Visual Presentation of SPARQL Queries in ViziQuer *

Kārlis Čerāns, Jūlija Ovčiņņikova, Mikus Grasmanis, Lelde Lāce, and Aiga Romāne

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

Abstract. Visual presentation of information artefacts can ease their perception. There are several solutions allowing visual-centered composition of SPARQL queries over RDF data endpoints by an end-user. We describe a "reverse" method and tool for generating visualizations in a UML-style diagram notation for a rich set of existing SPARQL queries (including BGPs, value filters, optional and negated constructs, as well as unions, aggregation, grouping and subqueries). Such a visualization is expected to enhance the reading, understanding, sharing and re-use ability for SPARQL queries, as well as it can be helpful in learning SPARQL itself. The prototype for visual notation generation from a SPARQL query is implemented within the ViziQuer tool environment. We describe and demonstrate the SPARQL query visualization possibilities on example query sets over Europeana cultural heritage data and other SPARQL endpoints.

Keywords: RDF · SPARQL · Visual queries · Query visualization · ViziQuer

1 Introduction

Visual presentation of information artefacts can ease their perception. Visual query composition paradigm (cf. [2, 8, 9, 12, 14]) along with facet-based ([10, 13]) and controlled natural language based (cf. [7]) approaches offers a promising avenue for involving end-users in query composition over SPARQL endpoints (cf. [12]). Although most of the visual query composition tools support visual composition of just the simplest forms of conjunctive SPARQL queries, with OptiqueVQs [12] and LinDA [9] supporting also outer-level aggregation, the ViziQuer notation and environment (cf. [2, 4, 5]) provides visual means for rich query definition, involving BGPs, value filters, optional and negated constructs, as well as unions, aggregation, grouping and subqueries, coming close to the full SPARQL 1.1 SELECT query visualization [2].

In this paper we present a "reverse" method for creating a visual presentation of an existing SPARQL query that complements the visual query creation process e.g., by offering an option to start the query creation from an already existing SPARQL form. It can help understanding a textual SPARQL query, therefore facilitating query sharing, communication, and re-use, as well as learning and adopting the visual query notation and eventually also the SPARQL notation itself. Our proposal is to use the ViziQuer notation [2, 4] as the SPARQL query visualization target, as it is sufficiently rich to

cover a wide range of query constructs, and it has an execution environment[1] [5] that can translate the visual query back into SPARQL and execute it over the data endpoint.

Automated visualization of (rich) SPARQL queries can be expected to facilitate the existing query comprehension. It would also allow combining the visual query creation approach with textual query writing and possibly also with other end-user-oriented query writing approaches, as e.g., SPARKLIS [7] that supports query creation in controlled natural language, to obtain integrated multi-paradigm data query environments.

We show in this paper that a SPARQL query can be rendered in a visual UML class diagram style notation, taking advantage of presenting a query variable or URI together with its class assertion and presenting its properties in attribute or link notation; such work has not been done before for rich sets of SPARQL queries.

Visualizing of textual queries is common within the realm of relational databases, where visual query builders for most DB management systems can be initialized by a textual SQL query and the visual and textual query presentations are kept in synchronization. There are efforts regarding SPARQL query visualization [11] in Optique VQs context, yet these consider only the simple query forms covered by [12].

We presume that adding a compact visual presentation to a SPARQL query can contribute to easing its perception (if comparing textual SPARQL query together with the visual presentation to the textual SPARQL query alone) if the visual query presentation is of similar or lower structural complexity than the original textual query. The target group of the query visualization would include both the "lay users" that would be able to obtain a structural presentation of the SPARQL query, as well as data management professionals that would be able to use the visual query presentation together with the textual one (in SPARQL).

Within the rest of the paper, Section 2 reviews the ViziQuer visual query notation, Section 3 describes the principles of mapping the SPARQL constructs onto the visual ones, Section 4 presents and discusses the SPARQL visualization evaluation, Section 5 describes the related work and Section 6 concludes the paper. The resources supporting the paper are available from http://viziquer.lumii.lv/examples/sparql2viziquer.

2 Visual UML Style Query Notation Overview

We start with a review of the visual RDF data query notation, as available in ViziQuer [2,4]. The visual queries are formulated in the context of a data dictionary that is a part of the data schema used by the ViziQuer tool and maps the entity short names used in the visual query presentations to their full IRIs.

Table 1 lists a few visual query examples over the Europeana cultural heritage data SPARQL endpoint. Further visual query examples are in [2, 3] and the tool web-site. The principal elements within a query are its ***nodes*** and ***edges***, the ***main node*** in a query is depicted as an orange round rectangle. A node in a query is either a ***data node*** that describes a variable or an IRI to be used as a subject or object within a triple pattern in a SPARQL query, or a ***control node*** use to further shape the query structure, if necessary (cf. the [ ] node in Example 4 in Table 1). The variable name or IRI can be explicitly specified for a data node, or the variable name can be implicit.

[1] cf. http://viziquer.lumii.lv

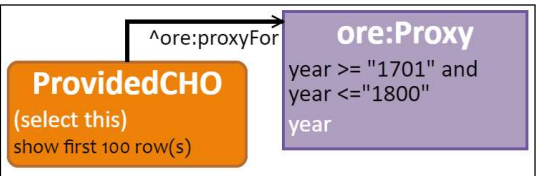
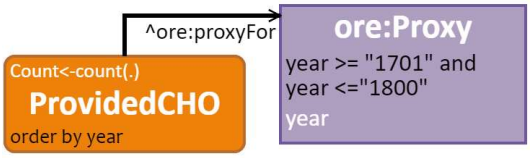
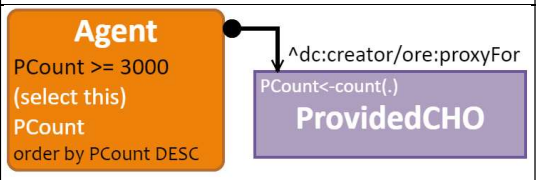
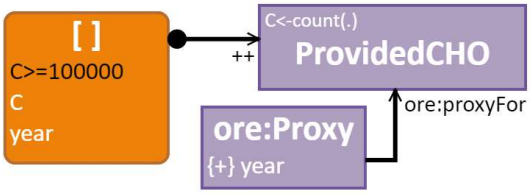
1. *List top 100 provided cultural heritage objects from the 18th century.* Each query node is a data instance pattern, listing its class name and conditions, and the selection items; the links show the instance connections.	
2 *Count the provided cultural heritage objects for each year during the 18th century*. The grouping is automatic by all non-aggregated selection fields.	
3. *Find all agents that are creators of at least 3000 provided cultural heritage objects.* The subquery construct (on *^dc:creator/ore:proxyFor* link) is used for calculating the CHO count for every agent separately.	
4. *Find all years corresponding to more than 100000 cultural heritage objects.* The subquery (over *ProvidedCHO* and *ore:Proxy*) computes the CHO count for each year. The outer query with unit node [ ] filters the subquery results.	

Table 1: Visual Query Examples

The following optional information can be specified for each data node:

- A ***class name*** (encoding the *rdf:type* assertion of the node variable of IRI),
- An ordered list of ***attribute fields***; each field specifies a variable in the query, together with its relation to the node variable or IRI, either via a Basic Graph pattern (BGP) triple, or as a BIND-expression; there are markers {+} for a required attribute value and {h} for not including the variable into the query select list (such a variable remains a helper for other value and condition description),
- A list of ***conditions*** (filters), each a Boolean-valued expression to be evaluated in the context of the node (shown in dark font), and
- A list of additional ***grouping markers*** (including a dedicated option to mark the node variable itself as grouping); this applies to aggregated queries.

The ***principal*** (bold) ***edges*** of a query determine its ***tree-shape structure*** with the main query node being its root. An edge can be labelled by a property path denoting its end node variable/URI connection, or it can be marked by == denoting the same variable on both ends, or ++ standing for no specified data connection between the edge ends; further connections between nodes can be specified by explicit references to the node and field aliases. There can be also ***auxiliary edges*** besides the query principal edge tree-shape structure to specify the extra data connections visually. An edge can also

carry an ***explicit variable*** in the form *?p* for a SPARQL triple with a variable in the predicate position (use the form *??p* if the variable is not included in the selection set).

A principal edge can correspond to a ***join link***, or an edge can link a host node to a ***subquery*** (starting with a black dot, as in Table 1, examples 3 and 4); the scope of a subquery involves the entire graph fragment beyond the subquery link, as well as the link itself and a reference to the variable or IRI in the host node (the node just above the subquery link). An edge can be ***required***, ***optional*** or ***negated***. Auxiliary edges always are join links; it does not make sense to have an auxiliary edge optional.

The main query node, as well as the head nodes of subqueries can have the following information (that applies to the entire main query or subquery level), as well:

- A list of ***aggregate fields*** (each with an aggregate function, aggregation expression, and a distinct element marker); these are positioned above the node class name,
- A ***distinct values*** specification to apply the DISTINCT modifier to the solution set (not to be used together with aggregate fields)
- An ordered list of ***result order modifiers*** (ORDER BY)
- Result ***slicing modifiers*** LIMIT and OFFSET (main query node only, or a subquery marked with an additional 'global' option).

There are two control nodes available in ViziQuer: unit [ ] and union [+]. A unit node can serve as a non-data bound container for outer-level processing of data (e.g., filtering or further aggregating) returned by a single or several subqueries. A union node corresponds to the union of the data sets corresponding to its branches (a branch can be linked to the union either by a join, or by a subquery link).

The visual query notation is explained in further detail and translation of a visual query into SPARQL are described in [4].

3 SPARQL Query Visualization

The visualization of a SPARQL SELECT query produces a visual extended UML-style diagram that describes the entire query contents. For a simple query consisting of the graph patterns, the pattern subject and object variables and resources are depicted as the query graph nodes and the graph edges correspond to their connections, with the important optimizations to represent the variable/resource ***classes*** in the dedicated class name compartments and ***single-use objects*** of SPARQL triples within node attribute fields, so obtaining a compact query presentation. Figure 1 shows an example query adopted from [1] in SPARQL form, and automatically rendered in the visual notation (with manual tuning of the node placement); the common namespace prefix definition in the SPARQL query is omitted for brevity.

The nodes in the visual query correspond to the variables *?ProvidedCHO*, *?Aggregation* and *?Proxy*. The variable *?year* is rendered into the visual notation in the attribute form. There are no class names in the visual query, as the original query contains no class specification. The SPARQL query visualization algorithm can create also visual query forms (up to the placement of the nodes and edges) from the SPARQL query texts for all examples, shown in Table 1. In the case of a query consisting of several nested blocks, the visual representation is first created for each of the blocks separately,

followed by merging the block representations into a single visual query. The rendering of expressions is generally carried over from the SPARQL textual notation into the textual compartments of the visual query notation.

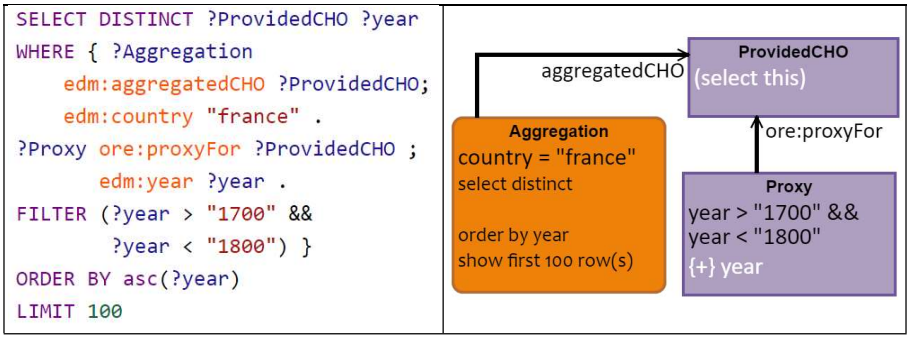


Fig. 1: Objects provided to Europeana from the 18th century from France

In what follows, we outline the SPARQL SELECT query visualization algorithm in more detail. The current version of the algorithm assumes that the SPARQL query to be visualized conforms to the following restrictions:

1. there are no HAVING clauses in the query[2],
2. the query does not use the FROM, GRAPH and SERVICE clauses, and
3. the EXISTS/NOT EXISTS clauses in the FILTER expressions are on the top expression level (i.e., they are not within other logical expressions).

The queries not satisfying 1 can be re-written into an equivalent form, satisfying the condition. The issue 2 is a limitation induced by the current state of the visual notation not providing the support of the respective constructs. The issue 3 can be resolved by resorting to richer textual expression fields within the visual query frames; it has not been considered at the current point.

The translation of a SPARQL query into the UML-based ViziQuer visual notation is based on the inclusion hierarchy of the following ***blocks*** within the query:

1. the query itself and its subqueries, and
2. the ***inline blocks*** that are group graph patterns behind group-or-union construct (simple parenthesis { } or UNION clause), OPTIONAL[3], MINUS, or EXISTS[4]/NOT EXISTS fragments within FILTER clauses.

[2] There is no direct counterpart for the HAVING clauses in the visual notation currently, as it is preferred to model their effects by means of subqueries

[3] An OPTIONAL block that consists of a single triple is considered to be a part of the enclosing block and the options of rendering the triple in the attribute notation are examined. If the attribute-form presentation of the triple fails, the block is processed as a separate one, just as larger OPTIONAL blocks.

[4] A FILTER EXISTS block can be incorporated into a textual filter within a node, if it introduces a variable, followed by a filter on its value.

The visual query is a graph. Its construction starts with its ***node and edge creation*** for a ***single block*** identified in the query structure, followed by ***connecting the block visualizations***.

3.1 Query Block Visualization

We start the processing of a query block by considering all query variables, URIs and blank nodes that appear in the subject or object positions in the basic graph pattern (BGP) triples in the block as the ***node candidates*** in its graph presentation. The ***edges*** connecting the created node candidates are created based on the block BGP triples and are decorated by the triple predicates (the decorations can be property paths or variables).

To obtain a compact query presentation, we transform the obtained node candidate and edge structure, as follows:

1. the presentation of any triple with *rdf:type* property and a variable or URI in the object position is transformed into the ***class specification*** for the node of the triple subject (variable, URI or blank node); note that we do not apply the transformation for the class specifications that are blank nodes; we are also not able to account for more than one class specification for a subject node (in such a case the other class assertions stay in the visual shape),
2. variables (not URIs and not blank nodes) that are mentioned in a single object position within the block triples (and are not mentioned in any subject or predicate position within the block triples) are rendered into the ***attribute notation*** for the node corresponding to the subject of the respective triple [5] [6]. This also includes the case of single-triple OPTIONAL blocks that are rendered as optional attributes.

To fit the created query block node and edge structure into the visual query shape with a spanning tree consisting of the principal edges, we mark all created edges as ***required join links***, except when the edges create a non-tree structure; in such case some edges are marked as ***auxiliary***. If the graph corresponding to a block is not connected, add empty-labelled (++ in the concrete syntax) edges to create the connections.

Note. The decisions on the main node in the visual query corresponding to the query block and on exact ways of marking some of edges auxiliary or adding the empty-labelled edges can be taken arbitrarily as this does not change the query fragment semantics (some considerations regarding the choice of the main node of a query fragment shall arise from connecting the fragments at a later point).

Figure 2 illustrates a single-fragment SPARQL query (with a single-triple OPTIONAL sub-fragment) and its visualization by the outlined algorithm (handling the query selection list shall be addressed shortly).

[5] The visualizer can be instructed not to create attribute notation for triples corresponding to the object properties defined in the data schema.

[6] The attributes corresponding to the BIND-expressions located in a block shall be created at a later stage.

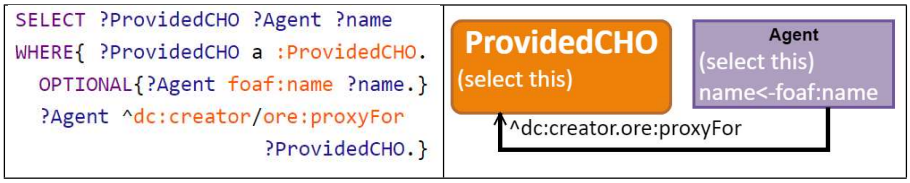


Fig. 2: Agents and their created CHOs that are provided to Europeana

3.2 Attributes, Filters and Aggregations

To complete the visual presentation of query blocks (as visual query fragments) the ***attribute lists*** for nodes are to be finalized, the ***filters*** are to be introduced, and the ***aggregates*** projected out of the main query and all its structural blocks that are subqueries are to be defined. The defined attribute and aggregate lists shall also describe the ***selection lists*** for the query and its subqueries. We arrange the process by traversing the query block graph (as observed in the textual SPARQL query) in a bottom-up manner since the attributes of an outer block may refer to the selection out of an inner block.

The ***attribute list*** for a node in a visual query fragment corresponding to a block shall involve:

1. the variables already presented in an attribute notation,
2. the variables computed as BIND-expressions within the fragment (each variable goes to a single node where it belongs most naturally), and
3. the variables that are projected out of the subquery-style fragments that are direct children of the current fragment; only those variables that are to be selected out of the current fragment are included.

The ***aggregates*** of nodes in a visual query fragment (that is the main query or is a subquery-style fragment) are created based on the aggregate expression binding in the selection list of the corresponding query block (the variables used as arguments to the binding should have been visualized in the created node attribute lists at this point).

The ***filters*** in the query nodes are created in accordance with the textual filter definition based on the computed attributes (some existential filters that involve a richer structure than a triple together with its value checking filter correspond to separate query fragments).

We perform an optimization to the created attribute structure to exclude from the node attribute lists those attributes that are used just in a single place within a BIND-expression, or an aggregate expression, or a filter, and that are not selected out of the query fragment themselves; these variables become ***implicit*** within the respective BIND-expression attribute, the aggregate expression, or the filter.

To ensure that the created attribute list defines the appropriate selection list for the query fragment (the attributes are selected by default), we mark all attributes that are not included in the selection list as helpers ({h} notation in the concrete syntax) and add explicit *(select this)* attributes for nodes whose variables are to be included in the select list (a node variable is not included in the select list by default).

3.3 Connecting Block Visualizations

The connection of (the visual fragments corresponding to) including blocks is based on identifying usage of the same variables in both the outer (the including) and the inner (the included) block. The primary connection between the blocks is based on just ***one variable corresponding to a node in each of the two blocks***. If there is no such variable (and there is no option to obtain one by reversing the earlier decisions to represent some variables as attributes), we introduce an empty-labelled edge (++) to connect some nodes within the blocks and do not consider further optimizations.

If such same-variable nodes, say N in the inner block's fragment and, say E, in the outer block's fragment exist, we initially draw an edge, labelled by '==' (the same-variable edge) between E and N (in this way we achieve the possibility to connect the fragments in any situation).

The principal idea of the block merging is to ***merge*** the nodes N and E into a ***single node in the visual query***, whenever this is possible. For the merging N should not have any attribute fields (except *(select this)*) and the inner block should remain non-empty after the removal of N. In this case we choose a node N' in the inner fragment (preferably, N' should be connected to N, otherwise an empty link with the ++ label is to be created) to become the ***principal node*** of the inner fragment. We consider the link connecting N (merged with E) and N' to be the connection between the outer and inner fragments and mark it with the type of the ***outer-to-inner block connection type*** (e.g., subquery/optional/not/union branch); mark other edges (if any) going from N to the other inner block nodes as auxiliary.

Figure 3 shows an example of merging nodes corresponding to the variable *?Agent*: the query fragments (a) and (b) together give the compound query (c).

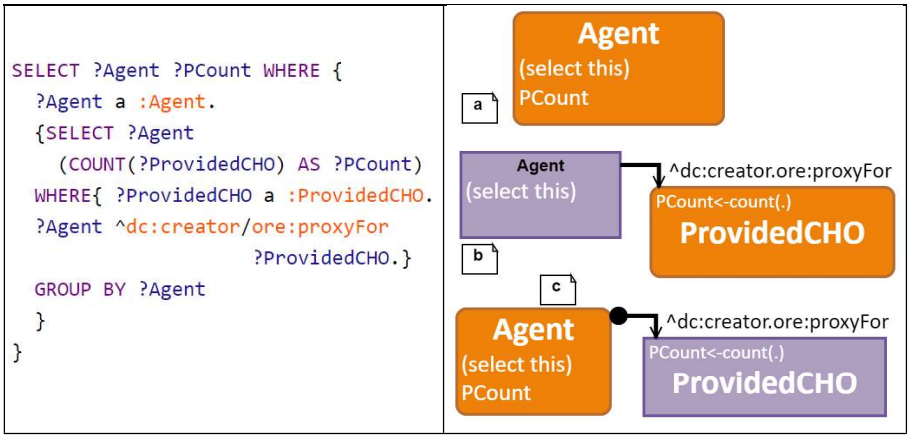


Fig. 3: Merging of nodes of including blocks

To optionally visualize other inter-block connections, if there are blocks X in Y in the query block structure, where X is (either directly or indirectly) above Y and there

are nodes E in X and N in Y, both marked with the same variable, connect E and N with an auxiliary edge, labelled by ==. Further on, if N has only one edge G in Y and N does not have any attribute (except *(select this)*), remove N and re-connect its edge G to E (as an auxiliary edge) instead of N.

After the block fragment merging, add the solution modifiers, as appropriate. In the case if a subquery has a slicing modifier (LIMIT or OFFSET), mark it a global subquery.

3.4 Implementation Notes

The SPARQL query visualization algorithm prototype has been implemented in Javascript programming language within the open source ViziQuer environment providing access possibility to the implementation[7]. The concrete syntax rendering of the query is provided by the API functions for visualizing the ViziQuer abstract syntax.

To ease the local ViziQuer server usage, a pre-built Docker environment with the tool is offered.

The visualization of a SPARQL query can be done within the visual query environment by copying the SPARQL query text into the SPARQL query pane and activating a 'Visualize SPARQL' context menu item.

The visualization of a SPARQL query is available in the context of any SPARQL endpoint for which the schema has been collected and the visual query environment has been created. There is an open-source tool available for retrieving the schema from a SPARQL endpoint[8] for the usage in the visual query tool.

4 Evaluation and Discussion

A simple notion of a successful visualization of a SPARQL query over a given data set would be to require that the visual query, when translated back into SPARQL and executed, would produce the same result as the execution of the original query. This notion may admit the dependency of the visualization and the visual query execution on the particular data set (e.g., if a property is known to have a maximum cardinality 1, both the visualization and visual query execution processes can possibly be simplified) and it can be contrasted with the mathematical notion of visualization success that would lead to a mathematically equivalent SPARQL query after the visualization and SPARQL query generation round-trip disregarding any data set specificity.

In the present work we consider the simple notion of the visualization success and work to expand the set of practical SPARQL queries that can be successfully visualized.

The current implementation of the visualization algorithm can be shown to successfully visualize the queries from the following query sets:

- the SPARQL queries generated from the visual queries in existing ViziQuer methodology and tool demonstrations, including [2, 3, 5], as well as the ViziQuer user manual (the "reverse" visualization of originally visual queries);

[7] https://github.com/LUMII-Syslab/viziquer
[8] https://github.com/LUMII-Syslab/obis-schemaextractor

- the example queries targeting the Europeana SPARQL endpoint [9] and described in its documentation [1];
- the example queries targeting the BNB SPARQL endpoint [10].

The live environments with Europeana and BNB example query visualizations are available from the paper's support page http://viziquer.lumii.lv/examples/sparql2viziquer. The chosen approach to consider the reverse visualization of the queries generated initially from the visual form is a natural one, as this would show the reversibility of the visual query implementation and show that the concepts of the visual query creation can be identified within SPARQL queries presented initially in the textual form.

Visualization of example query sets over different data endpoints indicate the possibility to obtain the visual form from a query that is expected to be understood by an end-user. We consider the addition of the visual form to an existing SPARQL query to bring another possible benefit in regarding the SPARQL query understandability (by comparing the comprehension of the textual query form alone to the comprehension of the textual query form together with the visual one). Still, the contribution of the visual notation to the SPARQL query comprehensibility remains to be evaluated.

Since the translation of a SPARQL query into the visual notation, as described in Section 3, goes on a construct-by-construct basis with very limited new construct introduction options, involving the optimizations related with the class and attribute form introduction in the visual presentation, the visual query would be generally of comparable or lower structural complexity than the textual query. Furthermore, the visual query presentation has appearance benefits if compared to the textual query presentation form by virtue of structuring the query across several interlinked graph nodes.

5 Related Work

A prominent UML-style visual SPARQL query creation environment is Optique VQs [12]. If compared to ViziQuer, the visual query constructs in Optique VQs are more limited, as they do not support subqueries, or the optional and negation modalities of join queries. There is also a much more limited expression language and expressions are not shown explicitly in the Optique VQs visual query presentation. Although the approach chosen by Optique is clearly relevant for usage of non-IT experts, the visual query presentation alone could not be regarded as a re-formulation of the textual SPARQL query. There are studies regarding textual SPARQL query rendering into a visual form in Optique VQs [11], however, our approach provides means for visualizing a much wider scope of the SPARQL queries.

The option of creating a visual presentation for a SPARQL query has been considered also in Sparqling [6], however, it is not using the UML-style presentation of the visual queries, and the scope of supported constructs is limited.

The ability of rendering a textual query visually is common also for the major relational database frameworks, however it is usually limited to simple join-filter-select queries that can be visually presented, as well as the visual presentation is not intended

[9] http://sparql.europeana.eu

[10] http://ldodds.github.io/bnb-queries/, the endpoint: https://bnb.data.bl.uk/sparql

to cover every aspect of the query definition (typically excluding any visual description for advanced expression computation, e.g., to describe the filters involved in the query).

The other major frameworks for visual SPARQL query creation, as [8, 9] do not provide explicit focus on obtaining a visual presentation from a textual SPARQL query expression.

6 Conclusions

The translation of a textual SPARQL query into a visual notation provides a new query viewing perspective, that can be beneficial in the query understanding. By contributing to the understandability of SPARQL queries, the query visualization has a potential to enhance the options for query sharing among the users and query reuse. The visualization of SPARQL queries would clearly enhance the options of learning and adopting the visual notation; it might be of help also in understanding and learning SPARQL itself.

The visual query environment allows further tuning of a query after its visualization, e.g., by adding the class information to some of the variables used in the query, in this way potentially further enhancing the end-user comprehension of a query.

An interesting perspective of using a SPARQL query visualization mechanism would be to integrate it with SPARQL query development tools, as SPARKLIS [7] for natural language-based query creation, or some form-based SPARQL query creation tool, as PepeSearch [13] or WYSIWYQ [10], to obtain a multi-modal SPARQL query management environment with integrated SPARQL query visualization option.

An important future work would be to assess the real benefits that the SPARQL query visualization could bring to an end-user; the potential end-users would be expected to be involved in such a study. As the provided SPARQL query visualization algorithm is primarily experience-based, a further study of its formal properties would be beneficial, as well.

In the general context of visual support to SPARQL query presentation, an interesting development avenue would be to also create a visual notation for SPARQL CONSTRUCT queries, coupled with their corresponding visualization tools.

7 Acknowledgements

This work has been partially supported by a Latvian Science Council Grant lzp-2020/2-0188 “Visual Ontology-Based Queries”.

References

1. Europeana sparql api, `https://pro.europeana.eu/page/sparql`, accessed on 24.09.2021
2. Čerāns, K., Bārzdiņš, J., Šostaks, A., Ovčiņņikova, J., Lāce, L., Grasmanis, M., Sproģis, A.: Extended uml class diagram constructs for visual sparql queries in viziquer/web. In: VOILA@ISWC. pp. 87–98. No. 1947 in CEUR Workshop Proceedings (2017), `http://ceur-ws.org/Vol-1947/paper08.pdf`

3. Čerāns, K., Ovčiņņikova, J., Lāce, L., Hodakovska, J., Romāne, A., Grasmanis, M., Kalniņa, E., Sproģis, A., Šostaks, A.: Visual query environment over rdf data. In: Posters and Demos at SEMANTiCS 2019. pp. 156–160. No. 2451 in CEUR Workshop Proceedings (2019), http://ceur-ws.org/Vol-2451/paper-32.pdf
4. Čerāns, K., Šostaks, A., Bojārs, U., Bārzdiņš, J., Ovčiņņikova, J., Lāce, L., Grasmanis, M., Sproģis, A.: Viziquer: a visual notation for rdf data analysis queries. In: Research Conference on Metadata and Semantics Research. pp. 50–62. No. 846 in CCIS, Springer (2018)
5. Čerāns, K., Šostaks, A., Bojārs, U., Ovčiņņikova, J., Lāce, L., Grasmanis, M., Romāne, A., Sproģis, A., Bārzdiņš, J.: Viziquer: a web-based tool for visual diagrammatic queries over rdf data. In: The Semantic Web: ESWC 2018 Satellite Events. pp. 158–163. No. 11155 in LNCS, Springer (2018)
6. Di Bartolomeo, S., Pepe, G., Savo, D.F., Santarelli, V.: Sparqling: Painlessly drawing sparql queries over graphol ontologies. In: VOILA@ISWC. pp. 64–69. No. 2187 in CEUR Workshop Proceedings (2018), http://ceur-ws.org/Vol-2187/paper6.pdf
7. Ferré, S.: Sparklis: An expressive query builder for sparql endpoints with guidance in natural language. Semantic Web **8**(3), 405–418 (2017)
8. Haag, F., Lohmann, S., Siek, S., Ertl, T.: Queryvowl: Visual composition of sparql queries. In: The Semantic Web: ESWC 2015 Satellite Events. pp. 62–66. No. 9341 in LNCS, Springer (2015)
9. Kapourani, B., Fotopoulou, E., Papaspyros, D., Zafeiropoulos, A., Mouzakitis, S., Koussouris, S.: Propelling smes business intelligence through linked data production and consumption. In: On the Move to Meaningful Internet Systems: OTM 2015 Workshops. pp. 107–116. No. 9416 in LNCS, Springer (2015)
10. Khalili, A., Merono-Penuela, A.: Wysiwyq-what you see is what you query. In: VOILA@ISWC. pp. 123–130. No. 1947 in CEUR Workshop Proceedings (2017), http://ceur-ws.org/Vol-1947/paper11.pdf
11. Mumtaz, Y.: Collaborative Query Formulation in a Visual Query System. Master's thesis (2015)
12. 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 **15**(1), 129–152 (2016)
13. Vega-Gorgojo, G., Giese, M., Heggestøyl, S., Soylu, A., Waaler, A.: Pepesearch: semantic data for the masses. PloS one **11**(3), e0151573 (2016)
14. Zviedris, M., Barzdins, G.: Viziquer: a tool to explore and query sparql endpoints. In: The Semantic Web: Research and Applications. pp. 441–445. No. 6644 in LNCS, Springer (2011)