Introducing RDF Graph Summary with application to Assisted SPARQL Formulation

Stéphane Campinas*, Thomas E. Perry*, Diego Ceccarelli†, Renaud Delbru* and Giovanni Tummarello*‡

*Digital Enterprise Research Institute
National University of Ireland, Galway
Galway, Ireland
firstname.lastname@deri.org

†ISTI-CNR
Dipartimento di Informatica
Università di Pisa
diego.ceccarelli@isti.cnr.it

‡Fondazione Bruno Kessler
Trento, Italy

***Abstract*—One of the reasons for the slow adoption of SPARQL is the complexity in query formulation due to data diversity. The principal barrier a user faces when trying to formulate a query is that he generally has no information about the underlying structure and vocabulary of the data.**

In this paper, we address this problem at the maximum scale we can think of: providing assistance in formulating SPARQL queries over the entire Sindice data collection - 15 billion triples and counting coming from more than 300K datasets. We present a method to help users in formulating complex SPARQL queries across multiple heterogeneous data sources. Even if the structure and vocabulary of the data sources are unknown to the user, the user is able to quickly and easily formulate his queries. Our method is based on a summary of the data graph and assists the user during an interactive query formulation by recommending possible structural query elements.

***Keywords*-SPARQL; Graph; Analytics; Summary; Query Formulation; Recommendation;**

I. Introduction

One of the reasons for the slow adoption of SPARQL is the complexity in query formulation. While the language itself might not be more complex than SQL, formulating correct and complex queries is a challenge for the very reason that makes RDF and Web Data fascinating, i.e., the data is very diverse and does not follow any predefined structure and vocabulary. However, the usefulness of Web Data is clearly dependent on the ease by which data can be consumed by others. In order to use such a massive amount of data, people must be able to easily and effectively query data across multiple web data sources. Therefore, one of the main challenges is to develop effective methods to help users formulate complex queries across possibly unknown data sources.

The principal barrier a user faces when trying to formulate a query is that he generally has no information about the underlying structure and vocabulary of the data. While this information is indispensable, it is frequently missing and difficult to obtain due to the heterogeneity of Web Data. Another obstacle in writing effective queries that combines information across multiple data sources is the lack of knowledge about how these data sources are interconnected.

A. Contribution

In this paper we address this problem at the maximum scale we can think of: providing assistance in formulating SPARQL queries over the entire Sindice data collection - 15 billion triples and counting coming from more than 300K datasets. We present a method to help users in formulating complex SPARQL queries across multiple heterogeneous data sources. Even if the structure and vocabulary of the data sources are unknown to the user, the user is able to quickly and easily formulate his queries without having to explore and study each data source separately. For users having a good knowledge about the data, the querying experience is improved and the query writing time reduced. Our method is based on a summary of the data graph and assists the user during an interactive query formulation by recommending possible structural query elements. Our main contributions are: 1) We formally define our model for the data graph and the data graph summary in Section III; 2) We develop an algorithm to automatically recommend graph summary elements to the user during query formulation based on the current query state in Section IV.

II. Background

Schema summarisation [9] is the task of automatically generating a more concise representation of a data schema by presenting only important schema elements while achieving broad information coverage. Such summary is principally used for visual exploration of the data schema and for query discovery. While this summary approach is similar in essence with our approach, the main differences are: 1) we present a model for data graph summarisation while they present a model for schema summarisation, i.e., we generate the summary directly from the data graph and are not relying on an existing schema; 2) our model supports the notion of data sources and enables the summarisation across multiple data sources.

With respect to query formulation, the closest approach to our assisted SPARQL editor is MashQL [4] which is developed for querying graph-based data (e.g., RDF) without prior knowledge about the structure and vocabulary of the data collection. However, MashQL defines its own query language while our approach is directly integrated within SPARQL. Also, even if MashQL enables the formulation of query across multiple data sources, it requires the selection of data sources prior to the query formulation, while our approach can recommend possible datasets to query through the *GRAPH* operator of SPARQL. In other words, our approach can recommend possible relationships across datasets. MashQL's approach is very similar to structural summary approaches such as [3], [7], [6], [5] and can recommend both structural and content elements. The system relies on a "Graph Signature" which consists of two structural summaries of the data graph: the "O-Signature" is a summary of the original graph such that nodes that have the same outgoing paths are grouped together and the "I-Signature" summarises a graph by grouping nodes that have the same incoming paths. Their approach requires specific indexing methodologies to provide fast response times. In contrast, our data graph summary can be converted into RDF and can be queried efficiently by any RDF database since the data graph summary is very small. There are many other approaches to query formulation from form-based query to visual query editor. A review of the main approaches and how they relate to MashQL can be found in [4].

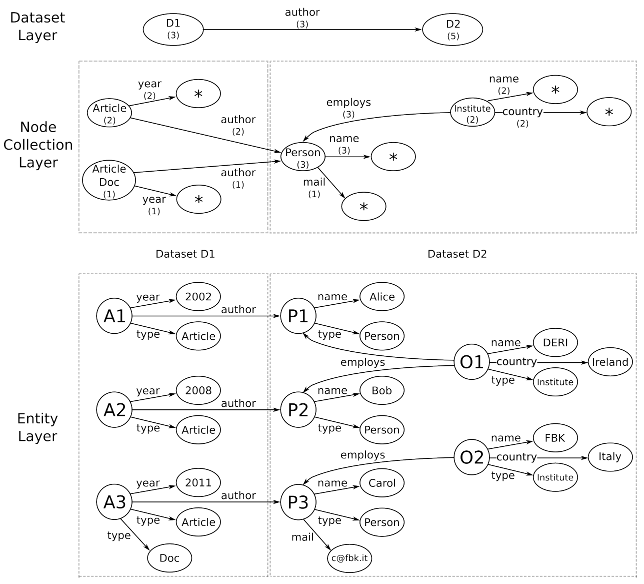


Figure 1: A three layer representation of our Web Data Graph Summary model. On the node collection layer, nodes labelled with a star $*$ represents blank collections.

III. A Model for Web Data Graph Summary

For the purpose of this work, we need a generic graph model that supports the various scenarios found on the Web of Data. First, we define a labeled directed graph model that covers the various type of Web Data sources, i.e., Microformats, RDFa, RDF databases, etc. We then define the concept of "node collections" which is a set of nodes sharing similar characteristics. The Figure 1 depicts the layers of this graph model. The entity and dataset layers represent datasets, entities and their relationships. The node collection layer extends the model introduced in [2] by representing node collections and their relationships.

1) Data Graph: Let V be a set of nodes and A a set of labelled edges. The set of nodes V is composed of two non-overlapping sets: a set of entity nodes V^E and a set of literal nodes V^L. Let $\mathcal{L}$ be a set of labels composed of a set of node labels $\mathcal{L}^V$, a set of edge labels $\mathcal{L}^A$ and a set of dataset labels $\mathcal{L}^D$.

Web Data is defined as a graph G over $\mathcal{L}$, and is a tuple $G = \langle V, A, l_v \rangle$ where $l_v : V \to \mathcal{L}^V$ is a node labelling function. The set of labelled edges is defined as $A \subseteq \{(e, \alpha, v) | e \in V^E, \alpha \in \mathcal{L}^A, v \in V\}$. The components of an edge $a \in A$ will be denoted by $source(a)$, $label(a)$ and $target(a)$ respectively.

2) Dataset: A dataset is defined as a subgraph of the Web Data graph:

Definition 3.1 (Dataset): A dataset D over a graph $G = \langle V, A, l_v \rangle$ is a tuple $D = \langle V_D, A_D, \mathcal{L}_D^V, l_v \rangle$ with $V_D \subseteq V$, $A_D \subseteq A$ and $\mathcal{L}_D^V \subseteq \mathcal{L}^V$.

3) Terminal Nodes: We identify a node that does not have outgoing edges as *terminal*. Let V^T be a set of terminal nodes. V^T and V^E (resp. V^L) are not mutually exclusive and their nodes may overlap.

Definition 3.2 (Terminal Node): A node $v \in V$ is said to be terminal if $\nexists a \in A | source(a) = v$.

4) Node Collection: A *node collection* is a set of nodes sharing particular characteristics. Let $\mathcal{C}$ be a set of node collections. The set of node collections $\mathcal{C}$ is composed of two non-overlapping sets: a set of *entity collections* $\mathcal{C}^E$ and a set of *blank collections* $\mathcal{C}^B$. Let $\mathcal{L}^{\mathcal{C}}$ be a set of node collection labels and $l_c : \mathcal{C} \to \mathcal{L}^{\mathcal{C}}$ a node collection labelling function.

An *entity collection* is a set of entity nodes within the same dataset sharing similar characteristics. In this paper, we consider two types of characteristics: class-based and attribute-based. The class-based characteristic groups together entity nodes which share identical classes, such as the class Person in the node collection layer in Figure 1. The resulting entity collection C is labelled with the set of class labels, i.e., $l_c(C) \subseteq \mathcal{L}^V$. The attribute-based characteristic groups together entity nodes which share the same set of attribute labels. Attribute-based grouping is necessary when an entity node does not have a class definition. The assumption is that entities having similar attributes are likely to belong to the same class. The resulting entity collection C is labelled with the set of attribute labels, i.e., $l_c(C) \subseteq \mathcal{L}^A$.

Definition 3.3 (Entity Collection): An entity collection $C \in \mathcal{C}^E$ over a dataset $D = \langle V_D, A_D, \mathcal{L}_D^V, l_v \rangle$ is a tuple $C = \langle V_C, I_C, l_c \rangle$ with $V_C \subseteq V_D^E$ and $I_C : V \to \{0, 1\}$ an indicator function for node collections such that $\forall v \in V_C, I_C(v) = 1$

We define two different set indicator functions for entity collections. The first indicator function I_C^c is based on the *class definition* of an entity node. The class definition of an

Table I: The set of edge labels $\mathcal{L}^{A_c}$ that defines a class attribute. The reported frequency is relative to the Sindice data collection.

Label	***Frequency***
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	123,994,777
http://opengraphprotocol.org/schema/type	61,202,581
http://ogp.me/ns#type	17,184,227
http://opengraph.org/schema/type	2,443,773
http://purl.org/dc/elements/1.1/type	11,367,218
http://dbpedia.org/property/type	120,044
http://dbpedia.org/ontology/type	65,796

entity node $v \in V_D^E$ is defined by its set of *class node* labels. A class node of v is defined as:

Definition 3.4 (Class Node): A node $v' \in V_D$ is said to be a class node of an entity node $v \in V_D^E$ if $\exists a \in A_D | label(a) \in \mathcal{L}^{A_c}, source(a) = v, target(a) = v'$.

The set $\mathcal{L}^{A_c}$ is a set of "class attribute" labels such as the ones in Table I. These attribute labels have been extracted from the Sindice data collection [1] and are generally used to define the class of an entity node. We denote by $class(v) \subseteq \mathcal{L}^V$ the set of labels of class nodes for an entity node $v \in V_D^E$. The class-based indicator function I_C^c is defined as:

Definition 3.5 (Class Indicator): Given a dataset D, an entity node $v \in V_D^E$ and an entity collection C,

$$I_C^c(v) = \begin{cases} 1 & \text{if } class(v) = l_c(C) \\ 0 & \text{otherwise} \end{cases}$$

The second indicator function I_C^a is based on the *attribute definition* of an entity node. The attribute definition of an entity node $v \in V_D^E$ is defined by its set of attribute labels, i.e., the labels of its outgoing edges. We denote by $attribute(v) \subseteq \mathcal{L}^A$ the set of labels of outgoing edges for an entity node $v \in V_D^E$.

Definition 3.6 (Attribute Indicator): Given a dataset D, an entity node $v \in V_D^E$ and an entity collection C,

$$I_C^a(v) = \begin{cases} 1 & \text{if } attribute(v) = l_c(C) \\ 0 & \text{otherwise} \end{cases}$$

We define a *blank collection* as a set of terminal nodes within a same dataset being target of edges with the same label and with the same entity collection source. The label for a blank collection is a tuple $\langle \mathcal{L}^C, \mathcal{L}^A \rangle$ with $l_1 \in \mathcal{L}^C$ the label of the source entity collection and $l_2 \in \mathcal{L}^A$ the label of the edge.

Definition 3.7 (Blank Collection): A blank collection $C \in \mathcal{C}^B$ over a dataset $D = \langle V_D, A_D, \mathcal{L}_D^V, l_v \rangle$ is a tuple $C = \langle V_C, I_C^b, l_c \rangle$ with $V_C \subseteq V_D^T$ and $I_C^b : V \rightarrow \{0, 1\}$ an indicator function for blank collections such that $\forall v \in V_C, I_C^b(v) = 1$.

The indicator function I_C^b for blank collections is defined as follows:

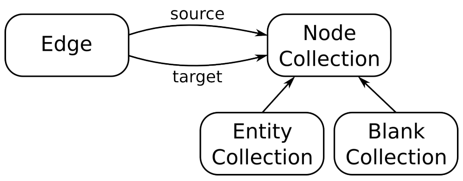


Figure 2: RDF vocabulary for the data graph summary.

Definition 3.8 (Blank Indicator): Given a dataset D, a terminal node $v \in V_D^T$, an entity collection C_i and a blank collection C_j,

$$I_{C_j}^b(v) = \begin{cases} 1 & \text{if } \exists a \in A_D | target(a) = v, \\ & \quad I_{C_i}^c(source(a)) = 1 \text{ or } I_{C_i}^a(source(a)) = 1, \\ & \quad l_c(C_j) = \langle l_c(C_i), label(a) \rangle \\ 0 & \text{otherwise} \end{cases}$$

5) Node Collection Edge: We group edges into *c-linksets* on the node collection layer. For example, in Figure 1 the edges *employs* between institutes (i.e., O_1 and O_2) and people (i.e., P_1, P_2 and P_3) are aggregated to form the linkset $L_{employs,Institute,Person}$ on the node collection layer.

Definition 3.9 (C-Linkset): Given two entity collections C_i and C_j, we denote a linkset between C_i and C_j with $L_{\alpha,i,j} = \{a | label(a) = \alpha, source(a) \in C_i, target(a) \in C_j\}$ the set of edges having the same label α and connecting the node collection C_i to the node collection C_j.

A. Data Graph Summary

In the rest of the paper, we denote by data graph summary an instantiation of the node collection layer. The node collection layer is composed of node collections and c-linksets. In order to make this data graph summary compatible with RDF databases and SPARQL, we introduce a RDF vocabulary to represent it as a RDF graph. This RDF model has some similarities with and can be mapped to the VoID[1] vocabulary.

The RDF vocabulary, pictured in Figure 2, enables to capture the metadata about the nodes and edges of the node collection layer. The vocabulary defines two main classes: *Edge* and *NodeCollection*. A *NodeCollection* can be subclassed into either an *EntityCollection* or a *BlankCollection*. Edges and NodeCollections are associated through the properties *source* and *target*. Edges and NodeCollections also have the properties 1) *origin* which indicates the dataset in which they have been defined; 2) *cardinality* which indicates the cardinality of their respective c-linkset or node collection; 3) *label* which defines the label(s) associated to an edge or a node. Using this RDF model, the indicator functions I_C^c, I_C^a and I_C^b can be easily and efficiently implemented with SPARQL in order to retrieve all node collections that match a subset of the labels.

The data graph summary is pre-computed offline using Hadoop. Computing the summary and the associated statistics on the fly turned out to be impracticable due to the

[1]VoID: http://www.w3.org/TR/void/

computational complexity as it required large joins and many data aggregates. Our Hadoop-based implementation requires a day of computation on a cluster of 10 machines on the full Sindice dataset (15 billion triples). The resulting data graph summary is composed of 100M edges.

IV. Query Recommendation

When a user formulates a SPARQL query, the user is in fact trying to summarise his information needs. However, this task can be very difficult and time consuming as it requires the user to have a good knowledge of the structure and vocabulary of the dataset he is trying to query. This task becomes even more complex if the user is trying to formulate a query across multiple data sources. In order to gain such knowledge, the user must explore and investigate the data itself before querying it. To save the user from such a tedious work we develop an *Assisted SPARQL Editor*, an application that leverages the data graph summary to help the user into effectively formulating complex SPARQL queries even without prior knowledge about the structure and vocabulary of the data sources. In this section, we first give an overview of the possible recommendations supported by the SPARQL editor. Then we introduce some of the main concepts in SPARQL before explaining how the current state of the SPARQL query is used to query the data graph summary and to retrieve possible structural query elements for recommendation.

A. Recommendations Overview

The Assisted SPARQL Editor supports four kinds of recommendations: class, predicate, relationships between variables and named graphs. Examples of such recommendations are pictured in Figure 3. Recommendations of entity node labels as well as literal node labels are not supported since such content data is discarded from the data graph summary.

During query formulation, the assisted editor provides one of these four different types of recommendations to the user based on the state of the edited query. The state of the edited query is composed of an incomplete graph pattern and the cursor position. The cursor position materialises the *Point Of Focus* (POF), i.e., the unknown element of the graph pattern for which the user requests recommendations.

Figure 3a depicts the recommendations of possible classes for a variable. Given that the variable *?Article* is associated to a predicate *akt:hasAuthor*, the system will only recommend classes that are mentioned with this predicate in the data graph. Figure 3b depicts the recommendation of additional predicates for the class *akt:Article-Reference* which co-occur with the property *akt:hasAuthor*. In the case of class recommendation, elements of the node collection labels are retrieved from the data graph summary and presented as possible recommendations. In the case of predicate recommendation, c-linkset labels $\mathcal{L}^A$ are retrieved from the data graph summary and presented to the user. Figure 3c depicts recommendations of possible relationships between two variables. The current implementation of the Assisted SPARQL Editor only supports direct relationships, but in future work we expect to provide recommendations about possible shortest paths between two variables. In Figure 3d, possible named graphs are recommended to the user. In this case, dataset labels $\mathcal{L}^D$ are retrieved from the data graph summary.

B. SPARQL Graph Pattern

In this section, we introduce the main concepts of SPARQL that are used later in the description of our recommendation engine. SPARQL is the standard query language for RDF data and is based around graph pattern matching. Triple Pattern (TP) [8] is the building block in SPARQL:

Definition 4.1 (Triple Pattern): A triple pattern is a tuple $t \in (\mathcal{L}^{V^E} \cup Var) \times (\mathcal{L}^A \cup Var) \times (\mathcal{L}^V \cup Var)$ where $\mathcal{L}^{V^E}$ is the set of the entity node labels and Var is an infinite set of variables.
The components of a triple pattern t are denoted $subject(t)$, $predicate(t)$ and $object(t)$, respectively.

Triple patterns can be combined into a Basic Graph Pattern (BGP) [8]. More complex graph patterns can be formed by combining BGPs in various ways [8]:*Group Graph Pattern*, *Optional Graph Pattern*, *Alternative Graph Pattern* and *Patterns on Named Graphs*.

A SPARQL query can be translated into an Abstract Syntax Tree (AST). The AST is a tree structure composed of all the logical operators of the query and where leaf nodes are triple patterns to be evaluated. Such an AST is the data structure used by the system to translate the current user needs into possible recommendations. In our implementation, the AST can contain an incomplete TP and a special symbol '$<$' to indicate the POF.

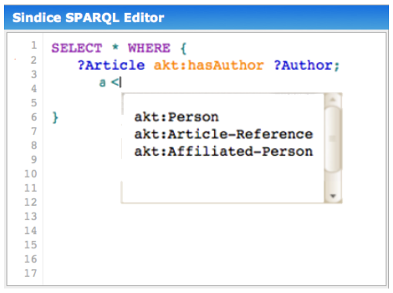
C. From Data Graph to Data Graph Summary

In order to suggest the possible structural elements to the user with respect to the current state of his query, we need to translate the AST of the query into another AST compatible with the data graph summary. This new AST is then evaluated on the data graph summary and the possible structural elements are retrieved and presented to the user. The AST translation is performed in three steps: 1) transformation of the POF symbol '$<$' into a variable to project as the query solution; 2) removal of content elements from the AST; and 3) mapping of triple patterns into *summary patterns*. Figure 4 depicts a translation of a data graph query (Figure 4a) to a data graph summary query (Figure 4b).

1) Summary Pattern: Similar to Triple Patterns in SPARQL, Summary Patterns (SPs) are the building blocks to construct a data graph summary query.

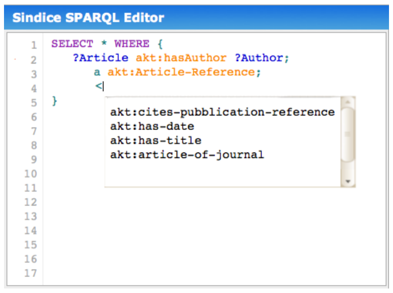
Definition 4.2 (Summary Pattern): A *Summary Pattern* is a tuple $t \in (\mathcal{L}^C \cup Var) \times (\mathcal{L}^A \cup Var) \times (\mathcal{L}^C \cup Var)$.

With respect to our RDF data model of the data graph summary, such a summary pattern is translated into a SPARQL BGP. For example, given the SP



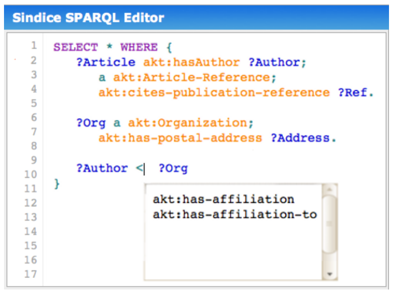
```
SELECT * WHERE {
   ?Article akt:hasAuthor ?Author;
      a <|

}
```

(a) Class



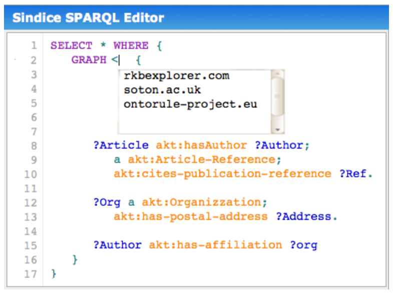
```
SELECT * WHERE {
   ?Article akt:hasAuthor ?Author;
      a akt:Article-Reference;
      <|
}
```

(b) Predicate



```
SELECT * WHERE {
   ?Article akt:hasAuthor ?Author;
      a akt:Article-Reference;
      akt:cites-publication-reference ?Ref.

   ?Org a akt:Organization;
      akt:has-postal-address ?Address.

   ?Author <|  ?Org
}
```

(c) Relationships



```
SELECT * WHERE {
   GRAPH <|  {



      ?Article akt:hasAuthor ?Author;
         a akt:Article-Reference;
         akt:cites-publication-reference ?Ref.

      ?Org a akt:Organizzation;
         akt:has-postal-address ?Address.

      ?Author akt:has-affiliation ?org
   }
}
```

(d) Named Graphs

Figure 3: Overview of the possible recommendations for the http://www.rkbexplorer.com dataset depending on the Point of Focus. The Point of Focus is displayed by the green angle bracket $<$.

```
1   SELECT * WHERE {
2    ?a a :Article ;
3       :title ?t .
4
5
6
7    ?i a :Institute ;
8       :employs ?p .
9
10
11
12   ?p :name "Renaud" ;
13
14
15
16      <
17
18
19  }
```

(a) SPARQL query over the data graph. The Point of Focus is depicted by the angle bracket $<$.

```
1   SELECT ?POF WHERE {
2    ?a :label :Article .
3    ?x :label :title ;
4       :source ?a ;
5       :target ?t .
6
7    ?i :label :Institute .
8    ?y :label :employs ;
9       :source ?i ;
10      :target ?p .
11
12   ?z1  :label :name ;
13        :source ?p ;
14        :target ?_z1 .
15
16   ?z2  :label ?POF ;
17        :source ?p ;
18        :target ?_z2 .
19  }
```

(b) SPARQL query over the data graph summary. The Point Of Focus defines the solution set, i.e., the variable $?POF$ in the SELECT clause.

Figure 4: Mapping from a data graph query to a data graph summary query.

$< Institute, employs, ?p >$, the corresponding SPARQL BGP is equal to the BGP in Figure 4b from the line 7 to line 10.

2) Projection of POF: The first step consists of defining the variable to project as the query solution. In the TP containing the POF, we transform the POF symbol into a variable $?POF$ and complete that TP with a wildcard variable if needed (we denote by wildcard variable a variable that is unique in the query). For example in Figures 4a and 4b at line 16, the POF symbol '$<$' is translated into the variable $?POF$, and the wildcard variable $?\_z2$ is added to complete the triple pattern. The initial *Query Form* [8] of the AST is replaced by a projection of the POF variable using the *SELECT* form.

3) Removal of content elements: The second step consists in removing all content elements from the AST. A content element is an element that describes a specific aspect of an entity, thus it does not inform about its structure. We replace with a wildcard variable Literals and URIs that appear in a TP at a subject or object position, except if the TP is a *Class Triple Pattern* (CTP). In that case, only the element at the subject position is replaced by a wildcard variable. For instance, the literal "Renaud" in Figure 4a is replaced by the variable $?\_z1$ in Figure 4b.

Definition 4.3 (Class Triple Pattern): A *Class Triple Pattern* is a triple pattern t such as $predicate(t) \in \mathcal{L}^{A_c}$.

4) Mapping: The third step consists in mapping all triple patterns into summary patterns, according to the following two rules. If the triple pattern $< var_s, p, o >$ is a CTP, then it is replaced with a new triple pattern $< var_s, label, o >$. Otherwise, the triple pattern $< var_s, p, var_o >$ is replaced with the following BGP:

1) A new wildcard variable var_x, representing an edge node x, is created;

2) A triple pattern $< var_x, source, var_s >$ is created to set the source of the edge;
3) A triple pattern $< var_x, target, var_o >$ is created to set the target of the edge;
4) A triple pattern $< var_x, label, p >$ is created to set the label of the edge;

In the case where the triple pattern $< var_s, p, var_o >$ belongs to a *Graph Graph Pattern* [8], i.e., a group graph pattern associated to a named graph URI g with the *GRAPH* operator, then a triple pattern $< var_x, origin, g >$ is added to the previously defined BGP to set the named graph of the edge. However, if the named graph is the POF variable, the triple pattern $< var_x, origin, var_{POF} >$ is created instead in order to restrict each edge to be from the same dataset and to bind the dataset label to the POF variable.

Applying these mapping rules on the query of Figure 4a, the first TP $< var_a, a, Article >$, being a CTP, is translated into the SP $< var_a, :label, Article >$. The second TP $< var_a, :title, var_t >$ is translated into the BGP from line 3 to line 5 of the query displayed in Figure 4b.

D. Recommendation Scope

In certain cases during the formulation of a query, the query may contain multiple BGPs, among which one does not return any result. The system will therefore no longer produce recommendations, as the evaluation of the data graph summary query will also not produce any results. However, this can be interpreted incorrectly by the user since he might believe that the dataset does not contain any other information. In order to minimise this issue, we introduce the notion of *recommendation scope*.

The *recommendation scope* helps to reduce the extent of the area that is relevant for the recommendation. Instead of taking into account the full SPARQL query, the recommendation engine will take only a relevant subset. The recommendation scope is defined recursively by including all the triple patterns with a path to the POF variable. A *breadth-first search* algorithm on the query, starting on the POF node, is performed in order to find all the graph components that are connected to the POF. All the graph components that are not connected to the POF are removed. This prevents non-relevant (to the POF) triple patterns from limiting the recommendations. For example, in Figure 4a, the recommendation scope does not contain the first BGP since it is not connected to the POF variable, whether directly or indirectly. Another possible solution for this issue is to provide to the user an estimation of the cardinality of TPs or BGPs that can lead to an empty result set. This is possible given the cardinality information provided by the data graph summary. We are currently planning to investigate this solution for future works.

V. Conclusion

In this paper, we have addressed the problem of SPARQL query formulation over a very large data collection containing 15 billions triples coming from over 300K datasets. We have proposed a formal model for a data graph summary that captures the necessary information to assist users in formulating queries across multiple data sources. We have presented an algorithm to recommend to the user the possible structural query elements with respect to the current state of his query. This paper presents an early prototype and preliminary version of our system. Future works include: 1) the investigation of ranking algorithms for structural elements in the data graph summary; 2) the recommendation of possible shortest paths between two variables. A prototypical demo of the assisted SPARQL editor is available at http://hcls.sindicetech.com/sparql-editor/, along with a screencast demonstrating the editor in action. This demo is built on top of the open-source Flint SPARQL editor.

Acknowledgements

This material is based upon works supported by the European FP7 projects LOD2 (257943) and SEMLIB (FP7-SME-2010-1-SEMLIB 262301) and by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

References

[1] S. Campinas, D. Ceccarelli, T. E. Perry, R. Delbru, K. Balog, and G. Tummarello. The Sindice-2011 Dataset for Entity-Oriented Search in the Web of Data. July 2011.

[2] R. Delbru, S. Campinas, and G. Tummarello. Searching Web Data: an Entity Retrieval and High-Performance Indexing Model. *Web Semantics: Science, Services and Agents on the World Wide Web*, 10(0), 2012.

[3] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 436–445, Aug. 1997.

[4] M. Jarrar and M. D. Dikaiakos. A Query Formulation Language for the Data Web. *IEEE Transactions on Knowledge and Data Engineering*, (99):1–1, 2011.

[5] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data - SIGMOD '02*, pages 133–144, New York, New York, USA, June 2002. ACM Press.

[6] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 129–140, Feb. 2002.

[7] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory*, pages 277–295, Jan. 1999.

[8] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[9] C. Yu and H. V. Jagadish. Schema summarization. In *Proceedings of the 32nd international conference on Very Large Data Bases*, pages 319–330, Sept. 2006.