Interactive Exploration and Querying of RDF Data

Martin Kryl[12], Petr Vcelak[12] and Jana Kleckova[12]

[1]NTIS - New Technologies for the Information Society, University of West Bohemia, Czech Republic
[2]Department of Computer Science and Engineering, University of West Bohemia, Czech Republic
email: {kryl, vcelak, kleckova}@kiv.zcu.cz

***Abstract*—Many contemporary healthcare information systems incorporate and utilize Resource Description Framework (RDF) datasets, which are characteristic by their flexibility and ability to form complex data networks. Users might find themselves overwhelmed when trying to understand the data layout since there are no apparent rigid structures such as tables in relational databases. In this paper, we present a prototype data exploration tool, that enables users to grasp the data structure by exploring a simplified RDF model. The solution does not rely on ontological description. The visualization has four modes of interaction defined that allow exploration in different levels of detail. One of the modes can be used to interactively create a SPARQL SELECT query. The proposed solution combines graph visualization and data extraction techniques into a single tool and allows users without expert SPARQL knowledge to extract data from RDF graph.**

Keywords–RDF; visualization; data exploration; interactive SPARQL builder.

I. Introduction

The current trend in data storage is to use various nonrelational databases (NoSQL) and data models beside classical relational databases. Big Data are common in health domain and relational databases are not considered suitable for dealing with them since they lack horizontal scalability, and need hard consistency [1]. One of the currently popular NoSQL data models is RDF, which has found its use as the preferred model for Open Data [2] and it is also used in Medical Information Systems for its ability to integrate heterogeneous data. Medical RDF applications can range from custom prospective study databases [3] to systems for inter-hospital data exchange [4]. RDF model is directed multigraph, and can be queried by special query languages, e.g., SPARQL.

There are some disadvantages of keeping data in RDF model. The main issue seems to be the lack of RDF support in common analytical and Business Intelligence software. Users usually need to transform the data into tabular format before being able to do the analytics, which requires additional knowledge of RDF query language. Another issue arises when data analysts want to understand the content of a dataset by performing data exploration. The structure of raw data is complex and there is only limited support in some graph database systems for navigating through the graph [5]. Various semantic browsers for traversing RDF graphs exist, such as Tabulator [6], but it can be difficult to use them for large graphs. The user can only see immediate surrounding nodes and may get lost during graph traversal. One can try to visualize the dataset but with the growing number of resource nodes, the legibility of visualization quickly decreases. Providing a method for interactive RDF analytics, that would not require the user to manually write queries, is currently considered to be an important area of research [5].

We have been working on a prototype solution allowing users to explore a general RDF data model and interactively define a data projection above the dataset without the need of extensive knowledge of RDF and query languages. The solution is composed of three components. The first one is an RDF model crawler, which analyses the model structure, determines property cardinalities and prevalence of RDF types. The second one is a web visualization which utilizes findings of the crawler to provide an aggregate graph view with a possibility of interactive model exploration. The third is a query builder that provides auto-generated SPARQL queries based on the user interactions with the visualization. The user can select objects of interests in the visualization and get the transformed underlying data.

There are multiple related projects that either help users to build SPARQL queries [7][8] or provide an aggregated visualization of data model [9]. However, to the best of our knowledge, there are no solutions that would assist in data exploration and extraction by combining the two techniques.

Both of the query builders work with a fixed set of SPARQL endpoints and have no visual tool that would help the user understand the relations in dataset. SPARQL Builder [7] constructs the query in two steps. First, the user selects two RDF classes from the dataset. The application prints all the possible paths between the resources of selected classes and the user has to choose one of them forming the query pattern. SPARQLGraph [8] offers an intuitive drag & drop query builder allowing the modeling of more complex query patterns than SPARQL Builder. The visualization [9] uses aggregated model based on the RDF classes of resources and provides good overview of relations in the dataset. However, it is not possible to use the visualization to interactively generate a query.

In the following Section II, the structure and functionality of the proposed system is described. The prototype implementation details are provided in Section III. The achieved results are discussed in context of other related solutions in Section IV before the conclusion and future outlook in Section V.

II. Proposed system

The system is composed of data preprocessor, graph visualization and query builder. Every RDF dataset needs to be transformed into corresponding aggregated model first. The aggregated model describes general structure and relations found in the dataset and serves similar purpose as widely accepted Entity-Relational model in relational databases. Visualization draws the model and allows user to interact with it. Query builder allows selection and projection of source data.

A. Data preprocessing

RDF data utilize property *rdf:type* to indicate that the resource is an instance of the specified class. Knowing the classes of resources, aggregated model is formed by finding all distinct combination of (Sc, P, Oc), where Sc is the class of a subject resource, P is the property and Oc is the class of an object resource. Resources having no class defined are ignored at this time. Literal values of datatype properties are considered to be belonging to a pseudo-class with the same id as the property, i.e., (Sc, P, P). In theory, it would be possible to generalize this aggregation by choosing arbitrary property instead of *rdf:type*, but in practice, there seems to be no property with such a prevalence.

Additionally, cardinalities of all properties in respect to subject and object classes are calculated, i.e., number of different values of type Oc are counted for each instance of Sc in (Sc, P, Oc). Visualization uses information about minimum and maximum cardinalities, as well as histogram of cardinality values. Lastly, total number of instances of each class are calculated.

B. Visualization

Metadata collected in previous step are incorporated into interactive directed graph visualization. Nodes represent resource classes or literal pseudo-classes, while edges represent RDF properties. Color of the node is used to distinguish between regular and pseudo-classes. Existence of an edge labeled P directed from A to B means that there is at least one RDF triple in the original data, where an instance of A is related to an instance of B by property P. Edges are visualized as arcs with clockwise orientation instead of lines. This way inverse properties do not overlap and it is also possible to visualize multiple edges in the same direction by assigning different radius for each arc. By using this technique, it is possible to draw loop edges the same way as other edges. Styling of the edge indicates the minimum and maximum cardinality of the property in context of classes A and B:

- Solid arc represents the minimum cardinality of 1, i.e., each instance of A has at least one value of class B assigned by the property.
- Dashed arc indicates minimum cardinality of 0, i.e., there are instances of A that have no instance of B assigned by the property.
- Empty arrow marker means that the maximum cardinality of the property is 1, i.e., there is always at most one value of B.
- Filled arrow marker means that the maximum cardinality is greater than 1, i.e., the property may have multiple values.

Following the Visual Information Seeking Mantra: ‘overview first, zoom and filter, then details-on-demand’ [10], the visualization offers four modes of interaction.

Overview: By default, the visualization shows only nodes representing RDF classes and object properties between them. This is the minimal possible configuration showing the relations in the entire dataset. Only the local names of a resources, the last part of object URIs (Uniform Resource Identifier), are displayed, unless there is an ambiguity. The user can scroll through the visualization canvas and move individual nodes. If there are multiple properties with the same orientation between any two resources, they can be merged into a single arc with numerical label indicating the number of merged properties

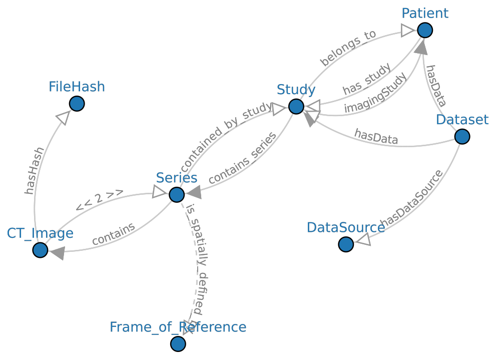


Figure 1. Visualization of an aggregated model of medical imaging data consisting of 8 RDF types and showcasing all possible cardinality types. Literal nodes and datatype properties are hidden.

as can be seen in Figure 1 between nodes labeled *CT_Image* and *Series*, or they can be drawn as separate arcs as shown in Figure 1 between *Patient* and *Study* nodes.

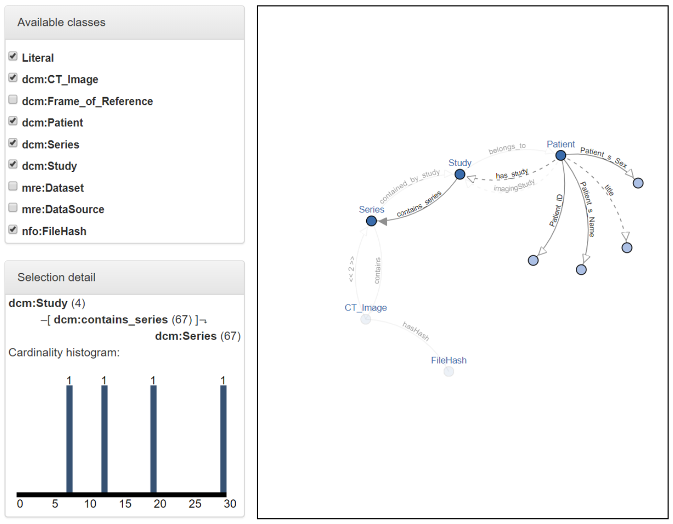
Zoom: This mode allows an expansion of detail in an area of interest. By clicking on the node, the user can choose to display datatype property edges leading from the specified class to respective pseudo-class nodes. Merged properties can be expanded or merged back by clicking on the arc.

Filter: The user can filter displayed graph elements by selecting classes of interest via a set of checkboxes provided next to the visualization canvas. All pseudo-classes are tied to a single checkbox. Only the nodes of selected classes and the edges between them are displayed. Additionally, one can set all adjacent nodes of a given node to be visible by clicking on the node and thus updating the filter.

Details-on-demand: An infobox containing information about the selected property prevalence and histogram of its cardinalities is displayed next to the graph visualization as shown in Figure 2a. It is also possible to interactively query the original dataset in this mode. The user can highlight nodes and edges that will serve as a subgraph pattern in the SPARQL query. The underlying query string is displayed and dynamically updated as the user interacts with the visualization. Details on SPARQL creation are provided in a further section. Some segments of the generated SPARQL query might be highlighted in some cases as seen in Figure 2b. The user can click on the text and tweak the query by selecting alternative auto-generated segment. Results are displayed in tabular form under the visualization and can be exported to a CSV (comma-separated values) file.

C. SPARQL query builder

Query builder works with an active selection in the visualization. Variables in the SPARQL query use the same name as the nodes they represent, thus the same variable name is always used for one node. Highlighted subgraph edges and nodes are collected and the first node to be traversed is chosen. The subgraph is traversed by depth-first approach with edges of datatype properties having priority. For each traversed edge, a new triple is added to the WHERE clause declaring the relation between the two variables via URI of the edge. A triple definition of variable type is added if the *rdf:type* of the variable has not been defined before and the variable is not literal. This default behavior works well when using only properties with cardinality of 1.


Available classes

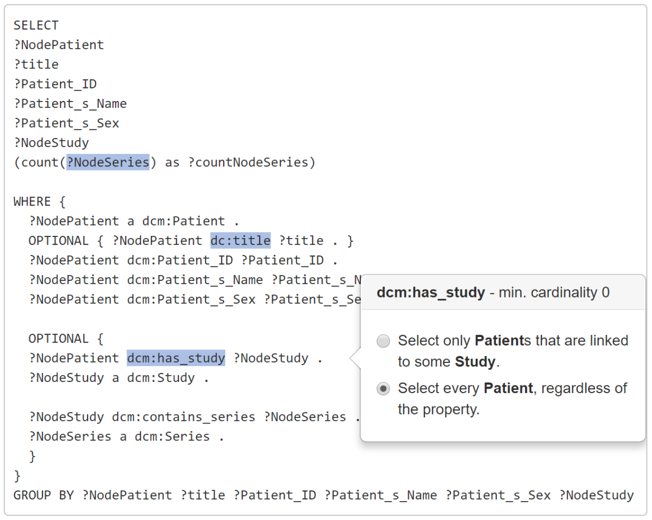
- [✓] Literal
- [✓] dcm:CT_Image
- [ ] dcm:Frame_of_Reference
- [✓] dcm:Patient
- [✓] dcm:Series
- [✓] dcm:Study
- [ ] mre:Dataset
- [ ] mre:DataSource
- [✓] nfo:FileHash

Selection detail

dcm:Study (4)
–[**dcm:contains_series** (67)]¬
dcm:Series (67)

Cardinality histogram:

(a)


```
SELECT
?NodePatient
?title
?Patient_ID
?Patient_s_Name
?Patient_s_Sex
?NodeStudy
(count(?NodeSeries) as ?countNodeSeries)

WHERE {
  ?NodePatient a dcm:Patient .
  OPTIONAL { ?NodePatient dc:title ?title . }
  ?NodePatient dcm:Patient_ID ?Patient_ID .
  ?NodePatient dcm:Patient_s_Name ?Patient_s_N
  ?NodePatient dcm:Patient_s_Sex ?Patient_s_Se

  OPTIONAL {
  ?NodePatient dcm:has_study ?NodeStudy .
  ?NodeStudy a dcm:Study .

  ?NodeStudy dcm:contains_series ?NodeSeries .
  ?NodeSeries a dcm:Series .
  }
}
GROUP BY ?NodePatient ?title ?Patient_ID ?Patient_s_Name ?Patient_s_Sex ?NodeStudy
```

dcm:has_study - min. cardinality 0

- ○ Select only **Patient**s that are linked to some **Study**.
- ● Select every **Patient**, regardless of the property.

(b)

Figure 2. Interactive query generation. Selection detail box in lower left part of (a) provides information about a single graph edge. Numbers in brackets indicate the prevalence in the dataset. The histogram shows cardinality spectrum of the property. The query in (b) is generated based on the highlighted subgraph from (a). There are three highlighted opportunities to tweak the query via popover box.

When dealing with properties having minimum cardinality of 0, the part of SPARQL query is highlighted, thus visually prompting the user to make a decision on a way this property should be handled. Without further changes, the property is considered mandatory and any resource that does not have the property assigned will not match the pattern and will not be present in the result. An alternative to this is to wrap the triple with OPTIONAL clause. This is sufficient for datatype properties, but for object properties it might lead to cartesian product evaluation in case the potentially unbound variable is used further in the query. To prevent this behavior, entire query pattern generated by traversing the target node and its children is wrapped with the optional clause.

All the variables declared in pattern statements are inserted into SELECT clause of the query. Clicking on the variable name will insert a simple filter at the end of the where block, which could be further edited by the user. This allows for additional conditioned selection in the query, thus limiting the result space.

Query builder displays a warning text in two specific situations. First, if the current selection does not form a pattern of connected graph. Proceeding with the query execution would result in a cartesian product of the disjoint components, which is undesirable in most cases. There is no clear-cut solution to this problem and the user needs to change the subgraph selection manually. The second case is when multiple edges between two nodes are selected. Only instances, that has all the properties leading to the same instance will be in the resulting selection. This is usually not the behavior the user wants, but it might be desirable in some cases.

III. Implementation

The prototype is implemented using client-server architecture. Server part is written in Java using Spring Framework [11] for web communication and Apache Jena [12] for RDF manipulation. The server accepts either an RDF file in serialization format supported by Jena or an URL address of SPARQL endpoint that implements the SPARQL 1.1 Graph Store Protocol [13]. Currently, the server application analyses the contents of the dataset programatically via Jena API. This provides parallel processing support of the input model if needed in the future. The chosen approach is in contrast to other contemporary solutions, which gains the graph metadata purely by analytically querying of public SPARQL endpoints built upon the datasets. Our metadata collection workflow could be transformed to a batch of analytical queries as Jena framework enables to query loaded RDF models, as well as remote SPARQL endpoints. The resulting dataset description metadata are returned as a JavaScript Object Notation (JSON) file. The server also provides an interface for SPARQL SELECT query execution above the dataset.

The client is written in HTML5 and utilizes D3 JavaScript library [14] to create a force-directed graph based interactive visualization. The visualization uses metadata JSON file provided by the server. The client is platform independent and only requires modern web browser with enabled JavaScript support.

IV. Discussion

The main focus of this project was on creating a tool that would assist the user in filtering and transforming the RDF dataset into tabular data structure for further use in other analytical software. The visualization interactions were inspired by the visual interface of Microsoft Access Query Designer, which users can successfully operate with only a basic understanding of relational database theory. Entity-Relationship model is the fundamental diagram in relational databases to understand the structure of dataset and Query Designer uses it to define joins between tables. In RDF case, we have used *rdf:type* values of resources as the entity labels and constructed an aggregate graph explaining the relations present in the dataset between such entities. By highlighting segments of the graph, the user intuitively defines a query projection.

There are minimum and maximum cardinalities calculated in preprocessing phase, which are used during query generation to determinate the need for optional clause and aggregation definitions. These values could in theory be gained from ontological definitions of properties, however we would have to accept the assumptions that the ontologies are available, the ontological restrictions are defined and data respect the ontologies. To allow the user to work with datasets lacking the ontological description, we have decided to use the ontologies only as a supplementary source of annotations. Similarly, the entire aggregate graph could in theory be created based on the ontological definitions of classes, properties and their domain and range spaces, but again we have chosen to rather describe the exact state of data as presented in an input file.

The prototype solution uses a generic approach that is applicable for any RDF file or SPARQL endpoint allowing access via Graph Store Protocol. Users are thus able to explore and extract data from various sources and are not limited to predefined databases, which is a common issue in other SPARQL builder solutions [7][8]. To support on-demand viewing of data and export of projected table, the server needs to store the input datasets. Currently, the entire datasets are read into Jena in-memory model which is a major scalability concern for future development because Java Virtual Machine memory capacity is limited.

Preprocessing of the dataset is computationally expensive operation and it scales linearly with the triple count. Depending on the input dataset size, there might be a noticeable delay before the user can interact with the visualization and it might be worth searching for possibilities on how to show results of partially processed dataset to the user. The visualization [9] avoids the problem by running the process in regular interval and having a cached results available on-demand. This is not applicable in our case, since user can input arbitrary datasets.

The quality and clarity of visualization depends mainly on two factors, the number of different RDF types and their hierarchy. With increasing node count, the visualization spans larger area and it becomes harder for users to grasp the overall shape or even find property paths between the nodes of interest. In such cases, it might be useful to incorporate a filter method that would only display the two selected nodes and the property path between them similarly as SPARQL Builder [7] does. User could then expand the neighboring nodes and the two nodes would serve as starting point for further exploration. The problem with hierarchy could be partially alleviated by using the drill down and drill up operations on the hierarchy of RDF types as was done in [9]. However, this would conceal some of the types and a user not knowing about the subtypes and supertypes of searched term might be confused.

V. Conclusion and Future Work

We have introduced the prototype software solution for interactive RDF data exploration and transformation to tabular format in this paper. The proposed interactions are in accordance with the overview first, zoom and filter, then details-on-demand principle. SPARQL generator is a part of the solution and is used for building basic SELECT queries with support of optional blocks, filter clause and group by clause. The user is prompted to choose from several offered query snippets rather than freely edit the query string, and is not expected to know the query language in detail.

The solution currently works well for datasets containing low number of RDF classes, but it needs to be improved before deploying and integrating in other systems. The solution might assist doctors in extracting and transforming relevant data for their clinical research or it might help in initial orientation in data structure of information system. It might find its use in applications for manipulating RDF data, e.g., an RDF editor.

Future work consists of extending the system architecture to include a proper database layer to store the input datasets and metadata generated during preprocessing. We will also look into the ways to optimize the preprocessing phase and to gather the model metadata by a batch of analytical SPARQL queries, thus allowing the visualization to be run against SPARQL endpoints built upon the datasets. Later on, we would like to perform usability testing.

Acknowledgment

The work was supported from European Regional Development Fund-Project “Application of Modern Technologies in Medicine and Industry” (No. CZ.02.1.01/0.0/0.0/17_048/0007280) and by Ministry of Education, Youth and Sports of the Czech Republic, project PUNTIS (LO1506) under the program NPU I and project SGS-2016-018.

References

[1] I. Merelli, H. Pérez-Sánchez, S. Gesing, and D. DAgostino, “Managing, Analysing, and Integrating Big Data in Medical Bioinformatics: Open Problems and Future Perspectives,” BioMed research international, vol. 2014, 2014.

[2] T. Berners-Lee, “Linked Data - Design Issues,” 2010, URL: https://www.w3.org/DesignIssues/LinkedData.html [retrieved: August, 2018].

[3] J. Schwarz et al., “Inflammatory bowel disease incidence in Czech children: A regional prospective study, 2000-2015,” World journal of gastroenterology, vol. 23, no. 22, 2017, p. 4090.

[4] P. Rajbhandari, R. Gosai, R. C. Shah, and K. Pramod, “Semantic Web in Medical Information Systems,” International Journal of Advances in Engineering & Technology, vol. 5, no. 1, 2012, p. 536.

[5] M. Junghanns, A. Petermann, M. Neumann, and E. Rahm, “Management and Analysis of Big Graph Data: Current Systems and Open Challenges,” in Handbook of Big Data Technologies. Springer, 2017, pp. 457–505.

[6] Berners-Lee et al., “Tabulator: Exploring and Analyzing Linked Data on the Semantic Web,” in Proceedings of the 3rd International Semantic Web User Interaction Workshop, vol. 2006. Citeseer, 2006, p. 159.

[7] A. Yamaguchi et al., “SPARQL Builder: Constructing SPARQL Query by Traversing Class-Class Relationships for Life Science Databases,” in JIST (Workshops & Posters), 2016, pp. 58–61.

[8] D. Schweiger, Z. Trajanoski, and S. Pabinger, “SPARQLGraph: a web-based platform for graphically querying biological Semantic Web databases,” BMC bioinformatics, vol. 15, no. 1, 2014, p. 279.

[9] F. Florenzano, D. Parra, J. Reutter, and F. Venegas, “An Interactive Visualisation for RDF Data,” in International Semantic Web Conference, 2016.

[10] B. Shneiderman, “The Eyes Have It: A Task by Data Type Taxonomy for Information visualizations,” in The Craft of Information Visualization. Elsevier, 2003, pp. 364–371.

[11] R. Johnson et al., “The Spring Framework–Reference Documentation,” Interface, vol. 21, 2004, p. 27.

[12] J. J. Carroll et al., “Jena: Implementing the Semantic Web Recommendations,” in Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters. ACM, 2004, pp. 74–83.

[13] C. Ogbuji, “SPARQL 1.1 Graph Store HTTP Protocol,” W3C REC, vol. 21, 2013.

[14] M. Bostock, V. Ogievetsky, and J. Heer, “D^3 Data-Driven Documents,” IEEE transactions on visualization and computer graphics, vol. 17, no. 12, 2011, pp. 2301–2309.