Aggregated Search of Data and Services

Mohamed Lamine Mouhoub

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à l'Université Paris Dauphine

Recherche Agrégée de Données et Services

Aggregated Search of Data and Services

École doctorale n°543

ÉCOLE DOCTORALE DE DAUPHINE

Spécialité INFORMATIQUE

Soutenue par
Mohamed Lamine MOUHOUB
le 11 Décembre 2017

Dirigée par **Daniela GRIGORI**

COMPOSITION DU JURY :

M. Bernd AMANN
Université Pierre et Marie Curie
Président du jury

Mme Daniela GRIGORI
Université Paris Dauphine
Directrice de thèse

Mme Maude MANOUVRIER
Université Paris Dauphine
Co-directrice de thèse

Mme Salima BENBERNOU
Université Paris Descartes
Rapporteure

M. Dan Vodislav
Université de Cergy-Pontoise
Rapporteur

M. Matteo Luigi PALMONARI
Université de Milan-Bicocca
Membre du jury

To my first baby boy Mohamed Abderrahmane
and for his brothers and sisters to come ...

Acknowledgements

As the arabic proverb says, *"Who doesn't thank people, doesn't thank God"*. I would like to express my sincere gratitude to my supervisors Pr. Daniela Grigori and Dr.HDR. Maude Manouvrier for their continuous support, motivation and their immense patience during the last five years. I will never be grateful enough and never forget our fruitful exchanges, our outstanding collaboration and more importantly our last-minute paper submissions.

Besides my supervisors, I would like to thank the rest of my thesis committee: Pr. Salima Benbernou who also was in my mid-term defense and had the patience to read me twice as well as Pr. Dan Vodislav and Pr. Bernd Amann for their insightful comments and encouragement. Very special thanks go to Dr Matteo-Luigi Palmonari for his previous research work that inspired my thesis for traveling from Milano to Paris just to attend my defense as a committee member.

I would like to thank my fellow doctoral students, the old and the new, for the time we spent together during those years, for the good mood that was dominating our desks and relationships. My thoughts and best wishes go for those who are still on the way. In addition I would like to express my gratitude to the staff Paris Dauphine University, especially the LAMSADE and Library staff for their helpfulness and .

I would like to thank my friends for expecting nothing less than excellence from me. Last but not the least, I would like to thank my family: my parents without whom I would have never been so far, my sister and my wife for supporting me spiritually throughout writing this thesis and throughout my whole life my as well as all my family members and in-laws.

Abstract

The last years witnessed the success of the Linked Open Data (LOD) project as well as a significantly growing amount of semantic data sources available on the web. However, there are still a lot of data not being published as fully materialized knowledge bases like as sensor data, dynamic data, data with limited access patterns, etc. Such data is in general available through web APIs or web services. Integrating such data to the LOD or in mashups would have a significant added value. However, discovering such services requires a lot of efforts from developers and a good knowledge of the existing service repositories that the current service discovery systems do not efficiently overcome.

In this thesis, we propose novel approaches and frameworks to search for semantic web services from a data integration perspective. Firstly, we introduce LIDSEARCH, a SPARQL-driven framework to search for linked data and semantic web services. Moreover, we propose an approach to enrich semantic service descriptions with Input-Output relations from ontologies to facilitate the automation of service discovery and composition. To achieve such a purpose, we apply natural language processing techniques and deep-learning-based text similarity techniques to leverage I/O relations from text to ontologies.

We validate our work with proof-of-concept frameworks and use OWLS-TC as a dataset for conducting our experiments on service search and enrichment.

Key words : Web Services, Service Discovery, Service Description Enrichment, Semantic Web, Linked Data, Natural Language Processing.

Résumé

Ces dernières années ont témoigné du succès du projet Linked Open Data (LOD) et de la croissance du nombre de sources de données sémantiques disponibles sur le web. Cependant, il y a encore beaucoup de données qui ne sont pas encore mises à disposition dans le LOD telles que les données sur demande, les données de capteurs etc. Elles sont néanmoins fournies par des API des services Web. L'intégration de ces données au LOD ou dans des applications de mashups apporterait une forte valeur ajoutée. Cependant, chercher de tels services avec les outils de découverte de services existants nécessite une connaissance préalable des répertoires de services ainsi que des ontologies utilisées pour les décrire.

Dans cette thèse, nous proposons de nouvelles approches et des cadres logiciels pour la recherche de services web sémantiques avec une perspective d'intégration de données. Premièrement, nous introduisons LIDSEARCH, un cadre applicatif piloté par SPARQL pour chercher des données et des services web sémantiques.

De plus, nous proposons une approche pour enrichir les descriptions sémantiques de services web en décrivant les relations ontologiques entre leurs entrées et leurs sorties afin de faciliter l'automatisation de la découverte et de la composition de services. Afin d'atteindre ce but, nous utilisons des techniques de traitement automatique de la langue et d'appariement de textes basées sur le deep-learning pour mieux comprendre les descriptions des services.

Nous validons notre travail avec des preuves de concept et utilisons les services et les ontologies d'OWLS-TC pour évaluer nos approches proposées de sélection et d'enrichissement.

Mots clés : Services Web, Découverte de services, Web Sémantique, Données Liées, Traitement Automatique de la Langue.

Contents

Chapter 1

Introduction

In the internet era, data is considered as one of the most valuable currencies. If it is the most valuable resource in the 21st century as oil was in the 20th century, then data search and information extraction are for data what oil exploration and extraction are for oil. Although data is somewhat cleaner than oil, a data search on the internet does not always return all the results to be found nor exactly what one is looking for. Data on the internet is raw, structured, semi-structured, clean, unclean, open, restricted, proprietary, static, on-demand, etc. Data on the internet is mainly human-readable but also machine-readable. Thanks to the advent of semantic web and Linked Data, a machine-readable version of the web is made possible and makes it possible for machines to answer search queries more efficiently. This thesis is about searching for data and data sources from a machine-as-a-user perspective.

1.1 Context

Today's World Wide Web (WWW) offers access to tons of data through different formats and access methods. General purpose search of data in such a huge network is today nowhere similar to looking for a needle in a haystack thanks to the advanced search engines like Google, Yahoo, Bing, etc and the underlying technologies used for this purpose. However, in order for search engines and the machines behind to better understand the user needs and respond efficiently to his queries, web data has to rely on structured models. Schema.org[1] is an effort by major internet companies towards such a machine-understandable web. Furthermore, when it comes to automating the generation, consumption, reasoning and integration of data, web data has to respect a number of conditions in order to make it readable, understandable and autonomously usable by machines. Linked data is another huge effort towards such aims.

[1] `http://schema.org/`

1.1.1 Linked Data and Linked Open Data cloud

Since the advent of the semantic web to make the web machine-understandable and the web data more structured, many organizations like DBpedia and Geonames started publishing their data in a semantic web format. Later in 2006 [Bizer et al., 2009], the World Wild Web inventor Tim Berners Lee coinded the term "*Linked Data*" and defined it as "semantic web done right". It is about a set of best practices and rules called "*Linked Data Principles*" to publish semantic web data (aka web of data) properly and making it interlinked, explorable and "universally" understandable.

The large adoption of the Linked Data principles resulted in the *Linked Open Data Cloud* (LOD), a large network of linked datasets released under open licenses. It includes inter alia public sector data published by several government initiatives, life science and scientific data facilitating collaboration, linguistic data, social media, geographic data, academic publications data such as dblp, cross-domain data like DBpedia, Freebase, YAGO, etc. It contains about 150 billion RDF triples (LODstats[2]). Figure 1.1 shows the growth of the Linked Open Data cloud since 2007. In the last three years (2014-2017), the number of linked datasets has doubled. A visual of the actual diagram of the LOD is given in Fig. 1.2.

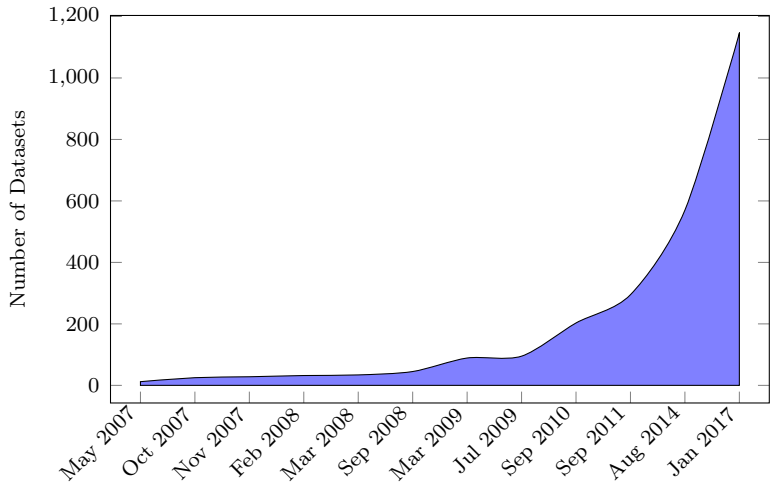


Figure 1.1: Growth in linked open datasets in the LOD since 2007

However, as mentioned in [Speiser and Harth, 2011a] there are still a lot of data that will not be published as a fully materialized knowledge base such as:

[2]Statistics as of July 2017. Source: `http://stats.lod2.eu/`

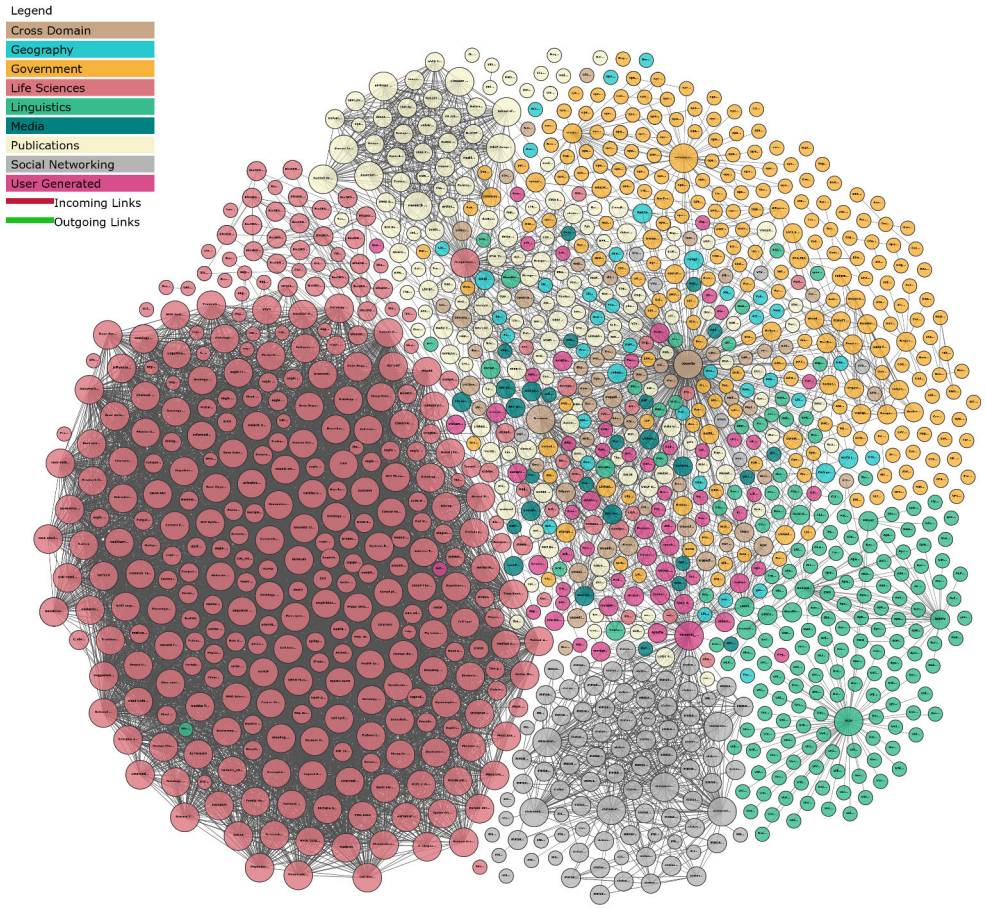
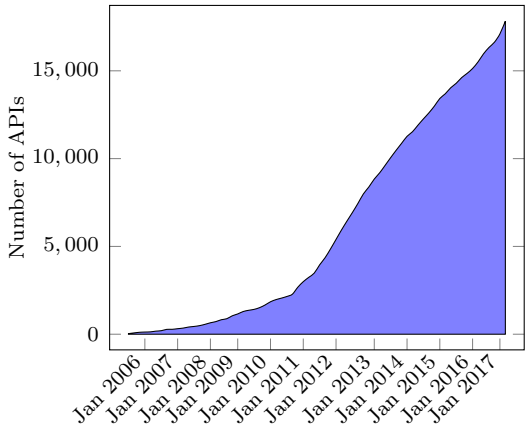


Figure 1.2: The linking Open Data Cloud Diagram as of August 2017[3]

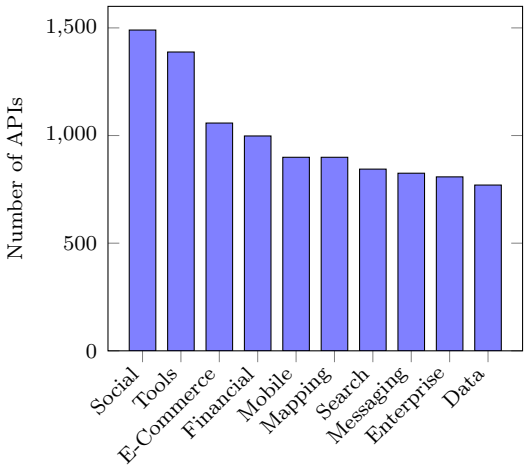
- dynamic data issued from sensors,
- data that is computed on demand depending on a large sets of input data, e.g. the faster public transport connection between two city points,
- data with limited access patterns, e.g. prices of hotels may be available for specific requests in order to allow different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

Such data is in general available through web APIs or web services. In the next subsection we present semantic web services and Web APIs and how they can be integrated with the LOD.

[3] `http://lod-cloud.net/`



(a) Growth in web APIs hosted listed by ProgrammableWeb since 2005



(b) Top 10 API categories in ProgrammableWeb in 2016

Figure 1.3: Web API statistics in ProgrammableWeb

1.1.2 Semantic Web services and APIs

Web services emerged long before the semantic web that we know today, however, they were originally motivated by a common purpose. The world wide web is static in a sense that it was designed for an interaction between humans and applications (machines) and doesn't allow applications to be easily reusable out of their pre-destined contexts and scenarios [Fensel et al., 2011]. Web services emerged to allow applications to interact, interoperate and share information while being reusable in different contexts. From this perspective, semantic web and web services both aim to democratize the web for machines, although the first addresses data while the second addresses applications.

Web APIs are a simplified category of web services that promote simplicity of design and ease of use by developers. They have known a great success over the last years as shown in Fig. 1.3a . According to ProgrammableWeb[4], the biggest repository of web APIs, the number of available web APIs have doubled in the last five years and it continues to grow significantly by more than 10% every year as indicated in Table 1.1.

Web services offer the possibility to create mashups or composite services that interact with different services and data sources to provide new added value that takes form in aggregated data or new services.

In order to allow web services to be automatically discovered and composed, research

[4] `https://www.programmableweb.com/`

Table 1.1: API count growth in ProgrammableWeb as of 2017

Total new APIs added since 2014	5,946
Average new APIs added yearly	1,982
Average new APIs added monthly	165

works 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[5], WSMO[6] for complex business services or, more recently, simple vocabularies like MSM[7] to publish various service descriptions as linked data. Most of the SWS description languages are RDF[8]-based (such as OWL-S, MSM) or offer an RDF representation (WSML). Therefore, existing tools for publishing SWS like iServe[9] are basically RDF stores that allow access via SPARQL endpoints and therefore, they can be considered also as a part of the LOD.

1.2 Motivations and contributions

The integration of LOD data and semantic web services (SWS) offers great opportunities for creating mashups and automatic service compositions. Moreover, such integration solves some existing issues and open challenges of the LOD that affect its data quality such as:

- Missing data: Some entities do not exist on the LOD yet. For example, the lists of all movies by a director or the books of a given author are not complete on the LOD as many of them don't have dedicated wikipedia/dbpedia pages yet. However, for this example, such data can be easily found using Amazon API[10], OMDbAPI[11], etc.

- Incomplete data: Some information about an entity (some of its attributes) can be missing on the LOD. For example, the information on book prices, where to buy

[5] `http://www.w3.org/Submission/OWL-S`
[6] `http://www.w3.org/Submission/WSMO/`
[7] `http://kmi.github.io/iserve/latest/data-model.html`
[8] `http://www.w3.org/RDF/`
[9] `http://iserve.kmi.open.ac.uk/`
[10] `http://developer.amazonservices.fr/`
[11] `http://www.omdbapi.com/`

them, theaters in which movies are played, etc are additional pieces of information that don't exist on the LOD.

- Nonexistent data: These issues are due to the nonexistence of some categories of data in the LOD. For example, business repositories, the social networks data, social graphs, connections, tweets, dynamic data, on-demand data , etc.

- Outdated data: The LOD data can be outdated quickly depending on its type like statistics, prices, etc. For example, statistics about population and share prices of companies are not updated for many entries in DBpedia.

However, services that would be suitable for solving the aforementioned data quality issues need to be discovered, and in case they don't exist, they need be composed from atomic services. To achieve such a goal, a developer who wants to integrate data and services 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.

This manual process requires a lot of knowledge and effort from the user. We aim to provide a framework for searching data and related services on the LOD that might bring some added value to the data.

This thesis was initially inspired by the research work in [Palmonari et al., 2011] which portrays an approach to search for data and services at the same time using an SQL-like querying language by leveraging non-semantic web services with semantic annotations. This work builds on top of this motivation and makes the following contributions:

1. We introduce LIDSEARCH (**LI**nked **D**ata and **S**ervice Search)([Mouhoub et al., 2014], [Mouhoub et al., 2015]), a SPARQL-driven framework to search for linked data and relevant semantic web services over the LOD that might add new value to the data, all effortlessly from a user perspective and with the same query. LIDSEARCH presents some interesting features and novelties.

- LIDSEARCH works on-the-fly and searches for data and services in the LOD without any pre-processing.
- LIDSEARCH uses only SPARQL for querying data sources and service repositories for a maximum compatibility.
- LIDSEARCH is extensible to any RDF-based service descriptions.
- LIDSEARCH extends search results with semantically similar results using a semantic-similarity approach that relies on the linked data principles.
- We propose a word2vec-based (deep-learning-based) technique to rank relevant services based on their relevance to the data query.
- We sketch an adaptation of a service composition algorithm to automatically create composite web services relevant to data queries when no actual ones are found.

2. We introduce an approach that uses natural language processing techniques towards automatically enriching semantic web service descriptions with precise information about their functionality to facilitate their automatic discovery, composition and integration with other data ([Mouhoub et al., 2017]).

1.3 Manuscript Outline

In chapter 2, we briefly recall some background concepts, definitions and standards related to our work including semantic web, semantic web services and natural language processing. After that, we portray the state of the art in the aforementioned domains, especially the recent works related to our work or the works that have most inspired ours.

In chapter 3, we present the first part of our work related to data and service search called LIDSEARCH. We present its aforementioned features, we describe step-by-step its underlying process and we explain each step by a running example.We dedicate a section to our sketched approach for an automatic service composition which has not yet been fully implemented within LIDSEARCH but deserves some attention. We also present our published experimentation results on OWLS-TC services and discuss its limitations and potential improvements.

Chapter 4 is dedicated to our most recent work that aims to facilitate automatic service discovery by enriching its formal description. We also explain the underlying process step-by-step with a running example.

In the conclusion chapter, Chapter 5, we recall our main contributions and the challenges we. We also discuss some perspective ideas including a potential combination of our both major works.

Chapter 2

Background and State of the Art

2.1 Semantic Web Background

In this section, we recall some important notions and concepts in the semantic web domain that we use or make mention of in this thesis. For a short introduction to the semantic web technologies, the reader can refer to the reference book about semantic web in [Domingue et al., 2011].

2.1.1 Resource Description Framework (RDF)

Resource **D**escription **F**ramework (**RDF**)[1] ([Raimond and Schreiber, 2014]) is a simple data model for describing web resources. It is based on the concept of statements that represent facts about any subject on the web. Fig.2.1 summarizes a hierarchical view of some of the most important concepts in RDF.

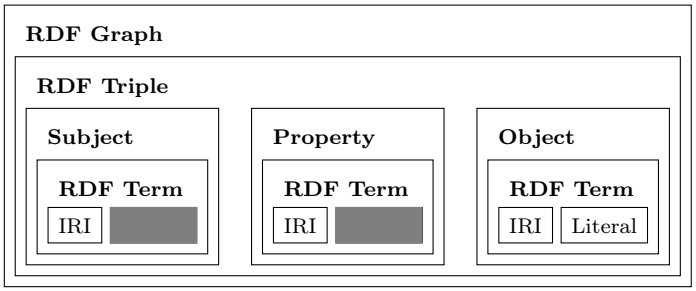


Figure 2.1: Partial hierarchical anatomy of the Resource Description Framework (RDF)

1. **RDF Resources (RDF Terms):** Any subject or any entity is a resource on the web. It can be a web-page, an image, a document, a number, a string, or even a

[1] http://www.w3.org/TR/rdf11-primer/

physical entity like a person, an organization, an event, an object, etc. An *RDF Term* can either be IRIs or literals[2].

(a) **I**nternationalized **R**esource **I**dentifier (**IRI**) is an extension of the **U**niform **R**esource **I**dentifier (**URI**) (for non ASCII characters) to identify a resource. It allows to uniquely identify and locate a resource on the web in order to help access and interact with it. For example, `<http://dbpedia.org/resource/Victor_Hugo>` corresponds to the entity representing Victor Hugo in DBpedia[3].

Namespaces are often used as prefixes to shorten IRIs of RDF resources in RDF files, the last example for instance would look like: `<dbr:Victor_Hugo>` in its short form. We shorten IRIs with prefixes in this thesis most of the time in the examples.

(b) **Literals** are basic values associated with XML Datatypes like strings, numbers, dates, etc and often used to describe textual or numerical attributes of IRI resources like names, titles, phone numbers, birth dates etc. For example, the title of the famous book by `<dbr:Victor_Hugo>` is `"Les misérables"` which is an `xsd:string`.

2. **RDF Triples:** To describe a resource, RDF uses a *statement* called an *RDF statement* to depict a relation called a *property*, or a predicate between this resource called a *subject* and another called an *object*.

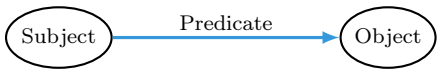


Figure 2.2: Anatomy of an RDF triple

RDF statements take the syntactic form of a *triple* that can be broken down into 3 elements: A subject, a relation and an object, hence the name *RDF triples*. It always has the following structure:

```
<subject> <predicate> <object>
```

For example, the famous book "Les_Misérables" is declared as written by Victor Hugo in DBPedia using the following RDF statement:

```
<dbr:Les_Misérables> <dbo:author> <dbr:Victor_Hugo>
```

[2]There exists a third form for resources in RDF, blank nodes, but we won't address this in this manuscript

[3]http://dbpedia.org/

Note that IRIs can be used as a subject, a predicate or an object in RDF triples whilst literals can only be used as objects as depicted in Fig. 2.1

3. **RDF Graphs** As a matter of fact, an RDF triple can be seen as an "atomic" graph consisting in a directed edge between two nodes linking and describing them as shown in Fig.2.2. Therefore, a set of RDF triples forms up an RDF graph. For example, a data-set about books and their authors is a set of RDF triples that all together form an RDF graph.

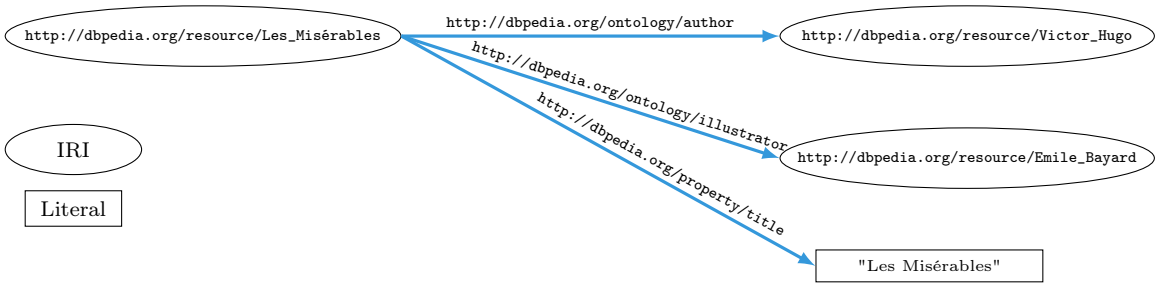


Figure 2.3: Example of an RDF graph

Fig.2.3 shows an example RDF graph consisting in 4 RDF triples that describe some basic information about `<dbr:Les_Misérables>`.

It is possible to group RDF triples in multiple RDF graphs, allowing to create named graphs that group RDF statements about a given subject within an RDF dataset as subsets of the latter. For example a books dataset can be divided into multiple graphs grouping books of the same theme or writer. In this thesis, we don't particularity address multigraphs and consider RDF datasets as a whole and single graph for clarity purposes. Therefore, the term RDF datasets refers to distinct multiple datasets from multiple sources.

2.1.2 RDF Vocabularies

As mentioned earlier, RDF allows to describe resources with RDF statements that depict relations between two resources. As we also mentioned earlier, RDF resources represent abstract or real word entities. In order to define what these entities are and what relations are, the ***RDF Schema***[4] (**RDFS**) provides vocabularies for defining data in RDF. Below, we mention the main elements of the RDFS vocabulary ([Brickley and Guha, 2014])

[4] `http://www.w3.org/TR/rdf-schema/`

1. **Class:** A class is an abstraction of a group of entities sharing the same characteristics. For example, the group of books belongs to the class of Books. In DBpedia, `<dbr:Les_Misérables>` belongs to the class `<dbo:Book>`. Entities belonging to a class are called *instances*. The group of classes is a class itself called `rdfs:Class`

2. **Property:** A property is a relation between two RDF resources, a subject and an object. It can be used to depict an attribute of an entity, like the title of a book, or a relation between two entities like the fact that a book is written by some author.

   Properties themselves are RDF resources. The class that groups properties is called `rdfs:Property`. Therefore, it can have properties of its own relations with other entities. There are two important properties that a `rdfs:Property` should have:

   (a) **Domain**: An `rdfs:Property` defines the relation between a subject and an object. `rdfs:domain` is an `rdfs:Property` that defines the class of subjects to be used with a property in RDF triples. For example, the `rdfs:Property` `dbo:author` defines the author of a work `dbo:Work` in DBpedia. Therefore we state that:

   ```
   dbo:author rdfs:domain dbo:Work
   ```

   (b) **Range**: `rdfs:range` is an `rdfs:Property` that defines the class of objects to be used with a property in RDF triples. For example, the `rdfs:Property` `dbo:author` from earlier considers that an author of a work `dbo:Work` is a `dbo:Person`. Therefore we state that:

   ```
   dbo:author rdfs:range dbo:Person
   ```

   Note that Properties in RDF are binary relations. To describe an n-ary attribute or relation of a resource, multiple RDF statements should be used. For example, to state that the book `<dbr:The_Frozen_Deep>` is written by `<dbr:Wilkie_Collins>` and `<dbr:Charles_Dickens>`, two RDF triples are required :

   ```
   <dbr:The_Frozen_Deep> <dbo:author> <dbr:Wilkie_Collins> .
   <dbr:The_Frozen_Deep> <dbo:author> <dbr:Charles_Dickens> .
   ```

3. **Type:** RDF allows to state that a resource is an instance of a class using the `rdfs:Property` `rdf:type`[5]. It is generally abbreviated in SPARQL with the word "a" For example, the following statement says that the resource `<dbr:Les_Misérables>`

[5] In SPARQL, `rdf:type` can be abbreviated with the article `"a"`.

is an instance of the class `<dbo:Book>`:

```
<dbr:Les_Misérables> <rdf:type> <dbo:Book>
```

4. **Sub-class:** It is common in abstract and real world entities that a class of entities could have one or more sub-classes. To depict this hierarchical relation between two classes in RDF, we use the <rdfs:subClassOf> property. It implies that all instances of a class are also instances of its super-class. For example, the class `<dbo:Book>` is a sub class of `<dbo:WrittenWork>`. We state:

```
<dbo:Book> <rdfs:subClassOf> <dbo:WrittenWork>
```

5. **Sub-property:** Similarly to classes, properties can also have sub-properties of their own. A sub-property `rdfs:subPropertyOf` is a `rdfs:Property` that states that all resources related by a property are also related by its super-property.

2.1.3 SPARQL

SPARQL[6] **P**rotocol **A**nd **R**DF **Q**uery **L**anguage[7] **(SPARQL)** is the defacto W3C standard querying language for retrieving and manipulating RDF data. It has an SQL-like syntax and relies on graph pattern matching with RDF graphs. Fig. 2.4 shows an anatomy of a SPARQL query along with the most important concepts around SPARQL: ([Seaborne and Harris, 2013])

1. **Variables:** A variable in SPARQL is a part of a variable binding pair (variable, RDF term) that aims to assign the value of an RDF term to this variable after the query execution ([Fensel et al., 2011]). Variables are prefixed with a `"?"` attached to their names (Eg.: ?book).

2. **Triple patterns** At the core of SPARQL queries are Graph patterns and at the core of graph patterns is the triple pattern. A triple pattern is a set of three whitespace-separated nodes commonly read *"subject property object"*. A property is either a variable, an `rdfs:property` or an `owl:property` node that links a subject to an object.

[6] `http://www.w3.org/TR/sparql11-query/`

[7] The **S** used to originally stand for **S**imple in the early proposals of SPARQL. However, after the language became more complex, SPARQL was substituted for Simple

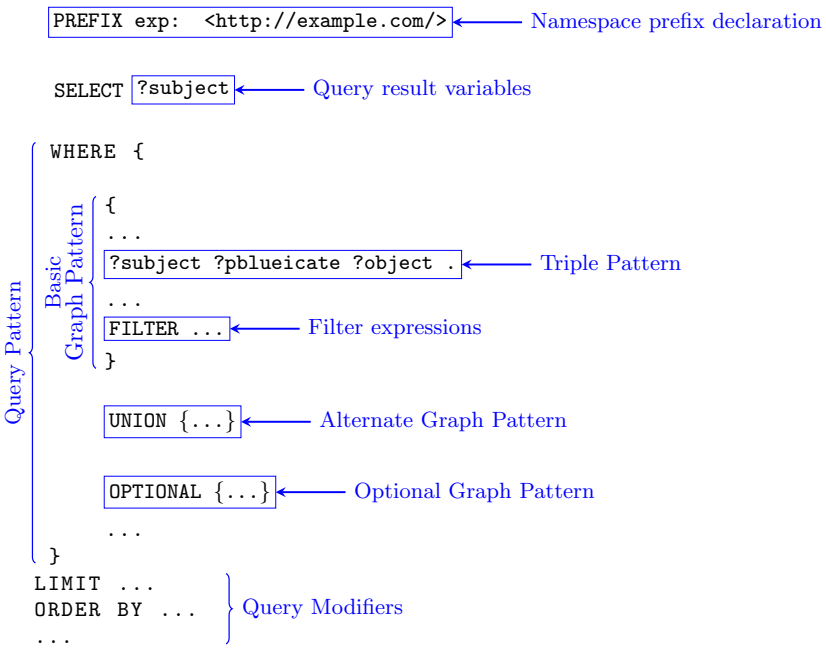


Figure 2.4: Anatomy of a SPARQL query

Similarly to RDF triples, subjects can only be an IRI or a variable whilst objects can be anything. In other words, a triple is similar to RDF triple but with zero or more bidden RDF terms, i.e, variables.

3. **Graph patterns** SPARQL uses graph matching for retrieving data. It defines a set of graph patterns for this purpose. A Graph pattern is a set of triple patterns that can take one of the following forms:

   (a) **Basic Graph Pattern (BGP):** a set of one or more triple patterns. In a SPARQL query, a query solution should match all the triple patterns of a BGP. A BGP can be delimited with brackets (" ").

   (b) **Optional Graph Pattern:** a graph pattern that can be coupled with another graph pattern to extend its solution. A query solution should either match both graph patterns or at least the non-optional graph pattern. In SPARQL, it is denoted with the a `OPTIONAL` keyword.

   (c) **Group Graph Pattern:** a set of graph patterns. A query solution should match all the graph patterns of a group graph pattern.

(d) **Union Graph Pattern:** also called an Alternate Graph Pattern is a set of graph patterns for which a query solution matches either or all of them. In SPARQL, alternate graph patterns are coupled using the `UNION` keyword.

The set of all graph patterns contained within the WHERE clause of a SPARQL query is called a *Query Pattern*.

4. **Query Forms:** There are four query forms in SPARQL:

   (a) **SELECT** Returns the values of all bound variables in the query result clause (not necessarily all the query variables).

   (b) **ASK** Returns a boolean (`TRUE`) if the query pattern has a match in the dataset, or (`FALSE`) otherwise.

   (c) **CONSTRUCT** Returns an RDF graph specified by a template in which result values are substituted for the construct variables.

   (d) **DESCRIBE** It is mainly used to return an RDF graph containing RDF statements in an RDF data-set about a resource.

5. **Filters:** SPARQL Filters, defined by a function (or an expression) in a `FILTER` clause, are used to restrict the query solution by eliminating solutions for which the filter expression returns FALSE (just like an algorithmic `return if`). Possible filters include String equality, regex matching, mathematical comparatives, etc.

6. **Modifiers:** Modifiers are used to modify the arbitrary sequence of the query results. Modifier operators include inter alia: `ORDER BY` which modifies the order of a solution either in the ascending `ASC` or descending `DESC` order of a variable's values. `LIMIT` limits the number of query solutions up to a defined integer.

To summarize, a SPARQL SELECT query consists, as shown in Fig. 2.4, in a set of one or more graph patterns called a query pattern contained within the `WHERE` clause that ought to be matched against a dataset (data graph). It also contains a set of variables contained within its `SELECT` clause that ought to appear in the query results.

Fig 2.1 shows an example of a simple SPARQL query that searches for books written by the author `<dbr:Victor_Hugo>`. The results of this query from DBpedia are shown in Table 2.1.

```
PREFIX dbo:<http://dbpedia.org/ontology/>
PREFIX dbr:<http://dbpedia.org/resource/>

SELECT ?book
WHERE {
    ?book    dbo:author
dbr:Victor_Hugo.
}
LIMIT 5
```

Listing 2.1. Example of a simple SPARQL Select query on DBpedia)

Table 2.1: Listing. 2.1's query results from DBpedia

?Book
http://dbpedia.org/resource/Les_Misérables
http://dbpedia.org/resource/Bug-Jargal
http://dbpedia.org/resource/Toilers_of_the_Sea
http://dbpedia.org/resource/Bug-Jargal
http://dbpedia.org/resource/The_Man_Who_Laughs
http://dbpedia.org/resource/Les_Rayons_et_les_Ombres
http://dbpedia.org/resource/Ninety-Three

2.1.4 Linked Data and LOD

Semantic Web Data or RDF data are not necessarily or fully machine-understandable unless they respect a set of four rules outlined by Tim Berners Lee[8] and called the *Linked Data Principles*:

1. Use URIs (IRI) as names for things,
2. Use HTTP URIs so that people can look up those names,
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL),
4. Include links to other URIs. so that they can discover more things.

These rules guarantee that RDF data can be accessed whenever looked upon, that a machine can access the schema/definition of an RDF resource, understand its nature (`rdfs:Class`), access other resources linked to this resource via some (`rdfs:Property`). It also favors the reuse of existing ontologies, schema and instances to facilitate data integration. Semantic web data that respect these rules are called *Linked Data*.

The adoption of the linked data principles by many organizations, governments, companies, developers etc by publishing their data as linked data using open licenses has allowed to create a large network of linked data called the **L**inked **O**pen **D**ata Cloud (**LOD**) [9]. In Chapter 1, we have portrayed the importance and the evolution of the LOD, please refer to this chapter for some interesting facts about the LOD.

[8] http://www.w3.org/DesignIssues/LinkedData.html
[9] httphttp://lod-cloud.net/

2.2 Semantic Data Querying in the LOD

There have been a lot of efforts carried by the database community and later on by the semantic web community to provide efficient techniques to run SPARQL queries over semantic data sources. Obviously, the amount of work in this area is considerable and it is not our aim to review all of it but it is rather to highlight the most referenced works in the literature in general and in the survey papers in particular. These works can be classified into 2 main categories as depicted bellow in Fig. 2.5 which summarizes and shows the typology of the works we cite here.

This literature review is based on the recent dedicated surveys [Özsu, 2016] and [Rakhmawati et al., 2013] [10].

2.2.1 Centralized (or Data Warehousing) approaches

At the early days of semantic web technologies, these approaches originated from the relational database techniques to manage the storage and the querying of RDF data. They are considered as *"centralized"* because the entire data is maintained in a single RDF store. [Özsu, 2016] has classified them into five categories of approaches:

1. Direct Relational Mappings:

   This approach consists of mapping RDF data directly into a relational database by putting all the triples in a single three columns table (Subject, Property, Object). For querying, SPARQL (1.0) queries are translated into SQL queries as it has been proven possible in [Angles and Gutierrez, 2008]. This approach exploits the mastered relational database management techniques and amongst the works that follow this approach we cite Oracle [Chong et al., 2005] and Sesame SQL92SAIL [Broekstra et al., 2002].

2. Single Table Extensive Indexing:

   In contrast with the first category, this approach consists of performing an extensive indexing while maintaining the single 3 column table. Works that fall into this category are Hexastore [Weiss et al., 2008] and RDF-3X [Neumann and Weikum, 2008]. The latter creates indexes for all possible combinations of the subject, property and object (i.e. spo, ops, sop, osp, pso, pos).

[10] Note that we don't keep track of approaches published after 2015 due to a change of interest but the typology of this state of the art still remains valid for 2017.

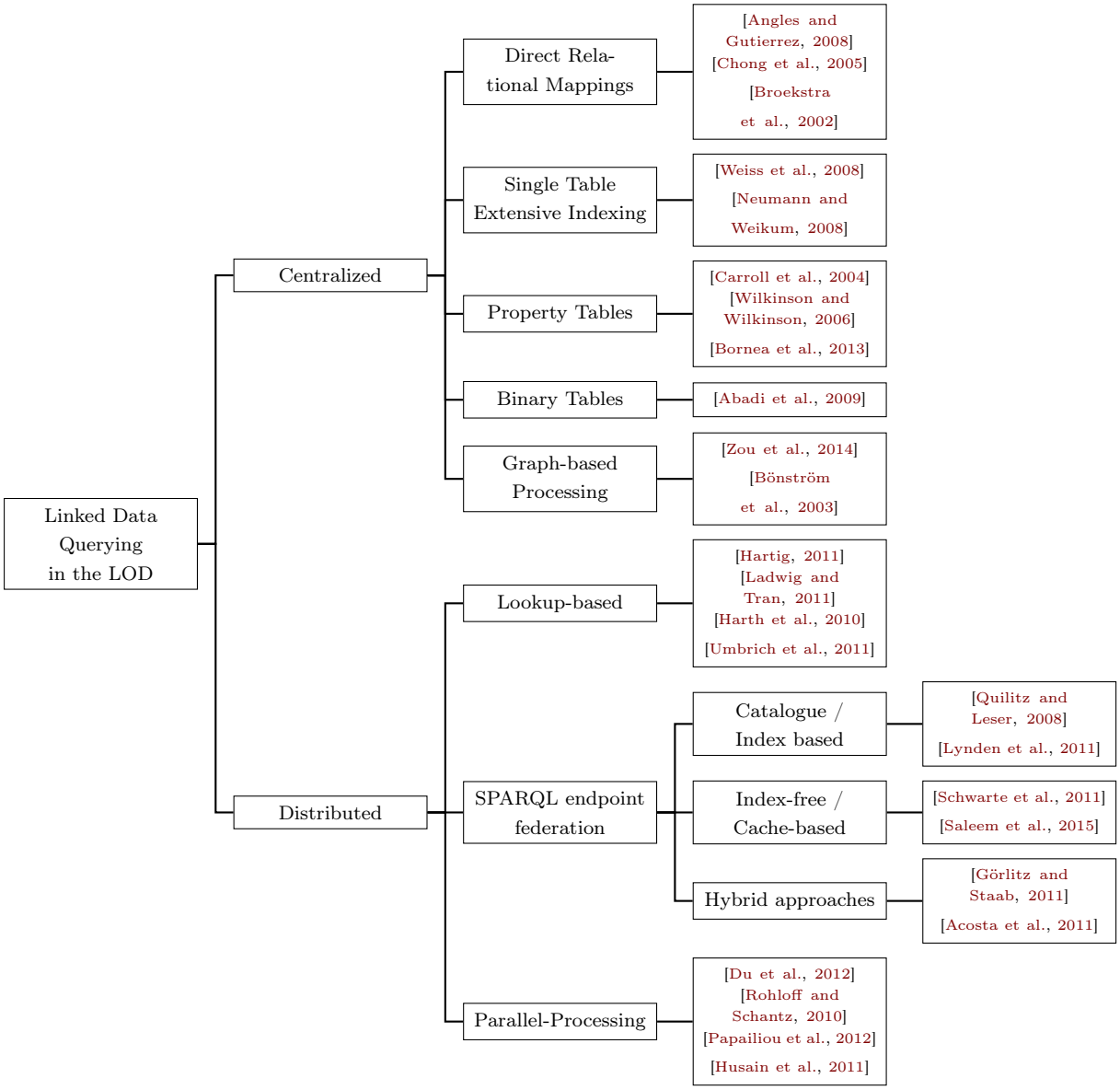


Figure 2.5: Typology of approaches for querying linked data in the LOD with some notable citations

3. Property Tables:

   This approach builds a database by creating property tables grouping the most common properties to a subject. A property table has a single column to store the subject of the statement. The remaining columns are named upon the co-occuring properties for that subject and store the values of the objects stated by the properties in the triples.The most commonly known works that fall into this

category are Apache Jena [Carroll et al., 2004] [Wilkinson and Wilkinson, 2006] and IBM's DB2RDF [Bornea et al., 2013]

4. Binary Tables:

   In contrast with the property tables approach above, this approach (SW-Store [Abadi et al., 2009]) builds a two column table containing Property(Subject,Object) for each property and names it after it. Therefore, there would be as many tables in the database as the number of existing properties in the initial RDF graph.

5. Graph based processing:

   This approach maintains the graph structure of the RDF data by representing it with adjacency lists. It converts the SPARQL query into a query graph, and performs sub-graph matching using homomorphism to evaluate the query against the RDF graph. Several systems follow this approach such as gStore [Zou et al., 2014] and [Bönström et al., 2003].

Advantages and limits

While the centralized approaches are very efficient thanks to the advances in the relational database management, they are not adequate for applications involving very large RDF datasets or involving the Linked Data Cloud sources. Therefore, scalability in terms of size or distribution of the sources is a major issue for these approaches [Özsu, 2016].

In the first approach, the single table quickly becomes very large and queries become quite complicated to process. The second approach of extensive indexing suffers from the overhead of updating the multiple indexes if the data is often updated. The fourth approach of binary tables require a lot of joins to answer queries since there is a table per property and insertions would involve a lot of tables at a time as well. The graph-based approach has many advantages but has a major shortcoming when it comes to query processing (sub-graph matching) because it is based on homomorphism which is an NP-Complete problem [Özsu, 2016].

To conclude, although some of approaches can be adapted for a better scalability, they are not initially designed for the LOD cloud. Therefore, the distributed category of semantic data querying approaches bellow (2.2.2) is more adequate in such a context.

2.2.2 Distributed approaches

These approaches target the emerging LOD sources and address the problem of how to run queries over these multiple independent and remote data sources. Three categories of approaches can be distinguished [Harth et al., 2012]:

2.2.2.1 Lookup-based approaches

This category of approaches relies on the Linked Data Principles which state that interlinked resources should have accessible URIs. Based on the URIs of the resources referenced in the query, or in the intermediate results, a URI lookup determines the data sources to be selected in the query processing at query runtime. Simply, the links in the URIs are followed to fetch new data from the addresses in these URIs and so on. Additional data summaries (catalogs) can be used locally to accelerate the process.

There are two major issues with this approach: the first is that the data should follow the Linked Data principles, and the second is the execution time of the query processing which is very slow compared to the other approaches.

One argument could be raised to highlight the advantage of these approaches: the heterogeneity of the access methods to the data sources in the LOD. In fact, only 68.14% of the data sources (RDF repositories) provide a SPARQL endpoint and 39.66% provide RDF dumps according to LOD cloud statistics[11].

We mention amongst them [Hartig, 2011; Ladwig and Tran, 2011; Harth et al., 2010; Umbrich et al., 2011]

2.2.2.2 SPARQL Endpoint Federation frameworks

Many surveys have studied and compared the existing federated systems, and the reader can refer to [Saleem et al., 2015] and [Rakhmawati et al., 2013] for a deep study and comparison.

The frameworks based on the federation of SPARQL endpoints are very common for processing SPARQL queries in the LOD context. Regardless of the differences between the famous existing frameworks, they all share the same basic pipeline of query processing.

Given a SPARQL query, the framework first performs a triple pattern-wise source selection to identify the set of data sources that contain relevant results against individual triple patterns of the query. The source selection information is then used to decompose

[11] http://sparqles.ai.wu.ac.at/

the query into multiple sub-queries. Each sub-query is optimized to generate an execution plan. The sub-queries are forwarded to the relevant data sources according to the optimization plan. The results of each sub-query execution are finally joined to aggregate the results of the query [Saleem et al., 2015]. The query decomposition into sub-queries is referred to as *"query rewriting"* in the literature.

However, the differences between the existing frameworks are most importantly at the source selection level. Therefore they can be classified into three categories:

1. Catalog/Index-Based approaches: These approaches perform a pre-processing on the data sources to create indexes called "Data-set Summaries". These indexes maintain information about the entities/properties/prefixes (or authorities) contained in each data-set to help make source selection decisions quickly. Therefore, query processing is very efficient but requires the index to be constantly updated to ensure the completeness of the results.

   Most of the Federated systems fall into this category. We cite the most commonly known amongst them bellow :

   (a) DARQ(Distributed ARQ) [Quilitz and Leser, 2008] is an extension of ARQ (Apache Jena's query processing engine). It employs an index that holds the list of predicates (properties) in a given data-set along with the SPARQL endpoint URL and other statistics. They refer to their index as "Service Description", referring to the SPARQL endpoint of a data source as a Service (which is distinct from a conventional web service). The shortcoming of such an index is that it is only capable of answering queries with bound predicates. The query planning algorithm is based on estimated cardinality cost and aims to reduce the cost of the query processing in terms of execution time and bandwidth.

   (b) ADERIS [Lynden et al., 2011]. At a cold start, ADERIS sends a SPARQL query to collect all the properties that appear in RDF triples in datasets. A cache keeps track of the locations of each property amongst the available datasets. A querying time, the user query is rewritten into several sub-queries, each one is routed to the corresponding data-source. One limitation of ADERIS is the lack of a full support of SPARQL.

2. Index-Free/Cache-Based approaches: These approaches do not perform any pre-processing on the datasets nor use any existing data-set summaries. The main idea

is to collect statistics and create data-set summaries on the fly as the queries come. The advantage of such approaches is that they guarantee the completeness of the results. The major issue here is the execution time because all the information about the data sources are collected right before executing the query. To my knowledge there is only one system that falls into this category:

FedX [Schwarte et al., 2011] is based on the Sesame Framework. The source selection is based on SPARQL ASK queries and a cache. The cache stores the ASK results to reduce the processing cost for successive similar queries. As shown in [Saleem et al., 2015], this reduces greatly the execution time of the next queries.

Because FedX is based on RDF4J[12] (formerly known as Sesame), it requires a list of pre-defined data sources at the initialization of the system without any statistical information about them.

At the query rewriting step, FedX clusters related triple patterns that can be answered exclusively by unique SPARQL endpoints and sends them as sub-queries to their corresponding endpoints. This allows the system to delegate the joins to the data sources (SPARQL endpoins), thus reducing the size of the intermediate results and the workload on the host server.

Due to its advantageous on-the-fly feature, we have chosen FedX as the federated query engine in our work as depicted in the next chapter.

3. Hybrid approaches: These approaches use some existing (pre-stored) data summaries such as statistics and collect others on-the-fly using SPARQL ASK queries for example.

   (a) SPLENDID [Görlitz and Staab, 2011] is an extension for Sesame [Broekstra et al., 2002] that employs VoiD[13] (Vocabulary of Interlinked Datasets, an RDF schema to describe linked datasets) Descriptions as a stored catalog along with SPARQL ASK queries to verify the VoiD information at the source selection step. The ASK queries verify weather an element from a triple (Subject, Predicate, Object) can be resolved by a given data source or not.

   (b) ANAPSID [Acosta et al., 2011] is a system that manages its query execution plan with respect to the data availability and the runtime conditions of the

[12] http://rdf4j.org/
[13] http://www.w3.org/TR/void/

SPARQL endpoints. Similar to the above mentioned index-based approaches, ANAPSID employs a index for the list of predicates contained by datasets. It sends ASK queries on-the-fly to verify the availability and updates its catalog accordingly. In addition, it keeps a track of the response-time of the SPARQL endpoints in the catalog. Therefore, it can use some heuristics for a data source selection that considers both functional and QoS properties of the data sources.

2.2.2.3 Parallel Processing

According to [Galárraga et al., 2012] there is an increasingly large number of approaches and systems that use MapReduce techniques to efficiently process RDF data in the cloud([Choi et al., 2009; Husain et al., 2011; Papailiou et al., 2012]).

The common basic idea behind using MapReduce is to split RDF datasets into small chunks and store them in a distributed fashion. At querying time, the graph pattern matching is performed in parallel across several machines in the cloud. The main difference on the other hand is the storage techniques of RDF triples. For example, HadoopRDF [Du et al., 2012] partitions RDF triples based on properties by storing in separate files the triples containing the same properties. On the other hand, SHARD [Rohloff and Schantz, 2010] partitions RDF triples based on the subject and stores every partition as a single line of one big file containing all partitions (triples) ([Özsu, 2016]).

According to [Galárraga et al., 2012], a common disadvantage shared by all of them is the poor response time of Hadoop MapReduce processes. A number of proposals combine distributed storage (Hbase[14] for instance) and MapReduce to improve performances. The recent work in [Naacke et al., 2017] uses Apache Spark[15] which is an alternative to Hadoop but performs much better in many cases. There is a considerable number of papers in this topic. The surveys in [Kaoudi et al., 2013] and [Özsu, 2016] are a good starting point for a literature review of these approaches.

2.3 Web Services Background

In this section we briefly recall the most important notions and concepts about web services that we make use of in the work presented in this thesis and redefine them

[14] http://hbase.apache.org/
[15] http://spark.apache.org

according to the context of our work.

2.3.1 Web Services

Before getting to define a web service, one should first define a service. The two terms are often interchangeable but do not stand for the same thing among all scientific communities.

Service: As defined in [Preist, 2004], a service is a provision of value in some domain. For example, a travel agency books travels for its clients to the destination and in the dates their clients want. In fact they offer a travel booking service. However, this booking service can be provided either by phone, directly at their office or through their internet website. Based on this definition, a service is the value provision no matter the way.

Throughout this thesis, we do not consider this vague definition. We use the term service to refer to web services or to semantic web services.

Web service: As defined by W3C [McCabe et al., 2004], a web service is "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL[16]). Other systems interact with the Web service in a manner prescribed by its description using SOAP[17] messages, typically conveyed using HTTP with an XML[18] serialization in conjunction with other Web-related standards".

In the light of this definition, we can say that a web service is a software that interacts with other software or machines over the internet to provides some services. It interacts with other systems based on a communication protocol defined by its description that facilitates communication and data exchange.

This thesis is motivated by the integration of data from data sources with data provided by services. Therefore, we mainly focus on data-providing services (DPS) that provide data rather than on transactional and process oriented services.

Service description: A service description is a document that describes what a service does, i.e, what service it provides, as well as how to communicate and exchange data with it. Services are mainly described in WSDL, the W3C standard for service descriptions.

[16] http://www.w3.org/TR/wsdl
[17] http://www.w3.org/TR/soap12/
[18] http://www.w3.org/XML/

From a narrow formalism-independent perspective, a service description depicts two categories of properties of a web service ([Klusch, 2008]):

1. **Functional properties** Also called "*service capability*", these properties describe what a service can do or what service it provides. They are generally expressed with four elements:

   (a) **Inputs:** Necessary parameters consumed by a service to provide its "*service*". For example, a car rental booking service requires a car pickup date and a return date.

   (b) **Outputs:** Results provided by a service at the end of its invocation. For example, a list of offers by different car renal companies for the provided dates.

   (c) **Preconditions:** conditions that are supposed to hold before the service is invoked in order to provide its service (outcome) as expected. For example, the car pickup date should be inferior to the car return date.

   (d) **Effects:** conditions that hold after the service provides its outcome. For example, in our previous car-rental booking service, after a client books a rental, the booking is confirmed and his credit card is charged.

2. **Non-functional properties** They include properties such as its name, provider, textual description, financial aspects such as a pricing policy, quality of service (QoS) aspects such as availability, performance, integrity, security, etc.

In the context of this thesis, we make an abstraction of the notion of service descriptions and redefine it from a restricted perspective that takes into consideration the contributions and approaches presented in this thesis. We define a service description as a set of three functional properties: Inputs, Outputs and Textual descriptions. We regard the latter as a functional property based on the assumption that it describes the inputs and outputs and their relationship in a natural language. For example, Fig. 2.6 illustrates our service description for a service that searches for books written by a given author.

We we would like to mention here that we do not take into account non-functional properties, especially quality of service (QoS) aspects which are widely considered in service discovery and composition (The reader may refer to [Kritikos et al., 2013] for a thorough literature review of these aspects).

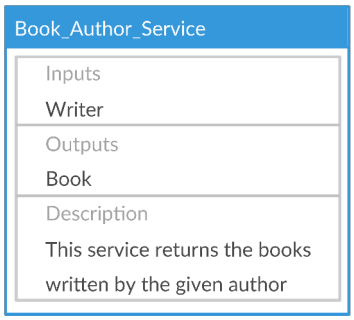


Figure 2.6: Example service description

2.3.2 Semantic Web Services (SWS)

Web services allow machines to interact and exchange data but do not give them the possibility to understand the semantics of their interactions or of the exchanged data.

Semantic **W**eb **S**ervices (often abbreviated with **SWS**) are a convergence of web services with semantic web. As per for the web and semantic web, semantic web services help machines acquire the semantics of their interactions. Such semantic layer on top of web services allows for an automatic discovery, composition, invocation and monitoring of web services. The main characteristic of SWS is the use of semantic-web-based descriptions, often based on a top-level ontology like OWL-S[19] or WSML[20] which are the two main efforts to bring a semantic layer on top of web services.

OWL-S (formerly DAML-S) is an ontology to describe web services using OWL vocabulary. It is based on three description levels: a *Service Profile* that advertises the capacity of a service, a *Service Model* that describes how a service works and how to interact with it and a Service Grounding that describes its communication protocol. OWL-S is a submission member to W3C since 2004.

Other efforts include SAWSDL[21] which is a W3C recommendation that adds semantic annotations to WSDL descriptions but is not a language and not RDF-based. The **M**inimal **S**ervice **M**odel[22] MSM, a lightweight ontology for modeling services, Linked USDL [Pedrinaci et al., 2014], etc.

In this thesis, we are interested in service search and service description enrichment. To match up with our interest in linked data, we only consider semantic web services

[19] http://www.w3.org/Submission/OWL-S/
[20] http://www.w3.org/Submission/WSML/
[21] http://www.w3.org/TR/sawsdl/
[22] http://kmi.github.io/iserve/latest/data-model.html

for their integration and automation virtues and we set our objectives and build our approaches based on that. Therefore, the term web service often refers to semantic web services.

2.4 Serice Discovery

Web service discovery, or service search, is a very important aspect for both service providers and users. A good web service is of no use if it cannot be discovered by potential users. Therefore, service search is as important for web services as web search is for websites. The web service discovery problem has been addressed since the early days of web services and has evolved accordingly. New advances in web service technology and standards, in semantic web as well as in other fields like artificial intelligence and information extraction have allowed for constant innovations and progress in service discovery. In this section, we cast some light on some recent or notable works on the discovery of Semantic Web Services [23].

The service search process can be seen as a two-fold process consisting in service description search and service selection. The service description search aims to find service repositories (or directories) that host service descriptions. This might seem like a trivial task but it has been seriously addressed in some works like in [Louati et al., 2016].

On the other hand, service selection, also called service matching aims to a) match the accessible service descriptions with a user query (also called service request) to measure their relevance for the latter then b) to rank the matched services based on their relevance.

Reviewing the existing service search approaches can be covered from two different aspects: the system architecture and the service selection technique. Fig. 2.7 shows the typology of service search approaches from this perspective with some examples of notable works. For further reading, [Ngan and Kanagasabai, 2013a] and [Klusch et al., 2016] provide very informative surveys of the SWS discovery approaches and give comparative summaries.

2.4.1 Search environment architectures

The architecture of the environment in which the service search is conducted plays an important role in the service search itself. Two major types of approaches can be distin-

[23]We only cite works that have been published before 2015, year in which we were actively working on service discovery. Otherwise it would have been very difficult to keep track of new works

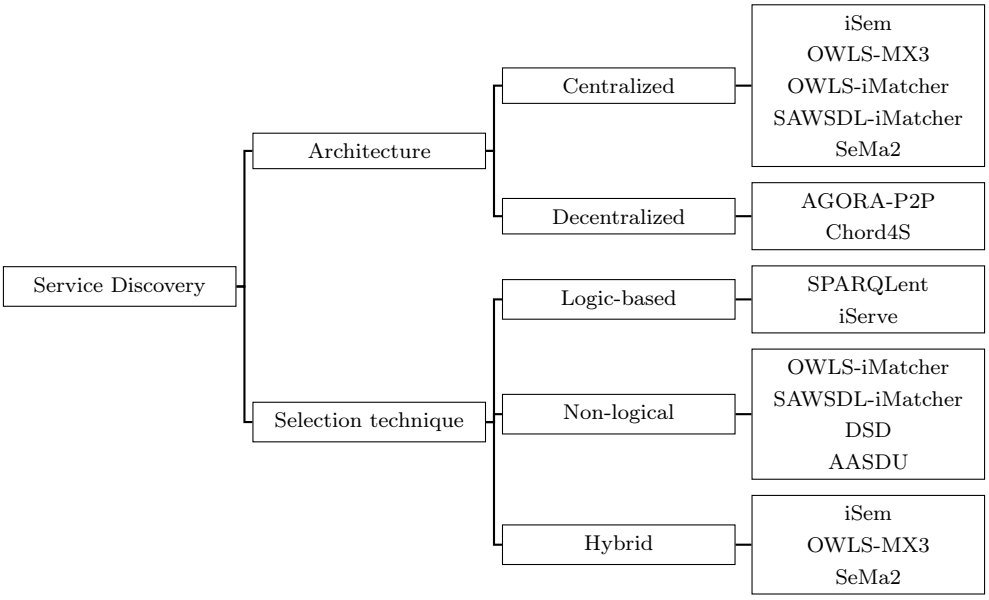


Figure 2.7: Typology of service discovery approaches

guished based on the architecture of the service search environment :

2.4.1.1 Centralized

The centralized approaches use central service repositories that can either be local or distant, single or distributed over a federation of known servers. Therefore, the discoverable web services are known to the discovery systems and the latter apply their matching techniques on the available services to match the user query. The advantage of this category of approaches is the reduced cost of updating the service repository(ies) and the guaranteed recall of service search.

Amongst works that operate in a centralized environment of service repositories we cite: iSem [Klusch and Kapahnke, 2012], OWLS-MX3 [Klusch and Kapahnke, 2009], OWLS-iMatcher [Kiefer and Bernstein, 2008], SAWSDL-iMatcher [Wei et al., 2011] and SeMa2 [Masuch et al., 2012].

2.4.1.2 Decentralized

The decentralized approaches are based on the fact that the knowledge about web services is distributed over a structured or an unstructured P2P network. In structured P2P networks, there is a query routing protocol and the knowledge about web services can be duplicated and spread over multiple peers. In unstructured P2P networks, each peer has

a knowledge of his web services and those known by his peers. To search for unknown web services, random walks should be performed across the network.

As in [Klusch et al., 2016], unstructured P2P networks do not guarantee to find all services within the network and the operation can be onerous whereas the structured P2P networks guarantee a 100% recall. We cite among the works based on P2P networks AGORA-P2P [Küngas and Matskin, 2005], Chord4S [He et al., 2013] and [Di Modica et al., 2011].

2.4.2 Service selection techniques

The service selection technique is the technique used by a service search system to compare and match the user query (commonly called a service request) with a service description to measure its relevance for the query. Depending on the matching technique, service search systems can be classified into three categories:

2.4.2.1 Logic-based approaches

The logic based approaches use semantic web matching techniques to match a service request with service descriptions. They use RDF and OWL entailment rules as a logical inference technique. More precisely, they compare the concepts (ontology classes) used in service descriptions, mainly the concepts of inputs, outputs, preconditions and effects (IOPE) with those of the service request.

There are two types of possible matches: a) an exact match in which the same classes are used in both the service description and the service query and b) an approximate match in which the service IOPE concepts are sub-classes of the query's. The latter is often called a subsume match because some concepts are subsumed by others. Another case of approximate match that is not addressed by all logic-based approaches is when the service description uses equivalent classes (i.e, via `owl:equivalentClassOf`) from the same or from different ontologies.

Among the exclusively logic-based approaches we mention: SPARQLent [Sbodio, 2012] that matches the IOPE of OWL-S services and iServe [Pedrinaci et al., 2010] that supports different description formalisms and languages including OWL-S, WSML, SAWSDL, etc thanks to the use of the MSM vocabulary that regroups many formalisms in one.

2.4.2.2 Non-logic-based approaches

This type of approaches is based on the textual and structural similarity between the service descriptions and the user request. Document-level techniques (such as TF-IDF[24]) and word-level techniques are applied in this approach to calculate textual similarities. Taxonomy and non-logical ontology matching techniques are also applied in this approach. They consist in comparing at a syntax and graph level the ontologies used to describe the service IOPE with those used for the service request.

Amongst the popular works that belong to this category of service matching we mention: OWLS-iMatcher [Kiefer and Bernstein, 2008] and SAWSDL-iMatcher [Wei et al., 2011], DSD [Palathingal and Chandra, 2004], AASDU [Klein and König-Ries, 2004].

2.4.2.3 Hybrid approaches

Hybrid approaches combine both the logic-based and non-logic-based matching techniques. They are the top performers in the benchmarks and test-beds because they take into account multiple criteria for matching. The most popular ones are:

- OWLS-MX3 [Klusch and Kapahnke, 2009] is one of the most notable hybrid service matchmakers. It applies logical and non logical techniques at three levels: a) logical matching: it compares the IO for exact and approximate matching (RDF entailment for sub-classes). b) textual matching: it applies a token-based text similarity matching with the query tokens. c) structural matching: it compares the ontologies of services and queries on a graph-structure lavel based on a syntax matching (similarly to some ontology matching works) but without any reasoning.

- iSem [Klusch and Kapahnke, 2012] is the successor of OWLS-MX3. It applies the same matching techniques (mostly) as its predecessor but takes into consideration the Preconditions and Effects (PE) in addition to (IO).

- SeMa2 [Masuch et al., 2012] is another hybrid service matcher that combines multiple logical, textual, taxonomic and reasoning techniques to compare service descriptions with service requests.

[24]Term Frequency-Inverse Document Frequency

2.5 Automatic composition of web services

Service composition is one of the most addressed topics in the domain of Web Services. Many survey articles have studied the recent and most prominent works in this domain like [Rao and Su, 2004], [Zeshan and Mohamad, 2011] and [Sheng et al., 2014]. The survey in [Lemos et al., 2016] portrays an interesting classification of the existing approaches from different perspectives.

In this section, we mention only a few notable works on automatic composition of web services and SWS that set some baselines upon which we build our automatic SWS algorithm later.

1. [Sycara et al., 2003] This paper is one of the reference papers in SWS research field. It introduces a baseline platform for SWS publishing, discovery and composition. Back in the time SWS used to be described in DAML-S [Ankolekar et al., 2002], the predecessor[25] of OWL-S.It highlights the facility given by semantics to service composition and shows a scenario of how different services can be composed and invoked. However, the paper doesn't describe any technique to perform such composition.

2. [Rao et al., 2006] This paper introduces a semi-automatic approach to compose SWS. The authors argue that SWS are not always fully and precisely described. In fact, automating the SWS compositions on the basis of the hypothesis of fully described services seems to unrealistic. For this purpose, this paper shows that human interaction at some discovery or composition steps would allow for efficient and accurate service compositions.

   The proposed approach uses an adapted version of GraphPlan [Blum and Furst, 1997]. GraphPlan is an an algorithm for automated planning given an initial state of the composition graph, a goal and a set of actions with their preconditions and effects. It takes into account mutually exclusive (mutex) states and actions to omit inconsistent composition plans. The aim of the algorithm is to find a plan that transforms the initial state of the graph into a state matching the composition goal by applying actions at different levels (times).

   The main contribution of this approach is the use of mutex information that indicate composition inconsistencies, conflicts, or the rules that require user intervention. A

[25] http://www.daml.org/services/owl-s/

desired automation level can be defined by users based on such mutex information.

3. [Lécué and Léger, 2006; Lécué et al., 2008] These works introduce a model for service composition called Casual Link Matrix (CLM). A casual link is a relevance link between an input of a service and an output of another represented by (Input, Similarity(Input, Output), Output). The CLM provides a solid background for planning algorithms to find accurate composition plans based on the similarity scores. For instance, a CLM reveals the types of match between two services as I mentioned in section 1.1. This model can be used to make composition plans by helping the algorithms to choose what service can be linked to the ones in the current steps. However, this model has to be calculated for all services before planning so that plans can be made on basis of such model.

   The authors introduced later a QoS-aware CLM+ model [Lécué et al., 2008] that takes into account QoS aspects of services.

4. [Rodriguez-Mier et al., 2012] This paper introduces a service composition algorithm based on A*. First, it generates a service dependency graph for all services in a repository. A service dependency graph (SDG) is a graph connecting services that depend on each other, i.e, some consume the outputs of others. This operation is based on a pre-calculated index of all services in the repository and their I/O. After that, it performs some optimizations to reduce the size of the service dependency graph and improve the execution time of the algorithm. This work has been evaluated using all the WS-Challenge test-sets (2001 to 2008) and shows some good results compared to the winners of these challenges.

   However, this work doesn't take into account the semantic layer on top of SWS In addition, the cost function in A* takes into account only the number of services per layer (step) and doesn't rely on any service-related features.

5. [Yan et al., 2008] This work introduces an approach that represents service dependency graphs as AND/OR graphs and uses an AO* search algorithm[26] to find execution plans through dependency graphs. It shows great performance on large repositories of services and was ranked at the top for three times in the late WS-Challenges (2007,2008,2009).

[26] AO* is an adaptation of A* to AND/OR graphs

Inspired by this work and the good results it shows, we introduce in this thesis an automatic semantic service composition algorithm based on AND/OR graphs that takes advantage of the semantics of SWS.

6. [Lee, 2013] This work introduces another AI-based algorithm for automatic composition of SWS. It is one of the few that distinguish RESTful APIs from SOAP services. At a pre-processing step, it leverages web APIs with semantic annotations using user-provided ontologies. After that, it generates a Directed Similarity Graph (a service dependency graph) that links all services in a repository based on the ontological similarities between their I/O parameters.

   When users submit a service composition request, the algorithm automatically finds (thanks to semantic annotations) composition plans using a BFS (Breadth-first search) algorithm. Unfortunately, the authors do not provide any metric measurements of their experiments that help us compare it to other works.

7. **PNBA***[Rios and Chaimowicz, 2011] This work presents an optimized version of A* adapted for parallel execution on multiple machines. They also present a bidirectional plan search algorithm to find the optimal composition plans faster. Experiments show that the execution time and the number of nodes browsed before finding the optimal plan is reduced to the half for the bidirectional version and by three quarters for the parallel bidirectional version. The authors give a good overview of the main efforts on adapting A* to work in parallel or in a bidirectional fashion.

In this chapter, we gave a brief introduction to the background of our thesis. We also presented the related works in data and service search as well as in service composition.

There are still some works very close in terms of approach or objectives to our work that we portray further in their corresponding chapters.

Chapter 3

Data and Service Search

The goal of this chapter is to present LIDSEARCH, a framework for searching linked data and semantic web services based on SPARQL. This chapter is structured as follows: In the first section, we relate in depth a few related works that we consider particularily similar or complementary to our work. Next, we present the context and the goals of searching both data and services. We also define some important notions and elements in such a context. In section 3.3, we present the search process applied by the framework and give details about each step and the algorithms used in it. After that, we present the implementation details of our framework and all the modules internal and external involved in. Finally, we evaluate the framework from different perspectives: as a whole system and at elementary levels separately. Some of the results presented in this chapter are published in Mouhoub et al. [2014] and [Mouhoub et al., 2015].

3.1 Related works on Data and Service Search

In this section, we talk about some worth-mentioning works that do not fall directly or exclusively into the categories of related works depicted in chapter 2. These works are particularly similar or related to our work in this thesis in many of its aspects.

1. [Palmonari et al., 2011] This work has inspired our thesis at its very beginning in establishing our first steps and objectives. The main objective of this work is to help users (developers) find useful web services for their data queries. It allows them to ask SQL-like queries to search for data in some static data sources, then finds relevant web services for their query. A relevant service to the query could be for example a service that provides some or all of the data requested in the query. During the process, semantic annotations and ontology generations are automatically generated to be used as a basis for data and service matching. The whole process of data and service search and aggregation in this paper is summarized in figure 3.1

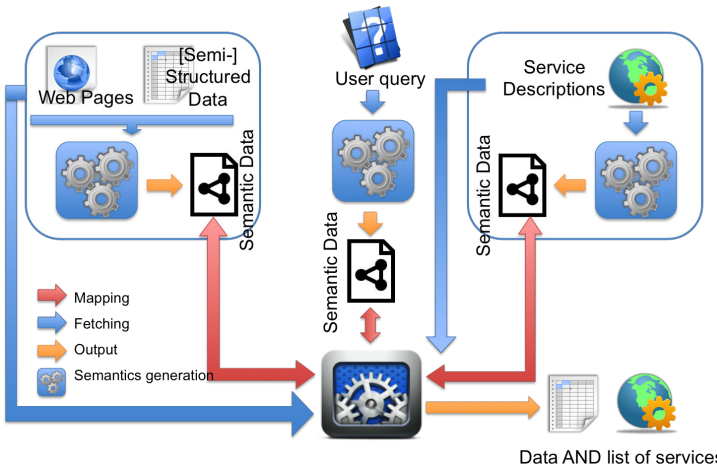


Figure 3.1: [Palmonari et al., 2011] approach overview

First, the system builds global ontologies for data (Data Ontology DO) and for services (Service Ontology SO). For data, it generates a domain ontology from the data schemas of the indexed data sources using WordNet[1]. The DO generation basically annotates the data schemas with WordNet to obtain the semantics. For services, it generates a global service ontology combining the ontologies used in OWL-S descriptions of all services in a repository. The global SO summarizes the functionality (I/O) offered by all web services known to the system.

At runtime, the system generates a Domain Ontology for the user query then maps it with the DO and SO to find relevant data and services. This mapping is based on syntactic and lexical (WordNet) similarity between the nodes of the graphs of ontologies. To find data, the system rewrites the user query into a set of equivalent queries corresponding to the schema of each data source. Relevant services are found by matching the query keywords to the SO graph nodes. Furthermore, the query keywords list is extended with similar keywords from the DO (WordNet similarity) to extend the service search space.

2. [Speiser and Harth, 2011a]

   This work aims to integrate web APIs as a part of the LOD cloud by adhering them to the Linked Data principles [Bizer et al., 2009]. The resulting APIs are called Linked Services (LIDS). LIDS have the following requirements and features:

[1] http://wordnet.princeton.edu/

(a) The invocation URL of a linked service might contain RDF resource IRIs as bindings for the input parameters (i.e. API calls take literals for input parameters as well as IRIs).

(b) The LIDS description (in RDF) describes the relations between the inputs and outputs and therefore discloses a part of the underlying schema of web APIs.

(c) The output format of linked services is RDF

(d) The output of an API call is considered as linked data because it can be accessed with its corresponding API call URI.

However, creating LIDS require manual efforts from API developers. First, developers should manually annotate web APIs with semantics and create a LIDS description using a provided template. In this template, I/O are described as a basic graph pattern (BGP). After that, they should implement wrappers in JAVA on top of APIs to create semantic versions of APIs called LIDS.

The authors also propose an algorithm to interlink the wrapped APIs (LIDS) with selected linked data in order to enrich the LOD.

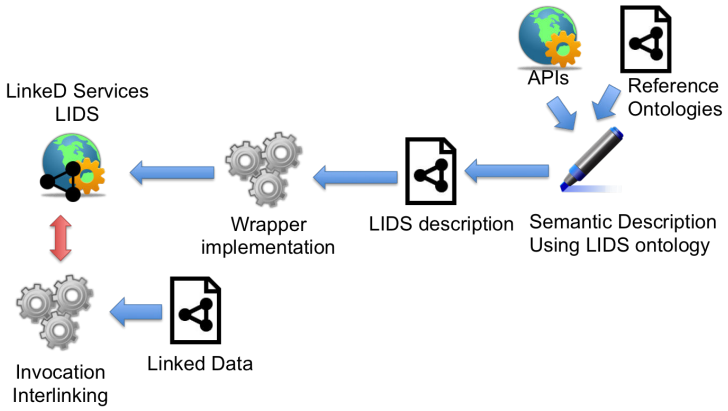


Figure 3.2: LIDS [Speiser and Harth, 2011a] approach overview and process summary

Although this work doesn't address the search problem, it defines a baseline for integrating Data and Services that helps not only search for services and data but also helps retrieve and integrate linked data with APIs data. Another advantage of this approach is the explicit definition of the relation between the inputs and outputs of a service within its description and results. This feature increases the precision of service discovery because it takes into account the relevance of the

outputs regarding the inputs. However, it requires a lot of intervention from the user for the semantic annotation. Hopefully, there are other works that tackle the automatic wrapping of semantic web APIs such as Karma [Taheriyan et al., 2012a] (see bellow).

3. Another similar work in [Preda et al., 2009] called ANGIE consists in enriching the LOD from with data 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.

4. Some recent works could complement our work such as [Harth et al., 2013] which proposes an approach that uses Karma [Taheriyan et al., 2012a] to integrate linked data on-the-fly from static and dynamic sources and to manage the data updates.

3.2 Data and Service querying

In the light of the motivations expressed in the introduction chapter, we propose a Framework that makes it possible to search for both data and relevant services in the LOD.

Let's consider the following example scenario illustrated in figure 3.3: a user wants to know all writers born in Paris as well as the list of all their books. This query is written in SPARQL in listing 3.1 Answers for this query in the LOD might supposedly contain all such 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 their prices, friends who read them, etc. In addition, there are some other relevant services that allow the user to buy a given book online. However, if the user wants to buy a given book from a local store and there is a service that takes only an ISBN number as input to return the available local stores that sell this book, then, in that case, a service composition can be made to return such information.

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. Such a search often requires distinct queries: a) data queries to lookup in the LOD to find data

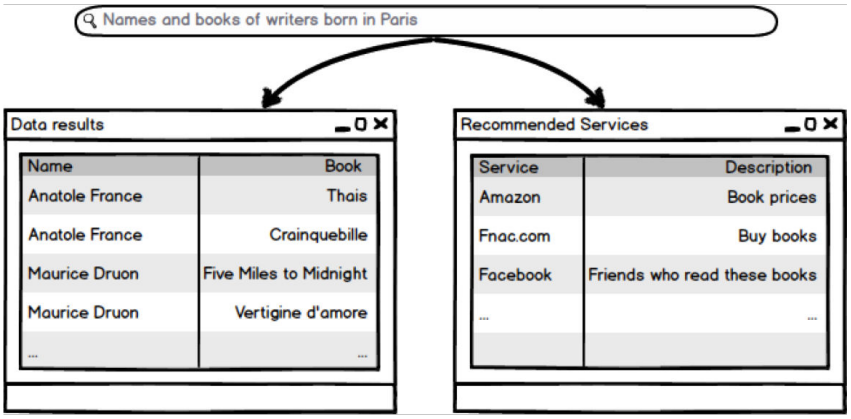


Figure 3.3: Process of discovering services with a data query

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.

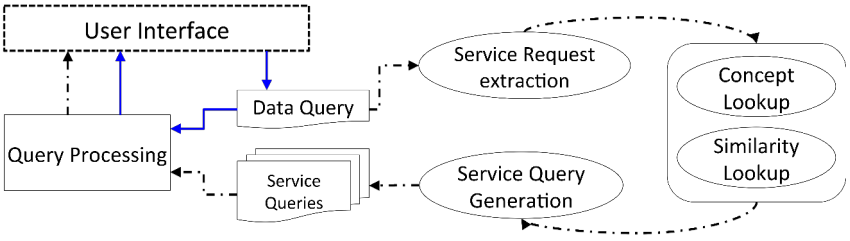


Figure 3.4: Process of discovering services with a data query

Figure 3.4 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 end-points. Thus, a SPARQL-federation approach along with the appropriate optimization and query rewriting techniques is used for this purpose.

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:birthPlace  dbpedia:Paris .
?book   dbpedia-owl:author      ?person ;
        dbpedia-owl:isbn        ?isbn .}
```

Listing 3.1: Example Data Query Q_D

Definitions

To better explain the details of our search framework, we first define some important concepts that we address and use in the context of our framework.

1. **Nodes:** In the context of SPARQL, we denote an RDF term or a variable as a node n. Nodes can either be literals like `"Les misérables"`, URI[2] references like `"http://dbpedia.org/page/Les_Miserables"`[3] or query variables that are hard to miss with their `"?"` prefix like `"?book"`.

2. **Triple patterns** A triple pattern is a set of three whitespace-separated nodes commonly read *"subject predicate object"* or *"subject property object"*. A property or a predicate is a variable, an `rdfs:property` or an `owl:property` node that links a subject to an object. Triple patterns are the elementary unit of

3. **Graph patterns**

   A Graph pattern is a set of triple patterns that can take one of the following forms:

   (a) Basic Graph Pattern (BGP): a set of one or more triple patterns. In a SPARQL query, a query solution should match all the triple patterns of a BGP.

   (b) Optional Graph Pattern: a graph pattern that can be coupled with another graph pattern to extend its solution. A query solution should either match both graph patterns or at least the non-optional graph pattern. In SPARQL, it is denoted with the `OPTIONAL` keyword.

[2]Since RDF 1.1, the standard refers to IRIs instead of URIs which are a subset of URIs that omit spaces

[3]There is a fourth type which is blank nodes that we won't address here

(c) Group Graph Pattern: a set of graph patterns. A query solution should match all the graph patterns of a group graph pattern.

(d) Union Graph Pattern: also called an Alternate Graph Pattern is a set of graph patterns for which a query solution matches either or all of them. In SPARQL, alternate graph patterns are coupled using the `UNION` keyword.

4. **SPARQL query**: A SELECT SPARQL query consists in a set of one or more graph patterns called a query pattern contained within the `WHERE` clause that ought to be matched against a dataset (data graph). It also contains a set of variables contained within its `SELECT` clause that ought to appear in the query results. It consists at its finest level of granularity in nodes as we defined them above.

5. **Nodes and Concepts** (n, c_n): We define a node $n \in N$ in the context of a SPARQL 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 Concept is the reference `rdfs:class` or `owl:class` used to describe the `rdf:type` of a node in its reference ontology Θ. It is obtained with the function $Concept(n)$.

6. **Data Query** (Q_D): A data query Q_D is a SPARQL query that was basically written to search for data in the LOD that match its graph pattern. Listing 3.1 shows an example of a data query for the provided example above.

7. **Service Request** (R_s): Given a data 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 3.3.1

8. **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.

9. **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.

10. **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.

3.3 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 [Bizer et al., 2009] b) SWS are described by RDF based languages such as OWL-S or MSM[Kopecky et al., 2008], 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 inputs of the request a service consumes, the more relevant it is. For example, taking into

account the location as an input might help the service 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 a full matching between all the desired outputs in the service request and the provided outputs in the service description. We refer to this matching strategy as $All_{(Out,Out)}$ and we define it as:

$All_{(Out,Out)}\#1(R_s, D_s) : \{\forall o \in Out_D : o \in Out_S\}[\implies (Out_D \subseteq Out_S)]$

In some cases, if the user wants to find a service that not only provides *all* the desired outputs but also consumes *all and only all* the given inputs in the user query, then we apply the special strategy $All_{(In,In)\wedge(Out,Out)}$:

$All_{(In,In)\wedge(Out,Out)}(R_s, D_s) : \{\forall i_d \in In_D, \forall i_s \in In_S, \forall o_d \in Out_D : i_d \in In_S \wedge i_s \in In_D \wedge o_d \in Out_S\}[\implies (In_D = In_S) \wedge (Out_D \subseteq Out_S)]$

2. Services that consume some of the inputs or the outputs of the request, or that provide 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 applying one of the strategies bellow:

   (a) Services that consume the some of the inputs of the service request:
   $Some_{(In,In)}(R_s, D_s) : (In_D \cap In_S \neq \phi)$

   (b) Services that consume some of the desired outputs of the service request:
   $Some_{(Out,In)}(R_s, D_s) : (Out_D \cap In_S \neq \phi)$

   (c) Services that provide some the outputs of the service request:
   $Some_{(Out,Out)}(R_s, D_s) : (Out_D \cap Out_S \neq \phi) \wedge (Out_D \nsubseteq Out_S)$

   (d) Services that provide some of the inputs of the service request:
   $Some_{(In,Out)}(R_s, D_s) : (In_D \cap Out_S \neq \phi)$

3.3.1 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 3.3.

Algorithm 1 Service Request Extraction

Input: Q_D
Output: In_D, Out_D

$Out_D.nodes \leftarrow$ GetSelectVariables(Q_D) ▷ Get the output variables
$triples \leftarrow$ GetAllQueryTriples(Q_D) ▷ Get all the query triples
for each t **in** $triples$ **do**
    if isConcrete(subject(t)) **then** ▷ check if URI or literal
        $In_D \leftarrow In_D \cup \{(\text{subject}(t), null)\}$
    else if isConcrete(object(t)) **then**
        if predicate(t) $\neq$ "$rdf : type$" **then**
            $In_D \leftarrow In_D \cup \{(\text{object}(t), null)\}$
        end if
    end if
end for

Table 3.1 shows an example a service request elements extracted from Q_D in listing 3.1.

Table 3.1: Example results of a service query extraction from a data query

In_D	Out_D
(dbpedia:Paris, null)	(?person, null) (?book, null)

3.3.2 Semantics Lookup

Once the service request elements are extracted from the query, we try to find the semantics of the previously extracted nodes with no concept: $(n, null)$. This is organized in two successive steps : a) concept lookup and b) similarity lookup which are detailed bellow :

Algorithm 2 Concept Lookup in Query

Input: $n \in (In_D \cup Out_D), Q_D$

Output: c_n ▷ Output a concept for a given node

1: $c_n \leftarrow$ **Null**
2: T $\leftarrow$ GetAllQueryTriples(Q_D) ▷ Get all the triples of Q_D
3: **for each** t **in** T **do**
4: **if** ($n =$subject(t)) $\wedge$ (predicate(t) = "rdf:type") **then**
5: $c_n \leftarrow$ object(t)
6: **end if**
7: **end for**
8: **return** c_n

3.3.2.1 Concept Lookup

In general, concepts can either be declared by the user in the data query (the user likely specifies what he is looking for) or in a graph (set of triples) in an rdf store in the LOD. The later can be the ontology triples (schema) or the instances (data). Concept lookup is applied on the three above sequentially and stops when the concept is found at any.

We do not distinguish between the input nodes in In_D and the output nodes in Out_D during the process of semantics lookup. Even tough the inputs are particularly concrete

values and not variables which seems to give more information on them, our semantics lookup process works for both.

1. Concept Lookup in Query:

   The concept 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. "$n \;\; rdf : type \;\; conceptURI$". Algorithm 2 summarizes the lookup in Q_D.

   In the example query in listing 3.1, the concept of $?person$ is given in Q_D as $dbpedia - owl : Writer$, but the concept of *book* is not given in Q_D.

2. Concept Lookup in Ontology:

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

   Basically, n appears as a subject linked to an object by a predicate p (or vice versa). The later is a property that has a definition in an ontology. Therefore, in the definition of the later, the concept of n, or a super/sub class of it, is used to describe the $rdfs : domain$ (resp. $rdfs : range$) of p. The concept c_n is obtained from the ontology graph of p with a SPARQL query q_c.

   As a matter of fact, if n appears in multiple triples in Q_D and is used with different predicates then the looking-up in ontologies might probably return a different concept for each predicate (eg. in table 3.2). These different concepts are likely a different abstraction level for the same concept; Some are sub-concepts of others. However, in some cases, very generic concepts such as $owl : thing$ or very specific ones can condemn service search either by returning non relevant services or by eliminating the chances of finding one. In this case, the choice of the proper concept depends on the description of the properties used in the data query. This is a design question that intervenes during the design of ontologies.

   Choosing the right concept will probably affect the service discovery process. However, if there are multiple predicates related to n, multiple queries can be issued, and the different results can be considered as similar concepts. Therefore, this will turn the concept lookup process into an All In One process that finds both concepts and similar concepts with the same queries.

In the example Q_D in listing 3.1, the concept of ?*book* is potentially declared in DBpedia ontology as the $rdfs : domain$ of the properties $dbpedia - owl : isbn$ (a) and $dbpedia - owl : author$ (b). Algorithm 3 constructs the SPARQL queries in listings 3.2 and 3.3 (resp.) to find the concept of *book*. However, each query returns a different concept as shown in table 3.2.

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

Listing 3.2: Concept Lookup in Ontology example (a)

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

Listing 3.3: Concept Lookup in Ontology example (b)

Table 3.2: Example results of a service query extraction from the data query in Listing 3.1

?bookConcept (a)	?bookConcept (b)
dbpedia:Work	dbpedia:Book

3. Concept Lookup in Data instances:

   If for any reason, the ontology schema are not accessible or don't provide an answer, then the concept of n might also be found in within the data instances. This depends on the content of the data sources, eg. DBpedia provides concepts within the instances. The concept is obtained by searching in the data source to which Q_D is destined using a SPARQL query.

   To generate this concept lookup query q_c, we take all the triples from Q_D in which n appears 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.

   In the instances data, for a given triple entity, multiple concepts can be attributed to the same entity using multiple $rdf : type$ triples. To increase the accuracy of this particular Concept Lookup, the most popular concept is selected among the rest. To get the most used concept, we count the number of instances that use each concept and select the most used one. This can be done by adding a

Algorithm 3 ConceptLookupInOntology(n)

Input: $n \in (In_D \cup Out_D), Q_D$ ▷ A node without a concept
Output: c_n ▷ Outputs a concept for a given variable
1: T ← GetQueryTriples(Q_D, n) ▷ Get all triples of Q_D where n appears
2: addTriples(q_c, T) ▷ Create q_c query and add the T triples in it
3: **for each** t **in** T **do**
4: **if** isConcrete(predicate(t)) **then**
5: $ns \leftarrow$ GetNameSpace(predicate(t))
6: **else** continue ▷ skip the current iteration if predicate p is a variable
7: **end if**
8: AddFromGraph(q_c, ns) ▷ Set FROM $< ns >$ in q_c
9: **if** $n =$ subject(t) **then**
10: $l \leftarrow$ "predicate(t) $rdfs : domain$ *?concept*"
11: **else**
12: $l \leftarrow$ "predicate(t) $rdfs : range$ *?concept*"
13: **end if**
14: AddTriple(q_c, l) ▷ add the triple l to q_c
15: $c_n \leftarrow$ GetQueryResults(q_c)
16: **if** $c_n \neq null$ **then**
17: **return** c_n ▷ return c_n once found using the current predicate
18: **end if**
19: **end for**
20: **return** null ▷ return null if no concept is found

$COUNT$(*?type as ?counter*) and sort the results in a descending order ($ORDERBY$ $DESC$ *?counter*)

Listing 3.4 shows an example concept lookup query to find the concept of *?book* within the instances of DBpedia. This query is generated using Algorithm 4. Table 3.3 shows the returned results of the later query. The returned concept is therefore the most used one with books that are written by people and have an isbn number.

```
SELECT  ?bookConcept (COUNT(?bookConcept) as ?cCount)
WHERE {
        ?book   dbpedia-owl:author      ?person .
        ?book   dbpedia-owl:isbn        ?isbn .
        ?book rdf:type ?bookConcept .
}
```

```
ORDER BY DESC (?cCount)
LIMIT 1
```

Listing 3.4: An example query of Concept Lookup in Data instances

Table 3.3: Example results of a concept lookup query in data instances

bookConcept	cCount
dbpedia-owl:Work	23851

Algorithm 4 ConceptLookupInData

Input: $n \in (In_D \cup Out_D), Q_D$

Output: c_n ▷ Outputs a concept for a given variable

1: $T \leftarrow$ GetQueryTriples(Q_D, n) ▷ Get all triples of Q_D where n appears
2: addTriples(q_c, T)
3: addSelectVar(q_c, ?*type*) ▷ Set ?type as a result variable
4: $t_1 \leftarrow$ "n rdf:type ?type"
5: addTriple(q_c, t_1)
6: setCountAs(q_c, ?*type*, ?*counter*)
7: setOrderBy(q_c, ?*counter*, $DESC$)
8: setLimit(q_c, 1)
9: $c_n \leftarrow$GetQueryResults(q_c)

3.3.2.2 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 3.2. We distinguish two types of similarities that rely strongly on the Linked Data principles : a) Equivalence similarity defined by $owl : sameAs$ and $owl : equivalentClass$ relations and b) Hierarchical similarity defined by rdfs:subClassOf relations.

1. Equivalence similarity:

   The similar concepts of c_n are defined in ontologies as objects or subjects for the predicates $owl : sameAs$ and $owl : equivalentClass$. Eg: "?$c_n owl : sameAs ?e$" or "?$e owl : sameAs ?c_n$". To find them, we run a SPARQL query on the ontologies

(schema). However, due to the fact that Linked Data is based on referencing other sources, the $owl : sameAs$ mentions can be anywhere in the LOD and not just in the main ontology. This is observed on DBpedia which declares sameAs links to a few sources while DBpedia entities are declared as sameAs in many other sources.

Listing 3.5 shows an example similarity lookup query that searches for owl:sameAs concepts in DBpedia ontology.

For the mentioned above reasons, we need to run a SPARQL query on all known schemas to find equivalent concepts. An alternative solution is to use the sameAs.org[4] API which provides sameAs URIs from the LOD for any given one.

```
SELECT  ?similarConcept
FROM <http://dbpedia.org/ontology/>
WHERE { { dbpedia-owl:Book owl:sameAs   ?similarConcept .}
        UNION
        { dbpedia-owl:Book owl:equivalentClass  ?similarConcept .}
        UNION
        { ?similarConcept rdfs:subClassOf       dbpedia-owl:Book .}
      }
```

Listing 3.5: An example query of Similarity Lookup in Ontology

2. Hierarchical similarity:

   The hierarchical similarity given by the $rdfs : subClassOf$ can be obtained from ontologies in the same way as the Equivalence similarity. In the example in listing3.5, we can add the following triple in a union block to fetch the subClasses of c_n : "$?similarConceptrdfs : subClassOfdbpedia - owl : Book$.".

   However, in some cases, the $rdf : type$ declarations of data instances are enriched with an entailment of all concepts and super-concepts. This is observed in DBpedia, which even provides a seperate rdf store for $rdf : type$ declarations. In such a case, the accuracy of the similarity lookup can be increased by getting the most popular sub-concepts for a given concept. However, sub-classes e_n can are more specific than c_n which makes some of them irrelevant for the context of the query. For example, the concept $dbpedia - owl : Work$ has many sub-concepts which are not relevant to the context of querying for books and authors such as Firm, Website, Musical, etc. In order to get only the relevant sub-concepts, i.e. WrittenWork and

[4]http://sameas.org/

Book, we use the query triples in which n is involved and look for the most popular sub-concepts among those used in the matching instances.

In order to get these sub-classes in data instances, we use algorithm 5 (similar to algorithm 4 but slightly different) in which we declare another variable as an $rdf : type$ of n and tell that "$?similarTyperdfs : subClassOf+?type$". We count the number of instances that use $?similarType$ among those matching the query triples and choose a limited number of the most popular ones. Listing 3.6 shows an example similarity lookup query that returns the top 10 most popular subclass of $bookConcept$.

```
SELECT  ?similarConcept (COUNT(?similarConcept) as ?cCount)
WHERE {
        ?book   dbpedia-owl:author      ?person .
        ?book rdf:type ?bookConcept .
        ?book rdf:type ?similarConcept .
        ?similarConcept rdfs:subClassOf+ ?bookConcept .
}
ORDER BY DESC (?cCount)
LIMIT 10
```

Listing 3.6: An example query of Hierarchical Similarity Lookup

Algorithm 5 SimilarityLookupInData

Input: $n \in (In_D \cup Out_D), Q_D$

Output: e_n ▷ returns a list of similar concept for a given variable

1: $T \leftarrow$ GetQueryTriples(Q_D, n) ▷ Get all triples of Q_D where n appears
2: addTriples(q_c, T)
3: addSelectVar($q_c, ?similarType$) ▷ Set ?type as a result variable
4: $t_1 \leftarrow$ "n rdf:type ?type ."
5: $t_2 \leftarrow$ "n rdf:type ?similarType ."
6: $t_3 \leftarrow$ "?similarType rdfs:subClassOf+ ?type ."
7: addTriple(q_c, t_1) ; addTriple(q_c, t_2) ; addTriple(q_c, t_3)
8: setCountAs($q_c, ?similarType, ?counter$)
9: setOrderBy($q_c, ?counter, DESC$)
10: setLimit($q_c, 1$)
11: $c_n \leftarrow$GetQueryResults(q_c)

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 3.6.2.

3.3.3 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 3.7 shows an example of a service query for the example scenario in section 3.2. 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 3.3) in the way we define them, describe the how tight($All_{(...)}$) or loose ($Some_{(...)}$) 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 .}
}
```

Listing 3.7: Example Service Query Q_S by applying the strategy $All_{(Out,Out)}$

3.4 Service Ranking

Once the service queries are generated and dispatched to service repositories, the later return back a list of web services without any further information regarding the their relevance to the user query. An additional step is required to determine the relevance of each service returned by each repository and then rank all the services based on this measurement. The relevance is measured in two ways which can be combined for a better accuracy :

3.4.1 Functional based ranking

This method of measuring the relevance is based on the number of matching I/O elements of the user query amongst the I/O of the service description. For each returned service, this relevance value for ranking is calculated and the list of all results is sorted in a descendant order of this value. This measure depends on the service discovery strategy and each strategy has its own formula for calculating it :

- $All_{(Out,Out)}$: The relevance measure in this strategy takes into account only the number of matched inputs because all the outputs should match in this strategy :

  $rank(S) = \frac{Card(In_D \cap In_S)}{Card(In_D)}$

- $All_{(In,In)\wedge(Out,Out)}$: in this strategy, all the Inputs and Outputs should match which means that the relevance is equal to 100% all the time of all the returned services. Therefore, the second ranking method should be applied here (see next subsection)

- $Some_{(In,In)}$: this strategy matches Inputs with Inputs only :

  $rank(S) = \frac{Card(In_D \cap In_S)}{Card(In_D)}$

- $Some_{(Out,In)}$:

  $rank(S) = \frac{Card(Out_D \cap In_S)}{Card(Out_D \cup In_S)}$

- $Some_{(Out,Out)}$: This strategy matches Outputs with Outputs only.

  $rank(S) = \frac{Card(Out_D \cap Out_S)}{Card(Out_D)}$

- $Some_{(In,Out)}$: $rank(S) = \frac{Card(In_D \cap Out_S)}{Card(In_D \cup Out_S)}$

3.4.2 Word2Vec based ranking

This ranking method relies on Natural Language Processing techniques to perform a query-service matching based on the service descriptions in natural language. This technique can be used alone for service discovery, but we cope it with out semantic web approach in the ranking phase for a better accuracy.

This accuracy is beyond the I/O matching levels because some identical web services in terms of I/O can have different goals and behaviors. Therefore, the description of a service in natural language can serve as basis for an implicit level of matching undeclared functional features. (This will be revisited in chapter (4 to enrich formal service descriptions using this informal description). To calculate this similarity measure between the user query and the service description in Natural Language, we use word2vec (in next chapter) similarity and apply it on a sentence-to-sentence level rather than on a word-to-word level. The sentence description matching is calculated on the basis of a Sentence-similarity calculation method. We propose three different methods for sentence-similarity calculation :

1. **Average of Maximums method:** First an individual word-word similarity calculation matrix $n * m$ is established for all the possible pairs of words from the two sentences s_1 and s_2. (user query and service description in NL) Word2vec cosine similarity is used here and is defined by the function $sim(w_i, w_j)$

   After that, the maximums for all the rows are calculated then the average of maximums is delivered as the final value for sentence matching.

$$AvgMaxSim(s_1, s_2) = \frac{\sum_{i=1}^{n} Max_{j=1}^{m}(sim(w_i, w_j))}{n} \tag{3.1}$$

2. **Vector sum method:**

   The sum of the word vectors for each sentence s is calculated first

$$sentVec(s) = \sum_{i=0}^{n} wdVec_{wi} \tag{3.2}$$

   then the sentence similarity is given by the cosine similarity between the two aggregated vectors.

   Obviously the stop words are not included in the vectors sum and product (below). Other approaches like (paper reviewed by Daniela) propose *idf* weights for words to

assign low scores for stop words and non relevant words and high scores for unique words.

3. **Vector product method:**

   The product of the word vectors for each sentence s is calculated first

$$sentVec(s) = \prod_{i=0}^{n} wdVec_{wi} \tag{3.3}$$

   then the sentence similarity is given by the cosine similarity between the two aggregated vectors

3.5 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 [Bülthoff and Maleshkova, 2014], 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 3.6.2 to speed-up this online process for the queries that have 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.

3.5.1 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 [Yan et al., 2008] 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 the output 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 [Yan et al., 2008]. 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, therefore, boosting the SDG construction.

The algorithm stops after exploring all the possible dependencies between services or a limited number of them. At this point, if the user inputs In_D are met then the algorithm succeeds and the graph contains at least one possible solution. Otherwise the algorithm fails.

For example, to find services that provide O_1 and/or O_2, a service request $R_s(null, \{O_1, O_2\})$

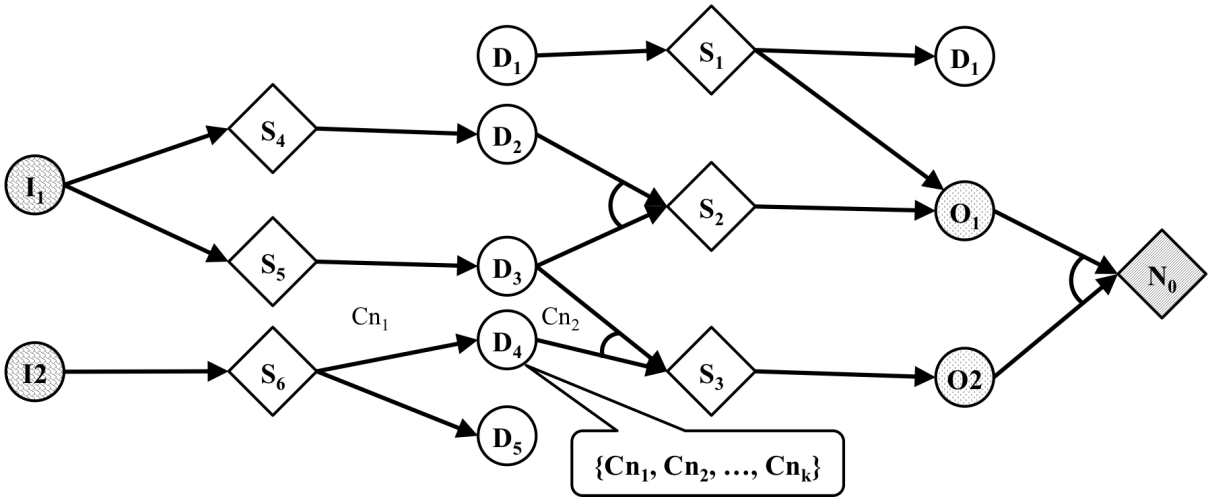


Figure 3.5: An example Service Dependency Graph

is used by applying $Some_{(Out,In)}$.

3.5.2 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 have 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.

Algorithm 6 Service Dependency Graph Construction(n)

Input: $R_s = \{In_D, Out_D\}, Q_D, t$

Output: SDG

1: $L \leftarrow \phi$ ▷ List of OR nodes that haven't been linked yet to a providing AND node.
2: $P \leftarrow \phi$ ▷ List of OR nodes that are equal or similar to data in In_D
3: $t \leftarrow t$ ▷ Maximum searching attempt threshold to limit the number of iterations of the algorithm.
4: $L \leftarrow$ Create OR nodes for all Out_D
5: $N_0 \leftarrow$ Create fictitious AND node N_0
6: LinkIncoming(N_0, L) ▷ Link N_0 to L
7: Create OR nodes for all In_D
8: $i \leftarrow 0$
9: **while** $(L \neq \phi) \wedge (t > i)$ **do**
10: $S \leftarrow Some_{(Out,In)}(R_s(null, L))$ ▷ List of discovered services
11: Create AND nodes for all discovered services S
12: LinkIncoming(S, L) ▷ Link nodes $s \in S$ that are not yet linked to L
13: Create OR nodes for $Out_S - L$ ▷ Create OR nodes for new outputs of all $s \in S$
14: $L \leftarrow \phi$
15: $L \leftarrow$ Create OR nodes for In_s of all $s \in S$
16: **if** $\exists l \in L : l \in In_D$ **then**
17: $P \leftarrow P \cup \{l\}$
18: **end if**
19: **if** $In_D \supseteq P$ **then**
20: SUCCESS
21: **end if**
22: $i \leftarrow i + 1$
23: **end while**
24: **if** $P \nsubseteq In_D$ **then**
25: FAIL
26: **end if**

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:

$$sim(c_{n_i}, c_{n_{i+1}}) = \begin{cases} 0 & \text{if } c_{n_i} = c_{n_i+1} \\ 1 & \text{if } c_{n_i} = Similar(c_{n_i+1}) \end{cases} \tag{3.4}$$

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:

$$g(n) = \sum_{i=0}^{n} cost(n_i) \tag{3.5}$$

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.

$$h(n) = Distance(n, n_0) \tag{3.6}$$

3.6 Implementation and experiments

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

3.6.1 Framework architecture

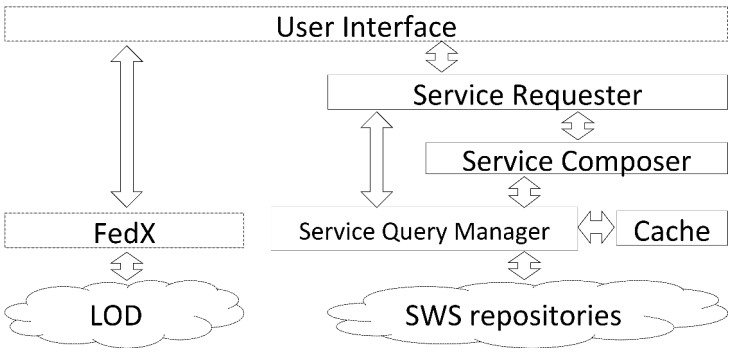


Figure 3.6: Framework Architecture

Figure 3.6 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 open source SPARQL federator, FedX [Schwarte et al., 2011], that we integrated within our framework. FedX uses its own query rewriting to optimize the data querying for each source. Therefore, the LOD can be seen as a federation of SPARQL endpoints of different data sources such as DBpedia.

FedX relies on RDF4J[5] (formerly known as Sesame by the OpenRDF team). One particularity of this framework is that it requires a declaration of all the SPARQL endpoints to be used before initializing the system. Therefore, our framework takes into account this requirement and allows the user to edit the list of SPARQL endpoints to be used within the session before launching the first query.

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 manage the federation of SPARQL endpoints of the SWS repositories separately from the Data federation. Therefore the user can edit the list of service repositories separately. Besides, the query management in the SWS SPARQL federation is separated from the Data federation and is handled using another FedX instance.

We have implemented our framework in Java using Apache Jena[6] framework to manage SPARQL queries and RDF. The implementation materials (demonstration video and executables) can be found at the link bellow[7].

The framework GUI allows the user to write his query, edit the SPARQL endpoints of both data sources and service repositories, choose different strategies for service search (from tight to loose service search , see section 3.3) and activate/deactivate similarity lookup.

In Figure 3.7, the user submits the data query in Listing 3.1 (green) through the "Query Tab". The framework guides the user through the processes described in section 3.2. The query can be executed against the SPARQL-endpoints that the user has selected and results will be displayed in the "Data Tab". On the other hand, he can visualize the generated service queries in the "Service Queries Tab" and manually edit them to fit his needs if necessary then launch service discovery. Once done, the discovered services

[5]http://rdf4j.org/
[6]https://jena.apache.org/
[7]http://sites.google.com/site/lidsearch/

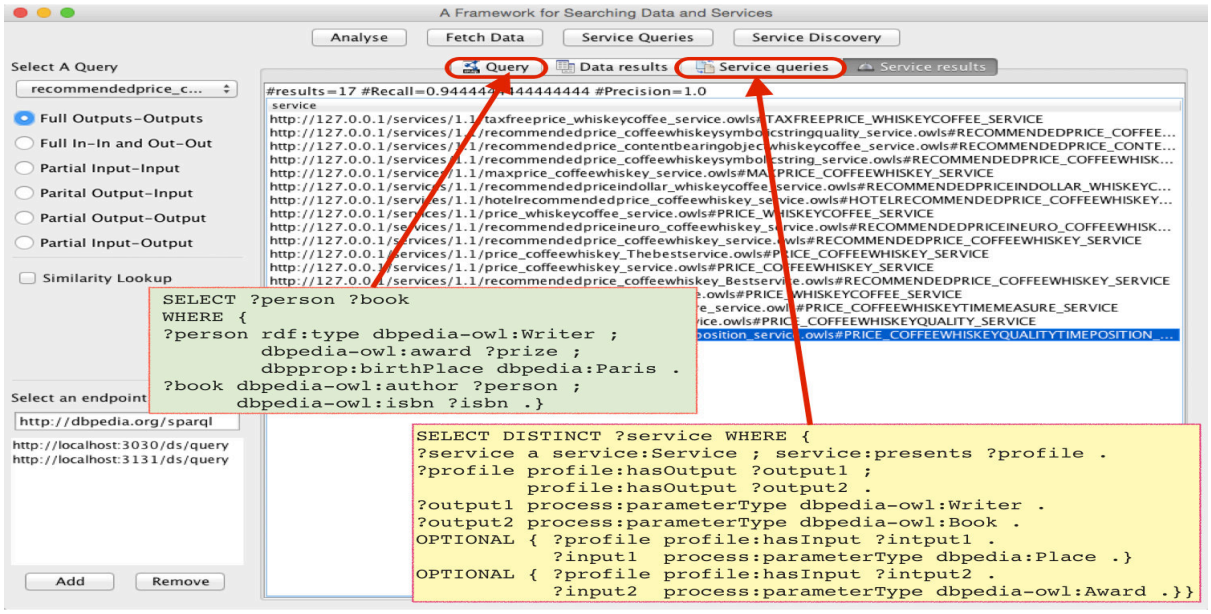


Figure 3.7: A screenshot showing the results of a service discovery process

are returned to the user and sorted according to their relevance according to our ranking method.

In Figure 3.7, in the left panel, the user has selected the strategy $All_{(Out,Out)}$. Therefore the generated service query (in yellow) searches for services that should return the data that the user is looking for as required by the strategy$All_{(Out,Out)}$. The list in the background shows the discovered services hosted in two local repositories.

3.6.2 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 frame-

work. Cache maintenance can be scheduled for automatic launch or triggered manually.

The cache is not yet implemented for the time being. In the future, We will enrich this section with the details of the applied caching method and the evaluation section will include experiments with and without cache.

3.6.3 Evaluation

Our main challenge in evaluating our framework is to find suitable benchmarks that provide SPARQL queries over real world data and to find SWS repositories of that offer real world services. Furthermore, to properly measure the performance of service query writing from data queries, benchmark queries should have missing concept declarations.

Unfortunately, to our best knowledge, there is no such benchmark that allows a full evaluation for the whole process of our framework. Therefore, some benchmarks and measures can be used to evaluate each phase in our process. To prove the feasibility of our approach to search services on the LOD, we have made an implementation as a proof-of-concept that integrates the whole process described in section 3.2. This section covers an evaluation of the execution time of service query generation as well as an evaluation of the service discovery. These evaluations are performed independently one from the other because the test data and the test services are not the same.

3.6.3.1 Service query generation

We evaluate the execution time of service query generation from the extraction of the service request, through semantics lookup and finally to the query wrapping in SPARQL. For such purpose, we use a set of SPARQL queries written written manually to meet the constraint of missing concept definitions within the query.

Figure 3.6.3.2 shows a summary of our experiments on a set of 10 queries with an increasing number of undefined concepts. 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.

3.6.3.2 Service discovery

To measure the effectiveness of our service discovery, we put our framework to the test using OWL-S-TC[8] benchmark. OWL-S-TC provides a set of OWL-S descriptions of virtual services as well as a set of service selection queries accompanied with a set of reference results defined by human experts based on the functional descriptions as well as the textual description of services. The queries in OWL-S-TC are basically expressed as imaginary OWL-S services so that Service Discovery tools would perform discovery as a matching between concrete services and the query service.

The reference results provide scores of matching services for a given query sorted by their relevance degree : 0) non relevant, 1) services that might be helpful, 2) services that might answer the query partially and 3) services that are exactly what the user asked for.

Figure 3.6.3.2 shows the recall and precision values of the service discovery on a set of set of OWL-TC queries by applying strategy $All_{(Out,Out)}$. We have chosen a set of 12 queries in the domains of travel, food, economy and education as it was chosen in [Palmonari et al., 2011]. To make them usable within our framework, we have rewritten these queries manually in SPARQL. We have also slightly adapted the service request extraction algorithm to extract the definitions of inputs directly from the query because the OWLS-TC queries provide all concepts of I/O elements. The results show an overall good recall, however, the precision value is bellow average. This can be explained by the fact that some of the I/O parameters in the queries are very generic (Price or example) which causes the discovery algorithm to have high number of false positives. The reference results in such a case rely on the textual description of the service which restricts the domains of candidate services.

To improve the output of our service discovery, we can use our current algorithm as a pre-filtering system to select primary candidates then apply a more sophisticated hybrid approach like OWLS-MX[Klusch and Kapahnke, 2009] as introduced in [GarcíA et al., 2012].

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

Figure 3.8: Average execution Time in MS per number of undefined variables in a random query

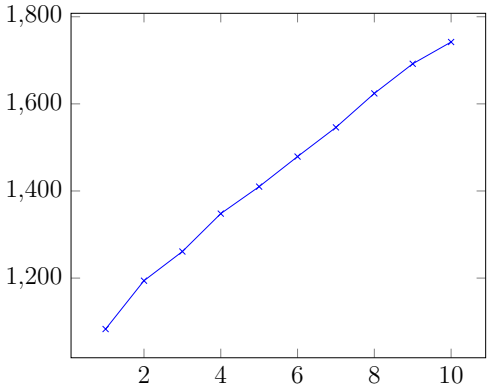
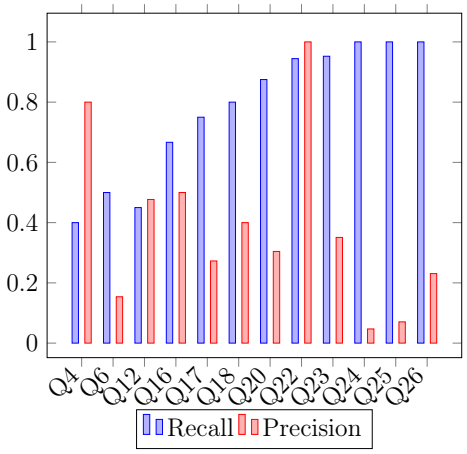


Figure 3.9: Recall and precision of service discovery in OWLS-TC using $All_{(Out,Out)}$



3.7 Conclusion

In this chapter we presented LIDSEARCH, 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 Taheriyan et al. [2012a] or SmartLink Dietze et al. [2011] 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.

Chapter 4

Enriching Service Descriptions with I/O relations

4.1 Introduction

In the previous chapter, we presented LIDSEARCH, our framework that searches for linked data with a data query and automatically finds relevant semantic web services for the data. This automated service discovery can sometimes be a tricky task in some scenarios because the standard service description formalisms do not provide sufficient information about the functionality of a service or its underlying data model. Such incompleteness precludes a full understanding of what a service really does and how its inputs are related to its outputs. One of the use cases that raise the issue is one where two services have identical described functionality in terms of I/O but have totally different behaviours regarding the relations between the inputs and the outputs. Hopefully, such ambiguous functionality can be enlightened in the textual descriptions and documentations of web services in an informal natural language but an additional effort is required to be formalize the I/O relations found within.

To better illustrate this problem, let's consider two semantic web services Service#1 and Service#2 described in OWL-S[1] that provide information about books (see Fig. 4.1). Service#1 (Fig. 4.1a) returns books written by a given author while Service#2 (Fig. 4.1b) returns books for which a given author has written a preface. Both services consume the same input `Writer` and provide the same output `Book` and therefore have identical I/O blocks in their semantic descriptions in OWL-S as shown in Listing 4.1 and 4.2. A typical service discovery algorithm would consider them as similar if requested to provide services with `Writer` as an input and `Book` as an output. However, after looking at their textual descriptions, we quickly figure out that the output `Book` is related differently to the input `Writer` in the two services. Therefore, it would be useful to point out the ontological

[1] http://www.w3.org/Submission/OWL-S/

relations between `Writer` and `Book` that correspond to the ones mentioned in the textual descriptions of each service.

(a) Service description #1 (b) Service description #2

Figure 4.1: An example of two web services with identical I/O types but with totally different functionality

Running Example

2/12

```
...
<profile:Profile rdf:ID="BOOK_AUTHOR_PROFILE">
  <service:isPresentedBy rdf:resource="#
       BOOK_AUTHOR_SERVICE"/>
  <profile:serviceName xml:lang="en">
       BookAuthorService</profile:serviceName>
  <!-- TEXTUAL DESCRIPTION -->
  <profile:textDescription xml:lang="en">
  This service returns the books written by the
       given author
  </profile:textDescription>
  ...
</profile:Profile>
...
<!-- INPUTS -->
<process:Input  rdf:ID="_AUTHOR">
  <process:parameterType rdf:datatype=xs:anyURI>
  http://127.0.0.1/ontology/books.owl#Writer
  </process:parameterType>
  <rdfs:label>Writer</rdfs:label>
</process:Input>
<!-- OUTPUTS -->
<process:Output rdf:ID="_BOOK">
  <process:parameterType rdf:datatype=xs:anyURI>
  http://127.0.0.1/ontology/books.owl#Book
  </process:parameterType>
  <rdfs:label>Book</rdfs:label>
</process:Output>
```

Listing 4.1. Example OWL-S description for Service #1 in Fig. 4.1a

```
...
<profile:Profile rdf:ID="
     BOOK_PREFACE_AUTHOR_PROFILE">
  <service:isPresentedBy rdf:resource="#
       BOOK_PREFACE_AUTHOR_SERVICE"/>
  <profile:serviceName xml:lang="en">
       BookPrefaceAuthorService</profile:
       serviceName>
  <!-- TEXTUAL DESCRIPTION -->
  <profile:textDescription xml:lang="en">
  This service returns the books whose the preface
       is written by the given author
  </profile:textDescription>
  ...
</profile:Profile>
...
<!-- INPUTS -->
<process:Input  rdf:ID="_AUTHOR">
  <process:parameterType rdf:datatype=xs:anyURI>
  http://127.0.0.1/ontology/books.owl#Writer
  </process:parameterType>
  <rdfs:label>Writer</rdfs:label>
</process:Input>
<!-- OUTPUTS -->
<process:Output rdf:ID="_BOOK">
  <process:parameterType rdf:datatype=xs:anyURI>
  http://127.0.0.1/ontology/books.owl#Book
  </process:parameterType>
  <rdfs:label>Book</rdfs:label>
</process:Output>
```

Listing 4.2. Example OWL-S description for Service #2 in Fig. 4.1b

In this chapter, we present an approach that aims to facilitate the automatic service discovery by enriching the service descriptions with detailed information about the functionality of services using the ontological relations between the inputs and outputs. The latter are endorsed by the relations found in the textual descriptions using natural language processing techniques. This approach is a step towards the automation of service discovery that would have many benefits in different domains like: a) improving the accuracy of automatic service discovery, b) facilitating the automatic service composition and automatic service replacement in faulty compositions, c) facilitating data integration when using data from data-providing web services (DPS), d) facilitating the creation of API mashups, e) improving service recommendation, etc.

Throughout this chapter, we will explain our approach step-by-step using the aforementioned example scenario as a running example. This running example is meant to be as simple as possible in every aspect to better explain each step of our approach. Whenever needed, other examples are used to illustrate how our approach deals with some more complex cases.

The rest of this chapter is structured as follows: We give a brief overview of the general process of our approach in section 4.3. In sections 4.4 and 4.5 we describe in detail the I/O relation extraction from both ontologies and textual descriptions respectively. The extracted relations are matched and then ranked as described in section 4.6. In section 4.7 we describe our implementation of a proof of concept that we evaluate later in section 4.8 on different aspects using OWLS-TC[2]. Lastly, we discuss the limitations, future improvements and some future perspectives for this work.

4.2 Related works

The service description enrichment framework is a cross-domain approach to enrich existing semantic web service descriptions with I/O relations. Works related to this framework can be classified into two categories: semantic annotation works and relationship extraction works. In this section we mention a few works of both categories.

4.2.1 Semantic annotation of web services

A lot of research works have been conducted on semantic annotation of web services. They can be classified from different perspectives into: a) dedicated annotation approaches and service discovery approaches with an intermediate annotation step, b) manual and semi-automatic approaches, or c) WSDL oriented and web API oriented approaches.

1. **Dedicated semantic annotation systems:**

   This category of approaches includes works that are solely dedicated to annotating web services and APIs using semantic web ontologies. Most of these works offer a graphical user interface to facilitate the selection and validation of annotations by users. They use machine learning, graph matching, text matching, information extraction, natural language processing, entity extraction, etc to automatically extract or generate ontological concepts for service description elements. The majority of these works focus on annotating functional properties of services, but some works also annotate non-functional properties such as QoS using dedicated vocabularies.

   Amongst the approaches of this category, we cite: [Zhang et al., 2013] that uses DBpedia Spotlight[3] to perform entity extraction and annotates service I/Os with

[2] http://projects.semwebcentral.org/projects/owls-tc/
[3] http://github.com/dbpedia-spotlight/dbpedia-spotlight/wiki

concepts from the DBpedia ontology. [Cheniki et al., 2016] also uses DBpedia to annotate functional properties and some extended ontology to annotate non-functional properties. The work in [Chen et al., 2017] uses WordNet to generate annotations for functional properties. [Patil et al., 2004] and [Heß et al., 2004] are amongst the first works that address the automatic annotation of web services.

WebKarma[4] [Taheriyan et al., 2012a] is a tool for annotating web APIs that do not have any formal descriptions. It asks the user for a few example API calls, then analyzes the call URL and the results to extract I/O. It also helps users in annotating the I/O with concepts from ontologies of their choice and facilitates this process by automatically recognizing ontological concepts. DORIS [Koutraki et al., 2015] and [Lucky et al., 2016] also create annotations for functional properties of Web APIs using LOD ontologies.

2. **Semantic annotation within service discovery systems:**

   These approaches are not dedicated to annotation but use annotation methods to improve service discovery. Instead of creating formal annotations that can be used to enrich service descriptions, they use non-logic based matching techniques to compare service requests and descriptions that do not use the same I/O concepts. This can be regarded as if they implicitly generate informal semantics to enrich the descriptions at service matchmaking time. We have already cited a few of these approaches in chapter 3 such as OWLS-MX [Klusch and Kapahnke, 2009] and iSem [Klusch and Kapahnke, 2012], etc.

   Other approaches like [Palmonari et al., 2011] generate WordNet-based annotations for services on a cold start and store them in order to use them later for service discovery.

   The work in [Speiser and Harth, 2011b] implicitly addresses the ontological I/O relations by proposing manual semantic annotation templates that describe the I/O of a service with Basic Graph Patterns linking them to web ontologies and eventually to each other. To the best of our knowledge, this is the only work that likely addresses annotating services with relations between their I/O but in a manual fashion.

In this section, we only cited a few works on semantic annotation of web services.

[4] http://www.isi.edu/integration/karma/

The reader may refer to the survey in [Tosi and Morasca, 2015] for a more complete literature review of semi-automatic approaches. It surveys dozens of works and yet it doesn't include the manual annotation tools.

In contrast to our work, the aforementioned approaches do no extract relations between I/O and do not apply any matching of these relations with the service description. Our work is complementary to these approaches, aiming to enrich the existing semantic annotations by automatically extracting and adding I/O relations from the ontologies they use.

4.2.2 Relationship extraction

On the other hand, there have been many efforts to enrich ontologies by extracting new semantic relations from text using Natural Language Processing (NLP) techniques [Nakashole et al., 2012; Arnold and Rahm, 2014; Angeli et al., 2015]. Authors of [Nakashole et al., 2012] propose an approach based on dependency grammar to extract new RDF properties for DBpedia from the English Wikipedia[5] using a scalable MapReduce-based algorithm. We apply a similar approach based on dependency grammar to extract I/O relations from textual descriptions but instead of adding them as new semantic relations to the service descriptions, we match them with I/O relations from ontologies and add the latter to service descriptions as basic graph patterns.

4.3 Approach Overview

To achieve the aforementioned desired service description enrichment, our system extracts parallelly two types of I/O relations. The first are the existing formal relations between the I/O of services from their underlying ontologies using SPARQL[6] . The other type is the informal I/O relations depicted in natural language in the text descriptions and documentation of services which requires Natural Language Processing techniques to be extracted. The two extraction processes are computationally independent and therefore are executed in parallel. At the end, the two types of extracted relations are matched against each other in the objective of endorsing the most relevant ontological relations (as multiple relations may exist between I/O concepts in ontologies). The ontological relations that have the best matches in the textual relations are considered convenient

[5] http://www.wikipedia.org
[6] http://www.w3.org/TR/sparql11-query/

for enriching the service description. They can later be added to the semantic service description after validation from the user. The overall process is illustrated in Fig. 4.2.

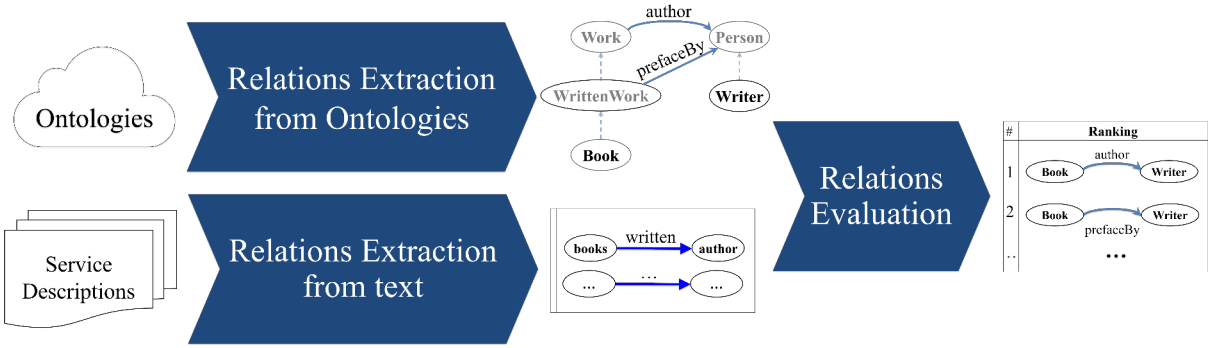


Figure 4.2: I/O relation extraction process

4.4 Extracting I/O relations from ontologies

The inputs and outputs of a semantic web service describe what a service needs in order to be invoked (the type of data that it consumes) and what it produces after its invocation while referring to concepts or nodes in one or more ontologies. Let $I_i \in I$ be an input of a semantic service referring to a node N_i in some ontology and $O_j \in O$ be an output referring to another node N_j. We denote this reference by $I_i = N_i$ and $O_j = N_j$. Given the connected-graph nature of an ontology, there can be at least one path (relation) that links N_i and N_j. Obviously, exceptions may apply if the two nodes are from different but not interlinked ontologies.

The first step of our approach aims to extract all existing relations between the input and output elements of the service within a predefined maximum distance. The extraction operates in a pairwise fashion (for each combination pair of an input and an output) but the extracted relations can all be combined and expressed in the form of a basic graph pattern.

Fig. 4.3 illustrates the relations between two example I/O nodes: $I_1 =$ `Writer` and $O_1 =$ `Book`. As seen in this figure, there is no edge (i.e Property) directly linking the two nodes at a distance $d = 0$, i.e. without intermediate nodes. However, there are multiple paths at different distances ($d = 2, d = 3$) including different nodes and properties that link I_1 and O_1.

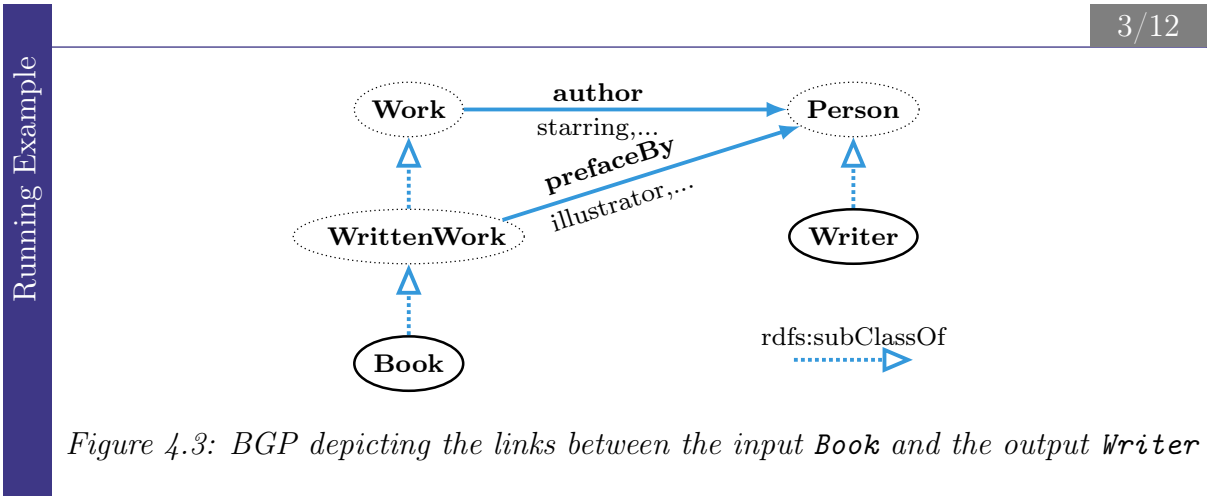


Figure 4.3: BGP depicting the links between the input `Book` *and the output* `Writer`

The relations graph can be extracted from the ontologies referenced in the service description to describe the I/O. For such an extraction, we can either apply graph search algorithms like *Breadth First Search* or *Greedy Best First* to ontology dumps or use a SPARQL based approach. In our process we use SPARQL because it works on-the-fly and allows to search in multiple ontologies in the Linked Data Cloud (LOD) in a federated fashion.

4.4.1 SPARQL-based extraction

We use SPARQL to query the endpoints hosting ontologies for the existing relations between all the input and output nodes in a pairwise fashion. Each combination of an input and an output requires its own SPARQL query consisting of a more or less complex template depending on the nature of the searched paths.

As depicted earlier, two nodes can have one or more relations that go through different paths. Each relation corresponds to a path of some length based on the distance between the two nodes. Depending on its length/distance $d = n$, a path involves one or more properties $p_k \in \{p_0, ..., p_n\}$ or reverse properties $q_k \in \{q_0, ..., q_n\}$ as well as zero or more intermediate nodes $M_k \in \{M_1, ..., M_n\}$.

The properties and reverse properties are `owl:ObjectProperty` or `rdf:Property` that link two source and target `owl:Class` or `rdfs:Class` nodes representing either the I/O nodes or the intermediate nodes. The difference between the two is that reverse properties are properties in which `rdfs:domain` and `rdfs:range` are reversed (i.e. the

subject and the object are reversed). This link is established in the ontology following a specific pattern. Listings 4.3 and 4.4 depict two example property extraction patterns in DBpedia[7] and OWLS-TCv4 respectively between two nodes `?source` and `?target`. The latter is not straightforward as the first because it is meant to be inferred by an inference engine.

```
# ?source-->?target
{?p   rdfs:domain   ?source;
      rdfs:range    ?target .}
# ?source<--?target
{?q   rdfs:domain   ?target;
      rdfs:range    ?source .}
```

Listing 4.3: Property and reverse property extraction pattern in DBpedia

```
# ?source-->?target
{?source   rdfs:subClassOf   ?x .
?x         owl:onProperty    ?p ;
           ?y                ?target .
?target    rdf:type          owl:Class .
}
```

Listing 4.4: Property extraction pattern in OWLS-TCv4 ontology

To obtain all the existing relations in the ontology between an input node I_i and an output node O_j, we need a SPARQL query for each possible path. For a maximum user-defined length/distance $d = n$, there are 2^{n+1} combinations of paths, i.e. possible paths or relations (see Fig. 4.4). We use an algorithm that generates all the queries in an incremental fashion. To reduce querying costs of the ontology servers, we merge all the queries of all path combinations per Input/Output pair as sub-queries into a single SPARQL query using the `UNION` operator.

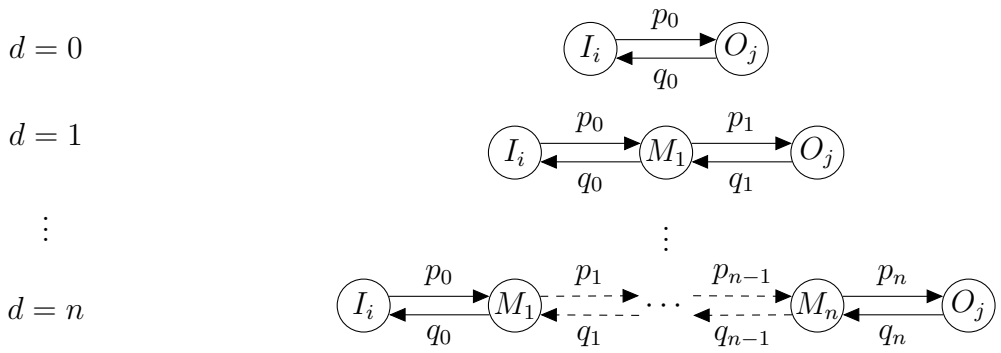


Figure 4.4: Combinations of paths between input and output nodes

An example of a relation extraction query is given in Listing 4.5. It searches for all possible combinations in a union fashion to find the relations between `otc:Author` and `otc:Book` from the OWLS-TC Book ontology[8] (see Fig 4.5) within a maximum distance of $n = 0$. The properties are extracted using the OWLS-TC pattern given in Listing

[7] http://wiki.dbpedia.org/

[8] The reader may notice that in this example, the I/O nodes refer to otc:Author and otc:Book in the

4.3. Table 4.2 shows the results of this query from the OWLS-TC ontology. It shows that there exists for this example a direct relation between `otc:Author` and `otc:Book` through the `otc:writtenBy` OWL property. This relation is represented as an array of strings like `[otc:Book, otc:writtenBy, otc:Author]`

Figure 4.5: A sub-graph from the OWLS-TC ontology showing the relation between `otc:Author` and `otc:Book`.

```
PREFIX otc:<http://127.0.0.1/ontology/books.owl#>
SELECT DISTINCT otc:Book ?p0 ?r0 otc:Author
WHERE {
{otc:Book  rdfs:subClassOf  ?x0 .
?x0        owl:onProperty   ?p0 ;
           ?y0              otc:Author}
UNION {
otc:Author rdfs:subClassOf  ?x0 .
?x0        owl:onProperty   ?q0 ;
           ?y0              otc:Book}}
```

Listing 4.5. Example I/O relation extraction query from OWLS-TC using the pattern in Listing 4.4)

Table 4.1: I/O relation extraction query results

otc:Book	?p	?q	otc:Author
otc:Book	otc:writtenBy		otc:Author

4.4.2 Extraction Enhancements

When observing the most recurrent patterns found in the relation paths between concept nodes from many domains in ontology schemas (not instances), one comes immediately under the light spot: `?x rdfs:subClassOf ?y`. This hierarchical pattern is very common in DBpedia. Fig. 4.3 gives a perfect example of these hierarchical patterns. In fact, the `author` property is not exclusive to Books and Writers but is a general relation between any `Work` and `Person`.

Such a phenomenon would make the relation paths longer and the queries more complex. Therefore, it is important to bind some parts of the paths between nodes with this pattern in order to narrow the search space and reduce the complexity. Instead of searching for all the properties and intermediate nodes, the extraction algorithm applies hierarchy patterns for up to a user-defined maximum hierarchical depth $h = m$ before

OWLC-TC Book ontology. This example query is slightly different from our running example that comes next in Listing 4.6

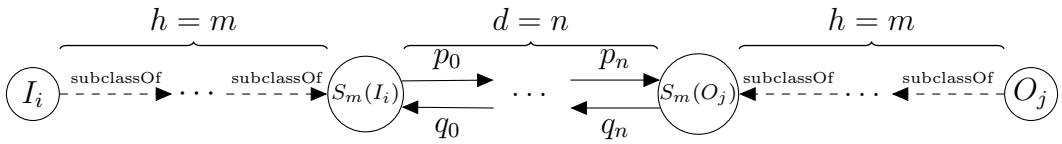


Figure 4.6: Combinations of hierarchical patterns on top of path combinations

looking for general properties and intermediate nodes at a distance $d = n$. Therefore, the number of unknown intermediate nodes in the path is decreased by the introduction of super-class nodes. In addition, the maximum distance value $d = n$ can be decreased as well by the user because most of the path is covered by super-classes (see Fig. 4.6). This improves the overall performance of the system by reducing the number of property variables as well as the number of required joins during the execution of SPARQL relation extraction queries.

For our running example, we search for relations between `Writer` and `Book` within a maximum distance $d = 0$ and a maximum depth $h = 2$ using the query in Listing 4.6. Table 4.2 shows the results of this query from DBpedia. Note that the results table doesn't show the `rdfs:subClassOf` property name for brevity reasons. In fact, there are 15 possible relations with the parameters we used for this extraction. The relation that matches the textual description of Service#1 of our running example is highlighted in yellow in the table and its representation array is: `[dbo:Book, rdfs:subClassOf, dbo:WrittenWork, rdfs:subClassOf, dbo:Work, dbo:author, dbo:Person, rdfs:subClassOf, dbo:Writer]`.

```
SELECT DISTINCT  dbo:Book ?sIn1 ?sIn2 ?p0
?r0 ?sOut2 ?sOut1 dbo:Writer
WHERE{
  { ?p0        rdfs:domain       dbo:Book ;
               rdfs:range        dbo:Writer }
UNION {
  ?r0          rdfs:domain       dbo:Writer ;
               rdfs:range        dbo:Book }
UNION {
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?p0          rdfs:domain       dbo:Book ;
               rdfs:range        ?sOut1 }
UNION {
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?r0          rdfs:domain       ?sOut1 ;
               rdfs:range        dbo:Book }
UNION {
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?sOut1       rdfs:subClassOf   ?sOut2 .
  ?p0          rdfs:domain       dbo:Book ;
               rdfs:range        ?sOut2 }
UNION {
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?sOut1       rdfs:subClassOf   ?sOut2 .
  ?r0          rdfs:domain       ?sOut2 ;
               rdfs:range        dbo:Book }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  ?p0          rdfs:domain       ?sIn1 ;
               rdfs:range        dbo:Writer }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  ?r0          rdfs:domain       dbo:Writer ;
               rdfs:range        ?sIn1 }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?p0          rdfs:domain       ?sIn1 ;
               rdfs:range        ?sOut1 }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?r0          rdfs:domain       ?sOut1 ;
               rdfs:range        ?sIn1 }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?sOut1       rdfs:subClassOf   ?sOut2 .
  ?p0          rdfs:domain       ?sIn1 ;
               rdfs:range        ?sOut2 }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?sOut1       rdfs:subClassOf   ?sOut2 .
  ?r0          rdfs:domain       ?sOut2 ;
               rdfs:range        ?sIn1 }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  ?sIn1        rdfs:subClassOf   ?sIn2 .
  ?p0          rdfs:domain       ?sIn2 ;
               rdfs:range        dbo:Writer }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  ?sIn1        rdfs:subClassOf   ?sIn2 .
  ?r0          rdfs:domain       dbo:Writer ;
               rdfs:range        ?sIn2 }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  ?sIn1        rdfs:subClassOf   ?sIn2 .
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?p0          rdfs:domain       ?sIn2 ;
               rdfs:range        ?sOut1 }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  ?sIn1        rdfs:subClassOf   ?sIn2 .
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?r0          rdfs:domain       ?sOut1 ;
               rdfs:range        ?sIn2}
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  ?sIn1        rdfs:subClassOf   ?sIn2 .
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?sOut1       rdfs:subClassOf   ?sOut2 .
  ?p0          rdfs:domain       ?sIn2 ;
               rdfs:range        ?sOut2 }
UNION {
  dbo:Book     rdfs:subClassOf   ?sIn1 .
  ?sIn1        rdfs:subClassOf   ?sIn2 .
  dbo:Writer   rdfs:subClassOf   ?sOut1 .
  ?sOut1       rdfs:subClassOf   ?sOut2 .
  ?r0          rdfs:domain       ?sOut2 ;
               rdfs:range        ?sIn2 }
  }
```

Listing 4.6. Example I/O relation extraction query from DBpedia-like ontologies using the `rdfs:domain|rdfs:range` pattern (see Listing 4.3)

Running Example

5/12

Table 4.2: Relation Extraction query results from DBpedia for the query in Listing4.6

dbo:Book	sIn1	sIn2	p0	q0	sOut2	sOut1	dbo:Writer
dbo:Book	dbo:WrittenWork		dbo:illustrator			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork		dbo:prefaceBy			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork		dbo:firstPublisher		dbo:Agent	dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:composer			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:writer			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:author			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:narrator			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:coverArtist			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:translator			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:mainCharacter			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:chiefEditor			dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work		dbo:created		dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work		dbo:debutWork		dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:publisher		dbo:Agent	dbo:Person	dbo:Writer
dbo:Book	dbo:WrittenWork	dbo:Work	dbo:producer		dbo:Agent	dbo:Person	dbo:Writer

4.5 I/O relation extraction from textual descriptions

The textual description of a web service as well as its documentation tend to tell more about the service functionality by revealing some of its underlying data model and the conceptual relations between its inputs and outputs. Based on this assumption, the relation extraction process from text descriptions aims to discover these implicit relations in the text in order to match them later with the existing explicit relations found in the ontologies.

This process is executed in parallel to the first one and consists in three sequential tasks as illustrated in Fig. 4.7: text pre-processing, I/O words recognition and relation extraction. The pre-processing task consists in annotating the textual description of a service using Natural Language Processing (NLP) techniques in order to facilitate the next two tasks. I/O recognition consists in recognizing the words used for I/O elements in the text using *word2vec*[Mikolov et al., 2013] word-word similarities. The final task is the extraction of I/O relations from the text using the previous annotations. The following subsections provide more details about the three tasks of this process.

4.5.1 Service description's text pre-processing

The textual part of a service description is written in an "informal" natural language. It might consist of a few sentences (a single sentence in OWLS-TC) up to many para-

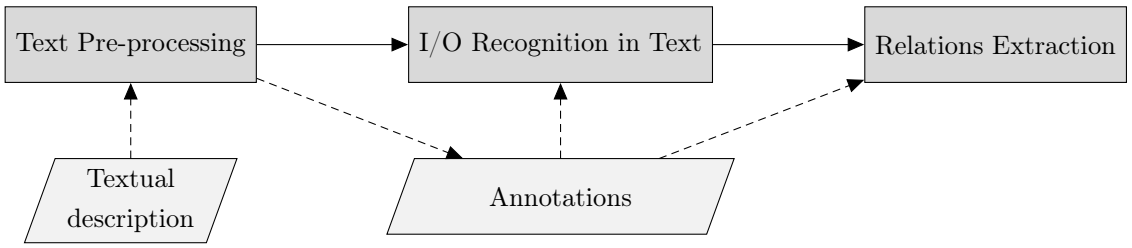


Figure 4.7: Process overview of I/O relation extraction from text.

graphs. The pre-processing task is a pipeline that aims to annotate the text in order to highlight some useful features that help understanding the text and extracting the I/O relations. It consists in four sequential tasks as illustrated in Fig. 4.8 and results in a set of annotations for the service description. The four sub-tasks are detailed in the following sub-sections.

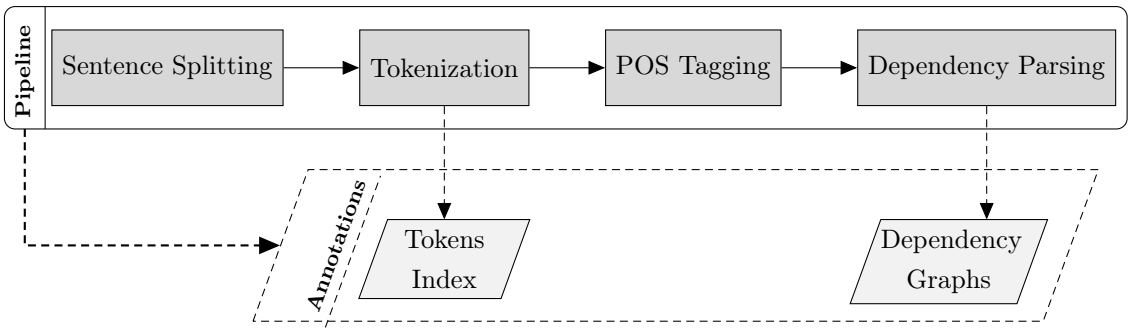


Figure 4.8: Text pre-processing pipeline

4.5.1.1 Sentence Splitting

Sentence splitting or segmentation consists in dividing the text into sentences. Punctuation characters (., !, or ?) that mark a sentence ending are used as a delimiter for splitting. However, this is not a trivial task because of two main reasons: 1) the lack of punctuation due to poor writing quality or 2) the use of the full stop character (.) in abbreviations can be misleading as in "Mr." for example. Faulty sentence splitting affects the dependency parsing that comes later and systematically impedes the extraction of I/O relations in the text.

4.5.1.2 Tokenization

Tokenization consists in splitting a string into sequences of characters that form a useful semantic unit called a token, often loosely called a word or a term. [Manning et al., 2008]. In addition to language words, tokens include inter alia punctuation, numerals and symbols. While tokenization appears to be a fairly easy problem, it is not a trivial task, especially in non-Latin-script languages such as Arabic-script and Asian-script languages. In many Latin-script based languages including English, words are mostly atomic and therefore can be delimited with a space. However, there are many compound words and unusual terms that require some attention. We distinguish 3 types of compound words in English [Straus et al., 2014]:

1. *Closed forms* that have been established in the language and used since long ago such as football, sometimes, airport, etc.
2. *Hyphenated (dashed) forms* as in camera-ready, computer-aided, to ice-skate, 20-year-old, four-wheeled car etc.
3. *Open (spaced) forms* like in deep learning, New York, Los Angeles, etc. Some of these compound words are entity names that require Named Entity Recognition (NER) techniques to be grouped together as a single unit for preserving their intended meaning. However, at this point we need them split for the next steps.

Running Example 6/12

Table 4.3: Tokens index for the example service #1 in Fig. 4.1a

		token# in sentence									
		1	2	3	4	5	6	7	8	9	10
sent#	1	This	service	returns	the	books	written	by	the	given	author

Upon sentence splitting and tokenization we store the tokens in a dynamic matrix to keep track of their position in the text regarding their order of appearance in the sentence and the order of their sentence in the text. Table 4.3 and Table. 4.4 represent two example tokens indexes for a single sentence (from our running example) and a two-sentence length service descriptions respectively.

Table 4.4: Tokens index for an example service description with 2 sentences

		token# in sentence								
		1	2	3	4	5	6	7	8	9
sent#	1	This	service	returns	novels	written	by	the	given	author
	2	Their	recommended	price	is	also	informed			

4.5.1.3 Part-Of-Speech Tagging

Part-Of-Speech (POS) Tagging is the task of marking each word of the text with its proper part of speech. Parts of speech can be defined as categories of words sharing the same grammatical properties. Linguists mostly agree that there are three primary parts of speech: verb, noun and adjective, exceptions may apply to some languages as always. Obviously, the list of parts of speech is an ongoing subject of debate. In natural language processing, there are many POS tag sets derived mostly from the three major parts of speech. The most popular one for English is the Penn Treebank POS tags set [Santorini, 1990] which consists of 36 main tags in addition to punctuation tags (45 in total) as listed in Table 4.5.

Table 4.5: Alphabetical list of POS tags used in the Penn Treebank Project

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun

Tag	Description
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

POS Tagging is not a task that we rely upon directly in I/O relation extraction but it is required for the next and final task of text pre-processing, thus, we give it a brief

mention here. It is an important prerequisite because POS tags provide information about words and their neighbors as well. A verb for instance is often preceded by a noun and a noun by a determiner or an adjective.

Fig. 4.9 shows the POS tags of the sentence from Service#1 of the running example. Each POS tag is mentioned below its corresponding word. For instance, the word *"returns"* is a verb (VBZ), the word *"service"* is a singular noun (NN) and the word *"books"* is a plural noun (NNS). The reader may refer to Table 4.5 for a complete list of descriptions for the used tags.

4.5.1.4 Dependency parsing

Dependency parsing is at the core of the I/O relation extraction process from the text. It is the task of extracting dependencies between the words of a sentence and representing them in a graph formalism called a dependency graph based on dependency grammars. Dependency grammars are a set of theories that rely on the notion of dependency, a binary grammatical relation (dependency) between two syntactic units (words) that defines the role that one word plays with respect to the other within the context and structure of the sentence. For example, a noun can be related to a verb as its subject or as its object. First introduced by [Tesnière, 1959], dependency grammars have evolved ever since and linguists have developed different taxonomies (tagsets) and representations (directed graphs) of dependencies that are used today by NLP toolkits and dependency parsers [De Marneffe and Manning, 2008].

The dependency parsing relies on different algorithms and techniques used by dependency parsers. The most famous techniques rely on statistical models such as in the Stanford CoreNLP[9] parser [Manning et al., 2014]. Recently, advances in neural networks and deep learning allowed to create new dependency parsers that perform very well like the one in [Chen and Manning, 2014] which is used in the recent versions of CoreNLP.

The dependency parsing is applied to all the sentences of the textual descriptions to obtain dependency graphs. Fig. 4.9 illustrates the extracted dependency graph from the single sentence of the running example. Words are linked to each other via directed edges which are labeled with (abbreviated) dependency relations that correspond to the dependency type between them. Table 4.6 lists the full form of dependency relations that apprear as abbreviations in Fig. 4.9. For instance, the word *"service"* is a nominal subject (*nsubj*) for the word *"returns"* and the word *"books"* is a direct object (*dobj*) of

[9] http://stanfordnlp.github.io/CoreNLP/

the word *"returns"*. For the complete and detailed list of dependency relations, please refer to [De Marneffe and Manning, 2008].

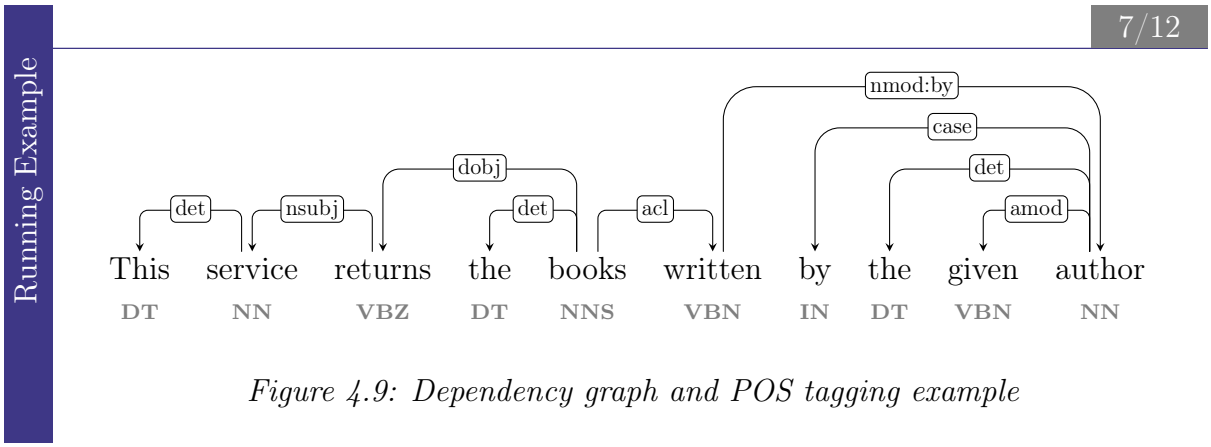


Figure 4.9: Dependency graph and POS tagging example

Table 4.6: Example dependency relations from Fig. 4.9

Abbreviation	Dependency relation
acl	clausal modifier of noun
amod	adjectival modifier
case	case marking
det	determiner
dobj	direct object
nmod:by	nominal modifier
nsubj	nominal subject

4.5.2 I/O recognition in text descriptions

Before starting to search for relations between I/O in the dependency graphs, we first need to recognize the corresponding tokens of I/O words in the text. The I/O recognition is a twofold task that consists in :

1. Extracting the I/O words from the Inputs/Outputs declaration in the service description. Given the nature of the semantic service descriptions in which the I/O elements are declared as URIs referring to some nodes in some ontologies, I/O words can be extracted from two possible locations:

(a) *RDFS Labels:* By fetching their labels from their reference ontologies through the `rdfs:label` property. For example, an I/O element that refers to the `dbo:Book`[10] class in DBpedia has the label *"book"* retrievable from the property `rdfs:label book` which fits perfectly as a word for this I/O.

(b) *Local names:* In the absence of labels, the local names in the URIs of I/O elements are often meaningful words that generally require a special treatment. For instance, local names can contain compound words, dash-separated words or concatenated words in CamelCase[11] as is the case in OWLS-TC, etc. A custom regex (regular expression[12]) pattern can be added manually to the enrichment system to deal with each case individually. Obviously, exceptions may apply here as well.

2. Recognizing the I/O tokens. After knowing what to search for in the previous step, we apply a *"semantic"* string similarity algorithm based on *word2vec* [Mikolov et al., 2013] to find the best matches for each I/O word in the text. Each input or output can be mentioned multiple times in different sentences and contexts, or can be composed of different words.

   word2vec allows to represent words in a vector space model and provides interesting features such as cosine similarity between two words. The advantage of using *word2vec* is its syntax-independent and dictionary-free representation as well as its ability to find semantic similarities based on the context. This fits perfectly in our use-case because the text can refer to I/O words with their synonyms, subclasses, super-classes, etc.

   For our recognition purposes, we calculate the cosine similarity between each I/O and each token in the text and keep all matches above some threshold value. For compound I/O elements composed of multiple words, each word is matched individually in the text. The outcome of this task is a mapping of all I/O words with the indexes of their matches (their positions in the text) and the similarity value.

Fig. 4.10 illustrates the recognized I/O elements for our running example from Service#1 in Fig. 4.1a. The input and output words are highlighted in the sentence after

[10] `http://dbpedia.org/ontology/Book`

[11] Practice of writing compound words in a concatenated fashion without any separators where the first letter of each word is capitalized. see `http://en.wikipedia.org/wiki/Camel_case`

[12] `http://en.wikipedia.org/wiki/Regular_expression`

being matched with the I/O of the service. The colored tables within the figure show the mappings of I/O tokens to the indexes of their matches in the text. For instance, the input `Writer` has a match in the text at the 10th position in the sentence, the word *"author"* which is a synonym of *"writer"*.

Running Example

8/12

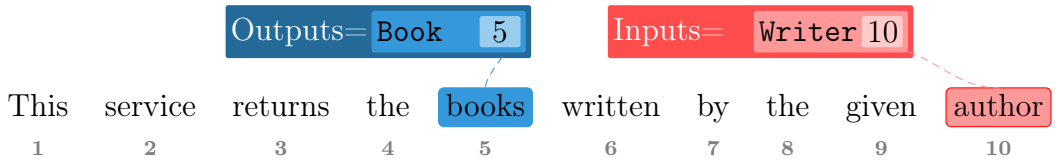


Figure 4.10: I/O words recognition example from Service#1 in Fig. 4.1a

Fig. 4.11 shows the I/O recognition results for another service where the output element `Four_Wheeled_car` is an underscore-separated compound word extracted from the local name of its URL. It has three matches in the text that form a sequence $\{14, 15, 16\}$ highlighted in blue.

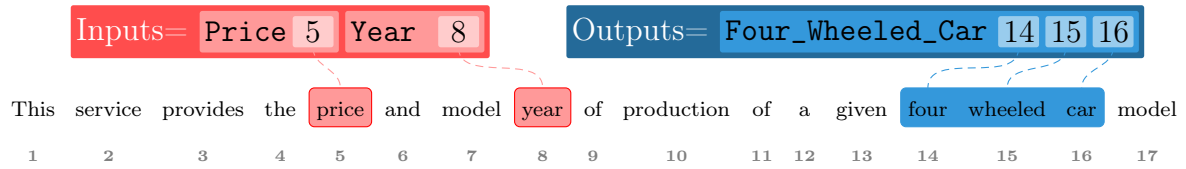


Figure 4.11: Compound I/O words recognition example

4.5.3 Relation extraction

After the two previous tasks, extracting I/O relations is now straightforward. Starting from the generated dependency tree, a BFS (Breadth First Search) graph search algorithm finds all the dependency paths between the inputs and the outputs in a pairwise fashion.

The extracted paths are represented as lists of strings containing only the involved words in the path without the dependency relations. The I/O tokens are replaced by their ontology labels extracted previously to help the matching process that comes next in focusing only on the intermediate words in the path.

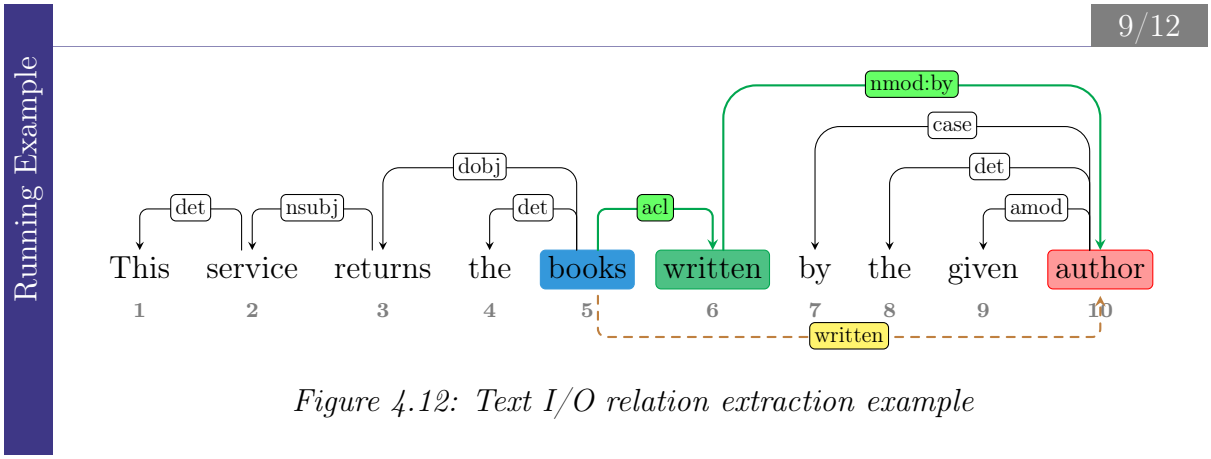


Figure 4.12: Text I/O relation extraction example

Fig. 4.12 illustrates the extracted relation in the form of a triple from the example Service#1 in Fig. 4.1a. The only dependency path between the I/O [(author ← written ← books] is converted into a relation array as in Table 4.7b where the ontology tokens for I/O are substituted for the ones from text.

4.6 Evaluating I/O relations

The final step of the service enrichment process consists in evaluating and ranking the extracted ontological I/O relations before passing them to the user to validate and add them to the service description. More precisely, this step aims to evaluate the ontology relations based on their matching with the text relations.

At this point, we have two relations matrices, $\boldsymbol{L} = (Orel_{ij})$ for ontology relations and $\boldsymbol{T} = (Trel_{ij})$ for text relations where each cell contains a list of ontology and text relations respectively (see the example in Table 4.7). The rows and columns of these matrices represent inputs and outputs respectively. The evaluation algorithm iterates over all the pairs of ontology and text relations and calculates the similarities between each pair.

Calculating similarities between I/O relations of different type is not a trivial task. First, we need to homogenize the format of text and ontology relations before comparing them. For our chosen similarity calculation approach, we require each relation to be represented as a list/array of all the words in their order of appearance in the relation path. The text relations are already represented as an array (for each relation) as in the example

Running Example

10/12

Table 4.7: Extracted I/O relations for Service#1 in Fig.4.1a

(a) Extracted ontology relations matrix $\boldsymbol{L} = (Orel_{ij})$

		Outputs
		Book
Inputs	Writer	[dbo:Book, dbo:WrittenWork, dbo:illustrator, dbo:Person, dbo:Writer]
		[dbo:Book, dbo:WrittenWork, dbo:prefaceBy, dbo:Person, dbo:Writer]
		[dbo:Book, dbo:WrittenWork, dbo:Work, dbo:author, dbo:Person, dbo:Writer]

(b) Extracted text relations matrix $\boldsymbol{T} = (Trel_{ij})$. Note that I/O words are substituted for text tokens

		Outputs
		Book
Inputs	Writer	[Book,written,Writer]

in Table 4.7b. However, the ontology relations have a different format upon extraction. We create an array form for each relation by substituting the local names or labels for the URIs of nodes and properties except `rdfs:subClassOf` and we preserve the order of the relation's path. For example, the relation array `[dbo:Book, rdfs:subClassOf, dbo:WrittenWork, rdfs:subClassOf, dbo:Work, dbo:author, dbo:Person, rdfs:subClassOf, dbo:Writer]` of the example from Table 4.2 is transformed into `[dbo:Book, dbo:WrittenWork, dbo:Work, dbo:author, dbo:Person, dbo:Writer]`. The relation `[otc:Book, otc:writtenBy, otc:Author]` from the example in Table 4.1 is preserved as is because it doesn't include any super-classes (i.e no `rdfs:subClassOf` properties).

4.6.1 Relation embedding using aggregated word2vec vectors

Because relations can have different lengths, the similarity calculation should be size-independent. To achieve such size-independence, we take advantage of *word2vec*'s cosine similarity and vector space model. In fact, aggregating the word2vec vectors of words of two relations containing similar words results in two aggregated word2vec vectors with a pertinent cosine similarity measure. In Fig. 4.13, we create two aggregated vectors

V_{r_1} and V_{r_2} for an ontology and a text relation from Table 4.7 consisting of six and three words respectively. The cosine similarity of the two is calculated later to measure their similarity. The aggregation of *word2vec* vectors' values is guaranteed to be size-independent because it is only impacted by the semantics of words and not by their number.

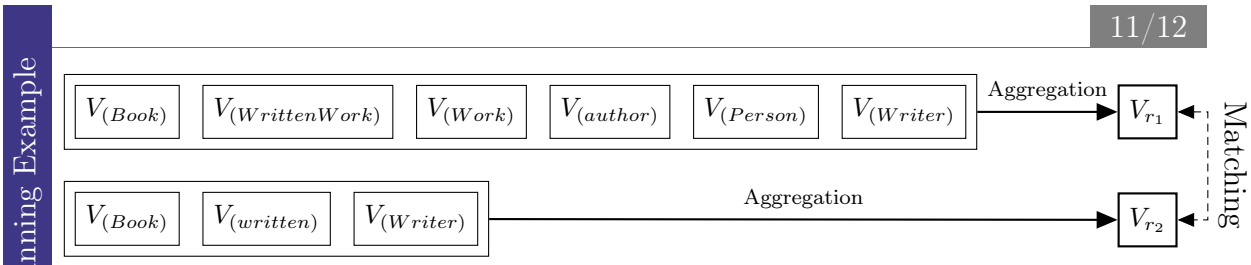


Figure 4.13: Example I/O Relations embedding using aggregated word2vec vectors

Let r be an I/O relation (it doesn't matter of what type because they are homogenized at this point) consisting of k words. Let V_{w_i} where $i \leqslant k$ be the *word2vec* vector embedding of each word w_i of r. It is obtained using the $wordVec_(w_i)$ function as in the following formulae:

$$V_{w_i} = wordVec(w_i) \tag{4.1}$$

To create an embedding for r using aggregated word2vec vectors we use the two following word2vec-based formulae:

1. *Element-wise Vector sum method:*

   To obtain the aggregated vector V_r of a relation r with k words, we calculate the element-wise vector sum of all V_{w_i} vectors of w_i words of r as follows:

   $$relVec^{\oplus}(r) = \sum_{i=0}^{k} wordVec(w_i) \tag{4.2}$$

2. *Hadamard product method:*

   We calculate the element-wise vector product (aka Hadamard product) of all V_{w_i} vectors of w_i words of r as follows:

   $$relVec^{\odot}(r) = \prod_{i=0}^{k} wordVec_(w_i) \tag{4.3}$$

4.6.2 Relation matching

After calculating word2vec vectors for relations and their words, we can calculate the similarity between relations using of the following formulae:

1. *Cosine similarity:* Given two word2vec vectors V_1 and V_2, The cosine similarity is based on the cosine distance given by:

$$similarity(V_1, V_2) = cos(V_1, V_2) = \frac{V_1 \cdot V_2}{||V_1||_2||V_2||_2} \tag{4.4}$$

   The cosine distance between two word2vec vectors gives a relevant similarity value as proven in [Mikolov et al., 2013]. At this step we use it for calculating similarities between relations or between words both as in the next formulae. We have used it previously in the I/O recognition task to match words as previously mentioned in 4.5.2.

2. *Average of Maximums:*

   This method is a baseline approach for calculating similarities between relations without vector aggregation. Let r_1 and r_2 be an ontology and a text relation respectively represented as arrays as mentioned previously. Let $k = |r_1|$ and $l = |r_2|$ be their respective cardinality, i.e. number of words.

   To calculate their similarity, first we calculate the word-word similarity matrix of size $k * l$ of all the possible pairs of words (w_i, w_j) from r_1 and r_2.We use the cosine similarity function from the equation above (4.1) $similarity(w_i, w_j)$ for word-word similarity.

   After that, we calculate the maximum similarity values for each row (i.e we find the best matching word in the text relation for each ontology relation word) for a total of k rows. Finally we calculate the average of maximums on k as the similarity value between r_1 and r_2. The following equation formalizes this similarity calculation method:

$$AvgMaxSim(r_1, r_2) = \frac{\sum_{i=1}^{k} Max_{j=1}^{l}(similarity(w_i, w_j))}{k} \tag{4.5}$$

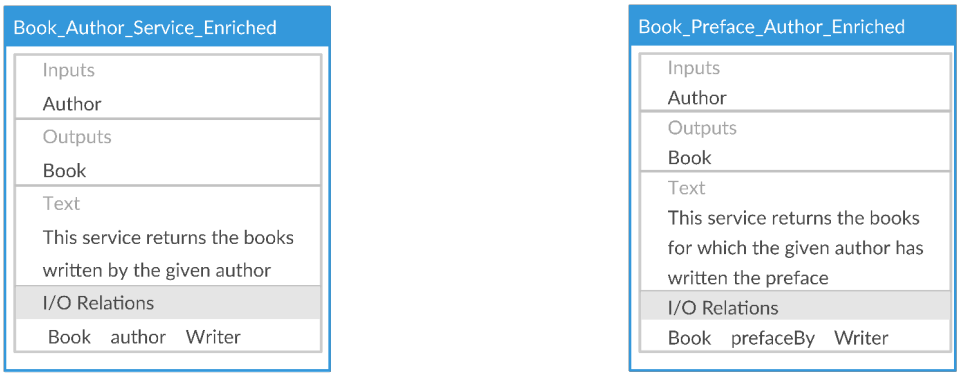
After calculating similarity between relations using the cosine similarity of their aggregated vectors (Equations (4.1) and (4.2)) or the average of maximums formulae (Equation (4.3)), the final similarity value is given by the maximum value of all the three. The

reader can refer to Fig. 4.16 located further in section 4.7 for some examples of similarity calculation between words and relations.

Upon the pairwise matching, we select the ontology relations that have the best matches in the text relations and suggest them to the user for validation before using them to enrich the semantic service description. Fig. 4.14 illustrates the enriched service descriptions of the example services in Fig. 4.1. Obviously, the extracted I/O relations are represented here in a simple way for illustration purposes only.

Running Example

12/12



(a) Enriched Service description #1 (b) Enriched Service description #2

Figure 4.14: An example of two web services (see Fig. 4.1a and Fig. 4.1b) after the description enrichment with I/O relations

4.7 Implementation

In this section we give some implementation details about our service description enrichment approach. Our approach is implemented in Java as a desktop application and is composed of different modules, a restful API and some external dependencies as shown in the software architecture in Fig. 4.15:

1. **User Interface** It is the entry point to trigger the service description enrichment either manually by the user who selects a specific service, or automatically by running the process on a whole service repository. It also includes an experimentation

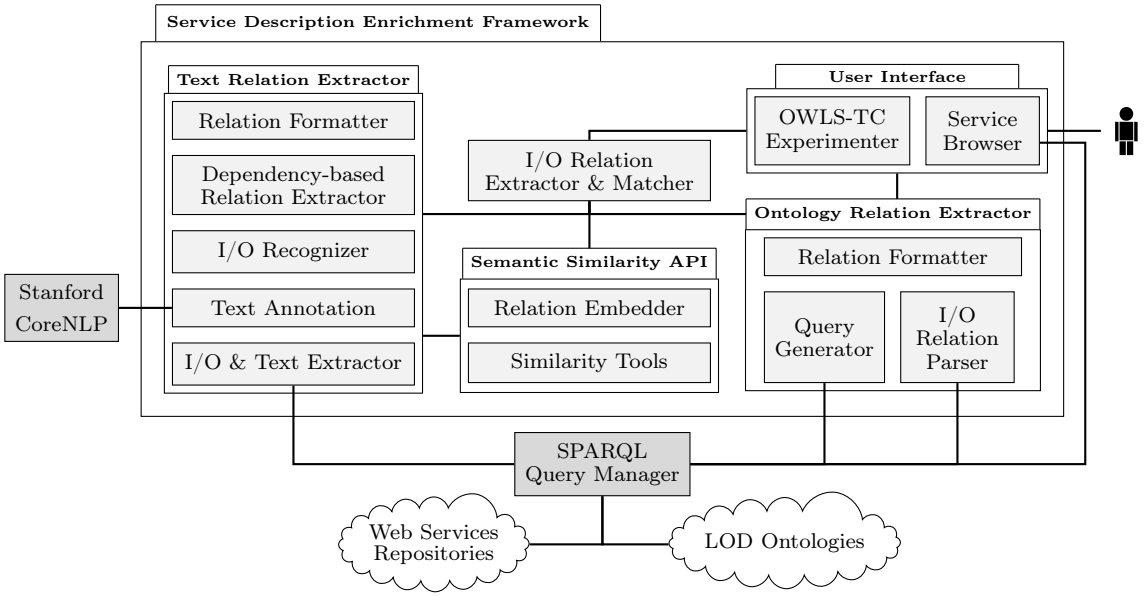


Figure 4.15: Approach implementation: Software architecture

module for OWLS-TC that helped us evaluate our system (Section 4.8). It offers the user the possibility to fine tune the service description enrichment by adjusting the similarity calculation method and thresholds, the ontology distance and depth thresholds as well as selecting specific ontology endpoints.

2. **Ontology relation Extractor** This module is responsible for I/O relation extraction from ontologies. It consists of three sub-modules: a) a query generator that generates SPARQL queries for each I/O pair to find relations based on the query parameters set by the user (maximum relation distance, maximum hierarchy depth and filters), b) a query result parser (I/O relation parser) that parses query results into sets of raw I/O relations then hands them over to c) the relation formatter which formats them in a String array containing I/O names (labels or local names) ordered in their path sequence order and substitutes super classes of I/O nodes with I/O names as mentioned before.

3. **Text relation Extractor** This module is responsible for I/O relation extraction from textual descriptions based on the process described in Fig 4.7. It consists of five sub-modules operating sequentially starting from: a) An extraction module that extracts I/O elements and text descriptions form semantic web service descriptions using SPARQL queries. b) A text pre-processing module responsible for

pre-processing and annotating the text (using Stanford CoreNLP). c) An I/O recognizer that uses some of the previous annotations and the semantic similarity API to recognize I/O words in the text. d) The dependency-based relation extractor extracts relations between pairs of recognized I/O words in the text using dependency graphs then hands them over to e) the relation formatter which formats them to String arrays and substitutes the I/O tokens from the text for the I/O words to reduce improve the relation matching.

4. **Relation Matcher** This module is responsible for managing the I/O relations extraction then for matching them by using the Semantic Similarity API. It returns the matched I/O relations to the user for validation.

5. **Semantic Similarity REST API** This module offers tools for word and sentence embedding as well as for measuring the similarities between words or sentences and relations. It receives word similarity requests from the I/O tokens recognition module and relation similarity requests from the relation matcher. Relation matching requests first go through the relation embedder to create custom embeddings (aggregated word2vec vectors) for relations before matching them.

   The semantic similarity API is implemented as a RESTful API deployed on a distant server for performance purposes. The used similarity method is based on word2vec which requires an important memory size. We use a Java implementation of the word2vec algorithm called Deeplearning4J[13] which is the only industrial-grade implementation of word2vec in Java for the moment.

   Fig. 4.16 gives an overview of the Semantic Similarity REST API usage with examples. The first example requests the API to return the similarity between the words *"author"* and *"writer"*. The second example requests the similarity between the sentences *"book hasAuthor writer"* and *"book writtenBy author"*. The API responds for both queries in JSON. The sentence similarity response contains three similarity values calculated according to the equations 4.5, 4.2 and 4.3 respectively.

6. **External dependencies** We use two external tools in our framework: a) Apache Jena ARQ as a SPARQL query Manager to send sparql queries to the LOD ontologies for ontology relations retrieval as wall as to Semantic Web Service repositories for service description retrieval. b) Stanford CoreNLP is a framework toolkit by

[13] http://deeplearning4j.org/

Figure 4.16: Similarity API documentation with examples

Stanford University that provides tools and models for natural language processing. We use it for annotating textual descriptions with POS tags and for dependency parsing.

Fig. 4.17 shows our Service Description Enrichment desktop application in action. The main window offers the possibility to load a service repository, select a service and extract I/O relations. The user can preview the extracted I/O words highlighted in the description text and see the matched I/O relations highlighted in green. A right palette allows the user to adjust the I/O relation extraction parameters starting from the ontology SPARQL endpoints URLs and the maximum ontology relation length to the word/sentence similarity method and threshold.

4.8 Evaluation and experimental results

In this section, we evaluate the feasibility and effectiveness of our proposed approach. We mainly seek to prove feasibility with the implemented Proof Of Concept but we also evaluate three aspects individually. These aspects correspond to the three underlying tasks of approach's process: extracting I/O relations from ontologies, extracting I/O relations from text and evaluating I/O relations.

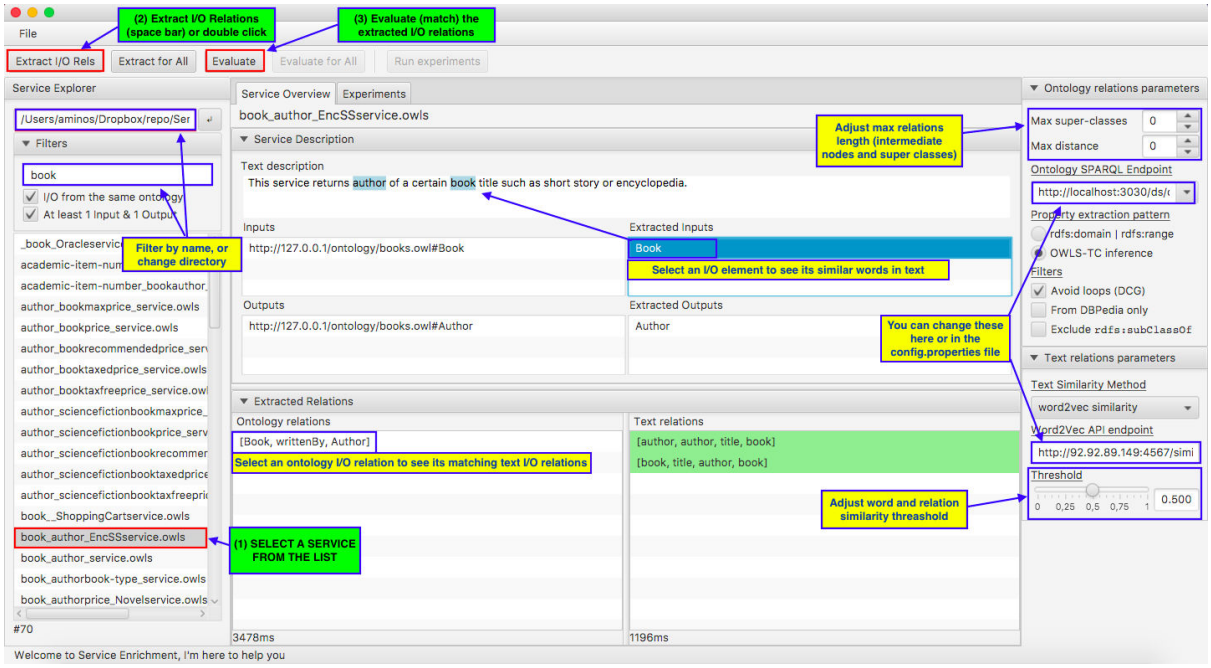


Figure 4.17: A screenshot of the implemented approach in action commented with some instructions

4.8.1 Evaluation setup

We use OWLS-TCv4 as a data-set for evaluating our approach because it provides a fair number of services (1008) and a number of ontologies (48) used by these services with more than 55900 triples. Its services have 1540 inputs and 1615 outputs in total as depicted in Table 4.8a. We host the ontologies and the services in a remote RDF repository accessible via a SPARQL endpoint provided by Apache Jena-Fuseki.[14]

For our *word2vec* string similarity approach, we use the *GoogleNews* pre-trained model[15] provided with the original *word2vec* paper [Mikolov et al., 2013]. We chose this model for its general purpose corpus and vocabulary. Probably, another trained word2vec model would have a better impact on our results, but again our purpose is to prove the feasibility rather than to obtain the best possible score. The semantic string similarity module is deployed as a RESTful API on a distant powerful server for a better performance given the size of the used model as we mentioned previously.

[14] http://jena.apache.org/documentation/fuseki2/index.html
[15] available at http://github.com/mmihaltz/word2vec-GoogleNews-vectors

4.8.2 Experimental results

4.8.2.1 Ontology I/O relation extraction

This first task aims to maximize its recall value by extracting *all* the *existing* relations in order to evaluate them later. Therefore, and given the nature of this graph search problem, the outcome has a boolean nature where relations either exist or not. However, the number of extracted relations depends on the user-defined maximum relation length and hierarchy depth.

The outcome of our proposed approach depends in terms of recall on the outcome of this first task which itself depends on the quality of OWLS-TC ontologies and services. For I/O ontology relations to be extracted, a service must have at least one input and one output and all its I/O must refer to the same ontology (since OWLS-TC ontologies are not interlinked). Table 4.8b shows some statistics about the I/O and ontology usage in OWLS-TC where: a) S^{all} is the set of all OWLS-TC services. S^{min} is the set of services that have the minimum requirements of at least one input and one output. S^{intra} are services whose all inputs and outputs refer to the same ontology (intra-ontology I/O). Therefore, a usable service that satisfies all the previously mentioned requirements must belong to $S^{usable} = S^{min} \bigcap S^{intra}$, the set of usable services. As in Table 4.8b, merely 19% of OWLS-TC services are usable for evaluating our I/O relation extraction approach.

Running our system on the set of usable services shows that the hierarchy depths have more impact on the extraction than the maximum lengths (number of intermediate nodes) as shown in the results in Table 4.9. In fact, the number of extracted ontology I/O relations increases more remarkably by increasing the maximum hierarchy depth values than by increasing the maximum path distance. Therefore, the ontology I/O extraction enhancement by hierarchical patterns has proven to be very efficient. The last two cells of the table are missing due to unreasonable execution time required to calculate them because of the increased algorithmic complexity at this point (depth and distance). Some optimizations are necessary to improve the algorithmic complexity of I/O relation extraction from ontologies. We will discuss this later in the conclusion.

Amongst the list of usable services S^{usable}, only 64/205 services (31%) made it through the ontology relations extraction experiments. This number is relatively low compared to the 1083 services of OWLS-TC. This is due to the fact that OWLS-TC ontologies were not designed for such a use-case and often lack relations even between close nodes.

Table 4.8: I/O of services in OWLS-TC and their ontology usage

(a) Total I/O in OWLS-TC

	Services	Inputs	Outputs
#	1083	1540	1615

(b) I/O and ontology usage

	S^{all}	S^{min}	S^{intra}	S^{usable}
#	1083	996	274	205
%	100	91,96	25,30	18,92

Table 4.9: I/O ontology relations extraction results in OWLS-TC

	max length							
max	$d = 0$		$d = 1$		$d = 2$		$d = 3$	
depth	#r	#s	#r	#s	#r	#s	#r	#s
$h = 0$	12	11	12	11	258	40	307	40
$h = 1$	39	34	57	38	2385	59	17611	59
$h = 2$	64	43	117	48	3852	64		
$h = 3$	72	47	195	52	4945	64		

$\#r$: extracted I/O relations

$\#s$: services with ≥ 1 extracted relation

4.8.2.2 I/O recognition evaluation

Recognizing the I/O tokens in the text descriptions is the basis for every textual I/O relation extraction. It depends on two major factors: a) the quality of the textual description and b) the quality of the used word2vec model and its sparseness. Table 4.10a shows the results of I/O recognition in OWLS-TC using a similarity threshold = 0.5. More than 80% of all OWLS-TC inputs and more than $87,3$ of all outputs are properly recognized in the text descriptions. These results are fairly good and can be improved by adapting the I/O recognition algorithm to take into account some particular cases in OWLS-TC. We have manually checked all the services to make sure the threshold value is adequate for most if not all services.

Table 4.10b shows the absolute frequency of the most frequent cases of non-recognized I/O tokens as well as their percentage amongst the total cases of non-recognition. (509 non-recognized I/O in total) The most frequent case ($24,16\%$ of them) is the usage of compound words in the text that correspond to variable names without camelCasing (eg. maxprice, taxedprice, etc.) while the variable names themselves in the I/O URIs are written in camelCase. This requires using a more sophisticated token splitting algorithm or using an edit-distance string similarity algorithm as an alternative to word2vec simi-

Table 4.10: Evaluation of I/O recognition in textual descriptions

(a) I/O recognition results using a similarity threshold $t_s = 0.5$

	Inputs	Outputs
#	1236	1410
%	80,25	87,30

(b) Top cases of non-recognized I/O

Case	#	%
compound words	123	24,16
unmentioned I/O words	98	19,25
special separators	42	8,25

larity. Another frequent case of non-recognition $(19,25\%)$ is the absence of any mention of the I/O or their synonyms or similar words in the text which unfortunately cannot be dealt with automatically unless the text description is manually enriched beforehand. The third frequent particular case $(8,25\%)$ is the use of unusual token separators such as (/, -, +, hidden characters, etc).

4.8.2.3 Extracted I/O relations evaluation

To evaluate this last step, we run the whole process on the list of usable services mentioned above. For the evaluation of I/O relations, we manually evaluate the matching results given by our three proposed similarity calculation methods (the aggregation-based cosine similarity and the average of maximums). For this experimentation, we let the system pick the maximum similarity value amongst the three similarities.

First of all, amongst the usable services set, only 32 of them had both text and ontology relations, 5 of which didn't have any valid I/O relations in the ontologies. Valid I/O relations are ontology relations that have correct matches among text relations. The last 5 services don't have any I/O relations that are related to the text in any way. Table 4.11a shows the obtained results for the subset of usable services with both text and ontology relations. It shows that the system has extracted 19 I/O relations in total out of 32 relations, 18 of which were valid whereas 1 was invalid. Therefore, there was only 1 false positive and no false negatives.

With an F-score of 0,76 and a precision value of 94,73, our I/O relation evaluation system shows promising results (see Table 4.11b). However, the recall is still to be optimized. For the most part, the low recall value is due to the quality of text descriptions sentences that do not produce *"good"* dependency graphs upon dependency parsing. It is mainly caused by some recurrent patterns in text descriptions like "This service returns

Table 4.11: Evaluation of extracted I/O relations

(a) Validity of the extracted relations

	Extracted	Valid	Invalid	Reference
#	19	18	1	28
%	67,85	64,2	3,5	100

(b) Recall and precision

Precision	Recall	F-score
94,73	64,28	0,76

... " and by the usage of pronouns to refer to I/O words and as well as some confusing conjunctions.

4.9 Conclusion

In this chapter, we presented an approach for enriching service descriptions to remedy to the problem of ambiguous service functionality. Our enrichment approach extracts existing relations between the inputs and outputs of the services from the underlying ontologies, using SPARQL, and from the text descriptions of services, using NLP techniques. We aimed to prove the feasibility of solving this problem in a way that has not been addressed before to ht best of our knowledge. We were motivated by the recent advances in deep-learning and NLP techniques that seemed to be very promising, which was proven true in our case.

Limits and future improvements

Ontology I/O relations extraction with SPARQL quickly shows its limits beyond some relation lengths because of the important number of joins required for all the possible combinations of paths. Even tough the most relevant relations are found at shorter distances, some ontologies like life-science ontologies have a fine granularity level with longer paths for I/O relations. Applying heuristics to overcome this problem is very important to solve this issue. One possible solution is to consider Ontology profiling. Profiling data can be used to automatically generate tailored relation extraction queries specific to each ontology.

We believe that the obtained results in terms of I/O tokens recognition or in relations evaluation can be significantly improved, either by optimizing the word/sentence similarity parameters, by training other word2vec models, or by adopting other alternatives,

like GloVe[16], WordNet[17], etc.

Since OWLS-TC ontologies and services were mainly designed for evaluating service matchmaking, they lack the interlinking of ontologies and many "real-world" relations between concepts are missing. These issues reduced the amount of usable web services for our experiments. Manually interlinking OWLS-TC ontologies to real world ontologies such as DBpedia would have permitted to extract more I/O relations. We have tried this on some OWLS-TC ontologies partially on several concepts and were able to extract more relations but we haven't included this in the experiments section.

[16] http://nlp.stanford.edu/projects/glove/
[17] http://wordnet.princeton.edu/

Chapter 5

Conclusion

5.1 Contributions

In this thesis, we portrayed two separate but complementary works:

5.1.1 LIDSEARCH: Linked Data and Service Search

We presented LIDSEARCH [Mouhoub et al., 2014, 2015], a framework for searching linked data and semantic web services at the same time with a single query. LIDSEARCH introduces the following features and contributions:

1. A method to derive service queries from a data query. It involves extracting I/Os for service queries from data queries and searching for their concepts (classes) in the LOD as well as their similar concepts to expand the search area.

2. Automated service discovery of services relevant to a data query, i.e, services that provide some or all searched data to complete it, or that consume some or all searched data to propose additional data or services.

3. Service search based on ontological similarity (logic) that takes into account hierarchical similarity (sub classes) as well as equivalence (rdfs:equivalentClassOf).

4. service ranking based on textual similarity of query's I/O with textual descriptions of services using a word2vec similarity measure.

5. Expressing service search queries without prior knowledge (from users) of service description languages, repositories or ontologies used to describe I/O of services.

6. Rewriting of service queries for multiple RDF-based service descriptions including OWLS and MSM.

We validated our proposition with a proof of concept and tested it with data queries that run against DBpedia and returns relevant services from OWLS-TC. We also separately evaluated our SPARQL-driven service search algorithm using a set of service queries from the OWLS-TCv4 benchmark.

5.1.2 Service Description Enrichment

We presented a framework and a baseline towards automatically enriching service descriptions with I/O relations. I/O relations are a descriptive element that allows for a precise understanding of service functionalities. It has the following features and contributions:

1. An approach to extract I/O relations from both textual descriptions and ontologies using:

   (a) A method to extract I/O relations from ontologies using SPARQL queries

   (b) A method to extract I/O relations from text using grammatical dependency graphs and word2vec textual similarity.

2. An approach for enriching service descriptions with I/O relations from ontologies. Ontology I/O relations are selected and ranked based on their match with textual relations.

3. A method for calculating textual similarity between I/O relations using word2vec cosine similarity. It is based on aggregated embeddings for relations created from wor2vec vectors of relations' words.

5.2 Perspectives

The presented work in this thesis is mainly validated using proofs of concepts that can be improved at different levels. In the conclusions of chapters 3 and 4 we discussed the main limits of our respective frameworks and suggested some possible solutions in the future. In this section, we will present some mid-term and long-term perspectives that could bring new features and contributions.

1. **Automatic Service Composition:** One of the objectives of LIDSEARCH is to automatically create service compositions in case the service search fails to find atomic relevant services. The introduced automatic service composition algorithm has been implemented but not fully integrated within the framework. Integrating the algorithm will allow us evaluate its efficiency and performance.

2. **Coupling with SWS annotation tools** LIDSEARCH is solely dedicated to semantic web services. However, in real world and to the best of our knowledge, there are almost no (if none at all) real world web services described and published as semantic web services. There are some web APIs listed on ProgrammableWeb.com that provide results in RDF format but do not have SWS descriptions. To make LIDSEARCH operable on real world services, a middleware layer can be used to leverage existing web services and APIs with SWS descriptions using a standard

language like OWLS. For instance, works like WebKarma[1] [Taheriyan et al., 2012a] or DORIS [Koutraki et al., 2015] can be integrated with our framework in an a priori fashion to create SWS descriptions for APIs that can be published in local or remote RDF repositories. This will extend the service search to real world web APIs.

3. **Data and service search in natural language:** One of the leads that has been explored but not accomplished is to add a natural language search layer on top of LIDSEARCH. We have investigated the existing approaches and sketched a baseline approach but we have put it aside in favor of the service description enrichment framework. Adding natural language queries support makes the search easier for the user but involves many challenges and domains such as natural language to SPARQL conversion, entity recognition, text to ontology matching, etc.

4. **QoS-aware service discovery and composition** In this thesis, we didn't take into account QoS aspects which are widely considered in service discovery and composition. In the future, it would be important to take into account these aspects.

5. **Integrating I/O relations extraction in LIDSEARCH:** In LIDSEARCH, a data query can be regarded as a BGP describing a desired set of outputs that satisfy a relation with each other or with other input concepts. Currently, LIDSEARCH only considers the I/O of data queries that could be used for service requests but doesn't make a use for their relationship in service search. Therefore, integrating I/O relations extraction in LIDSEARCH could unlock the possibility to search for a service with an underlying data model that satisfies that of the data query. This copes with the added value of I/O relations.

[1] http://www.isi.edu/integration/karma/

Bibliography

Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2009). Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal?The International Journal on Very Large Data Bases*, 18(2):385–406.

Acosta, M., Vidal, M.-E., Lampo, T., Castillo, J., and Ruckhaus, E. (2011). Anapsid: An adaptive query processing engine for sparql endpoints. In *The Semantic Web–ISWC 2011*, pages 18–34. Springer.

Akar, Z., Halaç, T. G., Ekinci, E. E., and Dikenelli, O. (2012). Querying the web of interlinked datasets using void descriptions. In *LDOW*.

Angeli, G., Premkumar, M. J. J., and Manning, C. D. (2015). Leveraging linguistic structure for open domain information extraction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 344–354.

Angles, R. and Gutierrez, C. (2008). *The expressive power of SPARQL*. Springer.

Ankolekar, A., Burstein, M., Hobbs, J. R., Lassila, O., Martin, D., McDermott, D., McIlraith, S. A., Narayanan, S., Paolucci, M., Payne, T., et al. (2002). Daml-s: Web service description for the semantic web. In *The Semantic Web—ISWC 2002*, pages 348–363. Springer.

Arnold, P. and Rahm, E. (2014). Extracting semantic concept relations from wikipedia. In *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics*, WIMS '14, pages 26:1–26:11, New York, NY, USA. ACM.

Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked data-the story so far. *Intl. journal on semantic web and information systems*, 5(3):1–22.

Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300.

Bönström, V., Hinze, A., and Schweppe, H. (2003). Storing rdf as a graph. In *Web Congress, 2003. Proceedings. First Latin American*, pages 27–36. IEEE.

Booth, D. and Liu, C. K. (2007). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. Technical report, W3C. http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626 - Ext. on Nov. 2015.

Bornea, M. A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., and Bhattacharjee, B. (2013). Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM.

Brickley, D. and Guha, R. (2014). RDF schema 1.1. W3C recommendation, W3C. http://www.w3.org/TR/rdf-schema/.

Broekstra, J., Kampman, A., and Van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying rdf and rdf schema. In *The Semantic Web—ISWC 2002*, pages 54–68. Springer.

Bülthoff, F. and Maleshkova, M. (2014). Restful or restless - current state of today's top web apis. In *11th ESWC 2014 (ESWC2014)*.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83. ACM.

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Chen, F., Lu, C., Wu, H., and Li, M. (2017). A semantic similarity measure integrating multiple conceptual relationships for web service discovery. *Expert Systems with Applications*, 67:19–31.

Chen, J., Feng, Z., Chen, S., Huang, K., Tan, W., and Zhang, J. (2015). A novel lifecycle framework for semantic web service annotation assessment and optimization. In *IEEE Int. Conf. on Web Services (ICWS)*, pages 361–368. IEEE.

Cheniki, N., Belkhir, A., and Atif, Y. (2015). Supporting multilingual semantic web services discovery by consuming data from dbpedia knowledge base. In *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, page 66. ACM.

Cheniki, N., Belkhir, A., and Atif, Y. (2016). Mobile services discovery framework using DBpedia and non-monotonic rules. *Computers & Electrical Engineering*, 52:49–64.

Choi, H., Son, J., Cho, Y., Sung, M. K., and Chung, Y. D. (2009). Spider: a system for scalable, parallel/distributed evaluation of large-scale rdf data. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 2087–2088. ACM.

Chong, E. I., Das, S., Eadon, G., and Srinivasan, J. (2005). An efficient sql-based rdf querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227. VLDB Endowment.

Damljanovic, D., Agatonovic, M., and Cunningham, H. (2012). Freya: An interactive way of querying linked data using natural language. In *The Semantic Web: ESWC 2011 Workshops*, pages 125–138. Springer.

De Marneffe, M.-C. and Manning, C. D. (2008). Stanford typed dependencies manual. Technical report, Stanford University.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Di Modica, G., Tomarchio, O., and Vita, L. (2011). Resource and service discovery in soas: A p2p oriented semantic approach. *International Journal of Applied Mathematics and Computer Science*, 21(2):285–294.

Dietze, S., Yu, H. Q., Pedrinaci, C., Liu, D., and Domingue, J. (2011). Smartlink: a web-based editor and search environment for linked services. In *The Semanic Web: Research and Applications*, pages 436–440. Springer.

Domingue, J., Fensel, D., and Hendler, J. A. (2011). *Introduction to the Semantic Web Technologies*, pages 1–41. Springer Berlin Heidelberg, Berlin, Heidelberg.

Du, J.-H., Wang, H.-F., Ni, Y., and Yu, Y. (2012). Hadooprdf: A scalable semantic data analytical engine. *Intelligent Computing Theories and Applications*, pages 633–641.

Elbassuoni, S., Ramanath, M., Schenkel, R., and Weikum, G. (2010). Searching rdf graphs with sparql and keywords. *IEEE Data Eng. Bull.*, 33(1):16–24.

Fensel, D., Facca, F. M., Simperl, E., and Toma, I. (2011). *Semantic web services*. Springer Science & Business Media.

Ferré, S. (2013). squall2sparql: a translator from controlled english to full sparql 1.1. In *Work. Multilingual Question Answering over Linked Data (QALD-3)*.

Freitas, A., Curry, E., Oliveira, J. G., and O'Riain, S. (2012). Querying heterogeneous datasets on the linked data web: Challenges, approaches, and trends. *Internet Computing, IEEE*, 16(1):24–33.

Galárraga, L., Hose, K., and Schenkel, R. (2012). Partout: A distributed engine for efficient rdf processing. *arXiv preprint arXiv:1212.5636*.

GarcíA, J. M., Ruiz, D., and Ruiz-CortéS, A. (2012). Improving semantic web services discovery using sparql-based repository filtering. *Web Semant.*, 17:12–24.

Goasdoué, F., Kaoudi, Z., Manolescu, I., Quiané-Ruiz, J., and Zampetakis, S. (2013). Cliquesquare: efficient hadoop-based rdf query processing. In *BDA'13-Journées de Bases de Données Avancées*.

Görlitz, O. and Staab, S. (2011). Splendid: Sparql endpoint federation exploiting void descriptions. *COLD*, 782.

Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.-U., and Umbrich, J. (2010). Data summaries for on-demand queries over linked data. In *Proceedings of the 19th international conference on World wide web*, pages 411–420. ACM.

Harth, A., Hose, K., and Schenkel, R. (2012). Database techniques for linked data management. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 597–600. ACM.

Harth, A., Knoblock, C., Stadtmüller, S., Studer, R., and Szekely, P. (2013). On-the-fly integration of static and dynamic sources. In *Proceedings of the ISWC 2013 workshop on Consuming Linked Data*. CEUR-WS.

Hartig, O. (2011). Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *The Semantic Web: Research and Applications*, pages 154–169. Springer.

Hatzi, O., Vrakas, D., Nikolaidou, M., Bassiliades, N., Anagnostopoulos, D., and Vlahavas, L. (2012). An integrated approach to automated semantic web service composition through planning. *Services Computing, IEEE Transactions on*, 5(3):319–332.

He, Q., Yan, J., Yang, Y., Kowalczyk, R., and Jin, H. (2013). A decentralized service discovery approach on peer-to-peer networks. *Services Computing, IEEE Transactions on*, 6(1):64–75.

Heß, A., Johnston, E., and Kushmerick, N. (2004). Assam: A tool for semi-automatically annotating semantic web services. In *International Semantic Web Conference*, volume 3298, pages 320–334. Springer.

Hitzler, P., Krötzsch, M., Rudolph, S., Parsia, B., and Patel-Schneider, P. (2012). OWL 2 web ontology language primer (second edition). Technical report, W3C. http://www.w3.org/TR/owl2-primer/.

Husain, M., McGlothlin, J., Masud, M. M., Khan, L., and Thuraisingham, B. M. (2011). Heuristics-based query processing for large rdf graphs using cloud computing. *Knowledge and Data Engineering, IEEE Transactions on*, 23(9):1312–1327.

Jaeger, M. C., Rojec-Goldmann, G., Liebetruth, C., Mühl, G., and Geihs, K. (2005). Ranked matching for service descriptions using owl-s. In *Kommunikation in Verteilten Systemen (KiVS)*, pages 91–102. Springer.

Kaoudi, Z., Manolescu, I., et al. (2013). Triples in the clouds. In *ICDE-29th International Conference on Data Engineering*.

Kiefer, C. and Bernstein, A. (2008). *The creation and evaluation of isparql strategies for matchmaking*. Springer.

Klein, M. and König-Ries, B. (2004). Coupled signature and specification matching for automatic service binding. In *Web Services*, pages 183–197. Springer.

Klusch, M. (2008). Semantic web service description. In *CASCOM: Intelligent Service Coordination in the Semantic Web*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 31–57. Birkhäuser Basel.

Klusch, M. (2014). Service discovery. In *Encyclopedia of Social Network Analysis and Mining*, pages 1707–1717. Springer.

Klusch, M. and Kapahnke, P. (2009). Owls-mx3: an adaptive hybrid semantic service matchmaker for owl-s. In *Proceedings of 3rd International Workshop on Semantic Matchmaking and Resource Retrieval (SMR2), USA*.

Klusch, M. and Kapahnke, P. (2012). The isem matchmaker: A flexible approach for adaptive hybrid semantic service selection. *Web Semantics: Science, Services and Agents on the World Wide Web*, 15:1–14.

Klusch, M., Kapahnke, P., Schulte, S., Lecue, F., and Bernstein, A. (2016). Semantic web service search: a brief survey. *KI-Künstliche Intelligenz*, 30(2):139–147.

Kong, W. and Allan, J. (2013). Extracting query facets from search results. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 93–102. ACM.

Kopecky, J., Gomadam, K., and Vitvar, T. (2008). hrests: An html microformat for describing restful web services. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM Intl. Conf. on*, volume 1, pages 619–625. IEEE.

Koutraki, M., Vodislav, D., and Preda, N. (2015). Doris: discovering ontological relations in services. In *The 14th International Semantic Web Conference*.

Kritikos, K., Pernici, B., Plebani, P., Cappiello, C., Comuzzi, M., Benrernou, S., Brandic, I., Kertész, A., Parkin, M., and Carro, M. (2013). A survey on service quality description. *ACM Computing Surveys (CSUR)*, 46(1):1.

Küngas, P. and Matskin, M. (2005). Semantic web service composition through a p2p-based multi-agent environment. In *Agents and Peer-to-Peer Computing*, pages 106–119. Springer.

Ladwig, G. and Tran, T. (2011). Sihjoin: querying remote and local linked data. In *The Semantic Web: Research and Applications*, pages 139–153. Springer.

Langegger, A., Wöß, W., and Blöchl, M. (2008). A semantic web middleware for virtual data integration on the web. In *The Semantic Web: Research and Applications*, pages 493–507. Springer.

Le-Phuoc, D., Polleres, A., Hauswirth, M., Tummarello, G., and Morbidoni, C. (2009). Rapid prototyping of semantic mash-ups through semantic web pipes. In *Proceedings of the 18th international conference on World wide web*, pages 581–590. ACM.

Lécué, F. and Léger, A. (2006). A formal model for semantic web service composition. In *The Semantic Web-ISWC 2006*, pages 385–398. Springer.

Lécué, F., Silva, E., and Pires, L. F. (2008). A framework for dynamic web services composition. In *Emerging Web Services Technology, Volume II*, pages 59–75. Springer.

Lee, Y.-J. (2013). Algorithm for automatic web api composition. In *WEB 2013, The First International Conference on Building and Exploring Web Based Environments*, pages 57–62.

Lemos, A. L., Daniel, F., and Benatallah, B. (2016). Web service composition: a survey of techniques and tools. *ACM Computing Surveys (CSUR)*, 48(3):33.

Levandoski, J. J. and Mokbel, M. F. (2009). Rdf data-centric storage. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 911–918. IEEE.

Liu, D., Li, N., Pedrinaci, C., Kopeckỳ, J., Maleshkova, M., and Domingue, J. (2011). An approach to construct dynamic service mashups using lightweight semantics. In *Current Trends in Web Engineering*, pages 13–24. Springer.

Lopez, V., Fernández, M., Motta, E., and Stieler, N. (2012). Poweraqua: Supporting users in querying and exploring the semantic web. *Semantic Web*, 3(3):249–265.

Louati, A., El Haddad, J., and Pinson, S. (2016). A multi-agent approach for trust-based service discovery and selection in social networks. *Scalable Computing: Practice and Experience*, 16(4):381–402.

Lucky, M. N., Cremaschi, M., Lodigiani, B., Menolascina, A., and De Paoli, F. (2016). Enriching API Descriptions by Adding API Profiles Through Semantic Annotation. In *Int. Conf. on Service-Oriented Computing (ICSOC)*, pages 780–794. Springer.

Lynden, S., Kojima, I., Matono, A., and Tanimura, Y. (2011). Aderis: An adaptive query processor for joining federated sparql endpoints. In *On the Move to Meaningful Internet Systems: OTM 2011*, pages 808–817. Springer.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J. R., Bethard, S., and McClosky, D. (2014). The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., et al. (2004). OWL-S: Semantic markup for web services. *W3C member submission*, 22:2007–04.

Masuch, N., Hirsch, B., Burkhardt, M., Heßler, A., and Albayrak, S. (2012). Sema2: a hybrid semantic service matching approach. In *Semantic web services*, pages 35–47. Springer.

McCabe, F., Ferris, C., Champion, M., Newcomer, E., Haas, H., Booth, D., and Orchard, D. (2004). Web services architecture. W3C note, W3C. https://www.w3.org/TR/ws-arch/.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Miller, G. A. (1995). Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41.

Motik, B., Patel-Schneider, P., and Parsia, B. (2012). OWL 2 web ontology language structural specification and functional-style syntax (second edition). W3C recommendation, W3C. http://www.w3.org/TR/owl2-syntax/.

Mouhoub, M., Grigori, D., and Manouvrier, M. (2014). A framework for searching semantic data and services with sparql. In *Service-Oriented Computing*, volume 8831 of *Lecture Notes in Computer Science*, pages 123–138. Springer Berlin Heidelberg.

Mouhoub, M. L., Grigori, D., and Manouvrier, M. (2015). Lidsearch: A sparql-driven framework for searching linked data and semantic web services. In *The Semantic Web: ESWC 2015 Satellite Events*, pages 112–117. Springer.

Mouhoub, M. L., Grigori, D., and Manouvrier, M. (2017). *Towards an Automatic Enrichment of Semantic Web Services Descriptions*. Springer International Publishing.

Mukhopadhyay, D. and Chougule, A. (2012). A survey on web service discovery approaches. In *Advances in Computer Science, Engineering & Applications*, pages 1001–1012. Springer.

Naacke, H., Amann, B., and Curé, O. (2017). Sparql graph pattern processing with apache spark. In *GRADES (Graph Data-management Experiences & Systems), Workshop, SIGMOD 2017*.

Nachouki, G. and Quafafou, M. (2011). Mashup web data sources and services based on semantic queries. *Information Systems*, 36(2):151–173.

Nakashole, N., Weikum, G., and Suchanek, F. (2012). Patty: A taxonomy of relational patterns with semantic types. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, pages 1135–1145, Stroudsburg, PA, USA. Association for Computational Linguistics.

Neumann, T. and Weikum, G. (2008). Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659.

Ngan, L. D. and Kanagasabai, R. (2013a). Semantic web service discovery: state-of-the-art and research challenges. *Personal and ubiquitous computing*, 17(8):1741–1752.

Ngan, L. D. and Kanagasabai, R. (2013b). Semantic Web service discovery: state-of-the-art and research challenges. *Personal and ubiquitous computing*, 17(8):1741–1752.

Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., and Tummarello, G. (2008). Sindice. com: a document-oriented lookup index for open linked data. *International Journal of Metadata, Semantics and Ontologies*, 3(1):37–52.

Özsu, M. T. (2016). A survey of rdf data management systems. *Frontiers of Computer Science*, 10(3):418–432.

Palathingal, P. and Chandra, S. (2004). Agent approach for service discovery and utilization. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 9–pp. IEEE.

Palmonari, M., Sala, A., Maurino, A., Guerra, F., Pasi, G., and Frisoni, G. (2011). Aggregated search of data and services. *Information Systems*, 36(2):134 – 150. Special Issue: Semantic Integration of Data, Multimedia, and Services.

Papailiou, N., Konstantinou, I., Tsoumakos, D., and Koziris, N. (2012). H2rdf: adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 397–400. ACM.

Patil, A. A., Oundhakar, S. A., Sheth, A. P., and Verma, K. (2004). Meteor-s web service annotation framework. In *Proceedings of the 13th international conference on World Wide Web*, pages 553–562. ACM.

Pedrinaci, C., Cardoso, J., and Leidig, T. (2014). Linked usdl: a vocabulary for web-scale service trading. In *The Semantic Web: Trends and Challenges*, pages 68–82. Springer.

Pedrinaci, C. and Domingue, J. (2010). Toward the next wave of services: Linked services for the web of data. *J. ucs*, 16(13):1694–1719.

Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J., and Domingue, J. (2010). iserve: a linked services publishing platform. In *Ontology Repositories and Editors for the Semantic Web Workshop at The 7th Extended Semantic Web*, volume 596.

Preda, N., Suchanek, F. M., Kasneci, G., Neumann, T., Ramanath, M., and Weikum, G. (2009). Angie: Active knowledge for interactive exploration. *Proc. of the VLDB Endowment*, 2(2):1570–1573.

Preist, C. (2004). A conceptual architecture for semantic web services. *The Semantic Web–ISWC 2004*, pages 395–409.

Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL Query Language for RDF. Technical report, W3C. `https://www.w3.org/TR/rdf-sparql-query/` - Ext. on June 2017.

Quilitz, B. and Leser, U. (2008). Querying distributed rdf data sources with sparql. In *The Semantic Web: Research and Applications*, pages 524–538. Springer.

Raimond, Y. and Schreiber, G. (2014). RDF 1.1 primer. W3C note, W3C. http://www.w3.org/TR/rdf11-primer/.

Rakhmawati, N. A., Umbrich, J., Karnstedt, M., Hasnain, A., and Hausenblas, M. (2013). A comparison of federation over sparql endpoints frameworks. In *Knowledge Engineering and the Semantic Web*, pages 132–146. Springer.

Rao, J., Dimitrov, D., Hofmann, P., and Sadeh, N. (2006). A mixed initiative semantic web framework for process composition. In *The Semantic Web-ISWC 2006*, pages 873–886. Springer.

Rao, J. and Su, X. (2004). A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, pages 43–54. Springer.

Rios, L. H. O. and Chaimowicz, L. (2011). Pnba*: A parallel bidirectional heuristic search algorithm.

Rodriguez-Mier, P., Mucientes, M., Vidal, J. C., and Lama, M. (2012). An optimal and complete algorithm for automatic web service composition. *Intl. Journal of Web Services Research (IJWSR)*, 9(2):1–20.

Rodriguez Mier, P., Pedrinaci, C., Lama, M., and Mucientes, M. (2015). An integrated semantic web service discovery and composition framework. *CoRR*, abs/1502.02840.

Rohloff, K. and Schantz, R. E. (2010). High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In *Programming support innovations for emerging distributed applications*, page 4. ACM.

Sakr, S. and Al-Naymat, G. (2010). Relational processing of rdf queries: a survey. *ACM SIGMOD Record*, 38(4):23–28.

Saleem, M., Khan, Y., Hasnain, A., Ermilov, I., and Ngonga Ngomo, A.-C. (2015). A fine-grained evaluation of sparql endpoint federation systems. *Semantic Web*, (Preprint):1–26.

Saleem, M. and Ngonga Ngomo, A.-C. (2014). Hibiscus: Hypergraph-based source selection for sparql endpoint federation. In *The Semantic Web: Trends and Challenges*, volume 8465 of *Lecture Notes in Computer Science*, pages 176–191. Springer International Publishing.

Santorini, B. (1990). Part-Of-Speech tagging guidelines for the Penn Treebank project (3rd revision, 2nd printing). Technical report, Department of Linguistics, University of Pennsylvania, Philadelphia, PA, USA. Available at: http://repository.upenn.edu/cgi/viewcontent.cgi?article=1603&context=cis_reports.

Sbodio, M. L. (2012). Sparqlent: a sparql based intelligent agent performing service matchmaking. In *Semantic Web Services*, pages 83–105. Springer.

Sbodio, M. L., Martin, D., and Moulin, C. (2010). Discovering Semantic Web services using SPARQL and intelligent agents. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):310–328.

Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011). Fedx: Optimization techniques for federated query processing on linked data. In *Intl. Semantic Web Conf. (1)*, pages 601–616.

Seaborne, A. and Harris, S. (2013). SPARQL 1.1 query language. W3C recommendation, W3C. http://www.w3.org/TR/sparql11-query/.

Sheng, Q. Z., Qiao, X., Vasilakos, A. V., Szabo, C., Bourne, S., and Xu, X. (2014). Web services composition: A decade's overview. *Information Sciences*, 280:218–238.

Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., and Manegold, S. (2008). Column-store support for rdf data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563.

Speiser, S. and Harth, A. (2011a). Integrating linked data and services with linked data services. In *Proc. of the 8th Extended Semantic Web Conf. on The Semantic Web: Research and Applications - Volume Part I*, ESWC'11, pages 170–184, Berlin, Heidelberg. Springer-Verlag.

Speiser, S. and Harth, A. (2011b). Integrating linked data and services with linked data services. In *Proceedings of the 8th Extended Semantic Web Conference on The Semantic Web: Research and Applications - Volume Part I*, ESWC'11, pages 170–184, Berlin, Heidelberg. Springer-Verlag.

Stollberg, M., Hepp, M., and Hoffmann, J. (2007). A caching mechanism for semantic web service discovery. In *Proceedings of the 6th International The Semantic Web and 2Nd Asian Conference on Asian Semantic Web Conference*, pages 480–493. Springer-Verlag.

Straus, J., Kaufman, L., and Stern, T. (2014). *The Blue Book of Grammar and Punctuation: An Easy-to-Use Guide with Clear Rules, Real-World Examples, and Reproducible Quizzes*. Wiley, 11 edition.

Studer, R., Grimm, S., and Abecker, A. (2007). *Semantic Web Services: Concepts, Technologies, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Suchanek, F. M., Kasneci, G., and Weikum, G. (2007). Yago: a core of semantic knowledge. In *Proc. of the 16th Intl. conf. on World Wide Web*, pages 697–706. ACM.

Sycara, K., Paolucci, M., Ankolekar, A., and Srinivasan, N. (2003). Automated discovery, interaction and composition of semantic web services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):27–46.

Syu, Y., Ma, S.-P., Kuo, J.-Y., and FanJiang, Y.-Y. (2012). A survey on automated service composition methods and related techniques. In *Services Computing (SCC), 2012 IEEE Ninth Intl. Conf. on*, pages 290–297.

Taheriyan, M., Knoblock, C., Szekely, P., and Ambite, J. (2012a). Rapidly integrating services into the linked data cloud. In Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J., Hendler, J., Schreiber, G., Bernstein, A., and Blomqvist, E., editors, *The Semantic Web – ISWC 2012*, volume 7649 of *Lecture Notes in Computer Science*, pages 559–574. Springer Berlin Heidelberg.

Taheriyan, M., Knoblock, C. A., Szekely, P., and Ambite, J. L. (2012b). Rapidly Integrating Services into the Linked Data Cloud. In *Proceedings of the 11th International Semantic Web Conference (ISWC 2012)*.

Tesnière, L. (1959). *Elements de syntaxe structurale*. Editions Klincksieck.

Thiagarajan, R., Mayer, W., and Stumptner, M. (2009). *Semantic service discovery by consistency-based matchmaking*. Springer.

Tosi, D. and Morasca, S. (2015). Supporting the semi-automatic semantic annotation of web services: A systematic literature review. *Information and Software Technology*, 61:16–32.

Umbrich, J., Hose, K., Karnstedt, M., Harth, A., and Polleres, A. (2011). Comparing data summaries for processing live queries over linked data. *World Wide Web*, 14(5-6):495–544.

Wei, D., Wang, T., Wang, J., and Bernstein, A. (2011). Sawsdl-imatcher: A customizable and effective semantic web service matchmaker. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9(4):402–417.

Weiss, C., Karras, P., and Bernstein, A. (2008). Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019.

Wilkinson, K. and Wilkinson, K. (2006). Jena property table implementation.

Yan, Y., Xu, B., and Gu, Z. (2008). Automatic service composition using and/or graph. In *E-Commerce Technology and the Fifth IEEE Conf. on Enterprise Computing, E-Commerce and E-Services, 2008 10th IEEE Conf. on*, pages 335–338. IEEE.

Zeshan, F. and Mohamad, R. (2011). Semantic web service composition approaches: overview and limitations. *International Journal of New Computer Architectures and their Applications (IJNCAA)*, 1(3):640–651.

Zhang, L., Liu, Q., Zhang, J., Wang, H., Pan, Y., and Yu, Y. (2007). *Semplore: an IR approach to scalable hybrid query of semantic web data*. Springer.

Zhang, Z., Chen, S., and Feng, Z. (2013). Semantic annotation for web services based on DBpedia. In *IEEE 7th Int. Symp. on Service Oriented System Engineering (SOSE)*, pages 280–285.

Zou, L., Mo, J., Chen, L., Özsu, M. T., and Zhao, D. (2011). gstore: answering sparql queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493.

Zou, L., Özsu, M. T., Chen, L., Shen, X., Huang, R., and Zhao, D. (2014). gstore: a graph-based sparql query engine. *The VLDB journal*, 23(4):565–590.

Zunino, A. and Campo, M. (2012). A Survey of Approaches to Web Service Discovery in Service—Oriented Architectures. *Innovations in Database Design, Web Applications, and Information Systems Management*, 107(1).

List of Figures

List of Tables

Résumé Étendu

Introduction

À l'ère de l'internet, les données sont devenues l'unes des ressources les plus importantes du 21ème siècle de même que ce qu'était le pétrole au siècle dernier. Les données sont présentes partout mais surtout sur internet sous différentes formes : structurées, semi-structurées, non structurées, ouvertes, propriétaires, statiques, à la demande, etc.

Les données que l'on trouve sur internet sont souvent destinées à être lues et interprétées par l'homme. Grâce à l'avènement du web sémantique et du paradigme des données liées, les machines sont devenues maintenant capables d'explorer et d'interpréter de manière précise les données.

L'adoption à grande échelle des principes des données liées a donné lieu au Linked Open Data Cloud (LOD), un vaste réseau de jeux de données liées publiés sous licences ouvertes. Il comprend entre autres des données du secteur public publiées par plusieurs initiatives gouvernementales, des données scientifiques facilitant la collaboration, des données linguistiques, des données géographiques, des publications académiques telles que DBLP, données pluridisciplinaires comme DBpedia, Freebase, YAGO, etc. Le LOD contient environ 150 milliards de triplets RDF (LODstats2). Au cours des trois dernières années (2014-2017), le nombre de jeux de données liées a quasiment doublé.

Cependant, il y a encore beaucoup de données qui ne sont et qui ne seront probablement pas publiées en tant que dépôts de données sur le LOD telles que :

- Les données dynamiques issues des capteurs et d'objets connectés.
- Les données calculées à la demande en fonction d'un certain nombre de paramètres d'entrée. Par exemple, l'itinéraire en transports publics le plus rapide entre deux points étant donnés en entrée ces deux points.
- Les données avec des modes d'accès restreints. Par exemple les prix des hôtels qui peuvent varier en fonction des différentes politiques de tarification (nouveaux clients, clients fidèles, agences, brokers, etc.).

Ces données sont généralement disponibles via des API Web ou des Services Web qui sont de plus en plus abondants aujourd'hui. Néanmoins, les machines sensées manipuler

des données sémantiques doivent par elles-mêmes chercher et utiliser les services web qui leur sont utiles. Afin de permettre la découverte et la composition automatique des services web, les travaux de recherche dans le domaine du web sémantique proposent de décrire ces derniers par le web sémantique lui-même, i.e avec des annotations sémantiques. Les services de ce type s'appellent les Services Web Sémantiques (SWS).

Motivations

L'intégration des données du Linked Data Cloud et des Services Web Sémantiques (SWS) offre de grandes facilités pour créer de mashups et de composer automatiquement des services web. De plus, une telle intégration permet de résoudre certains problèmes existants sur le Linked Data Cloud qui affectent la qualité de ses données tels que :

- **Données manquantes** : Certaines entités n'existent pas encore sur le LOD. Par exemple, les listes de tous les films d'un réalisateur ou les livres d'un auteur donné ne sont pas complètes sur le LOD car beaucoup d'entre eux n'ont pas encore de pages dédiées wikipedia / dbpedia. Cependant, pour cet exemple, de telles données peuvent être facilement trouvées en utilisant les APIs d'Amazon, IMDB, etc.

- **Données incomplètes** : Certaines informations sur une entité (certains de ses attributs) peuvent être manquantes sur le LOD. Par exemple, les informations sur les prix des livres, les magasins qui les vendent, les cinémas dans lesquels les films sont projetés, etc. sont des informations qui n'existent pas sur le LOD.

- **Données inexistantes** : Certains types ou catégories de données n'existe tout simplement pas sur le LOD en tant que données sémantiques. Par exemple, les annuaires d'entreprises, les données des réseaux sociaux, les tweets, les données dynamiques, les données à la demande, etc.

- **Données obsolètes** : Les données du LOD peuvent rapidement devenir obsolètes en fonction de leur type, comme les statistiques, les prix, etc. Par exemple, les statistiques sur la population et les prix des actions des entreprises ne sont pas mises à jour pour de nombreuses entrées dans DBpedia.

Cependant, les services qui seraient appropriés pour résoudre les problèmes de qualité des données mentionnés ci-dessus doivent être d'abord découverts, et dans le cas où ils

n'existeraient pas, ils doivent être composés à partir de services atomiques. Pour atteindre un tel objectif, un développeur souhaitant intégrer des données et des services devrait :

- Connaitre les annuaires de SWS existants sur le web,
- Connaitre les langages et formalismes de description SWS hétérogènes,
- Exprimer ses besoins en termes de vocabulaire utilisé par différents annuaires,
- Trouver des services pertinents à partir de différents annuaires et utiliser les outils de composition de services au cas où aucun service atomique satisfaisant n'existe.

Ce processus manuel nécessite beaucoup de connaissances préalables et d'efforts de la part de l'utilisateur. Dans cette thèse, nous visons à fournir un framework permettant à fois de chercher des données ainsi que des services web pouvant apporter une valeur ajoutée aux données.

Cette thèse est inspirée par le travail de recherche de [Palmonari et al., 2011] qui décrit une approche de recherche de données et de services en utilisant un langage d'interrogation de type SQL et en utilisant des services web non sémantiques avec des annotations sémantiques.

Contributions

Partie 1/2 : LIDSEARCH, recherche agrégée de données et de services

LIDSEARCH (LInked Data and Service Search) ([Mouhoub et al., 2014], [Mouhoub et al., 2015]) est un Framework piloté par SPARQL pour rechercher à la fois des données liées ainsi que des services web sémantiques complémentaires pour les données sur le LOD, le tout avec une seule et même requête pour les deux types recherche.

Pour mieux exprimer ses objectifs et son fonctionnement, considérons le scénario suivant : Un utilisateur veut connaître tous les écrivains nés à Paris ainsi que tous leurs livres. Cette requête destinée à chercher ces données sémantiques est écrite en SPARQL. Les réponses à cette requête depuis le LOD pourrait éventuellement contenir tous les écrivains parisiens dans DBpedia. Cependant, leurs livres publiés ne sont pas tous répertoriés dans DBpedia. Dans ce cas, les résultats sont incomplets et peuvent

être complétés avec une liste de services tels que l'API Amazon ou l'API Google Books qui puissent enrichir ces résultats. Certaines de ces API peuvent également fournir des informations complémentaires sur les livres, par exemple leurs prix, les amis qui les ont lus, les librairies proches qui les vendent, etc. Cependant, si l'utilisateur veut comparer les prix d'un livre dans plusieurs magasins et qu'il existe un comparateur de prix qui prend uniquement un numéro ISBN en entrée, alors dans ce cas, une composition de services peut être faite pour permettre l'interrogation de ce service et de répondre à la requête de l'utilisateur.

L'objectif du framework LIDSEARCH est d'étendre la recherche de données sémantiques par une découverte / composition de services utiles afin de permettre de trouver des services pertinents fournissant des données complémentaires. Une telle recherche nécessite souvent des requêtes distinctes: a) une ou plusieurs requêtes de données pour une recherche dans le LOD b) plusieurs requêtes de services pour découvrir des services pertinents dans des annuaires de services web sémantiques (SWS) et c) des requêtes de composition de services pour créer des compositions de service pertinentes dans le cas où aucun service répondant aux critères n'est trouvé. Notre framework recherche à la fois les données et les services à partir d'une seule requête de l'utilisateur appelée requête de données, c'est-à-dire une requête destinée à rechercher uniquement des données. À partir de cette requête, il émet automatiquement des requêtes de service et trouve des services pertinents ou génère des compositions de services.

LIDSEARCH présente les fonctionalités et les nouveautés suivantes:

1. Une méthode pour dériver des requêtes de services à partir d'une requête de données. Il s'agit d'extraire les entrées et sorties (E/S) pour les requêtes de service à partir de requêtes de données et de rechercher leurs concepts (classes ontologiques) dans le LOD ainsi que leurs concepts similaires pour étendre la recherche.

2. La découverte automatisée de services pertinents pour une requête de recherche de données, c'est-à-dire des services qui fournissent une partie ou la totalité des données recherchées pour la compléter, ou qui consomment tout ou partie des données recherchées pour proposer des données ou services supplémentaires.

3. Une recherche de services basée sur la similarité ontologique (logique) qui prend en compte la similarité hiérarchique (sous-classes) ainsi que l'équivalence (rdfs: equivalentClassOf).

4. Un classement des services trouvés basé sur les similarités textuelles entre les E/S de la requête et les descriptions textuelles des services en utilisant une mesure de similarité basée sur *word2vec*.

5. L'expression des requêtes de services, sans connaissance préalable de la part des utilisateurs, des langages, annuaires ou ontologies de description de services utilisés pour décrire les E/S des services recherchés.

6. La réécriture de requêtes de services pour plusieurs descriptions de services RDF, y compris OWLS et MSM.

Nous validons notre proposition avec une preuve de concept que nous testons avec des requêtes de données qui s'exécutent sur DBpedia et qui retournent des services pertinents parmi ceux du benchmark OWLS-TC. Nous évaluons également séparément notre algorithme de recherche de services piloté par SPARQL à l'aide d'un ensemble de requêtes de services issues du benchmark OWLS-TCv4.

Partie 2/2 : Enrichissement sémantique des descriptions de services

Dans cette partie, nous présentons une approche qui vise à faciliter la découverte automatique de services en enrichissant leurs descriptions sémantiques avec des informations détaillées sur leur fonctionnalité en utilisant les relations ontologiques entre leurs E/S. Ces dernières sont confirmées par les relations trouvées dans les descriptions textuelles en utilisant des techniques de traitement automatique du langage naturel.

Cette approche est une étape vers l'automatisation de la découverte des services qui présenterait de nombreux avantages dans différents domaines: a) améliorer la précision de la découverte automatique de services; b) faciliter la composition automatique de services et le remplacement automatique des services en panne ou non-opérationnels dans les compositions de services; c) faciliter l'intégration des données sémantiques provenant de services web, d) faciliter la création de mashups API, e) améliorer la recommandation de service, etc.

Afin d'enrichir la description d'un service donné, notre système extrait parallèlement deux types de relations entre les E/S. Les premières sont les relations formelles existantes entre les E/S dans leurs ontologies sous-jacentes et qui puissent être trouvées en utilisant SPARQL. L'autre type sont les relations informelles entre les E/S décrites en langage

naturel dans les descriptions textuelles et la documentation des services. Ces dernières nécessitent l'utilisation de techniques d'extraction basées sur le traitement automatique de la langue. Les deux processus d'extraction sont indépendants et sont exécutés en parallèle. Enfin, les deux types de relations extraites sont appariés dans l'objectif d'endosser les relations ontologiques les plus pertinentes (car des centaines de relations peuvent exister entre les concepts d'E/S dans les ontologies). Les relations ontologiques qui ont les meilleures corrélations dans le texte sont considérées commodes pour enrichir la description du service. Elles peuvent ensuite être ajoutées à la description sémantique du service après la validation de l'utilisateur.

Cette seconde partie de la thèse apporte les fonctionnalités et contributions suivantes :

1. Une approche pour extraire des relations entre E/S à partir de descriptions textuelles et ontologiques en utilisant :
   (a) Une méthode pour extraire les relations entre E/S à partir des ontologies à l'aide de requêtes SPARQL.
   (b) Une méthode pour extraire des relations entre E/S à partir du texte en utilisant des graphes de dépendance grammaticaux ainsi qu'une mesure de similarité textuelle basée sur word2vec.
2. Une approche pour enrichir les descriptions de services avec les relations d'E/S ontologiques. Ces dernières sont sélectionnées et classées en fonction de leur correspondance avec les relations textuelles.
3. Une méthode pour calculer la similarité textuelle entre les relations d'E/S en utilisant la similarité cosine de word2vec. Elle est basée sur des représentations agrégées pour les relations créées à partir des vecteurs wor2vec des mots de ces relations.

Plan de la thèse

Dans le 2ème chapitre, nous rappelons brièvement quelques concepts, définitions et standards liés à notre travail, y compris le web sémantique, les services Web sémantiques et le traitement automatique du langage naturel. Après cela, nous présentons l'état de l'art dans les domaines mentionnés ci-dessus, en particulier les travaux récents liés à notre travail ou les travaux qui ont le plus inspiré le nôtre.

Dans le 3ème chapitre, nous présentons la première partie de notre framework pour la recherche de données et de services (LIDSEARCH). Nous présentons ses fonctionnalités, nous décrivons pas à pas ses processus sous-jacents de fonctionnement et nous expliquons chaque étape par l'exemple. Nous dédions une section à notre approche esquissée pour une composition de service automatique qui n'est pas entièrement implémentée dans LIDSEARCH mais qui mérite d'être présentée. Nous présentons également nos résultats d'expérimentation sur les services du benchmark OWLS-TC et discutons les limites et des améliorations possibles de LIDSEARCH.

Le chapitre 4 est consacré à notre travail le plus récent qui vise à faciliter la découverte automatique de services en enrichissant sa description formelle. Nous expliquons également le processus sous-jacent étape par étape par l'exemple.

Dans le dernier chapitre, nous rappelons nos principales contributions et les défis auxquels nous étions confrontés. Nous discutons également quelques idées en perspectives comprenant une combinaison potentielle de nos deux travaux principaux.

Résumé

Ces dernières années ont témoigné du succès du projet Linked Open Data (LOD) et de la croissance du nombre de sources de données sémantiques disponibles sur le web. Cependant, il y a encore beaucoup de données qui ne sont pas encore mises à disposition dans le LOD telles que les données sur demande, les données de capteurs etc. Elles sont néanmoins fournies par des API des services Web. L'intégration de ces données au LOD ou dans des applications de mashups apporterait une forte valeur ajoutée. Cependant, chercher de tels services avec les outils de découverte de services existants nécessite une connaissance préalable des répertoires de services ainsi que des ontologies utilisées pour les décrire.

Dans cette thèse, nous proposons de nouvelles approches et des cadres logiciels pour la recherche de services web sémantiques avec une perspective d'intégration de données. Premièrement, nous introduisons LIDSEARCH, un cadre applicatif piloté par SPARQL pour chercher des données et des services web sémantiques.

De plus, nous proposons une approche pour enrichir les descriptions sémantiques de services web en décrivant les relations ontologiques entre leurs entrées et leurs sorties afin de faciliter l'automatisation de la découverte et de la composition de services. Afin d'atteindre ce but, nous utilisons des techniques de traitement automatique de la langue et d'appariement de textes basées sur le deep-learning pour mieux comprendre les descriptions des services.

Nous validons notre travail avec des preuves de concept et utilisons les services et les ontologies d'OWLS-TC pour évaluer nos approches proposées de sélection et d'enrichissement.

Mots Clés

Services Web, Découverte de services, Web Sémantique, Données Liées, Traitement Automatique de la Langue

Abstract

The last years witnessed the success of the Linked Open Data (LOD) project as well as a significantly growing amount of semantic data sources available on the web. However, there are still a lot of data not being published as fully materialized knowledge bases like as sensor data, dynamic data, data with limited access patterns, etc. Such data is in general available through web APIs or web services. Integrating such data to the LOD or in mashups would have a significant added value. However, discovering such services requires a lot of efforts from developers and a good knowledge of the existing service repositories that the current service discovery systems do not efficiently overcome.

In this thesis, we propose novel approaches and frameworks to search for semantic web services from a data integration perspective. Firstly, we introduce LIDSEARCH, a SPARQL-driven framework to search for linked data and semantic web services. Moreover, we propose an approach to enrich semantic service descriptions with Input-Output relations from ontologies to facilitate the automation of service discovery and composition. To achieve such a purpose, we apply natural language processing techniques and deep-learning-based text similarity techniques to leverage I/O relations from text to ontologies.

We validate our work with proof-of-concept frameworks and use OWLS-TC as a dataset for conducting our experiments on service search and enrichment.

Keywords

Web Services, Service Discovery, Service Description Enrichment, Semantic Web, Linked Data, Natural Language Processing