Querying Deep Web Data Sources as Linked Data

Vito W. Anelli
SisInf Lab, Politecnico di Bari
vitowalter.anelli@poliba.it

Vito Bellini
SisInf Lab, Politecnico di Bari
vito.bellini@poliba.it

Andrea Calí
DCSIS, Birkbeck, Univ. of London
and Oxford-Man, Oxford Univ.
andrea@dcs.bbk.ac.uk

Giuseppe De Santis
SisInf Lab, Politecnico di Bari
g.desantis6@studenti.poliba.it

Tommaso di Noia
SisInf Lab, Politecnico di Bari
tommaso.dinoia@poliba.it

Eugenio di Sciascio
SisInf Lab, Politecnico di Bari
eugenio.disciascio@poliba.it

ABSTRACT

The Deep Web is constituted by dynamically generated pages, usually requested through HTML forms; it is notoriously difficult to query and to search, as its pages are obviously non-indexable. 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, providing answers based on the underlying Deep Web sources. We present a software prototype that exposes RESTful services as Linked Data datasets thus allowing a smoother semantic integration of different structured information sources in a global data and knowledge space.

CCS CONCEPTS

• **Information systems → Information integration**; **Mediators and data integration**; **RESTful web services**; *Query languages for non-relational engines*; *Federated databases*;

KEYWORDS

Deep Web, Linked Data, SPARQL, RESTful Services

ACM Reference format:
Vito W. Anelli, Vito Bellini, Andrea Calí, Giuseppe De Santis, Tommaso di Noia, and Eugenio di Sciascio. 2017. Querying Deep Web Data Sources as Linked Data. In *Proceedings of WIMS '17, Amantea, Italy, June 19-22, 2017,* 7 pages.
https://doi.org/10.1145/3102254.3102290

1 INTRODUCTION

In the last years the Web has changed from a huge collection of unstructured textual documents to a vast repository containing a large amount of structured data. Structured data are often available as Deep Web sources, that is, as databases that can be queried through a Web interface, normally an HTML form. New ways of exposing structured data have recently emerged, which allow the composition of services for the creation of new integrated applications (mash-ups [14]) 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 [9] and Linked Data (LD) [3]. 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. 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[1].
- 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[2], datasets should be made 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, 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

WIMS '17, June 19-22, 2017, Amantea, Italy

ACM ISBN 978-1-4503-5225-3/17/06…$15.00
https://doi.org/10.1145/3102254.3102290

[1] Actually, with JSON-LD this issue could be solved but this format is not widely adopted yet.
[2] https://www.w3.org/DesignIssues/LinkedData.html
[3] https://www.w3.org/TR/sparql11-federated-query/

at the *knowledge* point of view the Linked Data paradigm represents a more suitable solution.

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.

The remainder of the paper is organized as follows. In the next section we report on some relevant related work on accessing the Deep Web; Section 3 illustrates the problem of querying Deep Web data. In Section 4 we give an overview of the PoLDo system. Section 5 concludes the paper.

2 RELATED WORK

The *Deep Web* (also known as *Hidden Web*) [6, 8, 12] 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 easy to search or access. Deep Web sources are naturally modeled as relational tables that can be queried only 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. The Deep Web is therefore separate 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 [11]. 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 [6] as well as keyword queries [7], 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. Processing structured query over federated Deep Web sources allows for the integration of such sources. The problem of query processing in this context poses several challenges due to the access limitations on the underlying sources. Answering a simple conjunctive (select-project-join) query on such sources requires, in general, the evaluation of a *recursive* Datalog query. Given that Deep Web sources, for their own nature, are normally rather slow in responding, it is crucial to minimize the number of accesses while processing a query.

In some cases one wants to pose keyword queries on a set of Deep Web sources; while the problem of keyword search on traditional relation databases has been studied in the literature (see e.g., [1, 10]), in the context of the Deep Web the same problem gets more complicated and require a suitable notion of answer to keyword queries as well as algorithms [7].

When searches need to be posed on the whole Web, including the Deep Web, the most common approach is *Surfacing* [12], that is, pre-computin answers from Deep Web sources by posing suitable queries, and indexing the results as static pages in a search engine.

Over the years, there have been many attempts in bridging the gap between Linked Data and RESTful services. The intuition that the two worlds are strongly connected is not new [13] and led also to the development of the Linked Data Platform[4] definition. Most of these works [2, 15, 16] focused mainly on how to describe a RESTful service in order to explicitly expose information about the data it may provide. To the best of our knowledge, PoLDo is the first attempt in exposing an orchestration of RESTful service through a SPARQL endpoint thus making possible the integration of more data coming from different and heterogeneous datasets via federated queries.

3 QUERYING DATA UNDER ACCESS LIMITATIONS

In this work we focus on RESTful services which can be modeled as relational tables with access limitations [4]. In fact, a RESTful service relates input data encoded in the query with output data usually returned as JSON or XML documents. In this respect, it results quite natural to adopt a relational approach to model the service where we have some attributes mapped to input parameters and others to outputs.

The relational model is basically constituted by two different sets of symbols: predicates and constants. Predicates denote the relations expressed in the database and constants are the values involved in the relations. Constant symbols belong to a fixed set Γ under the unique name assumption and a relational schema C is defined by an alphabet Γ of predicate symbols and a set of integrity constraints. Each predicate symbol has an arity associated, which denotes the number of attributes of the relation. An integrity constraint is an assertion on predicates symbols that must be satisfied on every database coherent with the schema. A relational database $\mathcal{DB}$ is a set of relations with values in Γ and with at least one relation $r^{\mathcal{DB}}$ of arity n for each predicate symbol $r \in \Gamma$ of arity n. This functional relation can be interpreted as $\mathcal{DB}$ is an instance of the schema C. This means that $\mathcal{DB}$ contains facts (relations and related values) that make true the corresponding predicate symbols in C with respect to a specific interpretation function. A relational query q specifies a set of tuples to retrieve from $\mathcal{DB}$. One of the most relevant query class is that of conjunctive queries, in which a query q of arity n can be expressed as

$$q(x_1, \ldots, x_n) \leftarrow conj(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

where $conj(x_1, \ldots, x_n, y_1, \ldots, y_m)$ is a conjunction of first-order atomic formulas involving $n + m$ variables and an arbitrary set of constants in Γ. The answer a to the query q consists of a set of constant tuples $(c_1, \ldots, c_n)$ of arity n such that when mapping $x_k \mapsto c_k$, with $k = (1, \ldots, n)$, the formula

$$\exists y_1, \ldots, y_m \; conj(x_1, \ldots, x_n, y_1, \ldots, y_m)[x_k \mapsto c_k]$$

results true in $\mathcal{DB}$.

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

[4] https://www.w3.org/TR/ldp/

(for *input* and *output* respectively) stating that for all the tuples $r(c_1, \ldots, c_n) \in \mathcal{DB}$ we have some of the arguments of r mapped to the inputs of the service and some to its outputs. Following [5], we denote access modes as superscript of the relation. As an example, a simple service can be expressed by a relation r_1 as:

$$r_1^{oio}(c_1, c_2, c_3)$$

in which c_2 is an input parameter and c_1 and c_3 are output parameters. Access limitation may affect the computation of the result of a query as all the constants $(c_1, \ldots, c_n)$ may be not accessible at the same time. Consider the conjunctive query q_1 defined as

$$q_1(x) \leftarrow r_1^{oio}(y, c_1, x), r_2^{ioo}(x, c_2, z)$$

This query can be solved by executing from left to right the services corresponding to the relations r_1 and r_2. Indeed, we have that by invoking the first service with input c_1 we obtains a set of pairs of constants (c_y, c_x) as output instantiating y and x respectively, meaning that $r_1(c_y, c_1, c_x) \in \mathcal{DB}$. For each pair, c_x is then used to execute the second service corresponding to r_2 thus obtaining as output the new set of pairs (c_k, c_z). Among these pairs we are interested only in those such that $c_k = c_2$. So this kind of query is *executable* and retrieves a complete answer which contains all the possible solutions to the query over $\mathcal{DB}$. It is noteworthy [5] that if the query is not executable, in some cases it can be executed reordering the subgoals. This kind of queries are called feasible(or orderable). Feasible queries return the *complete answer* to the query. If we had the relation r_3 defined as $r_3^{oio}(c_1, c_2, c_3)$ and a query $q_2(A) \leftarrow r_1^{oio}(t, c_1, x), r_2^{ioo}(x, c_2, z), r_3^{oio}(t, u, y)$ we could have a problem. The relation r_3 has as input parameter a variable and there is no way that variable can be set because u is not present in other subgoals. In this case we added artificially r_3, then we know that r_3 is not needed to answer the query because it can not affect the result set of the Variable x; hence it exists a subset of the subgoals that is not useful or redundant and without them the query is feasible, or better. This kind of queries are called stable.

If we exclude all the previous cases, the chance to retrieve the complete answer is no more guaranteed. The set of retrieved answers is usually only a subset of the complete answer and we refer to it as maximal answer set or reachable certain answer set. In general a recursive query plan is needed to retrieve the maximal answer set and this may result in a very expensive process.

From a practical point of view, if we know the domain of each input parameter and output result we may preselect possible constant values to invoke services. This is an important aspect of retrieving information in the Deep Web because the complete set of constants for a specific argument of a relation is usually unknown. By identifying the abstract domains of the arguments (attributes) the domain knowledge is increased and more input constants are available to retrieve information from the services thus resulting in a more complete answer.

3.1 Querying the Deep Web with SPARQL

The goal of this work is to express queries using a semantic language as SPARQL being agnostic with respect to the original deep web source. A SPARQL query, in details, tries to find the graph patterns in the underlying data that match to the query. The knowledge base is expressed in a standard W3C language, such as RDF. The atomic informational element of an RDF knowledge base is a triple. A triple is a ternary relation in which the three arguments can be digested to subject, predicate and object. The constant symbols that can be used in the relation are IRIs (I) and Literals (L) with the exception of predicates, in which only IRIs are allowed. An RDF database $\mathcal{DB}_{\mathcal{RDF}}$ is a set of triple with values in Γ. This functional relation can be intended as $\mathcal{DB}_{\mathcal{RDF}}$ is an interpretation of the underlying RDF schema.

The query language SPARQL extends triples with the introduction of variables (V) and constitutes patterns. The simple triple corresponds to the most elementar graph pattern (tp), formally defined as $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$. A generic SPARQL query (sq) can be expressed as

$$sq(x_1, ..., x_n) \leftarrow \pi_{(x_1,...,x_n)} sops(x_1, ..., x_n, y_1, ..., y_m, C_1, ..., C_p)$$

where $sops(x_1, ..., x_n, y_1, ..., y_m, C_1, ..., C_p)$ is a set of sets of triples resulted from the application of SPARQL operators among graph patterns that involve $n + m$ variables and p constants. The answer sa to the query sq consists of a set of relations of arity n that makes true the formula over $\mathcal{DB}_{\mathcal{RDF}}$. $\pi_{(x_1,...,x_n)}$ corresponds to the projection of each set of triples restricting the relation to a subset of the attributes.

In this scenario we want to achieve two main goals: the maximal answer set of a deep web source should be reformulated as an RDF knowledge base ($\mathcal{DB}_{\mathcal{RDF}}$) and the sparql query have to be translated in a appropriate sequence of queries over $\mathcal{DB}$. These two distinct operations must provide a vocabulary to query the knowledge base and a set of automatic operations with the purpose of populating the maximal answer set and hence the RDF knowledge base. Mapping answers (map_{ans}) of queries over $\mathcal{DB}$ to triples can be considered a simple but not trivial operation. In particular each deep web query q can be mapped to a triples set tr_{set}

$$q(x_1, ..., x_n) \rightarrow tr_{set}(x_1, ..., x_k, C_1, ..., C_p)$$

where the triples set tr_{set} involve k constants from the answer set (with $k \leq n$) and p new constants. Populating $\mathcal{DB}_{\mathcal{RDF}}$ operation, conversely, needs to be explained in details. Recently, many optimizations to a query planner for deep web sources were proposed and each of them tries to find the best sequence of executable queries. In a real-world scenario, where data sources have access limitation, only queries where input parameters are known can be executed. It is straightforward that the identification of the abstract domains is crucial. Once the abstract domains are known, state of the art planners can do the job. The system needs to analyze the query looking for a subset of triple patterns that can fulfill the input parameters of a query q_1. More formally, a translation map_{req} should exist for the query q_1 that expresses the input parameters abstract domains in terms of triples

$$q_1^{io\ldots o}(A, x_1, ..., x_n) \leftarrow tr_{set}(A, AD_1, y_1, ..., y_m, C_1, ..., C_p)$$

that contains a constant (A) for each input parameter associated with the appropriate abstract domain (AD_1), m variables and p constants.
In order to identify the appropriate abstract domain a formulation of q_1 in terms of abstract domains is needed.

$$q_1^{io\ldots o}(AD_1, ..., AD_n)$$

The number of possible mappings map_{req} for each query is potentially infinite since the way constants can be semantically associated with their abstract domains is arbitrary. In this work, we limit our modeling to two very representative types.

Type A. The input constant symbol is expressed as object of the predicate `<http://www.w3.org/2000/01/rdf-schema#label>` and an explicit membership to an abstract domain is provided through `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` predicate. The matching graph pattern will be `{?s rdfs:label [RDFLiteral | NumericLiteral | BooleanLiteral] . ?s rdf:type [iri | RDFLiteral]}` where `RDFLiteral`, `NumericLiteral`, `BooleanLiteral` and `iri` are the EBNF terms from SPARQL 1.1 syntax. As we said before, a formulation of q in terms of abstract domains is needed, in particular the abstract domain for $input_1$ of query q will be mapped as `{input1 rdf:type [iri | RDFLiteral]. }` in the mapping knowledge base.

In order to provide an example, we want to map a query q_2^{io} to a deep web source in which the input argument is a place and the output parameter corresponds to real time temperature value. A possible mapping for the input parameter is `{input1 a dbo:Place . }` This means that, if there is a subset of the graph patterns of a SPARQL query sq_2 that matches with the mapped graph pattern `{?s rdfs:label "Bratislava" . ?s rdf:type dbo:Place }` the query q_2 can be executed with the input constant `Bratislava`.

Type B. The input constant symbol is expressed as an object of a particularly defined predicate. The matching graph pattern is then `{?s [iri] [RDFLiteral | NumericLiteral | BooleanLiteral] .}` while the formulation of query in terms of abstract domains will be `{input1 a owl:DatatypeProperty; owl:samePropertyAs [iri]. }` where the abstract domain is not explicitly defined and the membership to the appropriate abstract domain can be inferred by the usage of the defined `iri` as predicate. As an example, we want to map a query q_2^{io} to a deep web source in which the input argument is a latitude and the output parameter corresponds to places associated to that specific latitude. A possible mapping for the input parameter would be represented by the RDF triples `{input1 a owl:DatatypeProperty; owl:samePropertyAs wgs84:lat. }`. As before, if there is a subset of the graph patterns of a SPARQL query sq_2 that matches with the mapped graph pattern, like `{?s wgs84:lat "41.00" .}` the query q_2 can be executed with the input constant `41.00`.

4 POLDO

The high level architecture of PoLDo is represented in Figure 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 [6], 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.

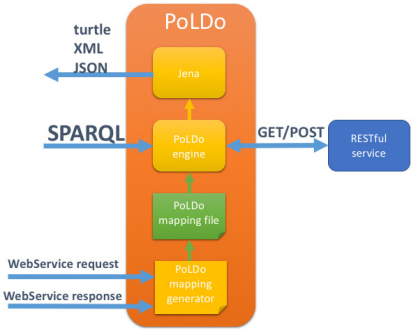

Figure 1: A high level view for the architecture of PoLDo

Jena The Jena Model is used to save a LD 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 It 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. A user is then allowed to manually assign a class of membership to resources. The Mapping Generator queries the ontologies (DBpedia and LOV) through the predicates `rdfs:domain` and `rdfs:range` and recommend all possible predicates that would link resources. Hence, when the recommended predicates have been accepted, the final mapping file is generated and can to be consumed 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 service. Moreover, it also describes the entities represented by inputs and outputs as well as their mutual relations.

4.1 PoLDo mapping language and RDF

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

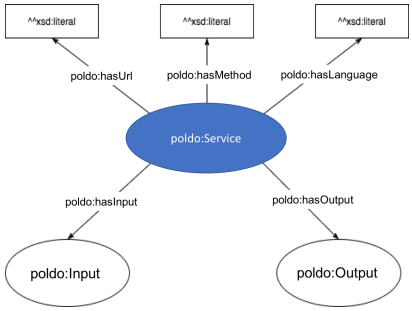

Figure 2: The class `poldo:Service` in the PoLDo mapping ontology.

service in terms of its base URL (`poldo:hasUrl`), the 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

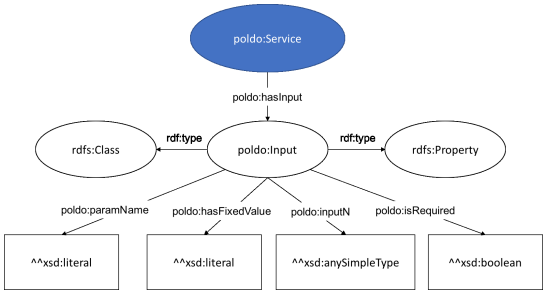

Figure 3: The class `poldo:Input` in the PoLDo mapping ontology.

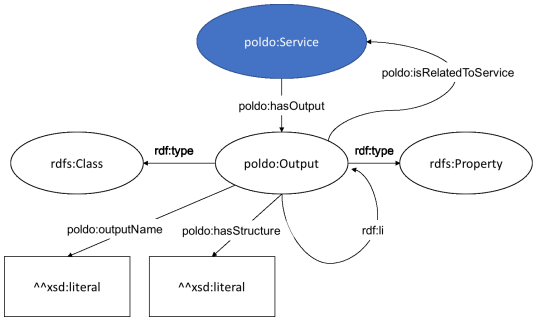

Figure 4: The class `poldo:Output` in the PoLDo mapping ontology.

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[5]. 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 is possible to model the situation when the service returns a single value or a list of values by means of the `poldo:hasStructure` and `rdf:li` statements.

4.2 PoLDo mapping generator

In this paragraph we describe all the steps required to extract parameters from an external Web Service request. These parameters form a query string, which is composed by pairs of `name-value` that are in the URL of an HTTP GET. Given a URL in a HTTP GET request, the starting location of all the pairs name-value is found after the special character ?. We can now extract all the parameters and split name and values with the character =. Found parameters, that represent input resources of the Web Service, are then used to populate the mapping file. When all the inputs have been identified, the generator will start to identify output resources from responses provided by the service itself. As responses are often in XML or JSON format, two different methods are implemented in PoLDo to handle them. After input and output have been mapped, it is possibile to assign for each resource its membership class. Generator will recommend all possible predicates to link resources, using `rdfs:domain` and `rdfs:range` and querying ontologies such as DBpedia and LOV. In the next phase, recommended predicates are accepted or edited. It either possible to add new predicates. The final mapping file can now be generated and deployed.

4.3 PoLDo engine

The core component of PoLDo is its query planner. It uses the algorithm presented in [6] to iteratively query the RESTful services and build a local cache containing the RDF version of the data. The transformations performed follow 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 exhaustive example which uses the data returned by the `flutrack` RESTful service. It detects in real time influenza symptoms, using Twitter.

service	input	output
`http://api.flutrack.org`	limit, keyword, time	latitude, longitude

Given `keyword`, `limit` and `time`, the service returns the latitude and longitude of the tweet's origin IP. The corresponding mapping file will then contain the following triples[6]:

```
poldo:api-flutrack-org- a poldo:Service ;
        poldo:hasInput      poldo:api-flutrack-org--input3 ,
                            poldo:api-flutrack-org--input2 ,
                            poldo:api-flutrack-org--input1 ;
        poldo:hasLanguage   "json" ;
        poldo:hasOutput     poldo:api-flutrack-org--output1 ;
        poldo:hasUrl        <http://api.flutrack.org/> .
```

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

[6] The prefix we use in the triples are those available via http://prefix.cc

```
poldo:api-flutrack-org--input1
        a poldo:api-flutrack-org--input1key ;
        poldo:hasFixedValue  "100" ;
        poldo:isRequired     true ;
        poldo:paramName      "limit" .

poldo:api-flutrack-org--input2
  a dbo:Disease ;
  poldo:isRequired  true ;
  poldo:paramName   "s" ;
  dbo:location poldo:api-flutrack-org--resource1 .
...
poldo:api-flutrack-org--output1
        a                          rdf:Bag ;
        rdf:li                     poldo:api-flutrack-org--output5 ,
                                   poldo:api-flutrack-org--output4 ,
                                   poldo:api-flutrack-org--output6 ,
                                   poldo:api-flutrack-org--output3 ,
                                   poldo:api-flutrack-org--output7 ,
                                   poldo:api-flutrack-org--output2 ;
        rdfs:label                 "jsonArrayRoot" ;
        poldo:hasStructure         "JSON_Array" ;
        poldo:isRelatedToService   poldo:api-flutrack-org- .

poldo:api-flutrack-org--output4
  a                          owl:DatatypeProperty ;
  rdfs:label                 "latitude" ;
  poldo:content              "String" ;
  poldo:hasStructure         "JSON_Data" ;
  poldo:isData               true ;
  poldo:isRelatedToService   poldo:api-flutrack-org- ;
  owl:samePropertyAs         geo:lat> .

poldo:api-flutrack-org--output5
   a                          owl:DatatypeProperty ;
   rdfs:label                 "longitude" ;
   poldo:content              "String" ;
   poldo:hasStructure         "JSON_Data" ;
   poldo:isData               true ;
   poldo:isRelatedToService   poldo:api-flutrack-org- ;
   owl:samePropertyAs         geo:long .
...
poldo:api-flutrack-org--resource1
   a                                  poldo:customResource ;
   poldo:api-flutrack-org--output4    xsd:double ;
   poldo:api-flutrack-org--output5    xsd:double ;
  poldo:findURI                      "poldo.GeocodeURI".
```

The first entity defined is the service itself, together with the service output language and the URL of its the endpoint. The mapping file shows how inputs and outputs are defined. Two of the three inputs (`limit` and `time`) corresponding to `input1` and `input3` are set to fixed values (100 and 7) while the keyword (parameter `s`) is defined as a `dbo:Disease`. This corresponds to Type A mapping style. The engine analyzes the query to find graph patterns such as: `{?disease a <http://dbpedia.org/ontology/Disease> . ?disease <http://www.w3.org/2000/01/rdf-schema#label> [ RDFLiteral | NumericLiteral | BooleanLiteral ] .}`. In this specific case a possible instantiation of the previous graph pattern is `{?disease a <http://dbpedia.org/ontology/Disease> . ?disease <http://www.w3.org/2000/01/rdf-schema#label> "fever" .}`

The engine recognize the pattern and `poldo:api-flutrack-org-` service is invoked. The mapping shows us the expected result: a JSON array associated to the object `jsonArrayRoot`. In this array, we can find five outputs that we can deal with as an RDF list. `output4` and `output5`, in particular, are latitude and longitude of the tweet and they are defined by following the Type B style. This mapping example provides an interesting feature, the definition of `poldo:api-flutrack-org--resource1`, a new resource generated by composing the informations from `output4` and `output5` and executing a custom function. In this case, we realized a look-up service that, given latitude and longitude, it calls Geocode.xyz and DBpedia Lookup and then it returns the corresponding DBpedia resource.

Now suppose we want to know the cities with more than 1M inhabitants in which fever symptoms were reported on Twitter. The information we get just form the service is not sufficient to answer the query but as it can now be queried through SPARQL 1.1, a federated query is now feasible:

```
SELECT  ?place
WHERE {
    ?disease a dbo:Disease  .
    ?disease rdfs:label "fever" .
    ?disease dbo:location ?place .

    SERVICE <http://dbpedia.org/sparql> {
        ?place rdf:type dbo:Place ;
               dbo:populationTotal ?population .
        FILTER (?population > 1000000).

    }
}
```

PoLDo engine also supports more services in the same mapping file. In this scenario, by using the outputs (constants) of the first invoked service, the engine queries the second service (by checking the abstract domains) and, if the remote call returns some results, these are added to the knowledge base. In this case the engine uses the constants returned by the second service to query again the first one thus continuing its search of the answer for 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. If we consider the previous SPARQL query, the answer to the query can be found, hence the following results are returned.

place	population
http://dbpedia.org/resource/Chennai	7088000
http://dbpedia.org/resource/Melbourne	4440000
http://dbpedia.org/resource/Kolkata	4496694
http://dbpedia.org/resource/Dubai	2459068
http://dbpedia.org/resource/Birmingham	1101360
http://dbpedia.org/resource/London	8538689
http://dbpedia.org/resource/Brooklyn	2621793
http://dbpedia.org/resource/New_York_City	8491079
http://dbpedia.org/resource/Varanasi	1201815
http://dbpedia.org/resource/Philadelphia	1560297
http://dbpedia.org/resource/Dallas	1197816
http://dbpedia.org/resource/Miyazaki_Prefecture	1128412
http://dbpedia.org/resource/Singapore	5535000
http://dbpedia.org/resource/Tokyo	13506607

5 CONCLUSION AND FUTURE WORK

In this paper we presented PoLDo, a system prototype 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] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. 2002. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *Proc. of ICDE*. 5–16.
[2] Rosa Alarcon and Erik Wilde. 2010. Linking Data from RESTful Services. In *Proceedings of the WWW2010 Workshop on Linked Data on the Web, LDOW 2010, Raleigh, USA, April 27, 2010.*
[3] Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked Data – The Story So Far. *Int. J. Semantic Web Inf. Syst* 5, 3 (2009), 1–22.
[4] Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. 2013. Data integration under integrity constraints. In *Seminal Contributions to Information Systems Engineering*. Springer, 335–352.
[5] Andrea Calì and Davide Martinenghi. 2008. Conjunctive Query Containment under Access Limitations. In *Proc. of ER 2008*. 326–340.
[6] Andrea Calì and Davide Martinenghi. 2008. Querying Data under Access Limitations. In *Proc. of ICDE*. 50–59.
[7] Andrea Calì, Davide Martinenghi, and Riccardo Torlone. 2016. Keyword Queries over the Deep Web. In *Proc. of ER 2016*. 260–268.
[8] Kevin Chen-Chuan Chang, Bin He, and Zhen Zhang. 2005. Toward Large Scale Integration: Building a MetaQuerier over Databases on the Web. In *Proc. of CIDR*. 44–55.
[9] Roy T. Fielding and Richard N. Taylor. 2002. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology* 2, 2 (2002), 115–150.
[10] Vagelis Hristidis and Yannis Papakonstantinou. 2002. Discover: Keyword Search in Relational Databases. In *Proc. of VLDB*.
[11] Govind Kabra, Zhen Zhang, and Kevin Chen-Chuan Chang. 2007. Dewex: An Exploration Facility for Enabling the Deep Web Integration. In *Proc. of ICDE*. 1511–1512.
[12] Jayant Madhavan, Loredana Afanasiev, Lyublena Antova, and Alon Y. Halevy. 2009. Harnessing the Deep Web: Present and Future. In *Proc. of CIDR*.
[13] Kevin R. Page, David C. De Roure, and Kirk Martinez. 2011. REST and Linked Data: A Match Made for Domain Driven Development?. In *Proceedings of the Second International Workshop on RESTful Design (WS-REST '11)*. ACM, 22–25.
[14] Ahmet Soylu, Felix M dritscher, Fridolin Wild, Patrick De Causmaecker, and Piet Desmet. 2012. Mashups by orchestration and widgetâĂŘbased personal environments: Key challenges, solution strategies, and an application. *Program* 46, 4 (2012), 383–428.
[15] Steffen Stadtmüller and Andreas Harth. 2012. Towards Data-driven Programming for RESTful Linked Data. In *Proceedings of the ISWC 2012 workshop on Programming the Semantic Web*. -.
[16] Steffen Stadtmüller, Sebastian Speiser, Andreas Harth, and Rudi Studer. 2013. Data-Fu: A Language and an Interpreter for Interaction with Read/Write Linked Data. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13)*. ACM, 1225–1236.