Exposing the Deep Web in the Linked Data Cloud

Andrea Calì[1], Giuseppe De Santis[2], Tommaso Di Noia[2] and Nicola Mincuzzi[2]

[1]London Knowledge Lab, Birkbeck, University of London
andrea@dcs.bbk.ac.uk
[2]SisInf Lab, Politecnico di Bari
{g.desantis6,n.mincuzzi}@studenti.poliba.it,tommaso.dinoia@poliba.it

Abstract. The Deep Web is constituted by dynamically generated pages, usually requested through a HTML form; it is notoriously difficult to query and to search, as its pages are obviously non-indexable. More recently, Deep Web data have been made accessible through RESTful services that return information usually structured in JSON or XML format. We propose techniques to make the Deep Web available in the Linked Data Cloud, and we study algorithms for processing queries, posed in a transparent way on the Linked Data, on the underlying Deep Web sources. The tool we developed mainly focuses on exposing RESTful services as Linked Data datasets thus allowing a smoother semantic integration of different structured information sources in a global data- and knowledge-space.

1 Introduction

Slowly but steadily, the Web has moved from a huge collection of unstructured textual documents to a gigantic repository of structured data. At a first stage, the original static Web pages have been replaced by dynamic ones fed by information coming from Deep web data sources which cannot be directly queried. Then we assisted to the flourishing of new services that expose structured data by exploiting standard Web technologies thus making possible their composition for the creation of new integrated applications (mash-ups [10]) and knowledge spaces. Among the various technological proposals and approaches for data publication on the Web which survived to the present days, the two most relevant ones are for sure: RESTful services [5] and Linked Data (LD) [2]. The former is a very agile way of exposing data in a request/response way over HTTP and has been widely adopted by programmers thanks to its easiness of implementation. Data are usually returned in XML or JSON documents after the invocation of a service. Among the issues related to pure a RESTful approach we may mention:

– no explicit semantics attached to the returned data[1];

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

– lack of a unique query language to invoke services. Each service exposes its own API which can considerably differ from each other even when they refer to the same knowledge domain;
– manual integration of different data sources.

On the other side, the Linked Data (LD) approach bases on the idea that data can be delivered on the Web together with their explicit semantics by means of common vocabularies. Following the Linked Data principles[2], datasets should be accessible through a SPARQL endpoint. Moreover, by using federated queries[3] 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:

– the effort in setting up a SPARQL endpoint is felt more difficult than a RESTful approach from service providers. Nowadays, it is much easier to find a JSON-based service than a LD-based one.
– programmers are more used to JSON services than to SPARQL endpoints;
– service providers are usually not interested in exposing all the data they have but only a small portion.

Based on the above points we can see that while from the *practical* point of view the champion is the RESTful approach, if we look at the *knowledge* point of view Linked Data represents a much better alternative. This was the leading observation that inspired us in the development of `PoLDo`. With `PoLDo` we can expose an already existing RESTful service, even a third-party one, as a SPARQL endpoint thus making it 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. Starting from the retrieved data it then builds the answer to the original SPARQL query.

The remainder of this paper is structured as follows. In the next section we report on some relevant related work on accessing the Deep Web while in Section 3 we describe `PoLDo` together with an explanatory example. Conclusion closes the paper.

2 Related Work

The term *Deep Web* (sometimes also called *Hidden Web*) [9,8,3,4] refers to the data content that is created dynamically as the result of a specific search on the web. For example, when we query a Yellow Pages website, the generated output is the result of a query posed on an underlying database, and cannot be indexed by a search engine. In this respect, the Deep Web content resides outside web pages, and is only accessible through interaction with the web site – typically via HTML forms.

[2] `https://www.w3.org/DesignIssues/LinkedData.html`
[3] `https://www.w3.org/TR/sparql11-federated-query/`

As an example of Deep Web source, the whitepages.com website presents a form where, when searching by name, the last name is a required field. In order to look for a person named Joseph Noto, living in New Jersey, we would fill the form which in relational terms, corresponds to issuing a SQL query to the underling database. Therefore, a Deep Web source can be naturally modeled as a relational table (or a set of relational tables) where not all queries are allowed; in fact, *access limitations* exist on such relations. In particular, they can be queried only according to so-called *access patterns*, each of which enforces the selection on some of the attributes (which corresponds to filling the corresponding field in the form with a value). It is believed that the size of the Deep Web is several orders of magnitude larger [7] than that of the so-called *Surface Web*, i.e., the web that is accessible and indexable by search engines. The information stored in the Deep Web is invaluable, but at the same time hard to collect automatically on a reasonably large scale.

Two main approaches for accessing the Deep Web exist [8]. In the *vertical data integration* approach, each Deep Web site is a data source, and sources are integrated and unified by a *global schema* representing the domain of interest. Normally, in this approach one deals with a relatively small number of sources, storing data related to a single domain of interest. In the *Surfacing* approach, instead, answers are pre-computed from Deep Web sources by posing suitable queries, and then the results are indexed as normal static pages in a search engine. Accessing the Deep Web requires a variety of techniques and tools from several areas of computer science. In this paper we survey some of the most relevant approaches to querying and searching the Deep Web. One important problem in accessing the Deep Web is the automatic understanding of the semantics of the so-called *query interfaces*, that is, the forms that provide access to Deep Web sources. Several approaches have been developed, based on heuristics or learning from samples; some techniques involve domain-specific knowledge encoded in ontological rules. Processing structured query over Deep Web sources is the key problem in the integration of such sources. Interestingly, when Deep Web sources are modeled, as mentioned, as relations with access patterns (i.e., having certain attributes that are necessarily to be selected in order to query the source), answering a simple conjunctive (select-project-join) query on such sources may require, in the worst case, the evaluation of a *recursive* Datalog query. This raises the issue of reducing the number of accesses to sources while answering a query.

The problem of answering a query over sources with access patterns falls into the setting of vertical data integration, where a limited number of Deep Web sources, whose interface is known (having been possibly understood automatically), are queried in a structured way. But if we want to search via keyword search the Deep Web together with the ordinary, "shallow" Web, we cannot rely on a set of known sources, and therefore we have to adopt the surfacing approach. Surfacing the Deep Web poses several challenges because it is not easy to get a good coverage of the content of a source while at the same time limiting the number of sampling accesses on it. Keyword search on relational databases has

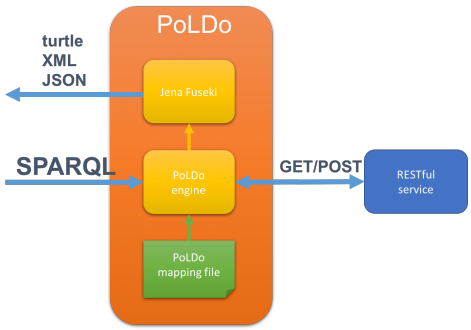

Fig. 1. A high level view for the architecture of PoLDo

been traditionally studied in several works in the literature (see e.g., [1,6]). It is interesting to consider keyword search also on Deep Web sources, in the context of vertical Deep Web integration. In this case, a suitable notion of answer to keyword queries is needed.

3 PoLDo

The high level architecture of PoLDo is represented in Fig. 1. 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 in details, we have:

PoLDo **engine** It receives the SPARQL query and extracts all the constants from the graph template in the WHERE clause. Then, by using the algorithm in [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.

Jena Fuseki The triple store is used to save a LD version of the data which are incrementally retrieved from the RESTful service. The availability of a third-party triple store 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 Jena Fuseki.

PoLDo **mapping file** This file contains information about how to map the URIs of the SPARQL query to inputs and outputs of the service. Moreover, it also describes the entities represented by inputs and outputs as well as their mutual relations.

3.1 PoLDo mapping language

PoLDo mapping file allows the designer to create a link between URIs contained in the SPARQL queries processed by the engine and, at the same time, to enrich their semantics by explicitly adding information about the corresponding OWL class or property that can be defined also in an external vocabulary (e.g. DBpedia). Mapping rules can also be created to describe RDF triples containing information on how to relate values of an input parameter with the outputs of the service invocation. All the rules contained in the PoLDo mapping file are, in turn, represented as RDF triples which refers to a corresponding RDF-S ontology. The main element of the PoLDo ontology is the class `poldo:Service` (see Fig. 2). It describes each service in terms of its base URL (`poldo:hasUrl`), the

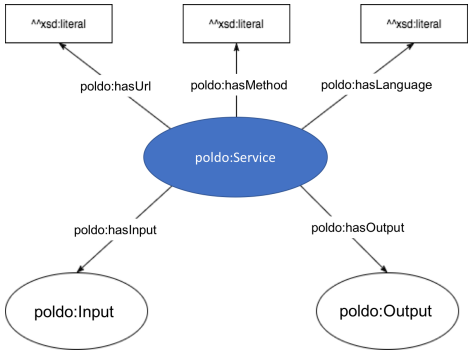

Fig. 2. The class `poldo:Service` in the PoLDo mapping ontology.

HTTP method GET or POST (`poldo:hasMethod`), the language of the answer to the service call, e.g. XML or JSON (`poldo:hasLanguage`) and its inputs and outputs (`poldo:hasInput` and `poldo:hasOutput`). The ontological description of `poldo:Input` and `poldo:Output` are represented respectively in Fig. 3 and Fig. 4. As we can see, both inputs and outputs of a services can be mapped as instances of a class or as a property. Indeed, especially when the type of the corresponding value is different from a string, we may have cases when the parameter is better represented by a property rather than the subject or object of a triple. We may think at a geographical service returning places based on their coordinates. In this case, coordinates are better mapped to the properties `geo:lat` and `geo:long` of the Basic Geo vocabulary[4]. The modeling of `poldo:Input` and `poldo:Output` classes try to catch all possible cases in the description of the inputs and outputs of a service. For instance, the parameter `poldo:hasFixedValue` is used when we need a key to access the RESTful service. As for `poldo:Output` we just highlight that it possible to model the

[4] `https://www.w3.org/2003/01/geo/`

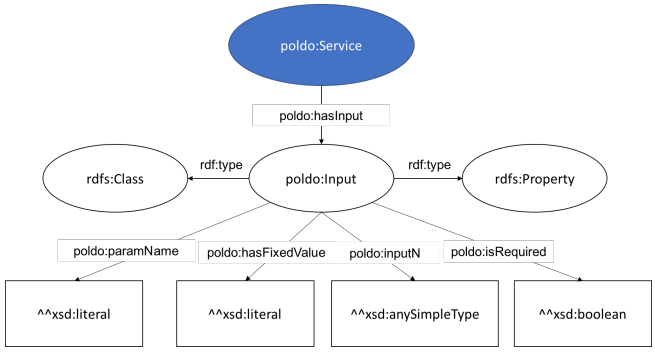

Fig. 3. The class `poldo:Input` in the `PoLDo` mapping ontology.

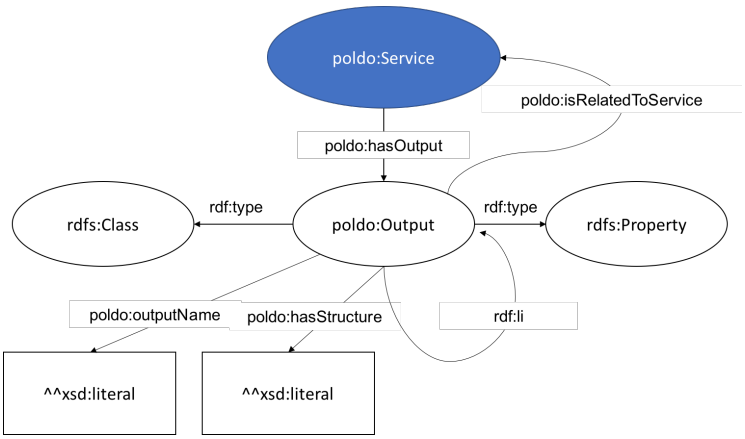

Fig. 4. The class `poldo:Output` in the `PoLDo` mapping ontology.

situation when the service returns a single value or a list of values by means of the `poldo:hasStructure` and `rdf:li` statements.

3.2 `PoLDo` engine

The main component of `PoLDo` is its query planner implemented in the `PoLDo` engine module. It uses the algorithm presented in [3] to iteratively query the RESTful services and build a local cache containing the RDF version of the data whose transformation follows the rules available in the `PoLDo` mapping file.

For a better understanding of the overall approach, we now describe how `PoLDo` engine works by means of an example. Suppose we have two services related to music events in a city as in the following.

service	**input**	**output**
`http://example.com/api/returnSinger`	city	title, singer
`http://example.com/api/returnPlace`	singer	city, country

Given a city, the first service returns the title of the events in that city together with the performing artist. The second service returns the birth place of an artist. The corresponding mapping file will then contain the following triples:

```
1   :returnArtist a poldo:Service ;
2                 poldo:hasURL ''http://example.com/api/returnSinger'' ;
3                 poldo:hasLanguage ''JSON'' ;
4                 poldo:hasMethod  ''GET'' ;
5                 poldo:hasInput :city ;
6                 poldo:hasOutput :title ;
7                 poldo:hasOutput :singer .
...
8   :returnPlace a poldo:Service ;
9                poldo:hasURL ''http://example.com/api/returnPlace'' ;
10               poldo:hasLanguage ''JSON'' ;
11               poldo:hasMethod  ''GET'' ;
12               poldo:hasInput :singer ;
13               poldo:hasOutput :city ;
14               poldo:hasOutput :country ;
...
15  :singer a dbo:Person .
16  :title a dbo:Event .
17  :city a dbo:Place .
18  :country a dbo:Country .
19  :singer dbo:birthPalce :city, :country .
20  :title dbo:artist :singer;
```

Now suppose we want to know the name of singers born in a specific city. The corresponding SPARQL query is then:

```
SELECT ?singer
WHERE {
   ?singer dbo:birthPlace ?city .
   ?city rdfs:label ''modena'' .
   ?singer a dbo:Person .
   ?city a dbo:Place .
}
LIMIT 1
```

The only entry points `PoLDo` engine has to query the services are constant symbols, in our case `modena` whose corresponding entity is declared to be a `dbo:Place` in the SPARQL query. By looking at the mapping file, the engine discovers it can query the service `:returnArtist`. Then, by using the outputs (constants) of the first service, the engine may query `:returnPlace` and, if lucky, it can get that one of the artists returned by `:returnArtist` was born in `modena`. If this is not the case, the engine uses the constants returned by `:returnPlace` to query again `:returnArtist` thus continuing its search of the answer for the original SPARQL query. It is worth noticing that while iteratively querying the services, `PoLDo` builds RDF triples by creating fresh entities corresponding to the arriving constants and by enriching and connecting them thanks to the rules states in rows 15-20 of the above mapping file. All the triples are saved in the triple store which is then natively queried by means of the original SPARQL query.

PoLDo engine stops querying the original services in the following cases: (i) the answer to the query is found; (ii) there are no more fresh constants and then the answer to the original query can not be found; (iii) the execution time exceeds a timeout set by the designer.

4 Conclusion

In this paper we presented PoLDo, that acts as a middleware between a RESTful service and SPARQL endpoint. By means of PoLDo we are allowed to expose the Deep Web data available via RESTful services as Linked Data that can be easily integrated in the so called Linked Data Cloud. The tool we developed adopts algorithms and techniques coming from the Deep Web literature to make possible the composition of services at a data level. Via a mapping file, PoLDo is able to interpret a SPARQL query in terms of a sequence of remote calls to external services and to translate the returned data in a temporary RDF graph which is locally stored in a triple store. The approach we developed is for sure a step forward the creation of a global, semantics-enabled, integrated, gigantic data graph as in the original view of the Semantic Web.

Acknowledgments. Andrea Calí acknowledges partial support by the EPSRC project "Logic-based Integration and Querying of Unindexed Data" (EP/E010865/1) and by the EU COST Action IC1302 KEYSTONE.

References

1. S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.
2. C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst*, 5(3):1–22, 2009.
3. A. Calì and D. Martinenghi. Querying data under access limitations. In *Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE 2008)*, pages 50–59. IEEE Computer Society Press, 2008.
4. K. Chen-Chuan Chang, B. He, and Z. Zhang. Toward large scale integration: Building a metaquerier over databases on the web. In *Proc. of CIDR*, pages 44–55, 2005.
5. R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002.
6. V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 670–681. VLDB Endowment, 2002.
7. G. Kabra, Z. Zhang, and K. Chen-Chuan Chang. Dewex: An exploration facility for enabling the deep web integration. In *Proc. of ICDE*, pages 1511–1512, 2007.
8. J. Madhavan, L. Afanasiev, L. Antova, and A. Y. Halevy. Harnessing the deep web: Present and future. In *Proc. of the 4th Conf. on Innovative Database Research (CIDR 2008)*, 2009.
9. D. Martinenghi. Access pattern. In H. C. A. van Tilborg and S. Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Edition*, pages 17–20, 2011.
10. A. Soylu, F. M dritscher, F. Wild, P. D. Causmaecker, and P. Desmet. Mashups by orchestration and widgetbased personal environments: Key challenges, solution strategies, and an application. *Program*, 46(4):383 – 428, 2012.