A Framework for Searching Semantic Data and Services with SPARQL

Mohamed Lamine Mouhoub, Daniela Grigori, and Maude Manouvrier

PSL, Université Paris-Dauphine, 75775 Paris Cedex 16, France
CNRS, LAMSADE UMR 7243
{mohamed.mouhoub,daniela.grigori,maude.manouvrier}@dauphine.fr

Abstract. The last years witnessed the success of Linked Open Data (LOD) project and the growing amount of semantic data sources available on the web. However, there is still a lot of data that will not be published as a fully materialized knowledge base (dynamic data, data with limited acces patterns, etc). Such data is in general available through web api or web services. In this paper, we introduce a SPARQL-driven approach for searching linked data and relevant services. In our framework, a user data query is analyzed and transformed into service requests. The resulting service requests, formatted for different semantic web services languages, are addressed to services repositories. Our system also features automatic web service composition to help finding more answers for user queries. The intended applications for such a framework vary from mashups development to aggregated search.

1 Introduction

The last years witnessed the success of Linked Open Data (LOD) project and the growing amount of semantic data sources available on the web (public sector data published by several government initiatives, scientific data facilitating collaboration, ...). The Linked Open Data cloud, representing a large portion of the semantic web, comprises more then 2000 datasets that are interlinked by RDF links, most of them offering a SPARQL endpoint (according to LODstats[1] as of May 2014) . To exploit these interlinked data sources, federated query processing techniques were proposed ([1]). However, as mentioned in [2] there is still a lot of data that will not be published as a fully materialized knowledge base like:

- dynamic data issued from sensors
- data that is computed on demand depending on a large sets of input data, e.g. the faster public transport connection between two city points
- data with limited access patterns, e.g. prices of hotels may be available for specific requests in order to allow different pricing policies.

Such data is in general available through web API or web services. In order to allow services to be automatically discovered and composed, research works

[1] http://stats.lod2.eu/

X. Franch et al. (Eds.): ICSOC 2014, LNCS 8831, pp. 123–138, 2014.

in the domain of the semantic web proposed to use machine-readable semantic markup for their description. Semantic web services (SWS) approaches include expressive languages like OWL-S[2], WSMO for complex business services or, more recently, simple vocabularies like MSM to publish various service descriptions as linked data. Most of the SWS description languages are RDF[3]-based (such as OWL-S, MSM) or offer a RDF representation (WSML). Therefore, existing tools for publishing SWS like iServe[4] are basically RDF stores that allow access via SPARQL endpoints and hence, they can be considered also as a part of the LOD.

The integration of LOD data and semantic web services (SWS) offer great opportunities for creating mashups and searching complementary data (data that does not exist on the LOD or that is incomplete or not updated). However, relevant services must be discovered first, and in case they don't exist, composed from atomic services. To achieve such a goal, an user should:

- have an awareness of the existing SWS repositories on the web,
- have a knowledge of the heterogeneous SWS description languages,
- express his needs in terms of the vocabulary used by different repositories
- find relevant services from different repositories and use service composition tools in case a service satisfying his goal does not exist.

As this manual process requires a lot of knowledge and effort for the user, our goal is to provide a framework for searching data and related services on the LOD. We are not aware of other federated approaches able to find data and related services in the LOD. An approach for aggregated search of data and services was proposed in [3], but it requires building global schemas for data and services and lacks a full support for the LOD and for semantic queries.

In this paper we make the following contributions:

- a SPARQL-driven framework to search data and related services in the distributed and dynamic setting characterizing the LOD
- a method to derive a service discovery query from a data query and enrich it in order to increase the number of retrieved services
- a method to find a web service composition on the fly, containing WS from different repositories.

The rest of this paper is structured as follows: In section 2, we highlight the overall functionality of the framework with a motivating scenario and give the important definitions. Section 3 is dedicated to the service discovery. Service composition is explained in section 4. The architecture and implementation details are described in section 5. The last sections are dedicated to the related works and the conclusion.

2 Data and Service Querying

The goal of our framework is to extend a search of linked data with a service discovery/composition to find relevant services that provide complementary data.

[2] `http://www.w3.org/Submission/OWL-S`
[3] `http://www.w3.org/RDF/`
[4] `http://iserve.kmi.open.ac.uk/`

Such a search often requires distinct queries: a) data queries to lookup in the LOD to find data b) service requests to discover relevant services in some SWS repositories and c) service composition requests to create relevant service compositions in case no single relevant service is found. Our framework searches for both (data and services) starting from a single query from the user called the data query, i.e. a query intended to search only for data. From this query, it automatically issues service requests and finds relevant services or generates service compositions.

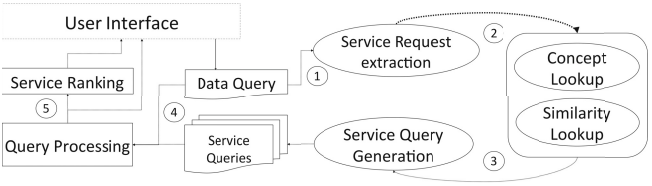

Fig. 1. Process of discovering services with a data query

Figure 1 shows an overview of our approach to search for services in parallel to data. When a SPARQL data query is submitted by a user or an agent, two parallel search processes are launched:

1. Data search process: A process to manage the query answering in the LOD data sources. These sources are distributed and accessible via SPARQL endpoints. Thus, a SPARQL-federation approach along with the appropriate optimization and query rewriting techniques is used for this purpose. This process is out of the scope of this paper.
2. Service search process: A process to discover and possibly compose services that are relevant to the data query. An analysis of the data query is required in order to transform it into one or multiple service requests.

```
SELECT ?person ?book
WHERE {
?person rdf:type    dbpedia-owl:Writer ;
    dbpedia-owl:award   ?prize ;
    dbpedia-owl:birthPlace  dbpedia:Paris .
?book   dbpedia-owl:author  ?person ;
    dbpedia-owl:isbn    ?isbn .}
```

Listing 1.1. Example Data Query Q_D

To explain the motivations and goals of our framework, we consider the following example scenario: A user wants to know all writers born in Paris and holding a Nobel prize as well as the list of all their books. This query is written in SPARQL in listing 1. Answers for this query in the LOD might supposedly find all these writers in DBpedia. However, their published books are not all listed in DBpedia. In this case, data is not complete and might need to be completed with full book listings from services like Amazon API, Google Books API, etc Some of the latter APIs can also provide complementary information on the books such as the prices, ISBN numbers, etc. In addition, there are some other

relevant services that allow the user to buy a given book given online. However, if the user wants to buy a given book from a local store, and there is a service that only takes an ISBN number as input to return the available local stores that sell this book, in that case, a service composition can be made to return such information.

2.1 Definitions

To better explain the details of service search we first give the following definitions for the context of the search in this section.

SPARQL Query Overview : A SPARQL query can be seen as a set of one or many graph patterns composed of nodes and edges. Nodes represent variables (prefixed by a '?') or concrete values (resource URIs, Literals) and edges represent properties that link nodes in a pairwise fashion. A subgraph composed of two nodes linked by a property edge is called a triple pattern and is read "subject property object". Multiple group graph patterns in SPARQL refers to queries containing multiple triple blocks separated or contained by `UNION, OPTIONAL,` brackets, etc. In case there is a single graph pattern, it is called a basic graph pattern.

Nodes and Concepts (n, c_n) : We define a node $n \in N$ in the context of a query as a part of tuple (n, c_n) where c_n is its corresponding concept formally defined by: $(n, c_n) : (n \in N, c_n = Concept(n))$.
A node is either a named variable or a concrete element of a triple pattern (a Literal or a resource URI).
A Concept is the reference rdfs:class that is used to describe the rdf:type of a node in its reference ontology Θ. It is obtained with the function $Concept(n)$.

Data Query (Q_D) : A data query Q_D is a SPARQL query composed of sets of triple patterns and selection variables. It is basically written by the user to fetch data from LOD that match these triples. Listing 1 shows an example of a data query for the provided example above. In this paper, we only consider SELECT queries that have a unique basic graph pattern.

Service Request (R_s) : Given a user SPARQL query Q_D, a service request $R_s = (In_D, Out_D)$ is a couple of two sets In_D, Out_D created by analyzing Q_D in order to extract inputs and outputs that could be considered as parameters of a service request to find relevant services for Q_D. $In_D = \{(n, c_n)\}$ is a set of service inputs provided implicitly by the user in Q_D in form of Literals or URIs in the triple patterns of the `WHERE` clause. $Out_D = \{(n, c_n)\}$ is a set of service outputs that are explicitly requested by the user in the query in form of variables in the `SELECT` clause. More details are provided in section 4.1

Service Descriptions (D_s) : In a service collection S, every service s is described by $D_s = (In_S, Out_S)$ where In_S is the set of inputs needed for a service s and Out_S is the set of outputs provided by the service. A service description can be in any known SWS formalism that is RDF/OWL based and that describes the functional and the non-functional features of a service. Currently in our work, we are only interested in the inputs and outputs of a service which are parts of the functional features.

Similar Concepts (e_n) : For a given concept of a node c_n, there exists a set of one or more equivalent (similar) concepts $e_n = Similar(c_n)$ where $Similar(c_n)$ is a function that returns the similar concepts of a given concept defined in its ontology by one of the following rdfs:property predicates: a) `owl:sameAs` b) `owl:equivalentClass` and c) `rdfs:subClassOf` in either directions.

Service Query (Q_s) : Similarly in the Q_D definition above, the service query is a SPARQL query written to select relevant services from their SWS repositories via their SPARQL endpoints . It consists of sets of triple patterns that match the inputs and outputs of R_s with inputs and outputs of a service in S. The triples of Q_s follow the SWS description model used by the repositories to describe services.

3 Service Discovery with SPARQL

To deal with the heterogeneity of the SWS descriptions and the distributed deployments of repositories containing them, we choose to issue service requests in SPARQL queries and adapt them to each description model based on the following assumptions: a) the data in question adheres to the principles of linked data as defined in [4] b) SWS are described by RDF based languages such as OWL-S or MSM[5], c) SWS repositories offer access via SPARQL endpoints to their content.

In addition, existing SWS repositories such as iServe are accessible via SPARQL endpoints. This allows to select SWS and perform explicit RDF entailment on their descriptions to extend the search capabilities. The RDF entailment is done explicitly by rewriting SPARQL queries since the existing implementations SPARQL engines don't offer this feature. Furthermore, using SPARQL allows to deal with the heterogeneous SWS descriptions more effectively without intermediate mapping tools.

We distinguish two kinds of service queries that can be relevant depending on the goal of the discovery. For a given service request R_s extracted from a data query Q_D, the user may want to find one of following kinds of services:

1. Services that provide all the information requested by the user, i.e provide all the requested outputs regardless of the given inputs. However, the more a service consumes the inputs of the request, the more relevant it is. For example, taking into account the location as input returns data that concerns

this location. Such services would be useful as an alternative or an additional data source to the LOD data. They are obtained by applying Strategy#1:
$Strategy\#1(R_s, D_s) : \{\forall o \in Out_D : o \in Out_S\}[\implies (Out_D \subseteq Out_S)]$
A specialization of this strategy, called $Strategy\#1_{exact}$, restricts the relevance on services that, in addition, consume only and only all the given inputs In_D.
$Strategy\#1_{exact}(R_s, D_s) : \{\forall i_d \in In_D, \forall i_s \in In_S, \forall o_d \in Out_D : i_d \in In_S \wedge i_s \in In_D \wedge o_d \in Out_S\}[\implies (In_D = In_S) \wedge (Out_D \supseteq Out_S)]$

2. Services that consume some of the inputs or the outputs of the request, or that return some of the inputs or the outputs of the request. Such services would be useful to: a) provide additional information or services to the data, b) discover candidate services for a mashup or composition of services that fit as providers or consumers in any intermediate step of the composition. The service request for such kind of services is obtained by one of the strategies bellow that satisfy the following:
$Strategy\#2_a(R_s, D_s) : (In_D \cap In_S \neq \phi)$
$Strategy\#2_b(R_s, D_s) : (Out_D \cap In_S \neq \phi)$
$Strategy\#2_c(R_s, D_s) : (Out_D \cap Out_S \neq \phi)$
$Strategy\#2_d(R_s, D_s) : (In_D \cap Out_S \neq \phi)$

3.1 Service Request Extraction

The data query is analyzed to extract elements that can be used as I/O for a service request. Outputs are simply the selected variables of the query. Inputs are the bound values that appear in the triples of the query.

The analysis of the data query Q_D allows to extract the inputs and outputs of Q_D using one of the following rules:

1. Variables in the `SELECT *`, (selection variables) are considered as outputs $o_d = (n, null) \in Out_D$. Simply because they are explicitly declared as desired outputs of the data query.
2. Bindings of subjects or objects in the `WHERE` clause of Q_D, i.e literals and RDF resources URIs, are considered as inputs $i_d = (n, null) \in In_D$. This can be explained by the fact that a user providing a specific value for a subject or an object simply wants the final results to depend on that specific value. The same way, a service requiring some inputs returns results that depend on these inputs.

The service request extraction consists of populating In_D and Out_D with the nodes of the elements mentioned above. Algorithm 1 gives an overview of the Service Request Extraction.

The SPARQL operators like OPTIONAL, UNION, FILTER, etc can reveal the preferences of the user for service discovery and composition. For instance, the I/O extracted from an Optional block mean that the user doesn't require services that necessarily provide/consume the optional parts. Therefore, the service request for such a data query is obtained using some of the loose strategies defined in section 4.

Algorithm 1. Service Request Extraction

Input: Q_D
Output: In_D, Out_D
1: $Out_D.nodes \leftarrow$ GetSelectVariables(Q_D) ▷ Get the output variables
2: $triples \leftarrow$ GetAllQueryTriples(Q_D) ▷ Get all the query triples
3: **for each** t **in** $triples$ **do**
4: **if** isConcrete(subject(t)) **then** ▷ check if URI or literal
5: $In_D \leftarrow In_D \cup \{(\text{subject}(t), null)\}$
6: **else if** isConcrete(object(t)) **then**
7: **if** predicate(t) $\neq$ "$rdf : type$" **then**
8: $In_D \leftarrow In_D \cup \{(\text{object}(t), null)\}$
9: **end if**
10: **end if**
11: **end for**

Listing 1.4 shows an example of a service query extracted from Q_D in listing 1.1 using *Strategy*#1 to find services that return the same data as the query.

3.2 Semantics Lookup

Once the service request elements are extracted from the query, we try to find the semantic concepts c_n that describe the previously extracted nodes with no concept: $(n, null)$.

Concept Lookup. In general, concepts can either be declared by the user in the data query (as the user probably specifies what he is looking for) or in a graph (set of triples) in an rdf store.

The semantics lookup process starts looking for the concept of a node n in the Q_D triples. The concept is the concrete value given by a URI and linked to n via the property $rdf : type$: i.e. "n $rdf : type$ $conceptURI$". In the example query in listing 1.1, the concept of ?*person* is given in Q_D as $dbpedia - owl : Writer$, but the concept of *book* is not given in Q_D

If c_n is not found in Q_D, a concept lookup query q_c is created to look for the concept of n in the ontology in which it is suspected to be.

To generate this concept lookup query q_c, we take all the triples from Q_D in which n is involved as a subject or as an object and then insert them in the WHERE clause of q_c. We add a triple pattern "n $rdf : type$?*type*" and set the ?*type* variable as the SELECT variable of q_c. The URLs of the ontology(ies) in which c_n can be extracted from the namespaces used in Q_D and are added to the From clause of the q_c .

Listing 1.2 shows an example concept lookup query to find the concept of ?book which is not declared in Q_D (listing 1.1).

If no concept is found for a given node (most likely because of a non working namespace URL), then the search space for q_c to find the missing concepts is expanded to the other known sources in the LOD.

```
SELECT  ?bookConcept WHERE {
SERVICE <http://dbpedia.org/sparql>{
?book   dbpedia-owl:author
?person ;
    dbpedia-owl:isbn  ?isbn ;
    rdf:type  ?bookConcept .}}
```

Listing 1.2. An example query of Concept Lookup in the LOD

```
SELECT  ?bookConcept
FROM <http://dbpedia.org/ontology/>
WHERE {
dbpedia-owl:author rdfs:domain
      ?bookConcept .
}
```

Listing 1.3. An example query of Concept Lookup in Ontology

Similarity Lookup. To extend the service search space, we use the similar concepts e_n of every concept c_n in the service search queries along with the original concepts. To find these similar concepts, we use the rules given by the definition in section 2.1. Based on this definition, we issue a SPARQL query q_e like the one in the concept lookup but slightly different by adding a triple that defines a silimarity link between c_n and a variable ?$similar$. The triple pattern has the form c_n ?$semanticRelation$?$similar$ where ?$semanticRelation$ is one of the following properties: a) owl:sameAs, owl:equivalentClass for similar concepts in other ontologies b) rdfs:subClassOf for hierarchically similar concepts within the same ontology.

The similarity lookup query q_e is executed on the sources used in Q_D as well as on the other sources of the LOD because the similar concepts can be found anywhere.

To optimize the search in other sources of the LOD, we use a caching technique to build an index structure on the go of the LOD sources content. The details of this caching is described in section 5.

3.3 Service Query Generation

Once all elements of the service request are gathered, service discovery queries are issued in SPARQL using rewriting templates. Such templates define the structure and the header of the SPARQL service query. There is a single template per SWS description formalism, i.e. OWL-S, MSM, etc. For instance, the OWL-S template defines a header containing triples that match the OWL-S model by specifying that the desired variable is an OWL-S service which has profiles with specific inputs/outputs. Listing 1.4 shows an example of a service query for the example scenario in section 1. It uses an OWL-S template to specify the required input and output concepts according to the OWL-S service model.

To generate the queries, all concepts c_n and their similar concepts e_n for every node $n \in In_D \cup Out_D$ are put together in a basic graph pattern of in a union fashion depending on the chosen selection strategy. More specifically, for every input $i_d \in In_D$ we write triple patterns to match service inputs with variables that have c_n as a concept and accordingly for every output $o_d \in Out_D$.

The service search strategies (c.f. section 3) in the way we define them, describe the how tight(Strategy $\#1, \#1_{exact}$) or loose (Strategies $\#2_{a,b,c,d}$)the service selection must be. Therefore, strict strategies require that one or more inputs or outputs are matched at the same time, thus, the query triples will be put in

a single basic graph pattern. On the other hand, loose strategies require only partial matching, hence, the query triples are be put in a UNION of multiple graph patterns.

```
SELECT DISTINCT ?service WHERE {
?service a service:Service ; service:presents ?profile .
?profile profile:hasOutput ?output1 ;
         profile:hasOutput ?output2 .
?output1 process:parameterType dbpedia-owl:Writer .
?output2 process:parameterType dbpedia-owl:Book .
OPTIONAL { ?profile profile:hasInput ?intput1 .
           ?input1  process:parameterType dbpedia:Place .}
OPTIONAL { ?profile profile:hasInput ?intput2 .
           ?input2  process:parameterType dbpedia-owl:Award .}}
```

Listing 1.4. Example Service Query Q_S with $Strategy\#1_{exact}$

4 Automatic Service Composition

In the previous section we showed how to make service requests to find relevant individual services for the data query. However, if no such services exist, service composition can create relevant composite services for the matter. In this section we describe our approach to make such compositions automatically.

In the context of our framework, service repositories are part of the LOD as SPARQL endpoints. Therefore, we think that the least expensive way to perform a service discovery and composition is on the fly without any pre-processing. This online composition consists of discovering candidate services at each step of the composition without a need to have a local index or copy of the service repositories. We argue that the approaches based on pre-processing the service repositories often require an expensive maintainability to stay up-to-date. Furthermore, according to [6], the web services are considerably growing and evolving either by getting updated, deprecated or abandoned.

However, some optimization based on caching are described further in section 5 to speed-up this online process for the queries that as already been processed in the past executions.

In this section, we describe our approach for an automatic composition of SWS based on a service dependency graph and an A*-like algorithm. The first subsection is dedicated to the Service Dependency Graph while the second describes the composition algorithm.

4.1 Service Dependency Graph

The Service Dependency Graph (SDG from now on) represents the dependencies between services based on their inputs and outputs. A service depends on another if the later provides some inputs for the former. In our work, we consider that a SDG is specific for each data query because it includes only services related to that query. In other works, the SDG might represent the dependencies for all the services in a repository, but this requires a general pre-processing for the LOD as we stated before.

We use an oriented AND/OR graph structure as in [7] to represent the SDG. Such a graph is composed of AND nodes - that represent services - and OR nodes - that represent data concepts - linked by directed edges. We slightly adapt this representation to include the similarities between concepts of data by : a) Each OR node contains the set of concepts that are similar to each other b) Each edge that links an AND node to an OR node is labeled with the concept that matches the service input/output concept among those in the OR node's concept set. A dummy service N_0 is linked to outputs nodes of Out_D to guarantee that a service composition provides all the requested outputs.

The AND/OR graph representation of the SDG is more adequate for the composition problem than ordinary graphs because the constraints on the inputs of services are explicitly represented by the AND nodes; A service cannot be executed if some of its inputs are not provided; thus, an AND node cannot be accessible unless all of its entering edges are satisfied. Furthermore, this graph has been utilized in a many previous approaches and has proven its efficiency as shown in [7]. However, a classical graph representation can be used to solve the composition problem.

To construct the SDG, we use our service discovery approach to find dependencies for each service in a bottom-up approach starting from the services that provide the final outputs of Q_D. In fact, the SGD construction searches for all services that provide all the unprovided-yet data at one time starting from Out_D nodes. Such a one-time search per iteration allows to reduce the number of service requests that are sent to the SWS repositories, hence, boosting the SDG construction.

For example, to find services that provide O_1 and/or O_2, a service request $R_s(null, \{O_1, O_2\})$ is used by applying $Strategy\#2_b$.

4.2 Service Composition Algorithm

Upon the construction of the SDG, one or many compositions can be found. The aim of the service composition algorithm is to find the optimal composition from the SDG for a given composition request.

For this purpose, we use an A*-like algorithm and adapt it for AND/OR graphs. Starting from the user input In_D nodes, the algorithm finds the optimal path to the target node N_0 (which is linked to the final outputs Out_D). Therefore, an optimal solution is a path that has the least total cost and that respects the AND/OR graph structure.

The total cost of a given path is the aggregation of the costs of each step from a node to another. Generally, the cost at a given step (at an AND node n) in an A* algorithm is given by the aggregation function: $f(n) = g(n) + h(n)$ where $g(n)$ is the total cost of the sub-path from the starting point to n and $h(n)$ is a heuristic that estimates the total cost from the n to the target node N_0.

Since the semantic web services has rich descriptions, the semantics of the Inputs/Outputs can be used for cost calculation to help finding an optimal solution. Therefore, we rely on the sets of similar concepts inside OR nodes and on the labels of the edges in SDG. Therefore, the cost of a move from an AND

node n_i to $n_i + 1$ is determined based on the similarity between the labels of the input and the output edges of the two AND nodes respectively. If the two labels (concepts) are the same, then the cost value is null. Otherwise if the two labels are different but similar concepts (sameAs, sub concepts) then the cost value is set to 1. This cost calculation can be resumed by the function: $cost(n_{i+1}) = sim(c_{n_i}, c_{n_{i+1}})$ where c_{n_i} is a concept used by the current service, $c_{n_{i+1}}$ is used by the next one and:

$$sim(c_{n_i}, c_{n_{i+1}}) = \begin{cases} 0 & \text{if } c_{n_i} = c_{n_i+1} \\ 1 & \text{if } c_{n_i} = Similar(c_{n_i+1}) \end{cases} \quad (1)$$

is a function that determines the similarity between two concepts.

From the functions above, the cost of the best known path to the current node subset is given by the following function:

$$g(n) = \sum_{i=0}^{n} cost(n_i) \quad (2)$$

where n_i are all the accessible services for the next step

The heuristic function $h(n)$ calculates the distance between the current node and the target AND node n_0 in the SDG graph. This is justified by the fact that, a better solution is the one that uses less services.

$$h(n) = Distance(n, n_0) \quad (3)$$

5 Implementation and Experiments

In this section, we show briefly the architecture of our framework and some experiments as a proof of concept.

5.1 Framework Architecture

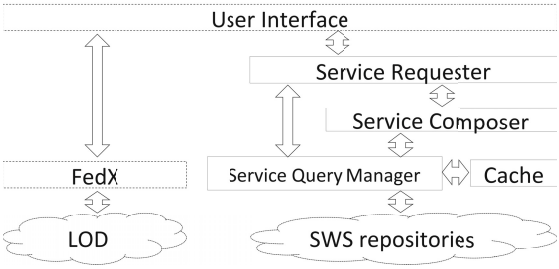

Fig. 2. Framework Architecture

Figure 2 shows an overview of the architecture of our framework. Through an interface, SPARQL queries are submitted to the system to be processed for data search and service search.

The data querying is managed by an external open source SPARQL federator, FedX [1]. FedX uses its own query rewriting to optimize the data querying for each source. Therefore, the LOD is a federation of SPARQL endpoints of different data sources such as DBpedia.

On the service side, queries are processed by the service requester to make service requests or service compositions. The SWS repositories which are SPARQL endpoints as well are considered as a particular part of the LOD. We use our own federation of SPARQL endpoints to query the SWS repositories separately. The reason why we don't simply reuse FedX is because we need specific optimization for service descriptions different than the general purpose optimization offered by FedX. A brief overview of our optimization is described in the next subsection.

We have implemented our framework in Java using Apache Jena[5] framework to manage SPARQL queries and RDF.

5.2 Optimizing Service Discovery with Cache

In order to optimize the service discovery in terms of response time, we use a caching for services and concepts. Such a cache indexes all the concepts and services that has been used in past requests.

We use three different types of cache : a) A cache for similar concepts to decrease the number the similarity lookup requests. b) A cache to index the concepts that have been used in the past and the URIs of services and repositories that use them. c) a local RDF repository to keep in cache the descriptions of services on the go once they are discovered. This later one can be queried directly via a local SPARQL endpoint.

Maintaining the cache costs much less than maintaining a whole index structure of all known SWS repositories and does not require any pre-processing prior to use the framework. Cache maintenance can be scheduled for automatic launch or triggered manually.

5.3 Experiments and Evaluation

Our main challenge to evaluate our framework is to find suitable benchmarks that provide SPARQL queries on real world data and to find SWS repositories of real world services. Furthermore, to properly measure the execution time of writing service queries from data queries, we need test queries that are more or less complex and have missing concept declarations.

Unfortunately, to our best knowledge, there is no benchmark that allows us to fully measure the performance of our framework. Therefore, to prove the feasibility of our approach to search services on the LOD, we have made an implementation as a proof-of-concept and some experiments to measure the execution time of query rewriting from a data query and through semantics lookup

[5] `https://jena.apache.org/`

to write service queries in SPARQL. For experiments we used a set of SPARQL queries that we wrote manually to have missing concepts.

Figure 3 shows a summary of our experiments on a set of queries. This set consists of a 10 queries, each with an increasing amount of undefined variables. We measured separately the total execution time of writing service queries including the execution time of the concept lookup process for each query. The results show that the concept lookup time increases linearly as the number of undefined variables increase.

We performed a partial evaluation for the effectiveness of our service discovery on OWL-S-TC[6] benchmark. Figure 4 show the number of false negatives (<0)and false positives (>0) of the service discovery on a set of 18 OWL-TC queries that have been used for evaluation in [3]. We have rewritten these queries in SPARQL to make them usable within our framework. The results show an overall good error rate. However, in some queries like Q22, some of the I/O parameters are very generic which explains the high number of false positives. In order to avoid such cases, the algorithm must be modified to select services that provide at least a non-generic I/O parameter. For the false negatives like in Q7, the reference matching results in OWLS-TC are set based on other features than I/O parameters such as the textual description of the service, etc.

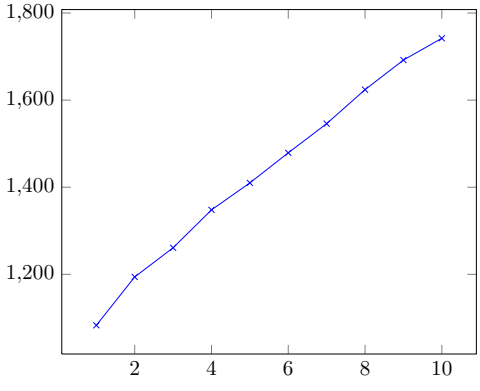

Fig. 3. Average execution Time in MS per number of undefined variables in a random query

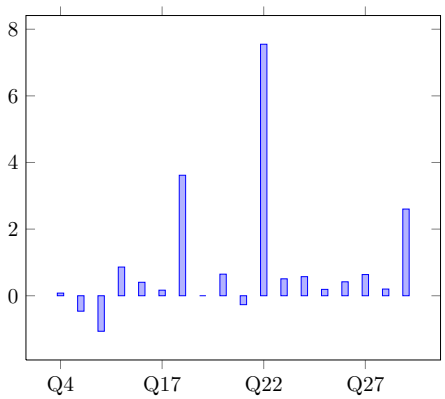

Fig. 4. False Negatives and false Positives on OWL-S TC queries

6 Related Works

The motivations and research questions of our work are tackled by many recent works. In fact, our work emerges from a crossing of many research topics in the semantic web and web services. We'll list few of the most recent and relevant works to our paper.

[6] `http://projects.semwebcentral.org/projects/owls-tc/`

SPARQL Query Management. Among the works that tackle the query management in the LOD, SPARQL federation approaches are the most relevant for our context. FedX[1] is one of the most popular works that has good performance results besides the fact that the tool is available in open source. FedX optimizes the query management by performing a cache-based source selection and rewriting queries into sub-queries and running them on the selected sources. Some recent works like [8] introduce some further optimization for FedX and other works by optimizing the source selection. We are actually using FedX as a part of our Framework for answering data queries because, as we stated in section 2, managing data queries is out of the scope of our work in this paper.

Service Discovery. Our context of service discovery involves exclusively the semantic web services. SWS discovery is the topic of interest of many recent works and benchmarks as shown in the survey[9]. The SWS discovery approaches are either semantically based, textual based or hybrid based. The first ones are the most relevant for our context because we operate on linked data which is meant to be properly described and linked. Among the recent works, [10] introduces a repository filtering using SPARQL queries to be used on top of the existing hybrid discovery approaches. However, in our context, SPARQL queries are sufficient for performing a service discovery in SWS repositories. In addition, [10] and the other existing approaches need a service request to operate, in contrast to our work in which service requests are implicit in a data query and have to be extracted first.

For discovery evaluation, OWL-S-TC is the reference benchmark for SWS. However, in our context, we need a benchmark that is based on real-world services because we need to find services for data that exists in the LOD. Unfortunately, for the moment, there are only a few SWS to consider in the real world as stated and agreed on by many researchers [6]. However, there are some tools such as Karma [11] that allow to wrap classical web APIs into semantic APIs and therefore help creating new SWS on top of the APIs.

Service Composition. Similarly to service discovery, the automatic composition of SWS has been the subject of many works, surveys [12] and challenges (WS-Challenge). In general, the automatic composition algorithms The most recent works like [13], [7] still use A*-based algorithms to find composition plans in an SDG graph which is mostly pre-constructed for all known services in a repository. In our paper we use a service composition approach that is very similar to the WSC challenge winner [7] that uses AND/OR graphs. However, we adapted their approach to take advantage of the semantics in cost calculation instead of using a static cost calculation (a fixed cost for all nodes).

Search of Data and Services. Our work is inspired by the work in [3] which aims to look for services that are related to a given query based on keywords comparison between an SQL-like query and a service ontology. This approach uses generated semantics for services to expand the search area.

Another similar work in [14] called ANGIE consists of enriching the LOD from RESTful APIs and SOAP services by discovering, composing and invoking services to answer a user query. However, this work assumes the existence of a global schema for both data and services which is not the case in the LOD. This assumption makes ANGIE domain specific and not suitable for general purpose queries.

Some recent works could complement our work such as [15] which proposes an approach that uses Karma[11] to integrate linked data on-the-fly from static and dynamic sources and to manage the data updates.

7 Conclusion and Perspectives

In this paper we presented a framework for finding data and relevant services in the LOD using a unique SPARQL query. Our framework helps the user to find services that he could exploit to construct mashups or to complement the data found in materialized knowledge bases. We implemented the proposed algorithms and we are evaluating them in terms of efficiency and quality. We plan to enrich the framework by storing and exploiting user actions (selected services and compositions for a given data query) in order to improve the efficiency of the algorithm and the relevance of the retrieved services.

Regarding the previously mentioned issue of lacking real-world SWS, Karma[11] or SmartLink[16] can be used to provide our experiments with SWS from real-world APIs. We plan to use such tools in the future to extend our experiments and have a clear measure of its effectiveness.

References

1. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization techniques for federated query processing on linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 601–616. Springer, Heidelberg (2011)
2. Speiser, S., Harth, A.: Integrating linked data and services with linked data services. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 170–184. Springer, Heidelberg (2011)
3. Palmonari, M., Sala, A., Maurino, A., Guerra, F., Pasi, G., Frisoni, G.: Aggregated search of data and services. Information Systems 36(2), 134–150 (2011)
4. Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. Intl. journal on semantic web and information systems 5(3), 1–22 (2009)
5. Kopecky, J., Gomadam, K., Vitvar, T.: hrests: An html microformat for describing restful web services. In: IEEE/WIC/ACM Intl. Conf. on. Web Intelligence and Intelligent Agent Technology, WI-IAT 2008, vol. 1, pp. 619–625. IEEE (2008)
6. Blthoff, F., Maleshkova, M.: Restful or restless - current state of today's top web apis. In: 11th ESWC 2014 (ESWC 2014) (May 2014)
7. Yan, Y., Xu, B., Gu, Z.: Automatic service composition using and/or graph. In: 2008 10th IEEE Conf. on E-Commerce Technology and the Fifth IEEE Conf. on Enterprise Computing, E-Commerce and E-Services, pp. 335–338. IEEE (2008)

8. Saleem, M., Ngonga Ngomo, A.-C.: HiBISCuS: Hypergraph-based source selection for SPARQL endpoint federation. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8465, pp. 176–191. Springer, Heidelberg (2014)
9. Ngan, L.D., Kanagasabai, R.: Semantic web service discovery: state-of-the-art and research challenges. Personal and ubiquitous computing 17(8), 1741–1752 (2013)
10. García, J.M., Ruiz, D., Ruiz-Cortés, A.: Improving semantic web services discovery using sparql-based repository filtering. Web Semantics: Science, Services and Agents on the World Wide Web 17, 12–24 (2012)
11. Taheriyan, M., Knoblock, C.A., Szekely, P., Ambite, J.L.: Rapidly integrating services into the linked data cloud. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 559–574. Springer, Heidelberg (2012)
12. Syu, Y., Ma, S.P., Kuo, J.Y., FanJiang, Y.Y.: A survey on automated service composition methods and related techniques. In: 2012 IEEE Ninth Intl. Conf. on. Services Computing (SCC), pp. 290–297 (June 2012)
13. Rodriguez-Mier, P., Mucientes, M., Vidal, J.C., Lama, M.: An optimal and complete algorithm for automatic web service composition. Intl. Journal of Web Services Research (IJWSR) 9(2), 1–20 (2012)
14. Preda, N., Suchanek, F.M., Kasneci, G., Neumann, T., Ramanath, M., Weikum, G.: Angie: Active knowledge for interactive exploration. Proc. of the VLDB Endowment 2(2), 1570–1573 (2009)
15. Harth, A., Knoblock, C.A., Stadtmüller, S., Studer, R., Szekely, P.: On-the-fly integration of static and dynamic sources. In: Proceedings of the Fourth International Workshop on Consuming Linked Data (COLD 2013) (2013)
16. Dietze, S., Yu, H.Q., Pedrinaci, C., Liu, D., Domingue, J.: SmartLink: A web-based editor and search environment for linked services. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part II. LNCS, vol. 6644, pp. 436–440. Springer, Heidelberg (2011)