on data instance queries 12-6 (out of 14), on simple aggregate queries 16-14 (out of 21), and there was one student in the ViziQuer group who completed all 5 advanced query tasks (including subqueries) while in the SPARQL group there was just 1 student who got to subquery tasks completing one of them.

While the achieved results do not allow to establish definite usability conclusions, they provide an indication about the potential applicability of the visual notation and its implementation to formulating SPARQL queries. We provide the materials for the conducted user study on the supplementary material site of this paper.
6 Implementation

The query environment, available at http://viziquer.lumii.lv/, allows for user sign-up. Users can create projects, consisting of query diagrams each capable of hosting multiple queries. Each project can be configured to work with a user-supplied data schema (in a tool’s native JSON serialization format). Services for ViziQuer schema extraction from an OWL ontology and from instance data (by investigating SPARQL endpoints) are work in progress and early prototypes are available from the main ViziQuer page.

The project has to be supplied also a SPARQL engine type to enable query translation optimizations for vendor-specific SPARQL endpoints. The practical tool usage up to now has been oriented towards OpenLink Virtuoso SPARQL endpoints, although a “General SPARQL” option is available, as well.

The query tool is open source and has been made available on GitHub, so enabling local installations of the ViziQuer query engine.

The tool is created using ajoo – a generic platform for web-based diagrammatic tool building [15] that handles diagram rendering and editing on the basis of JSON encoding of the diagrams. It relies on the Meteor [16] framework and MongoDB for diagram storage and exchange, as well as for user management and collaboration features.

The tool architecture allows for de-coupling of the ajoo platform and custom diagram handling code from the Meteor server, should there a need arise for integrating it into some other diagram serving infrastructure.

7 Conclusions

By presenting the visual query notation and example queries over hospital domain data within it we have shown that a unified graphical notation, based on extended UML class diagrams, is possible for both data instance and statistical queries, including aggregation and subqueries. This contrasts the existing graphical query notations both for OBDA (cf. [6]) and relational databases that typically cover non-aggregated queries only. The ViziQuer graphical notation, apart from organic incorporation of aggregation and subqueries, is equipped with advanced notation elements such as non-model query links and control nodes to be at service of professional data analysts and query builders. These advanced elements bring the expressive power of the ViziQuer visual notation close to that of the textual SPARQL 1.1 query language.

The performed initial pilot studies indicate both that it is well possible to introduce the ViziQuer visual notation, including aggregation constructs, to domain experts without specific IT training, and that the visual notation and its ViziQuer/web implementation can facilitate and speed up the formulation of SPARQL queries by IT-trained query developers who are not Semantic Web experts.

The visual notation can be further extended, for instance, by the practically important construct of local subqueries with slicing (e.g. limit), not supported in SPARQL 1.1, if alternative target query languages (such as SQL) for the visual notation are considered.

The query language proposed is implemented in a web-based tool, available at http://viziquer.lumii.lv/. We expect that the availability of the ViziQuer visual query
environment infrastructure would encourage use cases of the technology outside the group of its developers and further development of user-oriented query solutions.

References

2. Resource Description Framework (RDF), http://www.w3.org/RDF/