Processing SPARQL Queries on Deep Web sources

Andrea Calí[1,4], Tommaso Di Noia[2], Thomas W. Lynch[3,1], and Azzurra Ragone[5]

[1]Dept of Comp. Sci. and Inf. Syst. Birkbeck, Univ. of London, UK
[2]SisInf Lab Politecnico di Bari, Italy
[3]Reasoning Technology Ltd, UK
[4]Oxford-Man Inst. of Quantitative Finance University of Oxford, UK
[5] Independent Researcher
`andrea@dcs.bbk.ac.uk`, `tommaso.dinoia@poliba.it`,
`thomas.lynch@reasoningtechnology.com`, `azzurra.ragone@gmail.com`

Abstract. The Deep Web is constituted by data accessible through dynamic Web pages requested through a Web interface. While Deep Web data sources have been usually modelled as relational in the literature, in many cases it is useful to export Deep Web data as Linked Data sets. In this context, processing queries poses some algorithmic challenges due to the inherent limitations of Deep Web sources which, requiring some inputs to be queried, function as services. In this paper we present a framework and a system to export Deep Web data as Linked Data sets. Further, we characterise a class of SPARQL queries that are executable directly, in spite of the limitations.

1 Introduction

The *Deep Web* (also known as *Hidden Web*) [9, 6, 3] is constituted by structured data that are available as dynamically generated Web pages, generated upon queries usually posed through HTML forms. The Deep Web content cannot be indexed by search engines and is therefore not immediately searchable or accessible. The Deep Web is separated from the so-called *Surface Web*, the latter being the set of ordinary, static Web pages. It is also known that the Deep Web is orders of magnitude larger than the Surface Web [8]. Deep Web data are normally structured and of great value; however, the limitations in accessing them make them hard to search and query. Integrating Deep Web sources as a single database poses several challenges. Normally in this approach, which allows for processing structured queries [3] as well as keyword queries [5], sources are federated into a single schema. Normally, in this approach one deals with known sources, which contain data related to a single domain of interest. Deep Web sources have been naturally modelled as relational tables that can be queried only

SEBD 2018, June 24-27, 2018, Castellaneta Marina, Italy. Copyright held by the author(s).

according to so-called *access patterns* (or *access limitations*); more specifically, certain attributes are to be *selected* in the query in order to get an answer — such a selection corresponds to filling the corresponding attribute in the form with a value.

New ways of exposing structured data have recently emerged, which allow the composition of services for the creation of new integrated applications (mash-ups [11]) and knowledge spaces. Among the various technical proposals and approaches for data publication on the Web which survived to the present days, the two most relevant ones are: RESTful services [7] and Linked Data (LD) [2]. RESTful services provide an agile way of exposing data in a request/response fashion over HTTP, and has been widely adopted by programmers thanks to its easiness of implementation [1]. In this context, data are usually returned in XML or JSON documents after the invocation of a service. Among the issues related to the pure RESTful approach we mention the following:

- There is no explicit semantics attached to the returned data.
- There is no unique query language to invoke services. Each service exposes its own API, and APIs considerably differ from each other even when they refer to the same knowledge domain.
- The integration of different data sources is difficult and is often implemented ad-hoc.

On the other hand, the Linked Data approach is based on the idea that data can be delivered on the Web together with their explicit semantics, expressed by means of common vocabularies. Following the Linked Data principles, datasets should be made accessible through a SPARQL endpoint. Moreover, by using federated queries an agent is able to automatically integrate data coming from different sources thus creating a data space at a Web scale. Unfortunately, also the Linked Data approach comes with its drawbacks, among which we may mention:

- The effort in setting up a SPARQL endpoint is bigger than that of adopting a RESTful approach from service providers. Normally it is much easier to find a JSON-based service than a LD-based one.
- Programmers are usually more familiar with JSON services than with SPARQL endpoints.
- Service providers are usually not interested in exposing all the data they have; instead, they normally want to expose only a small portion of their data.

Based on the above points we can see that, while from the practical point of view the RESTful approach is the most efficient, if we look at the *knowledge* point of view the Linked Data paradigm represents a more suitable solution.

Actually, with JSON-LD this issue could be solved but this format is not widely adopted yet.

https://www.w3.org/DesignIssues/LinkedData.html

https://www.w3.org/TR/sparql11-federated-query/

Following the above observations, we built the `PoLDo` prototype system, which is able to export existing RESTful services, even third-party ones, as a SPARQL endpoint, thus making the underlying Deep Data sources part of the Linked Data cloud. Thanks to a configurable query planner, `PoLDo` is able to break down a SPARQL query into a sequence of RESTful service invocations and to orchestrate different services to provide a correct answer to the posed query. Starting from the data it retrieves from services, `PoLDo` builds a temporary RDF dataset used to compute the result set for the original SPARQL query.

In `PoLDo`, we expose Deep Web sources as services according to the aforementioned intrinsic restrictions. Rather than limiting ourselves to relational Deep Web sources, we model sources as Linked Data with access limitations, that is, SPARQL access points that are only accessible with queries that contain a certain fixed, ground (variable-free) pattern. This poses the problem of determining whether a query can be evaluated over a set of sources (in the form of SPARQL endpoints), while returning the *complete* answers, in the presence of the access limitations. For other queries, we can only expect to get partial answers.

2 Modelling and Querying

In this section we present a model for Linked Data sources that expose Deep Web data, with their inherent limitations. We characterise a class of queries that can be evaluated non-recursively on the data, while returning the complete set of answers (as if they were evaluated in the absence of limitations).

We first show how Deep Web sources are represented as relational data.

When dealing with databases with access limitations the relations can be expressed using the proper access modes. In the case of RESTful services we have the two access mode i and o (for *input* and *output* respectively) stating that for all the tuples $r(c_1, \dots, c_n) \in \mathcal{DB}$ we have some of the arguments of r mapped to the inputs of the service and some to its outputs. Following [4], we denote access modes as superscript of the relation. As an example, a simple service can be expressed by a relation r_1 defined as $r_1^{oio}(C_1, C_2, C_3)$, where C_2 is an input parameter and C_1, C_3 are output parameters. Let us assume we have also another service $r_2^{ioo}(C_3, C_4, C_5)$, where C_3 is an input parameter and C_4, C_5 are output parameters. Notice that we use the same name for parameters that are compatible, that is, have as instance values (constants) of the same type; this is expressed by the notion of *abstract domain* in Deep Web [3], that is a domain specifying the class the objects it represents (e.g. telephone number, person's name etc.) rather than the concrete domain (string, integer etc.). The annotations that specify input and output parameters are also called *access patterns*. Access limitation affect the computation of the result of a query as not all data, in the forms of facts of the form $r_1(c_1, c_2, c_3)$ where c_1, c_2, c_3 are constants, may be accessible. Consider the conjunctive query q_1 defined as

$$q_1(X) \leftarrow r_1^{oio}(Y, a_2, X), r_2^{ioo}(X, a_4, Z)$$

. This query can be processed by executing the services corresponding to the relations r_1 and r_2 from left to right. Indeed, by invoking the first service with input

c_1 we obtain as output a set of pairs of constants of the type $\langle c_1, c_3 \rangle$, each instantiating Y and X respectively, as a consequence of the fact that $r_1(c_1, a_2, c_3) \in \mathcal{DB}$. For each such pair, the second value c_3 is then used as input to invoke the service corresponding to r_2, thus obtaining as output a new set of pairs of the form $\langle c_4, c_5 \rangle$. Among these pairs, due to the selection with a_4 on the second atom of the query, we are interested only in those such that $c_4 = a_4$. So this kind of query is *executable* and retrieves a *complete* answer that contains all solutions to the query over $\mathcal{DB}$. Notice that if the query is not executable, in some cases it can be executed from left to right by reordering the subgoals (atoms) [4]. This kind of queries are called *feasible* (or *orderable*). Feasible queries can be evaluated so as to return the *complete answer* to the query. If we had in addition the relation r_3 (and associated service) defined as $r_3^{oio}(C_5, C_4, C_6)$ and a query

$$q_2(X) \leftarrow r_1^{oio}(Y, a_2, X), r_2^{ioo}(X, a_4, Z), r_3^{oio}(Z, U, W),$$

we may not be able to compute all the answers. For example, the instance $\{r_1(d_1, d_2, d_3), r_2(d_3, d_4, d_5), r_3(d_5, d_4', d_6)\}$ will provide the answer $\langle d_3 \rangle$ for q_2, but such answer cannot be retrieved due to the limitations. In general, the answers $q(D)$ to a query q on a database D (as computed if the data sources had no access limitations) is a superset of the answers $\mathrm{ans}(q, I, D)$ that can be actually retrieved through the access patterns, that is $\mathrm{ans}(q, I, D) \subseteq q(D)$, where I is a set of *initial constants* available to start the extraction of data from the sources.

When considering Linked Data sets, the limitations of the HTML forms (or others beyond the relational formalism) are reflected naturally as we explain below. We refer to the formalisation of SPARQL in [10].

Data are in the form of *triple patterns* of the form $\langle s, p, o \rangle \in (\mathbf{I} \cup B) \times \mathbf{I} \times (\mathbf{I} \cup B \cup \mathbf{L})$, where B is the set of blank nodes. We assume to have a partial relation ρ on predicates, where $\rho(p_1, p_2)$ means that p_1 and p_2 have compatible domains; this can be specified by the `rdfs:domain` of an `rdf:Property`, but we are not going to expose this in detail in this preliminary paper. The fact that two predicates have compatible domains reflects the notion of abstract domain in Deep Web.

Basic graph patterns (BGPs) are sets of *triple graph patterns* of the form $t \in (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ where $\mathbf{I}$, $\mathbf{V}$ and $\mathbf{L}$ are the sets of IRIs, variables and labels, respectively.

In the context of integrating set of Deep Web data sources and answering queries posed on a set of such sources as if it was a single database, we assume to have a schema $\mathcal{S} = \{S_1, \ldots, S_n\}$ of Deep Web sources. Each source S_i, with $1 \leqslant i \leqslant n$, is an access point that has associated a set Λ_i of triple patterns that are called *input triple patterns (ITPs)*. Intuitively, in order to query a source, we need to use a SPARQL query that provides constants for the object (the

Analogously to [3], in a "traditional" data integration setting, we assume that we are integrating a set of known sources related to a domain of interest. We do not address here the problem of automatic discovery of sources, nor the problem of designing a global schema that offers a seamless model of the underlying data. Instead, we take the union of the sources as our database.

third element in the corresponding triple) of *all* such patterns. This reflects the structure of Deep Web sources when represented as relational tables. More specifically, to query a source S_i, we need a query that contains all input triple patterns Λ_i for S_i, with the third element (the object) instantiated to a value.

To start our exposition, we assume that queries are of the form $q = P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n$, where, for all i s.t. $1 \leqslant i \leqslant n$, each P_i is a graph pattern query to be evaluated on S_i. We do not differentiate here between SELECT and CONSTRUCT query types. Now, we formalise the limitations on sources and we establish a criterion for determining queries that can be evaluated so that all answers are computed.

We consider a schema $\mathcal{S} = \{S_1, \ldots, S_n\}$ of Deep Web sources with access limitations $\Lambda = \{\Lambda_1, \ldots, \Lambda_n\}$. For each S_i, $1 \leqslant i \leqslant n$, Λ_i is a set of input triple patterns (ITPs) of the form $\lambda = \langle V_s, p, V_o \rangle \in \mathbf{V} \times \mathbf{I} \times \mathbf{V}$. Let $\mathcal{D} = \{D_1, \ldots, D_n\}$ be an instance for the schema $\mathcal{S}$, where D_i is the instance of S_i, , for all i s.t. $1 \leqslant i \leqslant n$.

Definition 1.

(a) A triple pattern t instantiates *an ITP $\lambda = \langle V_s, p - V_o \rangle$ if $\alpha = \mu(\lambda)$ such that μ is a mapping and $\mu(V_o) \in \mathbf{I} \times \mathbf{L}$. Notice that in this case $\mu(p) = p$.*
(b) A source S with a set Λ of input triple patterns can be accessed with a BGP P if for each $\lambda \in \Lambda$, there is $\alpha \in P$ such that α instantiates λ.

3 Processing Queries

We now focus on queries that can be executed on an instance $\mathcal{D}$ according to the access patterns

Definition 2. *Let t_1, t_2 be triple graph patterns defined as $t_1 = \langle W_1, p_1, V \rangle, t_2 = \langle W_2, p_1, V \rangle$, with $\{W_1, W_2\} \subseteq \mathbf{I} \cup \mathbf{L} \cup \mathbf{V}$ and $V \in \mathbf{V}$.*

(a) We say that t_1 feeds *t_2 if $\rho(p_1, p_2)$, that is, p_1 and p_2 are compatible (notice that both triple graph patterns have the same variable as third element).*
(b) We say that t_1 and t_2 match *if $p_1 = p_2$.*

We now identify a class of queries of the form $q = P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n$ that are executable from left to right, after ordering the graph patternss P_i, on an instance $\mathcal{D}$ for a schema $\mathcal{S}$; we call such queries *orderable* in accordance with the relational terminology for queries under access limitations.

Definition 3. *Let q be a query of the form $q = P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n$, posed on a schema $\mathcal{S}$ as previously defined. q is said to be* orderable *if $P_1, \ldots, P_n$ can be ordered as $P_{i_1}, \ldots, P_{i_n}$ such that:*

(a) For every ITP $\lambda \in \Lambda_{i_1}$, there is $\alpha \in P_{i_1}$ such that α instantiates λ;
(b) For all j such that $1 \leqslant j \leqslant n$, for every ITP $\lambda \in \Lambda_{i_j}$, either
(i) there is $\alpha \in P_{i_j}$ such that α instantiates λ;

(ii) there are $\beta \in P_{i_j}$ and $\gamma \in P_{i_\ell}$, witha $1 \leqslant \ell \leqslant i_j - 1$, such that (1) γ feeds β; (2) β and λ match.

Notice that this definition is analogous to the same one in the context of relational data under access limitations.

Proposition 1. *Every orderable query on a schema $\mathcal{S}$ of the form $q = P_1$ AND P_2 AND ... AND P_n can be executed on an instance $\mathcal{D}$, after suitably ordering $P_1, \ldots, P_n$, from left to right so that all answers $q(\mathcal{D})$ are retrieved.*

Proof (sketch). We know q can be ordered as in Definition 3; let us assume w.l.o.g. then that it is already ordered. It is easily seen that P_1 can be immediately evaluated on D_1. Then, for each j such that $2 \leqslant j \leqslant n$, P_j can be evaluated on D_j according to Λ_j because every ITPs λ of Λ_j is either *(1)* instantiated by some BGP in P_j or *(2)* instantiated by a triple pattern β', obtained from $\gamma \in P_\ell$, $1 \leqslant \ell \leqslant j-1$ (as per Definition 3) by replacing its third element with the values of the third element of $\mu(\gamma)$, where $\gamma \in P_\ell$, for every mapping μ resulting from the evaluation of P_ℓ on D_ℓ.

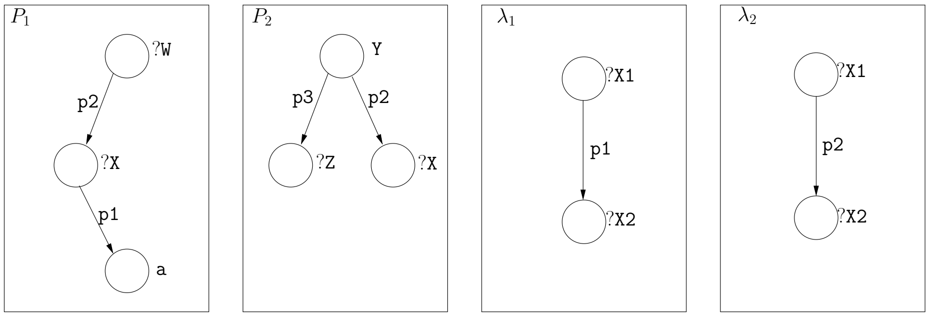

Fig. 1. Figure for Example 1

Example 1. Consider the query P_1 AND P_2, where P_1, P_2 are as in Figure 1 (we omit variables that appear only once), on a schema $\mathcal{S} = \{S_1, S_2\}$ with $\Lambda_1 = \{\lambda_1\}$, $\Lambda_2 = \{\lambda_2\}$, where λ_1, λ_2 are again as in Figure 1. The query q is processed by evaluating P_1 on S_1 directly and then, for all values v obtained by mapping ?X in the evaluation of P_1, by evaluating P_2' (see figure), obtained by replacing ?X with v in P_2.

4 PoLDo

In this section we briefly present the architecture of PoLDo, a prototype that processes queries over distributed Deep Web sources that are exposed as Linked Data sets.

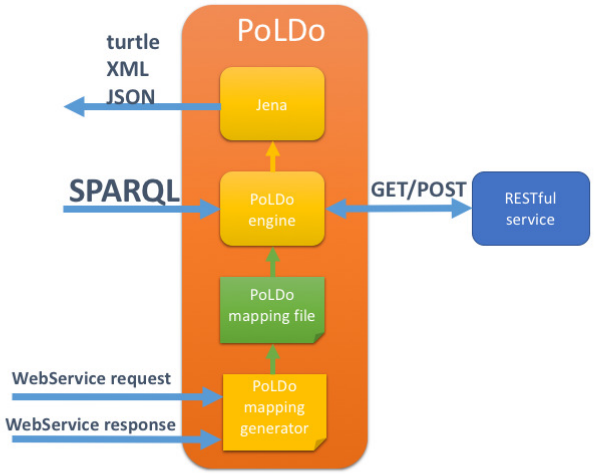

Fig. 2. A high level view for the architecture of `PoLDo`

The high level architecture of `PoLDo` is shown in Figure 2. The engine is responsible of getting the SPARQL query and breaking it down to a sequence of RESTful calls to a remote service. The transformation is made possible thanks to a mapping file that maps Linked Data URIs to the elements of the signature of the remote call. While querying the remote service, `PoLDo` feeds an RDF local triple store (Jena Fuseki, in its current implementation) which is in charge of processing the actual SPARQL query. More specifically, in our system we have the following components and models.

`PoLDo` **engine** The engine accepts the SPARQL query and extracts all the constants from the graph template in the WHERE clause. Then, by using the algorithm sketched in Section 3 , it uses the constants to query the external service and to get the data that will be used to create a local RDF representation of the data space. Thanks to the information encoded in the `PoLDo` mapping file, the engine is able to feed a local repository of RDF triples. The engine is also capable to exploit external services to get extracted resources' URI from mapped API services, that often return data related to the resource but not the URI.

Jena The Jena Model is used to save a Linked Data version of the data which are incrementally retrieved from the RESTful service. The availability of a third-party RDF model makes `PoLDo` able to support the full specification of SPARQL query language. Furthermore, it is able to return the data in all the formats supported by the query engine Jena ARQ.

`PoLDo` **Mapping Generator** The Mapping Generator is responsible for the generation of the mapping file. Given a RESTful service, it works in four steps. It analyzes request (HTTP GET) and response (JSON or XML) given to and by a web service, then extracts all the inputs and outputs. The user is then allowed to manually assign a class of membership to resources. The Mapping Generator queries the ontologies (DBpedia and LOV, in our first experimentation) and recommend how to link resources. After user confirmation, the final mapping file is generated and can to be used by the engine.

`PoLDo` **mapping file** This file contains information about how to map the URIs of the SPARQL query to inputs and outputs of the services. It also describes the access patterns as represented in Section 2 as well as their mutual relations (that is which triple graph patterns feed each other).

References

1. Vito Walter Anelli, Vito Bellini, Andrea Calì, Giuseppe De Santis, Tommaso Di Noia, and Eugenio Di Sciascio. Querying deep web data sources as linked data. In *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics, WIMS 2017, Amantea, Italy, June 19-22, 2017*, pages 32:1–32:7, 2017.
2. Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data – the story so far. *Int. J. Semantic Web Inf. Syst*, 5(3):1–22, 2009.
3. Andrea Calì and Davide Martinenghi. Querying data under access limitations. In *Proc. of ICDE*, pages 50–59, 2008.
4. Andrea Calì and Davide Martinenghi. Querying the deep web. In *Proc. of EDBT 2010*, pages 724–727, 2010.
5. Andrea Calì, Davide Martinenghi, and Riccardo Torlone. Keyword queries over the deep web. In *Proc. of ER 2016*, pages 260–268, 2016.
6. Kevin Chen-Chuan Chang, Bin He, and Zhen Zhang. Toward large scale integration: Building a metaquerier over databases on the web. In *Proc. of CIDR*, pages 44–55, 2005.
7. Roy T. Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002.
8. Govind Kabra, Zhen Zhang, and Kevin Chen-Chuan Chang. Dewex: An exploration facility for enabling the deep web integration. In *Proc. of ICDE*, pages 1511–1512, 2007.
9. Jayant Madhavan, Loredana Afanasiev, Lyublena Antova, and Alon Y. Halevy. Harnessing the deep web: Present and future. In *Proc. of CIDR*, 2009.
10. Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and complexity of SPARQL. *ACM Trans. on Database Systems*, 34(3):16:1–16:45, 2009.
11. Ahmet Soylu, Felix Mödritscher, Fridolin Wild, Patrick De Causmaecker, and Piet Desmet. Mashups by orchestration and widgetbased personal environments: Key challenges, solution strategies, and an application. *Program*, 46(4):383–428, 2012.