Pontifícia Universidade Católica
do Rio de Janeiro

Yenier Torres Izquierdo

Keyword Search over Federated RDF Graphs by Exploring their Schemas

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Marco Antonio Casanova

Rio de Janeiro
March 2017

Pontifícia Universidade Católica
do Rio de Janeiro

Yenier Torres Izquierdo

Keyword Search over Federated RDF Graphs by Exploring their Schemas

Dissertation presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the undersigned Examination Committee.

Prof. Marco Antonio Casanova
Advisor
Departamento de Informática – PUC-Rio

Profª. Simone Diniz Junqueira Barbosa
Departamento de Informática – PUC-Rio

Prof. Bernardo Pereira Nunes
Coordenação Central de Educação a Distância – PUC-Rio

Prof. Marcio da Silveira Carvalho
Vice Dean of Graduate Studies
Centro Técnico Científico – PUC-Rio

Rio de Janeiro, March 31th, 2017

Yenier Torres Izquierdo

The author graduated in Computer Science from University of Havana (UH), Havana - Cuba in 2012. He joined the Master in Informatics at Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 2015.

Bibliographic data

Izquierdo, Yenier Torres

Keyword Search over Federated RDF Graphs by Exploring their Schemas / Yenier Torres Izquierdo; advisor: Marco Antonio Casanova. – Rio de Janeiro: PUC-Rio, Departamento de Informática, 2017.

v., 66 f. : il. ; 29,7 cm

1. Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Busca por palavras-chave. 3. Dados conectados. 4. SPARQL. 5. RDF. 6. Consultas federadas. 7. Esquema mediado. I. Casanova, Marco Antonio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I would like to give a special thanks to my parents, Katiuska and Alberto, for their support and encouragement during all these years of study, to my family and true friends which contributed to the accomplishment of this challenge.

Thank you so much to Professor Marco Antonio Casanova, the best advisor I could ever have. I am admired for his professionalism and dedication to his students.

To PUC-Rio, CNPq and FAPERJ for funding my research.

To all my classmates, professors and staff from the Department of Informatics.

Thanks to all for your help and for always being so accommodating.

Thank you so much to all of you!

Abstract

Izquierdo, Yenier Torres; Casanova, Marco Antonio (Advisor). **Keyword Search over Federated RDF Graphs by Exploring their Schemas**. Rio de Janeiro, 2017. 66p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The Resource Description Framework (RDF) was adopted as a W3C recommendation in 1999 and today is a standard for exchanging data in the Web. Indeed, a large amount of data has been converted to RDF, often as multiple datasets physically distributed over different locations. The SPARQL Protocol and RDF Query Language (SPARQL) was officially introduced in 2008 to retrieve RDF datasets and provide endpoints to query distributed sources. An alternative way to access RDF datasets is to use keyword-based queries, an area that has been extensively researched, with a recent focus on Web content. This dissertation describes a strategy to compile keyword-based queries into federated SPARQL queries over distributed RDF datasets, under the assumption that each RDF dataset has a schema and that the federation has a mediated schema. The compilation process of the federated SPARQL query is explained in detail, including how to compute a set of external joins between the local subqueries, how to combine, with the help of the *UNION* clauses, the results of local queries which have no external joins between them, and how to construct the *TARGET* clause, according to the structure of the *WHERE* clause. Finally, the dissertation covers experiments with real-world data to validate the implementation.

Keywords

Keyword search; Linked Data; SPARQL; RDF; federated query; mediated schema.

Resumo

Izquierdo, Yenier Torres; Casanova, Marco Antonio. **Busca por Palavras-chave sobre Grafos RDF Federados Explorando seus Esquemas**. Rio de Janeiro, 2017. 66p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O *Resource Description Framework* (RDF) foi adotado como uma recomendação do W3C em 1999 e hoje é um padrão para troca de dados na Web. De fato, uma grande quantidade de dados foi convertida em RDF, muitas vezes em vários conjuntos de dados fisicamente distribuídos ao longo de diferentes localizações. A linguagem de consulta SPARQL (sigla do inglês de ***S**PARQL **P**rotocol **a**nd **R**DF **Q**uery **L**anguage*) foi oficialmente introduzido em 2008 para recuperar dados RDF e fornecer *endpoints* para consultar fontes distribuídas. Uma maneira alternativa de acessar conjuntos de dados RDF é usar consultas baseadas em palavras-chave, uma área que tem sido extensivamente pesquisada, com foco recente no conteúdo da Web. Esta dissertação descreve uma estratégia para compilar consultas baseadas em palavras-chave em consultas SPARQL federadas sobre conjuntos de dados RDF distribuídos, assumindo que cada conjunto de dados RDF tem um esquema e que a federação tem um esquema mediado. O processo de compilação da consulta SPARQL federada é explicado em detalhe, incluindo como computar o conjunto de *joins* externos entre as subconsultas locais geradas, como combinar, com a ajuda de cláusulas *UNION*, os resultados de consultas locais que não têm *joins* entre elas, e como construir a cláusula *TARGET*, de acordo com a composição da cláusula *WHERE*. Finalmente, a dissertação cobre experimentos com dados do mundo real para validar a implementação.

Palavras-chave

Busca por palavras-chave; dados conectados; SPARQL; RDF; consultas federadas; esquema mediado.

Table of contents

List of figures

List of tables

1
Introduction

1.1 Motivation

The Resource Description Framework (RDF) was adopted as a W3C recommendation in 1999 and today is a standard for exchanging data in the Web. At present, a huge amount of data has been converted to RDF (RAKHMAWATI, *et al.*, 2013b) and is rapidly increasing due to numerous organizations that are opening up their databases on the Web, following the Linked Data principles (BIZER, HEATH and BERNERS-LEE, 2008), often as multiple datasets physically distributed over different locations. The SPARQL Protocol and RDF Query Language (SPARQL) was officially introduced in 2008 to retrieve RDF and provide endpoints to query distributed sources.

Approaches to querying distributed RDF data with SPARQL-like queries typically exploit optimizations based on structural information (i.e. graph partitioning) (HUANG, ABADI and REN, 2011; QUILITZ, BASTIAN and LESER, 2008; ZENG, KAI, *et al.*, 2013). Furthermore, according to (RAKHMAWATI, *et al.*, 2013b), the existing tools and systems designed to address federated queries focus mostly on source selection and join optimization during federated SPARQL query execution.

An alternative way to access RDF sources is to use keyword-based queries, an area that has been extensively researched, with a recent focus on Web content. Indeed, keyword search is attracting the attention of Semantic Web practitioners, who want to support users in accessing Linked (Open) Data. In general, these users:

(i) are unaware of the way in which data is organized;
(ii) do not know how to interpret a Web ontology (if present); and
(iii) do not know the syntax of a specific query language (e.g., SPARQL).

Most approaches to address keyword-based queries assume that the RDF triples are stored in a centralized repository (ZHOU, *et al.*, 2007; MÖLLER, DRAGAN and AMBRUS, 2008; HUANG, ABADI and REN, 2011;

ELBASSUONI and BLANCO, 2011; ZENZ, *et al.*, 2009; GARCÍA, *et al.*, 2017). By contrast, the main motivation of this work is to address the problem of processing keyword-based queries over distributed RDF datasets.

1.2 Goal and Contributions

In more details, the goal of this dissertation is to develop a strategy to compile keyword-based queries into federated SPARQL queries over RDF triples stored in distributed databases, without user intervention, under the assumption that each dataset has an RDF schema and that the federation has a mediated schema.

The main contribution of this dissertation is to extend to federated environments the centralized algorithm to compile keyword-based queries to SPARQL queries implemented in (GARCÍA, *et al.*, 2017). In particular, this dissertation introduces:

- A model for keyword-based search over RDF graphs stored in distributed databases.
- A strategy to generate partial SPARQL queries against individual, centralized RDF graphs, which takes into account only the elements in their schema, without user intervention.
- A strategy to generate a federated SPARQL query from the partial queries.

1.3 Dissertation Structure

This dissertation is structured as follows. Chapter 2 provides an overview of the main concepts related to this dissertation. Chapter 3 summarizes related work. Chapter 4 presents an algorithm and its implementation to compile keyword-based queries into federated SPARQL queries. Chapter 5 covers experiments with the implementation. Finally, Chapter 6 presents the conclusions and proposes future work.

2
Background

This chapter provides an overview of the main concepts related to this dissertation. Section 2.1 introduces key definitions about RDF. Section 2.2 covers the latest version of the SPARQL Query Language. Section 2.3 summarizes SPARQL 1.1 Federated Query, the extension of SPARQL 1.1 to support queries that merge data distributed across the Web. Section 2.4 summarizes the concepts related to keyword-based queries for centralized RDF graphs. Finally, Section 2.5 defines the basic concepts of federated keyword-based queries and answers.

2.1 Resource Description Framework (RDF)

RDF is a framework for representing information about resources in the Web (CYGANIAK, WOOD and LATHANER, 2014). A global identifier that denotes a resource is named *Internationalized Resource Identifier* (*IRI*). A *literal* is a basic value, such a string, a number, or a date. Any *IRI* or *literal* denotes something in the world (the "universe of discourse"). The resource denoted by an *IRI* is called its *referent*, and the resource denoted by a *literal* is called its *literal value*. A blank node acts as a local identifier; a blank node can always be replaced by a new, globally unique *IRI* (a *Skolem IRI*). An *RDF term* is either an *IRI*, a *blank node* or a *literal*. The sets of IRIs, blank nodes and literals are disjoint and, unlike IRIs and literals, blank nodes do not identify specific resources.

RDF models data as triples of the form *(s, p, o)*, where *s* is the *subject*, *p* is the *predicate* and *o* is the *object* of the triple. An RDF triple *(s, p, o)* says that some relationship, indicated by *p*, holds between the subject *s* and object *o*. The subject of a triple is an IRI or a blank node, the predicate is an IRI, and the object is an IRI, a literal or a blank node. A triple is also seen as an edge in a *directed, labeled graph* where a directed edge (labeled predicate) connects the subject node to the object node.

A set *T* of RDF triples, or an *RDF dataset*, is equivalent to a labeled graph G_T, such that the set of nodes of *G* is the set of RDF terms that occur as subject or object of the triples in *T* and there is an edge *(s, o)* in *G* labeled with *p* iff the triple *(s,p,o)* occurs in *T*. Therefore, we will use the concepts of set of RDF triples and RDF graph interchangeably. Note that a predicate IRI can also occur as a node in the same graph.

RDF offers enormous flexibility but, apart from the `rdf:type` property, which has a predefined semantics, it provides no means for defining application-specific classes and properties. Instead, such classes and properties, and hierarchies thereof, are described using extensions to RDF provided by the *RDF Schema 1.1* (RDF Schema or RDF-S) (BRICKLEY and GUHA, 2014). In RDF-S, a *class* is any resource having an `rdf:type` property whose value is the qualified name `rdfs:Class` of the RDF Schema vocabulary. A *property* is any instance of the class `rdfs:Property`. The `rdfs:domain` property is used to indicate that a particular property applies to a designated class, and the `rdfs:range` property is used to indicate that the values of a particular property are instances of a designated class or, alternatively, are instances (i.e., literals) of an XML Schema datatype. Finally, RDF-S offers a property, `rdfs:comment`, used to associate a comment with an IRI, and a property, `rdfs:label`, used to assign a different name to a resource.

For example, the following set of triples describes the class *Drug* in DBpedia and specifies one of its instances.

```
(http://dbpedia.org/ontology/Drug,
        http://www.w3.org/1999/02/22-rdf-syntax-ns#type,
               http://www.w3.org/2002/07/owl#Class)
(http://dbpedia.org/ontology/Drug,
        http://www.w3.org/2000/01/rdf-schema#label,
               "drug")
(http://dbpedia.org/Drug/Amoxicillin,
        http://www.w3.org/1999/02/22-rdf-syntax-ns#type,
               http://dbpedia.org/ontology/Drug)
```

2.2 SPARQL 1.1 Query Language

SPARQL 1.1 is designed to tackle limitations of SPARQL 1.0, including update operations, aggregations, and federated query support. SPARQL is a query

language specifically designed to access sets of RDF triples (HARRIS and SEABORNE, 2013). SPARQL offers two types of queries. A *SELECT query* returns tabular data, whereas a *CONSTRUCT query* returns an RDF graph. The body of a SPARQL query is a *graph pattern* composed of *triple patterns*, defined like RDF triples, except that the subject, predicate or object can be a variable. The evaluation of a SPARQL query binds values to the variables using a *solution mapping*. The application of a solution mapping to a graph pattern b uniformly replaces each variable in b by the RDF term.

A simple example of a *SELECT* SPARQL query is shown below, in which the result is the set of all triples related to people who live in the cities of "Boston" or "New York", with their email address, if available.

```
PREFIX ex: <http://example.com/exampleOntology#>
SELECT  ?name ?city ?email
WHERE {
        ?name rdf:type  ex:Person .
        ?name ex:live   ?city .
        ?city rdf:type  ex:City .
        OPTIONAL{ ?name ex:email ?email }.
        FILTER( ?city IN ("Boston", "New York") )
      }
```

The `SELECT` clause identifies the variables that will appear in the result (in this case, `?name ?city ?email`). The `WHERE` clause contains the graph pattern that is matched with a RDF graph. The pattern in this example is a set of triples that join the class `ex:Person` to the class `ex:City` through the property `ex:live`, filtered by the specified cities names, and optionally returning the `ex:email` property value.

2.3 SPARQL 1.1 Federated Query Extension

SPARQL can be used to express queries across multiple data sources, whether the data is natively stored as RDF or viewed as RDF via middleware. This section summarizes the syntax and semantics of the SPARQL 1.1 Federated Query extension for executing queries distributed over different SPARQL endpoints. This extension allows for combining graph patterns that can be evaluated over several endpoints within a single query. Results are returned to the federated query processor and are combined with results from the rest of the query. The `SERVICE`

keyword instructs a federated query processor to invoke a portion of a SPARQL query against a remote SPARQL endpoint (PRUD'HOMMEAUX and BUIL-ARANDA, 2013; BUIL-ARANDA, *et al.*, 2013).

The following example shows how to query a remote SPARQL endpoint and join the returned data with the data stored into local RDF dataset. Consider a query to find the names of the people that we know and data about the names of various people available at the *http://people.example.org/sparql* endpoint:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name
FROM <http://example.org/myfoaf.rdf>
WHERE{
  <http://example.org/myfoaf/I> foaf:knows ?person .
  SERVICE <http://people.example.org/sparql>
  { ?person foaf:name ?name }
}
```

The execution of a `SERVICE` pattern may fail due to several reasons: the remote service may be down, the service IRI may not be available to be accessed, or the endpoint may return an error to the query. Normally, under such circumstances the invoked query containing a `SERVICE` pattern fails as a whole. Queries may explicitly allow failed `SERVICE` requests with the use of the `SILENT` keyword. The `SILENT` keyword indicates that errors encountered while accessing a remote SPARQL endpoint should be ignored while processing the query. The failed `SERVICE` clause is treated as if it had a result of a single solution with no bindings.

In this case, the above query will be as follows:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name
FROM <http://example.org/myfoaf.rdf>
WHERE{
  <http://example.org/myfoaf/I> foaf:knows ?person .
  SERVICE SILENT <http://people.example.org/sparql>
  { ?person foaf:name ?name }
}
```

Besides `SERVICE`, SPARQL 1.1 also introduces `VALUES` as one SPARQL Federation extension. It can reduce the intermediate results during query execution by giving constraints from the previous query to the next query.

2.4 Keyword-based Queries over Centralized RDF Graphs

In this section, we summarize the concepts related to keyword-based queries for centralized RDF graphs introduced in (GARCÍA, *et al.*, 2017).

Let T be an RDF dataset and G_T be the corresponding RDF graph. We assume that T follows an RDF schema S, with $S \subseteq T$.

A *keyword-based query* K is simply a set of literals, or *keywords*.

Let $\boldsymbol{L}$ be the set of all literals. Let *match*: $\boldsymbol{L}\times\boldsymbol{L} \rightarrow [0,1]$ be a similarity function between literals such that $match(s,t)=j$ indicates how similar s and t are: $j=1$ says that s and t are identical, and $j=0$ indicates that s and t are completely dissimilar. We also introduce a *similarity threshold* $\sigma\in(0,1]$. We leave *match* and σ unspecified at this point.

The set $\boldsymbol{MM}[K,T]$ of *metadata matches* between K and the metadata descriptions of the classes and properties in S (recall that $S \subseteq T$) is defined as:

$$\boldsymbol{MM}[K,T] = \{ (k,(r,p,v))\in K\times S \,/\, (r,p,v)\in S \wedge match(k,v)\geq\sigma \}$$

The set $\boldsymbol{VM}[K,T]$ of *property value matches* between K and property values of T is defined as (recall that $S \subseteq T$):

$$\boldsymbol{VM}[K,T] = \{ (k,(r,p,v))\in K\times T \,/\, (r,p,v)\notin S \wedge match(k,v)\geq\sigma \}$$

The set of *matches* between K and T is then defined as:

$$\boldsymbol{M}[K,T] = \boldsymbol{MM}[K,S] \cup \boldsymbol{VM}[K,T]$$

An *answer* for K over T is a subset A of T such that:

(1) There is a subset of K, denoted K/A, such that, for each $k\in K/A$:
   a. There are $(s,\text{rdf:type},c_n)$, $(c_n,\text{rdfs:subClassOf},c_{n-1})$,..., $(c_1,\text{rdfs:subClassOf},c_0)$ and (c_0,p_0,v_0) in A such that $(k,(c_0,p_0,v_0))\in\boldsymbol{MM}[K,T]$; or
   b. There are (s,q_n,v_n), $(q_n,\text{rdfs:subPropertyOf},q_{n-1})$,..., $(q_1,\text{rdfs:subPropertyOf},q_0)$ and (q_0,p_0,v_0) in A such that $(k,(q_0,p_0,v_0))\in\boldsymbol{MM}[K,T]$; or
   c. There is $(r,p,v)\in A$ such that $(k,(r,p,v))\in\boldsymbol{VM}[K,T]$.

(2) There is no other answer B for K over T such that $K/A \subset K/B$.

We say that K/A is the set of keywords *matched* by A.

Condition (1a) says that a keyword k has a class metadata match for a class c_0 and the answer A must contains an instance of c_0 or one of its sub-classes c_n, in which case A must include all triples indicating that c_n is a sub-class of c_0. Likewise, Condition (1b) says that a keyword k has a property metadata match for a property

q_0 and the answer A must contains an instance of q_0 or one of its sub-properties q_n, in which case A must include all triples indicating that q_n is a sub-property of q_0. Condition (1c) simply says that k matches the literal of a triple (r,p,v) in A. Also, Condition (1) does not require that all keywords in K be matched in an answer. Indeed, we say that A is *total* iff $K/A = K$, and *partial* otherwise. Condition (2) requires that an answer must match as many keywords in K as possible.

The definition of an answer is quite liberal. In particular, it allows an answer A to be a set of disconnected triples. To circumvent this problem, (GARCÍA, *et al.*, 2017) defines a partial order between answers as follows. Given a directed graph G, let $|G|$ denote the number of nodes and edges of G and $\#c(G)$ denote the number of connected components of G, when the direction of the edges of G is disregarded. A partial order "<" for graphs is defined such that, given two graphs G and G',

$$G < G' \text{ iff } \quad (\#c(G) + |G|) < (\#c(G') + |G'|) \text{ or}$$
$$(\#c(G) + |G|) = (\#c(G') + |G'|) \text{ and } \#c(G) < \#c(G')$$

In (GARCÍA, *et al.*, 2017) is used the partial order "<" between graphs to compare answers. We say that an answer A *is smaller than* an answer B iff $G_A < G_B$, where G_A and G_B are the RDF graphs of A and B (which may include metadata, since the RDF schema is part of the dataset). An answer A for K over T is *minimal* iff there is no other answer B for K over T such that $G_A < G_B$.

2.5 Keyword-based Queries over Federated RDF Graphs

Let R be the set of all IRIs and $\boldsymbol{L}$ be the set of all literals. Let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of distributed RDF datasets, where T_i is identified by a SPARQL endpoint s_i, for i=1,…,n. Let G_{Ti} be the RDF graph corresponding to T_i, and assume that T_i follows an RDF schema S_i, with $S_i \subseteq T_i$. Let S be a set of *inter-dataset* RDF triples of the form (s_i, p, o_j) where s_i is an IRI occurring in a dataset T_i and o_j is an IRI occurring in T_j, where $i, j \in [1,n]$ and $i \neq j$. The *global graph* corresponding to T and S is defined as the RDF graph corresponding to the set of triples $T_1 \cup T_2 \cup \ldots \cup T_n \cup S$.

A *keyword-based query* K is defined as for the centralized case as a set of literals, or *keywords*.

We define a *federated answer* for *K* over $T \cup S$ as a minimal answer for *K* over $T \cup S$, where the notion of minimal answer was defined in the previous section.

The problem of finding answers for keyword-based queries over sets of federated RDF graphs (or, briefly, the *RDF-FKwS problem*) is defined as: "Given a set *T* of federated RDF graphs and a keyword-based query *K*, find an answer for *K* over *T*". A stricter form is *the problem of finding minimal answers for keyword-based queries over a set of federated RDF graphs* (or, briefly, the *minRDF-FKwS problem*), defined as: "Given a set *T* of federated RDF graphs and a keyword-based query *K*, find a minimal answer for *K* over *T*".

For example, if we search data about a drug, we can collect data from DrugBank[1], DBpedia[2] and Kegg[3] SPARQL endpoints. Figure 1 shows each schema and the relation between these datasets. We obtain information about a drug and its indicated use from DrugBank and DBpedia by using *owl:sameAs*. In this example, Kegg stores the chemical composition and other information about drugs in the *Compound* class. DrugBank and Kegg are connected by the object property *keggCompoundId* that links the classes *drug* and *Compound*.

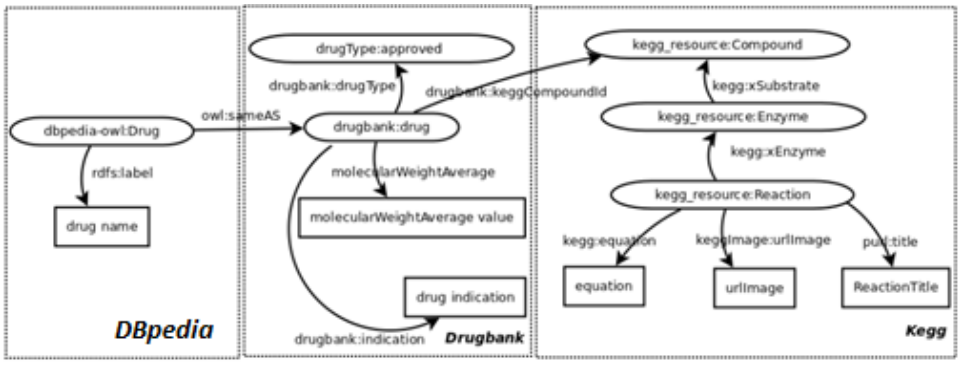


Figure 1. Schema and relationship between DBpedia, DrugBank and Kegg Dataset **(RAKHMAWATI, *et al.*, 2013b)**

Let *K = {drug, fever, acetamide, ethanamide}* be a keyword-based query and let *T* be the set of data sources shown in Figure 1. The feasible matches are: '*drug'* matches with the class name of the DBpedia class *Drug*. The keyword '*fever'* matches with a value of the DrugBank *drug:indication* property; and *'acetamine'* and *'ethanamide'* match with label of instances of the class *Compound* in Kegg dataset. An expected response to this query would then be **Paracetamol**[4].

[1] http://www4.wiwiss.fu-berlin.de/drugbank/sparql
[2] http://dbpedia.org/sparql
[3] http://kegg.bio2rdf.org/sparql
[4] https://en.wikipedia.org/wiki/Paracetamol

3
Related Work

Many approaches have been developed to help solve the keyword search problem over RDF graphs. The main challenge has been to synthesize SPARQL queries from a set of keywords because users are generally unaware of the query language and the RDF graph schema to be queried. This chapter provides an overview of different approaches, which use SPARQL queries to access Linked (Open) Data, in both centralized and distributed environments.

The set of works analyzed were selected based on their relevance for the discussion. Each analyzed work contains a description of the main characteristics of the approach, the elements that contribute to the development of current work and drawbacks of obtained results linked to the goal of our work.

3.1 Keyword Search over RDF Graphs in Centralized Environments

A first version of an extension to the *Konduit* tool (DRAGAN, *et al.*, 2009), that provides non-expert users with a way to visually specify SPARQL queries, is presented in (MÖLLER, DRAGAN and AMBRUS, 2008). Users avoid having to write the SPARQL query, which can be tedious and error prone. Additionally, data types for literals from a selection box can also be specified.

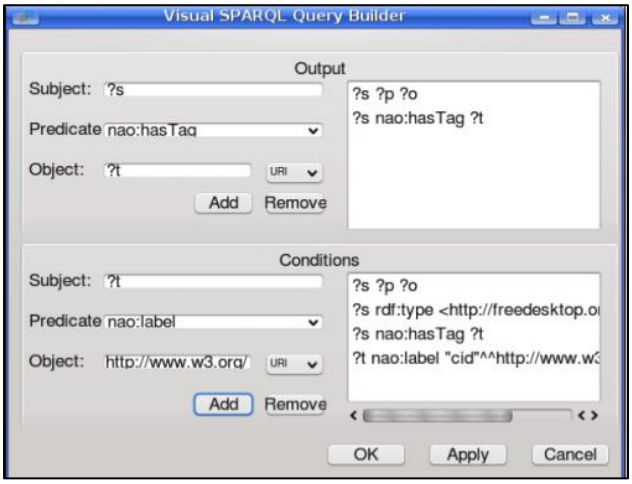


Figure 2. Building a SPARQL CONSTRUCT query in *Konduit* **(MÖLLER, DRAGAN and AMBRUS, 2008)**

However, this tool has the disadvantage that the user has to have a basic knowledge of the SPARQL query language and it is only possible to create SPARQL *CONSTRUCT* queries.

The *QUICK* (**QUery Intent Constructor for Keywords**) tool, described in (ZENZ, *et al.*, 2009), is a system for helping users construct semantic queries in a given domain. QUICK combines the convenience of keyword search with the expressivity of semantic queries. Users start with a keyword query and then are guided through a process of incremental refinement steps to specify the query intention. The intermediate queries are listed and ranked.

Figure 3 shows the user interface of *QUICK*, which consists of three parts: a search field (at the top), the construction pane showing the query construction options (on the left), and the query pane showing semantic queries (on the right). *QUICK* computes all possible semantic queries, presents the selected ones in the query pane, generates a set of query construction options, and presents them in the construction pane. The generated construction options ensure that the space of semantic interpretations is rapidly reduced with each selection. When the user selects the desired query, *QUICK* executes it and shows the results. This approach still has the drawback that the user must have knowledge of the concept of RDF schema.

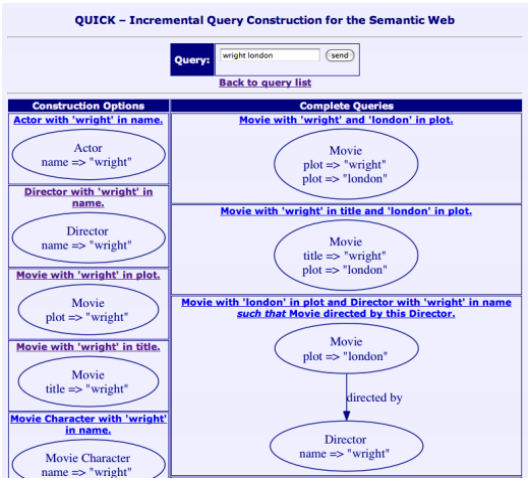


Figure 3. QUICK User Interface
(ZENZ, *et al.*, 2009)

However, unlike (ZENZ, *et al.*, 2009), in (GARCÍA, *et al.*, 2017) the translation of a set of keywords to a SPARQL query is fully automatic, although in both tools to synthesize SPARQL queries, the RDF schema is explored. This last approach features an algorithm to translate a keyword query into a SPARQL query

such that each result of the SPARQL query is an answer to the keyword query. Also the implemented tool in this approach allows the user to specify a keyword-based query which includes *filters*, which involve only comparison operators, expressed in symbolic form, such as *"<, >, =*", or using reserved words, such as "*between*". The syntax of the keywords and filters was specified by a grammar defined in ANTLR4 (ANother Tool for Language Recognition) (PARR, 2013).

Our work consists of the extension of the centralized algorithm presented in (GARCÍA, *et al.*, 2017) for a distributed scenario.

3.2 Federated Queries over SPARQL Endpoints

Several frameworks, such as ARQ, Sesame and Virtuoso, have been built on top of SPARQL query engines supporting SPARQL 1.1; but this field is still far from maturity. In (RAKHMAWATI, *et al.*, 2013a), a comparison of existing SPARQL federation frameworks is given. Table 1 shows only the existing frameworks to support SPARQL 1.1 Federation Extension. As we can see, all frameworks support the *SERVICE* keyword, but not all of them support *BIDING* and *VALUES* operators.

Framework	Platform	SERVICE	BINDINGS	VALUES
ARQ	Jena	√	×	√
SPARQL-FED	Virtuoso	√	×	√
Sesame	Sesame	√	√	√
SPARQL-DQP	OGSA-DAI OGSA-DQP	√	√	×

Table 1 - The Existing Frameworks Support SPARQL 1.1 Federation Extension

For instance, ARQ[5] is a query engine processor for Jena that supports federated query, providing *SERVICE* and *VALUES* operators. The framework implements nested loop joins to retrieve and combine result from multiple SPARQL endpoints. Also, it provides a set of Java packages[6] to build SPARQL query programmatically.

[5] http://jena.apache.org/documentation/query/index.html
[6] http://jena.apache.org/documentation/javadoc/arq/

Based on the data source location, the infrastructure for querying Linked Data can be divided into two categories: *central repositories* and *distributed repositories*. According to (RAKHMAWATI, *et al.*, 2013b), systems belonging to the distributed category can be grouped into two types: *Link Traversal* and *Federation*.

Federation systems use a query mediator to transform a user query into several sub queries and generates results from the integrated data sources. As the data sources need not be collected in a single repository, the data tend to be more up-to-date than in a central repository, but query processing time takes longer. There are two kinds of federation frameworks: federation over single repositories and federation over SPARQL Endpoints (Figure 4).

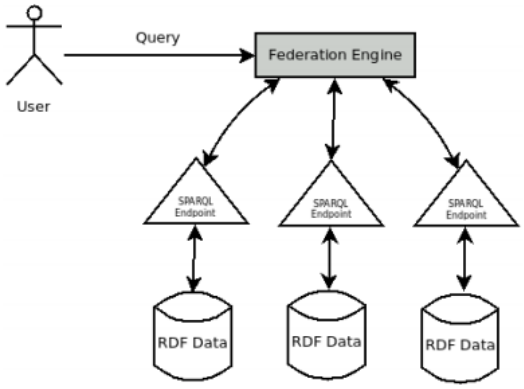


Figure 4. Federation over SPARQL Endpoints
(RAKHMAWATI, *et al.*, 2013b)

An example of such tool is *FedSearch*, a system to access distributed linked data (NIKOLOV, SCHWARTE and HÜTTER, 2013). This system implements a hybrid query engine based on the SPARQL federation framework *FedX* (SCHWARTE, *et al.*, 2011). This approach proposes an extension to the SPARQL query algebra that allows representing hybrid SPARQL queries in a triple-store-independent way and suggests query optimization techniques to match keyword search clauses to appropriate repositories, combining the retrieved results seamlessly, and reducing the processing time.

ANAPSID: AN Adaptive query ProcesSing engIne for sparql enDpoints (ACOSTA, *et al.*, October, 2011) is an adaptive query processing engine for RDF Linked Data accessible through SPARQL endpoint, which provides a set of physical operators and an execution engine able to adapt the query execution to the availability of the endpoints and to hide delays from users. *ANAPSID* is a system that accepts SPARQL query federation in SPARQL 1.0 format, but it was built on top of SPARQL query engine that supports SPARQL 1.1.

The tools described above, and already mentioned in the introduction, do not focus on compiling federated SPARQL queries from keywords, but on query optimization. The work reported in this dissertation extends the centralized algorithm developed in (GARCÍA, *et al.*, 2017) to compile federated SPARQL queries from keyword-based queries. Our approach is developed following the architecture for a federation of SPARQL Endpoints described in (RAKHMAWATI, *et al.*, 2013b). The Java packages that ARQ provides is used in the implementation to synthesize the SPARQL queries.

4
Compiling Keyword-based Queries into Federated SPARQL Queries

This chapter describes an algorithm and its implementation to compile keyword-based queries into federated SPARQL queries. Section 4.1 presents the system architecture. Section 4.2 explains the components present in the solution. Finally, section 4.3 details the construction process of the federated SPARQL query.

4.1 Architecture

Figure 5 depicts the architecture of the system that we propose in this dissertation, whose main elements are: (1) a Web interface that allows the user to perform keyword search; (2) a component called **Mediator**; (3) a component that stores data and metadata of the RDF graphs accessible via SPARQL endpoints; (4) a component that saves the *sameAs* definition, the external object properties and the mapping of the elements of different data sources to elements of the mediated schema; and (5) a set of available SPARQL Endpoints.

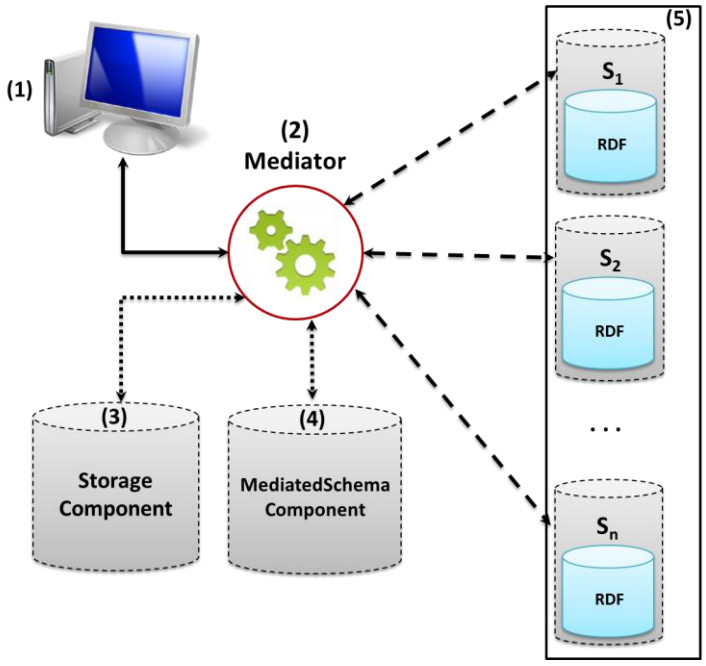


Figure 5. Architecture of Federated Keyword Search System

The **Mediator** is called when a keyword-based query is submitted, and the federated algorithm described in section 4.3.1 is executed. In general, the process is divided into three main phases:

1. The **Mediator** gets the set of keywords specified by user and the centralized algorithm is executed, for each endpoint, to compute a local subquery. In this process, the **Storage Component** is queried to find the data and metadata matches between the keywords.
2. The **Mediator** uses the **Mediated Schema Component** to find the external joins between the computed subqueries. If necessary, *UNION* clauses are created to combine the result of queries that are not linked by the joins found. Then, the federated SPARQL query is synthesized.
3. Finally, the federated SPARQL query is executed and the response returned to the user.

Figure 6 shows the sequence diagram of the construction process of the federated SPARQL query.

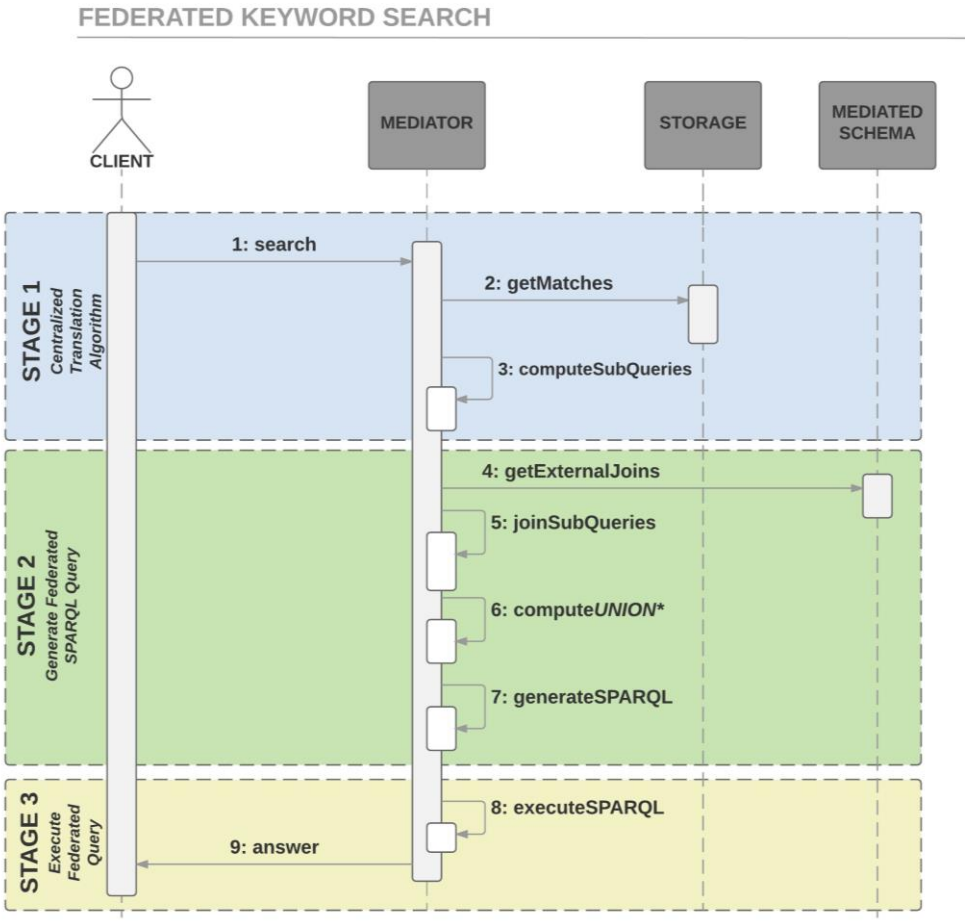


Figure 6. Sequence Diagram of Federated Keyword Search Process

4.2 Components Description

4.2.1. Storage Component

The **Storage Component** is responsible for recording the metadata schema of each RDF graphs and the indexed property values.

This component owns the following tables:

- **EndpointTable:** stores the URL of each SPARQL Endpoint (data source) and a flag to indicate if it is available.

and, for each SPARQL endpoint:

- **ClassTable:** saves all classes and their metadata (label and description).
- **PropertyTable:** stores all properties with their metadata (domain, label, description, range, etc.).
- **ValueTable:** stores the distinct searchable (indexed) property values with information about their property and class domain.
- **JoinTable:** saves the object properties of the RDF graph.
- **GroupTable:** lists all properties that will be shown to the user, organized into groups.

4.2.2. Mediated Schema Component

This component is responsible for saving the link definitions between two endpoints, which can be either external object properties or *sameAs* definitions, and the mappings between the local schema elements to elements of the mediated schema.

The **Mediated Schema Component** has three tables:

- **SameAsTable:** stores the properties in different sources that represent the *sameAs* definitions between two classes.
- **ExternalObjectPropertyTable:** saves the object properties that link two classes in different endpoints.
- **MapElementTable:** records a mapping of elements (classes and properties) of different data sources to homogeneous elements of the mediated schema that can be combined in *UNION* clauses.

The *sameAs* definitions in **SameAsTable** and the object properties in **ExternalObjectPropertyTable** describe which are the possible inter-dataset RDF triples introduced in Section 2.5. We consider that the strategies to maintain and update the auxiliary tables in the **Storage** and the **Mediated Schema Components** are outside the scope of this work. Furthermore, in our approach, the presence of materialized *owl:sameAs* properties will not be taken into account. That is, the *sameAs* links will be computed on the fly from the information stored in **SameAsTable**.

We say that there is an *external join* between classes c_i and c_j iff

1) There is a tuple of the form $(e_i, c_i, (p_{i1}, \ldots, p_{in}), e_j, c_j, (p_{j1}, \ldots, p_{jn}))$ in the **SameAsTable** table (see Table 2), in which case we say that c_i is the *source class* and c_j is the *destination class* of the external join; or

2) There is a tuple of the form $(e_i, c_i, p_m, e_j, c_j)$ in the **ExternalObjectPropertyTable** table (see Table 3), in which case we say that c_i is the *source class*, c_j is the *destination class*, and p_m is the *joining property* or the external join.

SameAsTable					
Endpoint Source	**Class Source**	**Properties Source**	**Endpoint Destination**	**Class Destination**	**Properties Destination**
			. . .		
e_i	c_i	$p_{i1}, \ldots, p_{in}$	e_j	c_j	$p_{j1}, \ldots, p_{jn}$
			. . .		

Table 2 - Definition and sample fragment of **SameAsTable**

ExternalObjectPropertyTable				
Endpoint Domain	**Class Domain**	**Object Property**	**Endpoint Range**	**Class Range**
		. . .		
e_i	c_i	p_m	e_j	c_j
		. . .		

Table 3 - Definition and sample fragment of **ExternalObjectPropertyTable**

We also say that:

1) Two classes c_i and c_j, respectively declared in the schemas of the datasets associated with endpoints e_i and e_j, are *associated to* (or *mapped to*) a class *X* of the mediated schema iff there are two tuples of the form *(X, c_i, NULL, e_i)* and *(X, c_j, NULL, e_j)* in the **MapElementTable** table (Table 4).

2) Two properties p_i and p_j, respectively declared with domains c_i and c_j in the schemas of the datasets associated with endpoints e_i and e_j, are *associated to* (or *mapped to*) a property *Y* of the mediated schema iff there are two tuples of the form *(Y, c_i, p_i, e_i)* and *(Y, c_j, p_j, e_j)* in the **MapElementTable** table (Table 4).

MapElementTable			
Mediated Schema Element	**Local Schema Class**	**Local Schema Property**	**Source**
	...		
X	c_i	*NULL*	e_i
X	c_j	*NULL*	e_j
Y	c_i	$\boldsymbol{p_i}$	e_i
Y	c_j	$\boldsymbol{p_j}$	e_j
	...		

Table 4 - Definition and sample fragment of **MapElementTable**

4.2.3. Mediator Component

The **Mediator** is the core component of our strategy. It runs the algorithm that compiles keyword-based queries into federated SPARQL queries. It receives and processes a set of keywords, submitted by the user, and connects with the databases where the **Storage Component** and the **Mediated Schema Component** are located. Finally, it returns to the user the response of the execution of the generated query.

4.3 Constructing the Federated SPARQL Query

4.3.1. Overview of the Federated Translation Algorithm

Figure 7 shows a high-level description of the **Federated Translation Algorithm**. It receives as input a keyword-based query K and a set of RDF datasets $T = \{T_1, T_2, \ldots, T_n\}$ and returns an answer of K.

FEDERATED TRANSLATION ALGORITHM
Input: A keyword-based query K
A set of RDF datasets $T = \{T_1, T_2, \ldots, T_n\}$
Output: A SPARQL query $\boldsymbol{Q}$ over T

STAGE 1 Compute the set of local SPARQL queries:
For each endpoint e_i:
1.1. Run the Centralized Translation Algorithm.
1.2. Return the local SPARQL query Q_i or *NULL*.

STAGE 2 Synthesize the federated SPARQL query $\boldsymbol{Q}$:
2.1. Discover the external joins which are candidates to link the local queries.
2.2. Create a federated graph G_F and compute a maximum spanning forest T_F of G_F, to select the external joins that will be present in the federated query.
2.3. Insert the triple patterns corresponding to the selected external joins into the respective queries.
2.4. If T_F is not connected, then check if it is possible to compute *UNION* clauses.
2.5. Synthesize and return $\boldsymbol{Q}$.

STAGE 3 Execute the federated query:
3.1. Execute **Q**.
3.2. Returns the result to the user.

Figure 7. Outline of the Federated Translation Algorithm

Stage 1 runs the **Centralized Translation Algorithm**, for each dataset T_i. The result may be a local query, Q_i, if dataset T_i contributes to answering K, or *NULL*, otherwise. Section 4.3.2 explains the process to compute the local queries.

Stage 2 synthesizes a federated SPARQL query $\boldsymbol{Q}$ from the local queries, and is the central contribution of this dissertation. Section 4.3.3 details the steps 2.1, 2.2 and 2.3, and section 4.3.4 covers Step 2.4. Step 2.1 finds the set of external joins that are candidates to link the set of local queries. Step 2.2 first creates the *federated multigraph* $G_F = <V_F, E_F, W_F>$ corresponding to the local queries and the candidate external joins, and then computes a maximum directed spanning forest T_F of G_F, considering the weight of the arcs in W_F. Step 2.3 inserts into the local queries the triple patterns associated to the external links corresponding to the arcs present in T_F. If T_F is not connected, then Step 2.4 tries to combine the subqueries represented

by the connected components of T_F through *UNION* clauses, as explained in section 4.3.4. Step 2.5 synthesizes the federated SPARQL query ***Q***, as described in sections 4.3.5 and 4.3.6.

Stage 3 executes ***Q*** and returns an answer of *K* to the user.

4.3.2. Computing the Set of Local Queries

To compute the set of local queries, Stage 1 of the **Federated Translation Algorithm** runs the **Centralized Translation Algorithm**.

The **Centralized Translation Algorithm** (see Figure 8) accepts a keyword-based query *K* and an RDF dataset *Ti*, and outputs a SPARQL query or *NULL*. After removing stop words from *K*, it matches the remaining elements in *K* with literals stored in tables **ClassTable**, **PropertyTable** and **ValueTable** corresponding to T_i. The match process creates a set of *metadata matches* $\boldsymbol{MM}[K,T_i]$ and a set of *property value matches* $\boldsymbol{VM}[K,T_i]$, as mentioned in Section 2.4. If $\boldsymbol{MM}[K,T_i]$ and $\boldsymbol{VM}[K,T_i]$ are empty, the centralized algorithm returns *NULL*. Otherwise, it synthesizes and returns a local SPARQL query Q_i, in which case we say that the dataset T_i *contributes* to answering *K*.

CENTRALIZED TRANSLATION ALGORITHM
Input: A keyword query *K*
An RDF dataset *T*, with a simple RDF schema *S*
Output: A SPARQL query *Q* over *T*
1. Keyword matching:
   Match the keywords in *K* with literals in *T*, creating the sets of matches ***MM[K,T]*** and ***VM[K,T]***.
2. Nucleus generation:
   Use ***MM[K,T]*** and ***VM[K,T]*** to compute a set ***M*** of nucleuses
3. Nucleus score computation:
   Compute the score of each nucleus in ***M***.
4. Nucleus selection:
   Compute a set of nucleuses ***N*** such that ***N*** covers as many keywords as possible.
5. Steiner tree generation:
   Let D_S be the RDF schema diagram of *S*.
   Compute a Steiner tree *ST* of D_S that contains the set of classes of the nucleuses in *N*.
6. Synthesis of the SPARQL query *Q*:
   Construct the *WHERE* and the *TARGET* clauses of *Q* from the nodes and edges in *ST*

Figure 8. Outline of the Centralized Translation Algorithm
(GARCÍA, *et al.*, 2017)

Assume that the *TARGET* clause of Q_i is composed of variables $v_1,\ldots,v_n$. Then, the **Centralized Translation Algorithm** also returns, for each variable v_k (k=1,…,n):

(1) the set of keywords K_k that v_k covers;

(2) the IRIs of the elements that v_k binds to, as follows:

- if v_k binds to instances of a class c_i, then the IRI of c_i;
- if v_k binds to property values of a property p_i, then the IRIs of p_i and c_i, where c_i is the domain of the property p_i.

The set of keywords that the query Q_i covers is given by $K_i = K_1 \cup \ldots \cup K_n$, such that $K_i \subseteq K$.

4.3.3. Computing the *External Joins* of the Federated Query

Recall that the result of executing the centralized algorithm for a dataset T_i is a local query Q_i or *NULL*. For the sake of simplicity and without loss of generality, by reordering the datasets, we may assume that the **Centralized Translation Algorithm** returns a non-null answer for datasets T_1, T_2, …, T_k, for $k \leq n$. Let $\boldsymbol{C_i}$ denote the set of classes presents in Q_i, for $1 \leq i \leq n$. Also, for each $c_i \in \boldsymbol{C_i}$, let *score*(c_i) be the score of the nucleus computed by the **Centralized Translation Algorithm** that contains the class c_i, as defined in (GARCÍA, *et al.*, 2017).

Recall from Section 4.2.2 that the external joins are defined using tables **SameAsTable** and **ExternalObjectPropertyTable**.

To identify which external joins are *candidates* to construct the federated SPARQL query, Step 2.1 checks if, for a pair of local queries Q_i and Q_j, with $1 \leq i \neq j \leq n$, there are classes $c_i \in \boldsymbol{C_i}$ and $c_j \in \boldsymbol{C_j}$, such that there is an external join from c_i to c_j. We also define the *score* of a candidate external join as the summation of the scores of $c_i \in \boldsymbol{C_i}$ and $c_j \in \boldsymbol{C_j}$.

The next procedure shows how to compute the set of candidate external joins and their scores.

Compute the set of *candidate* external joins (EJ):

```
Input: List of queries Qe
       ExternalObjectPropertyTable extObjProperty
       SameAsTable sameAsDef
Output: List of Candidate External Joins
List<ExternalJoin> EJ = ∅
```

```
for each Query qi in Qe:
  for each Class ci in qi.Classes():
    ei = qi.Endpoint()
    ∃qj ∈ Qe ∧ qi ≠ qj
    ej = qj.Endpoint()
    if(∃(ei,ci,(pi1,…,pin),ej,cj,(pj1,…,pjn))∈ sameAsDef)
      EJ.add(new SameAs(qi, qj, ci, cj))
    else if(∃(ei,ci,pi,ej,cj) ∈ extObjProperty)
      EJ.add(new ObjectProperty(qi, qj, ci, cj)
return EJ
```

After computing the set of *candidate* external joins, Step 2.2 first creates the *federated multigraph* $G_F = <V_F, E_F, W_F>$ corresponding to the local queries and the *candidate* external joins, where:

- $V_F = \{Q_1,\ldots,Q_k\}$ is the set of local queries that Stage 1 returns;
- there is an arc (Q_i,Q_j) in E_F with score $w((Q_i,Q_j)) = s$ iff there is a candidate external join, returned by Step 2.1, between a class c_i of Q_i and a class c_j of Q_j whose score is s.

Note that G_F is indeed a multigraph, since there can be more than one candidate external join linking the same pair of local queries. Also, G_F may have more than one connected component.

Step 2.2 then computes an approximation of a maximum spanning forest T_F of G_F, by calling a specific procedure, called MST, that we leave unspecified. The arcs in T_F represent the *selected external joins* that will be used to bind the local queries. The method to compute theses external joins is shown below.

Select the external joins with the highest score (SJ):

```
Input:  List of External Joins EJ
        List of endpoint queries Qe
Output: List of External Joins SJ
GF = CreateFederatedGrap(EJ, Qe);
TF = MST(GF)
SJ = TF.Arcs()
return SJ
```

Having already selected the external joins, the score value of the queries is calculated. We define the *score* of the query Q_i, denoted by $score(Q_i)$, as the highest score value of the external join outgoing of it:

$$score(Q_i) = \max\{w((Q_i,Q_j)), \forall Q_j \in V_F : (Q_i, Q_j) \in T_F\}$$

Then, for each selected external join, Step 2.3 adds the related triples pattern to the corresponding query. We refer to such triple patterns as *external join triple patterns*.

There are two cases to consider. If the selected external join is based on an external object property, the following code shows how the triple pattern is inserted into the local query corresponding to the domain endpoint.

Insert the triple patterns associated to external *objectProperty*:

```
Input: Set of External Joins E_F
       List of endpoint queries Qe
for each eJ in SJ:
  if (eJ IS ObjectProperty)
    Q_i = getDomainQueryEndpoint(eJ)
    c_i = getDomainClass(eJ)
    oP = getObjectProperty(eJ)
    c_j = getClassRange(eJ)
    Q_i.addTriplePattern(c_i, oP, c_j)
```

The following fragment shows how the triple pattern (s_i p_{im} s_j) is appended to the subquery Q_i, where some query elements that did not intervene in the discussion were omitted for convenience.

```
  SERVICE SILENT <e_i>{
          ...
      ?c_i rdf:type rdfs:Class .
      ?s_i rdf:type ?c_i .
      ?s_i p_im ?s_j .
          ...       }
  SERVICE SILENT <e_j>{
          ...
      ?c_j rdf:type rdfs:Class .
      ?s_j rdf:type ?c_j .
          ...       }
```

If the selected external join is a *sameAs* definition, the following code shows how the triple patterns are appended to the corresponding subqueries.

Insert the triple patterns associated to *SameAs* Definition:

```
Input: List of External Joins SJ
       List of endpoint queries Qe
for each eJ in SJ:
  if(eJ IS SameAs)
    Q_i = getDomainQueryEndpoint(eJ)
    Q_j = getRangeQueryEndpoint (eJ)
    c_i = getDomainClass(eJ)
    c_j = getRangeClass(eJ)
    for i = 0 to getProperties(eJ).size()
      Create a new variable value: v_i
      p_di = getDomainPropertyIndex(eJ, i)
      p_ri = getRangePropertyIndex(eJ, i)
      Q_i.addTriple(c_i, p_di, v_i)
      Q_j.addTriple(c_j, p_ri, v_i)
```

For each definition of *sameAs*, triple patterns of the form (s_i p_{ik} v_k) and (s_j p_{jk} v_k) are included in the subqueries Q_i and Q_j. For each pair of properties p_{ik} and p_{jk}, a variable v_k is created by matching the property values, where $c_i \in C_i$ is the domain of property p_{ik} and $c_j \in C_j$ is the domain of property p_{jk}. The fragment below highlights how the query pattern is created, where again some query elements that did not intervene in the discussion were omitted for convenience.

```
SERVICE SILENT <ei>{
        ...
    ?ci rdf:type rdfs:Class .
    ?si rdf:type ?ci .
        ...
    ?si pi1 ?v1 .
        ...
    ?si pin ?vn . }
SERVICE SILENT <ej>{
        ...
    ?cj rdf:type rdfs:Class .
    ?sj rdf:type ?cj .
    ?sj pj1 ?v1 .
        ...
    ?sj pjn ?vn . }
```

In the present solution, the property values in the *sameAs* definitions are compared using perfect matching. This approach can be changed by applying transformations (e.g. lower case function) and similarity measures (e.g. Levenshtein distance), as in tools that compute links between different datasets, like as the Silk Linking Framework (VOLZ, *et al.*, 2009).

4.3.4. Computing the *UNIONs*

Recall that Step 2.2 of the **Federated Translation Algorithm** creates a federated graph G_F and computes a maximum spanning forest T_F of G_F. If T_F is an unconnected graph, then Step 2.4 is executed to compute the feasible *UNION* clauses. In this section, we present the conditions under which queries can be combined using *UNION* clauses.

The computation of an *UNION* clause that combines the results of two unlinked queries in T_F requires that certain conditions be met. Let $\bar{Q}_1$ and $\bar{Q}_2$ be two SPARQL queries, assume that $S_1 = \{v_{11}, v_{12}, \ldots, v_{1n}\}$ is the *TARGET* clause of $\bar{Q}_1$, $S_2 = \{v_{21}, v_{22}, \ldots, v_{2n}\}$ is the *TARGET* clause of $\bar{Q}_2$, S_1 covers the keywords set K_1 and S_2 covers the keywords set K_2. It is possible to combine $\bar{Q}_1$ and $\bar{Q}_2$ with the help of a *UNION* clause iff

(1) $K_1 = K_2$,

(2) S_1 and S_2 have the same number of variables,

(3) The pair of variables $v_{1k} \in S_1$ and $v_{2k} \in S_2$ (for k=1,…,n) are bound to classes or properties that map to the same mediated schema element, as we defined in section 4.2.2. For the sake of simplicity, we also say that v_{1k} and v_{2k} map to the same mediated schema element.

The algorithm to compute a *UNION* clause is shown below.

```
Compute a UNION clause:
Input: Pair of subqueries: Qi, Qj
       MapElementTable mapTable
Output: A UNION query Q̄
if(Qi.variables() ≠ Qj.variables()) return null
for s=0 to Qi.variables().size()
  Var qi = Qi.variables(s)
  Var qj = Qj.variables(s)
  Element ei = mapTable.MediatedSchemaElement(qi)
  Element ej = mapTable.MediatedSchemaElement(qj)
  if(ei ≠ ej) return null
Q̄ = Qi ∪ Qj
return Q̄
```

To generate a federated SPARQL query with a *UNION* clause, a bind variable u_s (s=*0,…,k*) is created, for each pair of variables in the respective *TARGET* clauses that refer to the same element of the mediated schema, where k is the number of variables present in S_{Qi} and S_{Qj}.

An example template of the *UNION* pattern is shown below.

```
SELECT (?si AS ?u1) (?vi1 AS ?u2) ⋯ (?vik-1 AS ?uk)
WHERE{
   SERVICE SILENT <ei>{
     ?ci rdf:type rdfs:Class .
     ?si rdf:type ?ci .
          ...          }
 }
 UNION{
   SELECT (?sj AS ?u1) (?vj1 AS ?u2) ⋯ (?vjk-1 AS ?uk)
   WHERE{
     SERVICE SILENT <ej>{
       ?cj rdf:type rdfs:Class .
       ?sj rdf:type ?cj .
             ...         }
   }
 }
```

4.3.5. Defining the *WHERE* clause of the Federated SPARQL Query

Let Q_i (i=1,...,k) be the local SPARQL queries computed in Stage 1 of the **Federated Translation Algorithm**. Let $\boldsymbol{Q}$ be the federated SPARQL query to be constructed, and $\boldsymbol{W_Q}$ be the *WHERE* clause of $\boldsymbol{Q}$. The definition of $\boldsymbol{W_Q}$ is given by the expression $\boldsymbol{W_Q} = \cup\bar{\boldsymbol{Q}}_j$, where $\bar{\boldsymbol{Q}}_j = \bowtie_i \boldsymbol{Q_i}$, i=1,...,m; $m \leq n$, corresponds to a tree of the spanning forest, and j corresponds to the number of connected components.

For a better understanding of the above definition, consider the following example. Suppose that the following local SPARQL queries $\boldsymbol{Q_1}$, $\boldsymbol{Q_2}$, $\boldsymbol{Q_3}$ and $\boldsymbol{Q_4}$ were generated, where for convenience we omit the syntax of queries. Assume that $\boldsymbol{Q_1}$ and $\boldsymbol{Q_2}$ are joined by a *sameAs* definition, and $\boldsymbol{Q_3}$ and $\boldsymbol{Q_4}$ are joined by an object property, and that these are no other joins between these queries. Based on these assumptions, we can compute $\bar{\boldsymbol{Q}}_1$ and $\bar{\boldsymbol{Q}}_2$ as:

$$\bar{\boldsymbol{Q}}_1 = \boldsymbol{Q_1} \bowtie \boldsymbol{Q_2} \text{ and } \bar{\boldsymbol{Q}}_2 = \boldsymbol{Q_3} \bowtie \boldsymbol{Q_4}$$

where the symbol "⋈" concisely represents a join between two queries via a *sameAs* or an object property. Assume that the results of $\bar{\boldsymbol{Q}}_1$ and $\bar{\boldsymbol{Q}}_2$ can be combined by a *UNION* clause. Then, we can compute $\boldsymbol{W_Q}$ as:

$$\boldsymbol{W_Q} = \bar{\boldsymbol{Q}}_1 \cup \bar{\boldsymbol{Q}}_2 = (\boldsymbol{Q_1} \bowtie \boldsymbol{Q_2}) \cup (\boldsymbol{Q_3} \bowtie \boldsymbol{Q_4}).$$

Note that, when $\bar{\boldsymbol{Q}}_1$ and $\bar{\boldsymbol{Q}}_2$ cannot be combined by a *UNION* clause, that is, when they do not meet the conditions defined in Section 4.3.4, then the **Federated Translation Algorithm** will generate only one of the queries, $\bar{\boldsymbol{Q}}_1$ or $\bar{\boldsymbol{Q}}_2$.

4.3.6. Defining the *TARGET* clause of the Federated SPARQL Query

Let $\boldsymbol{Q}$ be the federated query with *TARGET* clause $\boldsymbol{S_Q}$ and *WHERE* clause $\boldsymbol{W_Q}$. The construction of $\boldsymbol{S_Q}$ consists mainly in the computation of a subset $\boldsymbol{Var(S_Q)}$ of the set of variables $\boldsymbol{Var(W_Q)}$ present in $\boldsymbol{W_Q}$. The computation of $\boldsymbol{Var(S_Q)}$ depends on $\boldsymbol{W_Q}$ and the coverage of the keywords set K.

The different situations that can occur are explained below.

Federated Query with a WHERE clause without external join triple patterns or UNION clauses

The first case occurs when the **Federated Translation Algorithm** creates a single local query $\boldsymbol{Q_1}$. Then, the federated SPARQL query $\boldsymbol{Q}$ will be $\boldsymbol{Q_1}$, with an additional *"SERVICE SILENT"* clause to query the target dataset, and $\boldsymbol{Var(S_Q)}$=$\boldsymbol{Var(S_{Q1})}$.

Federated Query with a WHERE clause with external join triple patterns, but without UNION clauses

The second case occurs when, for each pair of local queries Q_i and Q_j used to compose the federated SPARQL query $\boldsymbol{Q}$, there is an external join (generate either by an *object property* or by a *sameAs* definition). In this case, the *WHERE* clause of $\boldsymbol{Q}$ will be of the form $\boldsymbol{W_Q} = \bowtie_i \boldsymbol{Q_i}$.

To compute the set of variables $\boldsymbol{Var(S_Q)}$ of the *TARGET* clause of $\boldsymbol{Q}$, a greedy strategy is used, based on the score values of the subqueries, and taking into account the coverage of the keywords set K by the variables in $\boldsymbol{Var(S_Q)}$.

Let $\boldsymbol{CQ} = \{Q_1, \ldots, Q_m\}$ be the set of computed local queries.

The strategy starts with $\boldsymbol{Var(S_Q)} = \emptyset$ and a set of *covered keywords* $K' = \emptyset$.

Assume that the subquery $\boldsymbol{Q_i}$ $(1 \leq i \leq m)$ has the highest value score. Then, the variables in $\boldsymbol{Var(S_{Qi})}$ are added to $\boldsymbol{Var(S_Q)}$, and the keywords covered by $\boldsymbol{Var(S_{Qi})}$ to K', as mentioned in Section 4.3.2.

If $K = K'$ or all subqueries have been analyzed, the process stops.

Otherwise, the next subquery $\boldsymbol{Q_j}$ in decreasing score value order is analyzed and, if there is a variable $v_j \in \boldsymbol{Var(S_{Qj})}$ such that v_j covers a set of keywords $K_j \subseteq K$, and there is a keyword $k \in K_j$ and $k \notin K'$, then v_j is added to $\boldsymbol{Var(S_Q)}$ and k to K'.

Federated Query with a WHERE clause with only UNION clauses

The third situation occurs when the *WHERE* clause $\boldsymbol{W_Q}$ is of the form $\boldsymbol{W_Q}$=$\boldsymbol{Q_1} \cup \ldots \cup \boldsymbol{Q_n}$. That is, the *WHERE* clause $\boldsymbol{W_Q}$ of the federated query $\boldsymbol{Q}$ is composed entirely of *UNION* clauses.

As an example, suppose that a pair of local queries Q_1 and Q_2 satisfy the *UNION* conditions defined in the Section 4.3.4, so that the final SPARQL query $\boldsymbol{Q}$ is given by $\boldsymbol{Q} = Q_1 \cup Q_2$. Assume that $\boldsymbol{S_{Q1}} = \{v_1, \ldots, v_m\}$, $\boldsymbol{S_{Q2}} = \{w_1, \ldots, w_m\}$, and

there is a permutation π of 1,…,*m* such that each pair of variables v_i and $w_{\pi(i)}$ map to the same mediated schema element. Then, a new variable u_i is created to bind the results of variables v_i and $w_{\pi(i)}$ and the *TARGET* clause $\boldsymbol{S_Q}$ is composed by the bind variables $u_1,\ldots,u_m$.

Federated Query with All Elements in WHERE Clause

The last situation is when the *WHERE* clause $\boldsymbol{W_Q}$ is of the form $\boldsymbol{W_Q} = \cup\bar{\boldsymbol{Q}}_{\boldsymbol{j}}$, i.e. the *WHERE* clause of $\boldsymbol{Q}$ is composed of different types of external joins as well as *UNION* clauses. In this case, the strategy for choosing the variables is a bit more complex and is based on the composition of $\boldsymbol{W_Q}$.

As an example, suppose that the federated SPARQL query $\boldsymbol{Q}$ is of the form $\boldsymbol{Q} = (\boldsymbol{Q_1} \bowtie \boldsymbol{Q_2}) \cup (\boldsymbol{Q_3} \bowtie \boldsymbol{Q_4})$, that is, subqueries $\boldsymbol{Q_1}$ and $\boldsymbol{Q_2}$ are joined by an external join, as are the subqueries $\boldsymbol{Q_3}$ and $\boldsymbol{Q_4}$. We assume that the sets of variables $\boldsymbol{Var(S_{Q1 \bowtie Q2})}$ and $\boldsymbol{Var(S_{Q3 \bowtie Q4})}$ cover the keywords set *K*. Suppose that $\boldsymbol{S(Q_1 \bowtie Q_2)}=\{v_1,\ldots,v_n\}$ and $\boldsymbol{S(Q_3 \bowtie Q_4)}=\{w_1,\ldots,w_n\}$ and that these two sets of variables satisfy Conditions (1) and (2) defined in Section 4.3.4. To meet Condition (3), there must be a permutation π of 1,…,*n* such that each pair of variables v_i and $w_{\pi(i)}$ map to the same mediated schema element, as defined in 4.3.4. Then, the results of the queries $\boldsymbol{Q_1 \bowtie Q_2}$ and $\boldsymbol{Q_3 \bowtie Q_4}$ can be combined via an *UNION* clause, and a new variable u_i is created to bind the results of variables v_i and $w_{\pi(i)}$, for i=1,…,*n*. The set of variables $\boldsymbol{Var(Q)}$ is composed by new bind variables $u_1,\ldots,u_n$.

5
Experiments

This chapter presents the experiments performed to test the performance of an implementation of the **Federated Translation Algorithm**. Section 5.1 describes the configuration of the experiments. Section 5.2 shows the results obtained with selected datasets. Finally, section 5.3 summarizes and discusses the results.

5.1 Data Configuration

In order to test an implementation of the **Federated Translation Algorithm**, we selected data from three free RDF datasets: DBpedia, DrugBank, and Kegg Drug. Each one is exposed in different SPARQL endpoints following the steps that we explain in Appendix I.

To run the experiments using these RDF datasets, it was necessary to populate the tables allocated in the **Storage Component** and **Mediated Schema Component**. The details are explained in the following subsections.

5.1.1. *DBpedia* RDF Dataset Setup

DBpedia is a crowd-sourced community effort to extract structured information from Wikipedia and make this information available on the Web. Localized versions of DBpedia are available in 125 languages. The English version of the DBpedia knowledge base describes 4.58 million objects, out of which 4.22 million are classified in a consistent ontology, including 1,445,000 persons, 735,000 places (including 478,000 populated places), 411,000 creative works (including 123,000 music albums, 87,000 films and 19,000 video games), 241,000 organizations (including 58,000 companies and 49,000 educational institutions), 251,000 species and 6,000 diseases[7].

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

For our experiments, the classes **drug** (http://dbpedia.org/ontology/Drug) and **enzyme** (http://dbpedia.org/ontology/Enzyme) from the named graph http://dbpedia.org/resource/classes# were chosen, with the following schema.

Schema		
Properties	**Values**	
	Drug	**Enzyme**
rdf:type	owl:Class	owl:Class
rdfs:label	drug (en)	enzyme (en)

Table 5 - Schema of classes in DBpedia data source

The instances of the selected classes were obtained by querying the DBpedia SPARQL Endpoint[8] executing the query below.

```
SELECT DISTINCT ?s
WHERE {
  ?s rdf:type ?o .
  FILTER ( ?o IN (<http://dbpedia.org/ontology/Drug>,
                  <http://dbpedia.org/ontology/Enzyme>))
}
```

The query result and the triples that represent the RDF schema were exported into an N-triples file. This file was transformed into an SQL file with the objective of inserting each triple into the RDF model named **`dbpedia_drug_mat`** created in the Oracle Server using the **`rdf_data`** table. A sample fragment of the file with the transformation result is as follows:

```
INSERT INTO rdf_data (TRIPLE) VALUES (
       SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT', '<http://dbpedia.org/Drug>',
                        '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
                        '<http://www.w3.org/2002/07/owl#Class>'));
INSERT INTO rdf_data (TRIPLE) VALUES (
       SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT', '<http://dbpedia.org/Drug>',
                        '<http://www.w3.org/2000/01/rdf-schema#label>',
                        '"drug"'));
INSERT INTO rdf_data (TRIPLE) VALUES (
       SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT', '<http://dbpedia.org/Enzyme>',
                        '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
                        '<http://www.w3.org/2002/07/owl#Class>'));
INSERT INTO rdf_data (TRIPLE) VALUES (
       SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT', '<http://dbpedia.org/Enzyme>',
                        '<http://www.w3.org/2000/01/rdf-schema#label>',
                        '"enzyme"'));
INSERT INTO rdf_data (TRIPLE) VALUES (
       SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT',
                        '<http://dbpedia.org/Drug/Amoxicillin>',
                        '<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>',
                        '<http://dbpedia.org/Drug>'));
      ...
```

[8] http://dbpedia.org/sparql/

In this case, to populate the auxiliary tables allocated in the **Storage Component**, we followed the process:

- Insert the instance label, for each instance:

```
INSERT INTO rdf_data (triple)
SELECT SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT',
                subj$rdfterm, pred$rdfterm,
                '"' || upper(substr(lb, 1,1))||substr(lb, 2) || '"')
FROM(SELECT subj$rdfterm, pred$rdfterm,
      REGEXP_REPLACE(SUBSTR(obj, - INSTR(REVERSE(obj), '/') + 1),
      '([[:lower:]])([[:upper:]])', '\1 \2') lb
      FROM TABLE(SEM_MATCH(
         'CONSTRUCT{ ?s rdfs:label ?s}
          WHERE { ?s rdf:type ?c.
                  ?c rdf:type owl:Class
                }',
SEM_MODELS('DBPEDIA_DRUG_MAT'),NULL,NULL,NULL)));
```

- Create a property named ***name*** having as values the labels of classes instances:

```
INSERT INTO RDF_DATA (TRIPLE)
VALUES ( SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT',
         '<http://dbpedia.org/name>', 'rdfs:label', '"Name"' ));

INSERT INTO RDF_DATA (TRIPLE)
SELECT SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT',
                        subj$rdfterm, pred$rdfterm, obj$rdfterm)
FROM TABLE(SEM_MATCH(
'CONSTRUCT { ?s <http://dbpedia.org/name> ?lb }
 WHERE {
  { SELECT DISTINCT ?s ?lb
    WHERE { ?c rdf:type owl:Class .
            ?s rdf:type ?c .
            ?s rdfs:label ?lb
  } }
}',SEM_MODELS('DBPEDIA_DRUG_MAT'),NULL,NULL,NULL));
```

- Assign `rdf:Property` and `rdf:domain` to the new property:

```
INSERT INTO RDF_DATA (TRIPLE)
SELECT SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT',
                         subj$rdfterm, pred$rdfterm, obj$rdfterm)
FROM TABLE(SEM_MATCH(
 'CONSTRUCT { ?p rdf:type rdf:Property }
  WHERE { SELECT distinct ?p
          WHERE { ?s ?p ?o .
                  ?s rdf:type ?c .
                  ?c rdf:type rdfs:Class .
                  FILTER ( ?p != rdf:type )
                  FILTER ( ?p != rdfs:label )} }',
SEM_MODELS('DBPEDIA_DRUG_MAT'),NULL,NULL,NULL));
```

```
INSERT INTO RDF_DATA (TRIPLE)
SELECT SDO_RDF_TRIPLE_S('DBPEDIA_DRUG_MAT',
                        subj$rdfterm, pred$rdfterm, obj$rdfterm)
FROM TABLE(SEM_MATCH(
 'CONSTRUCT { ?p rdfs:domain ?c}
  WHERE { SELECT distinct ?p ?c
          WHERE { ?p rdf:type rdf:Property .
                  ?s ?p ?o .
                  ?s rdf:type ?c . } }',
SEM_MODELS('DBPEDIA_DRUG_MAT'),NULL,NULL,NULL));
```

5.1.2. *DrugBank* RDF Data Setup

The DrugBank database is a unique bioinformatics and cheminformatics resource that combines detailed drug (i.e. chemical, pharmacological and pharmaceutical) data with comprehensive drug target (i.e. sequence, structure, and pathway) information. The database was created following a given schema[9].

We downloaded the N-triples data file available in the **D2R Server publishing the DrugBank Database**[10], with an accessible SPARQL Endpoint[11]. The N-triples file was transformed into an SQL file and the triples were inserted into the RDF model named **`drugs_mat`** created in the Oracle Server using the **`rdf_data`** table.

Figure 9 shows a partial RDF schema diagram. The diagram depicts all 5 classes (rectangles), all 6 object properties (single arrows, starting on the domain and ending on the range), with their names omitted to avoid cluttering the diagram.

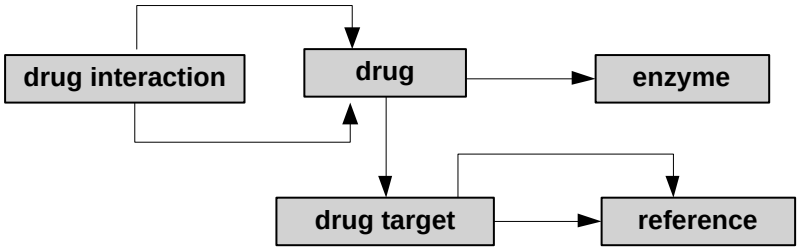


Figure 9. RDF Schema of **DrugBank**

Table 6 and Table 7 expose the statistics of the RDF dataset, with 765,936 triples and 19,770 class instances representing 4,472 drugs instances, 10,096 drug interaction entries, 53 enzymes, 96 references, and 4,553 drug target. The dataset

[9] http://download.bio2rdf.org/release/3/drugbank/drugbank.schema.owl
[10] http://wifo5-03.informatik.uni-mannheim.de/drugbank/
[11] http://wifo5-03.informatik.uni-mannheim.de/drugbank/sparql

has 118 datatype properties and a large number of RDF links to other Linked Data sources (59,661).

Classes (label)	# Instances
Enzymes	53
Reference	96
Drug	4.772
drug interaction	10.096

Table 6 - Statistics of **DrugBank** Classes

Triple Type	# Triples
Class declarations	5
Datatype property declarations	118
Class instances	19.570
Object property declarations	6
RDF links to other sources	59.661
Total number of triples	765.936

Table 7 - Statistics of **DrugBank** RDF Data

5.1.3. *Kegg Drug* RDF Data Setup

The Kyoto Encyclopedia of Genes and Genomes (Kegg) is a collection of databases and resources for studying high-level functions and utilities of the biological systems. These databases are broadly categorized into systems information, genomic information, chemical information and health information (Figure 10).

Category	Database	Content
Systems information	KEGG PATHWAY	KEGG pathway maps
	KEGG BRITE	BRITE functional hierarchies
	KEGG MODULE	KEGG modules of functional units
	KEGG ORTHOLOGY	KEGG Orthology (KO) groups
Genomic information	KEGG GENOME	KEGG organisms with complete genomes
	KEGG GENES	Gene catalogs of complete genomes
	KEGG SSDB	Sequence similarity database for GENES
Chemical information	KEGG COMPOUND	Metabolites and other small molecules
	KEGG GLYCAN	Glycans
	KEGG REACTION	Biochemical reactions
	KEGG RPAIR	Reactant pair chemical transformations
	KEGG RCLASS	Reaction class defined by RPAIR
	KEGG ENZYME	Enzyme nomenclature
Health information	KEGG DISEASE	Human diseases
	KEGG DRUG	Drugs
	KEGG DGROUP	Drug groups
	KEGG ENVIRON	Crude drugs and health-related substances

Figure 10. Main categories for Kegg databases

The Kegg data in the N-triples format were downloaded from the available FTP Kegg[12]. The N-triples file was transformed into an SQL file and the triples were inserted into the RDF model named **`kegg_mat`** created in the Oracle Server using the **`rdf_data`** table.

Figure 11 presents the schema of the Kegg RDF triples. The RDF dataset has 4 classes (represented by rectangles) and 4 object properties (single arrows, starting on the domain and ending on the range), with their names omitted to avoid cluttering the diagram.

[12] ftp://ftp.genome.jp/pub/kegg/medicus/drug/

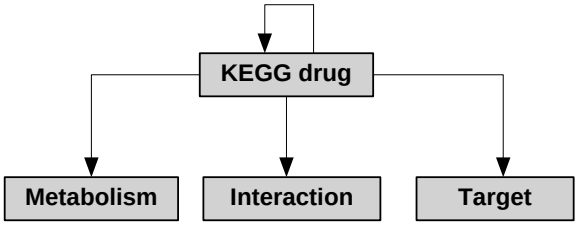


Figure 11. RDF Schema of **Kegg Drug**

Table 8 and Table 9 show the statistics about the Kegg RDF dataset with 21,873 class instances representing 9,773 Kegg drug instances, 996 metabolism entries, 4,808 interactions, and 6,296 targets. The RDF graph contains 713,737 triples and 40 datatype property declarations.

Classes (label)	# Instances
KEGG drug	9.773
Metabolism	996
Interaction	4.808
Target	6.296

Table 8 - Statistics of **Kegg** Classes

Triple Type	# Triples
Class declarations	4
Datatype property declarations	40
Class instances	21.873
Object property declarations	4
Total number of triples	713.737

Table 9 - Statistics of **Kegg** RDF Data

5.1.4. Common Settings

In order to fill in all auxiliary tables allocated in the **Storage Component,** we need to insert in the RDF graphs additional information about their respective schemas. The missing information is mainly related to the cardinalities of classes and properties, the definition of properties groups, and the property values that will be indexed.

This can be accomplished by running, for the respective RDF models (replacing the real values highlighted in bold), the following SQL queries (see the queries details in Appendix III):

Q1. Insert the order of the classes by the cardinality
Q2. Insert the order of the properties by the cardinality
Q3. Indexing **TRUE** the properties with STRING type
Q4. Insert the default group order
Q5. Insert all properties in the "default" group
Q6. Insert the triples with the pattern form
(`property rdfs:range owl:ObjectProperty`)

5.1.5. Mediated Schema Composition and Setting

The allocated tables in the **Mediated Schema Components** will be populated with the mediated schema information. Figure 12 depicts the links between the three data sources (represented by dashed type single arrows, when the dash type varies with the link type).

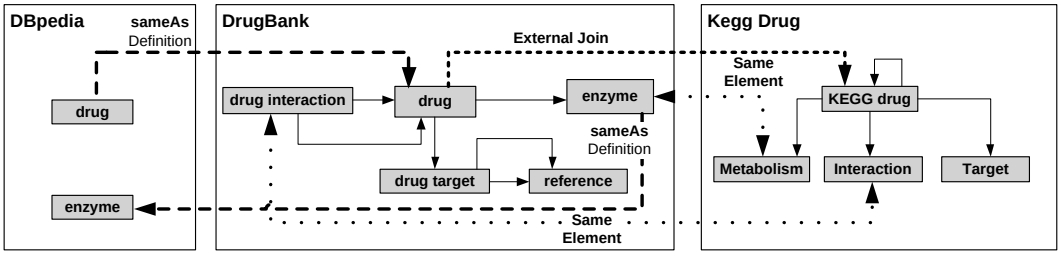


Figure 12. Mediated Schema of **DBpedia**, **DrugBank** and **Kegg Drug**

As is defined in Table 10, all drug information in DBpedia is connected with the class **drug** in DrugBank, and the enzyme instances in DrugBank are joined with the class **enzyme** in DBpedia, both by a *sameAs* definition.

SameAsTable					
Endpoint Source	**Class Source**	**Properties Source**	**Endpoint Destination**	**Class Destination**	**Properties Destination**
DBpedia	Drug	*Name*	DrugBank	drug	*rdfs:label*
DrugBank	enzyme	*rdfs:label*	DBpedia	enzyme	*Name*

Table 10 - SameAsTable populated with the *sameAs* definition in the selected data sources

The object property *drugbank:keggDrugId* has as domain the class **drug** in DrugBank and the class **Kegg Drug** in RDF Kegg as range, according Table 11 shows below.

ExternalObjectPropertyTable				
Endpoint Source	**Class Source**	**Object Property**	**Endpoint Destination**	**Class Destination**
DrugBank	Drug	*keggDrugId*	Kegg	Kegg Drug

Table 11 - ExternalObjectPropertyTable populated with the external joins in the selected data sources

The instances of classes ***enzyme*** (**DrugBank**) and ***Metabolism*** (**Kegg Drug**) were mapped to the same mediated schema element to be able to construct *UNION* clauses in the *TARGET* clause of a federated SPARQL query when these classes are involved in the federated query. The same happens with the instances of the classes ***drug interaction*** (**DrugBank**) and ***Interaction*** (**Kegg Drug**) (see Table 12).

MapElementTable			
Mediated Schema Element	**Local Schema Class**	**Local Schema Property**	**Source**
Enzyme	Enzyme	*NULL*	DrugBank
Enzyme	Metabolism	*NULL*	Kegg Drug
Drug Interaction	drug interaction	*NULL*	DrugBank
Drug Interaction	Interaction	*NULL*	Kegg Drug

Table 12 - MapElementTable populated with the elements maps of the Mediated Schema

5.2 Experiments with Selected Data

We ran a suite of keyword-based queries to assess the performance of the **Federated Translation Algorithm**. The keyword-based queries were selected to show the different compositions that the *WHERE* clause of the federated query can take, and the coverage of the set of keywords by the variables in the *TARGET* clause. In the queries, the IRIs of the different SPARQL endpoints will be replaced by the name of data source, as follows:

- **DBpedia** SPARQL Endpoint: **dbpedia**
- **DrugBank** SPARQL Endpoint: **drugbank**
- **Kegg Drug** SPARQL Endpoint: **kegg**

5.2.1. Translated Queries over a Single SPARQL Endpoint

The first case covers keyword-based queries for which the **Federated Translation Algorithm** generates SPARQL queries over a single data source.

The example keyword-based query K_1 = "*indication for backache*" expresses the search "drugs that are indicated for backache". The first stage of the translation algorithm eliminates the stop word "*for*", matches the keyword "*indication*" with the label of property ***Indication*** of the class ***drug*** in **DrugBank** dataset, and the keyword "*backache*" with the values of the same property ***Indication***. This stage then returns a single local query. The second stage of the algorithm generates the following SPARQL query.

```
1.  SELECT ?C_0_0 ?P_0_0
2.  WHERE{
3.    SERVICE SILENT <drugbank> {
4.      ?I_C_0_0 rdf:type <http://www4.wiwiss.fu-
5.  berlin.de/drugbank/resource/drugbank/drugs> .
6.      ?I_C_0_0 <http://www4.wiwiss.fu-
7.  berlin.de/drugbank/resource/drugbank/indication> ?P_0_0
8.      FILTER <http://xmlns.oracle.com/rdf/textContains>(?P_0_0,
9.  "fuzzy({backache}, 70, 1)", 1)
10.     ?I_C_0_0 rdfs:label ?C_0_0
11. } }
```

FQ 1 - Federated SPARQL query generated by responding to K_1

The query only accesses the **DrugBank** SPARQL Endpoint via the *SERVICE SILENT* clause in line 3. The **Centralized Translation Algorithm** returns the local query in lines 4 to 10. It founds a match with the label property ***Indication*** and generates the triple pattern in lines 6 and 7. Since the domain of property ***Indication*** is the class ***drug***, the variable *I_C_0_0* will bind to instances of this class (lines 4 and 5). The *FILTER* declaration in lines 8 and 9 matches the keyword "*backache*" with the value in *P_0_0*, using the Oracle fuzzy matching function with the appropriate parameters (70 and 1). Line 10 translates the URI in *I_C_0_0* to a label, which is hopefully user-friendly, and binds it to *C_0_0*. The *TARGET* clause in line 1 returns a table with the binding variables *C_0_0* and *P_0_0*, the same variables present in the *TARGET* clause of the local query.

5.2.2. Translated Queries with only *external joins* in the *WHERE* clause

The second case covers keyword-based queries for which the **Federated Translation Algorithm** generates SPARQL queries that retrieve data from different data sources linked by *external joins*. In more detail, the **Centralized Translation Algorithm** generates local subqueries, and the **Federated Translation Algorithm** synthesizes a federated SPARQL query whose *WHERE* clause contains *external joins*, generated by *sameAs* definitions or external object properties.

The example keyword-based query K_2 = "*drug target*' *of ibuprofen*" expresses the search "targets information (i.e. sequence, structure, and pathway) associated to ibuprofen". The first stage of the translation algorithm eliminates the stop word "*of*", matches the keyword "*drug target*" with the label of class ***drug***

target in **DrugBank** dataset, and the keyword "*ibuprofen*" with the values of an instance of the class ***Drug*** in **DBpedia** dataset and with an instance of class ***drug*** in **DrugBank** source. This stage then returns two local queries. The second stage of the algorithm generates the following federated SPARQL query.

```
1.  SELECT  ?C_0_0 ?C_1_1
2.  WHERE{
3.   SERVICE SILENT <dbpedia>{
4.      ?I_C_0_0 rdf:type <http://dbpedia.org/Drug> .
5.      ?I_C_0_0 <http://dbpedia.org/name> ?sA_1_0
6.      FILTER <http://xmlns.oracle.com/rdf/textContains>(?C_0_0,
7.  "{\"ibuprofen\"}", 0)
8.      ?I_C_0_0 rdfs:label ?C_0_0    }
9.   SERVICE SILENT <drugbank>{
10.       ?I_C_1_0 <http://www4.wiwiss.fu-
11.  berlin.de/drugbank/resource/drugbank/target> ?I_C_1_1 .
12.       ?I_C_1_0 rdfs:label ?sA_1_0
13.       FILTER <http://xmlns.oracle.com/rdf/textContains>(?sA_1_0,
14.  "{\"ibuprofen\"}", 0)
15.       ?I_C_1_1 rdfs:label ?C_1_1    }
16. }
```

FQ 2 - Federated SPARQL query generated by responding to K_2

The **Centralized Translation Algorithm** returns two local queries. Lines 3 to 8 show the local query to access the **DBpedia** SPARQL Endpoint, and lines 9 to 15 show the local query to access the **DrugBank** SPARQL Endpoint, both via a *SERVICE SILENT* clause. In what follows, we respectively use the terms **DBpedia** query and **DrugBank** query to refer us to these subqueries.

Related to the **DBpedia** query, the triple pattern in line 4 binds instances of class ***Drug*** to the variable *I_C_0_0*, due to the match with label of the class. The *FILTER* declaration in lines 6 and 7 matches the keyword "*ibuprofen*" with the value in *C_0_0*, using the Oracle *textContains* matching function. Line 8 translates the URIs in *I_C_0_0* to labels, binding them to *C_0_0*. Although not reflected in the local query, the *TARGET* clause is composed only of the variable *C_0_0*, because it covers the keyword "*ibuprofen*".

As for the **DrugBank** query, the (local) object property *drugbank:target* generates the triple pattern in lines 10 and 11. Note that, since the domain of the object property is the class ***drug*** and the range is the class ***drug target***, variables *I_C_1_0* and *I_C_1_1* bind to instances of these classes, respectively. Hence, it is not necessary to include triple patterns that force *I_C_1_1* to be of type ***drug target***, and *I_C_1_0* to be of type ***drug***. The variable *sA_1_0* translates the URIs in

I_C_1_0 to labels. As seen in **DBpedia** query, the *FILTER* declaration in lines 13 and 14 matches the keyword "*ibuprofen*" with the value in *sA_1_0*. Although it is not reflected, the **Centralized Translation Algorithm** constructs the *TARGET* clause of the local query with the variables *sA_1_0* and *C_1_1*. The variable *sA_1_0* covers the keyword "*ibuprofen*", and *C_1_1* covers "*drug target*".

DBpedia query and **DrugBank** query are joined by a *sameAs* definition, as Table 10 defines, so the **Federated Translation Algorithm** generates the triple patterns in lines 5 and 12. The variable *sA_1_0* binds, with a perfect matching, the properties values of the respective properties *http://dbpedia.org/name* and *rdfs:label* joining the instances class bound by the variable *I_C_0_0* with the instances class that the variable *I_C_1_0* binds.

The *TARGET* clause in line 1 returns a table with the binding variables *C_0_0* and *C_1_1*. To select the variables in the *TARGET* clause was followed the strategy described in the second situation of Section 4.3.6. As the **DBpedia** query is the query with the highest score value, then *C_0_0* belongs to the *TARGET* clause. The variable *C_1_1* is also added to the *TARGET* clause to cover the keyword-based query K_2.

5.2.3. Translated Queries with only *UNIONs* in the *WHERE* clause

The third case covers keyword-based queries for which the **Federated Translation Algorithm** generates SPARQL queries whose *WHERE* clauses contain only *UNION* elements.

The example keyword-based query K_3 = "*interaction*" expresses a search about the interactions of drugs. The first stage of the translation algorithm matches the keyword "*interaction*" with the labels of the classes ***drug interactions*** of **DrugBank** and ***Interaction*** of **Kegg Drug**. This stage then returns two local queries. The second stage of the algorithm generates the following federated SPARQL query.

```
1.  SELECT  ?U_0
2.  WHERE{
3.  { SELECT  (?C_1_0 AS ?U_0)
4.      WHERE{
5.        SERVICE SILENT <drugbank>{
6.          ?I_C_1_0 rdf:type <http://www4.wiwiss.fu-
7.  berlin.de/drugbank/resource/drugbank/drug_interactions> .
8.          ?I_C_1_0 rdfs:label ?C_1_0
9.    } }
10.  UNION{
11.    SELECT  (?C_0_0 AS ?U_0)
12.      WHERE{
13.       SERVICE SILENT <kegg>{
14.        ?I_C_0_0 rdf:type <http://bio2rdf.org/kegg:Interaction> .
15.        ?I_C_0_0 rdfs:label ?C_0_0
16. } } } }
```

FQ 3 - Federated SPARQL query generated by responding to K_3

The first subquery (**DrugBank** query), in lines 3 to 9, accesses the **DrugBank** SPARQL Endpoint and the second subquery (**Kegg** query), in lines 11 to 16, accesses the **Kegg Drug** SPARQL Endpoint. The queries results are combined by a *UNION* clause (line 10), since these queries satisfy the conditions defined in Section 4.3.4.

In the **DrugBank** query, the triple pattern in lines 6 and 7 binds instances of the class ***drug interaction*** to the variable *I_C_1_0*, due to the match with the label of the class. Line 8 translates the URIs in *I_C_1_0* to labels, binding them to *C_1_0*. The *TARGET* clause is composed of the variable *C_1_0*, binding their values to the new variable *U_0*.

In the **Kegg** query, the triple pattern in line 14 binds instances of class ***Interaction*** to the variable *I_C_0_0*, and line 15 translates the URIs in *I_C_0_0* to labels, binding them to *C_0_0*. The *TARGET* clause is composed of the variable *C_0_0*, which also binds their values to the new variable *U_0*.

The *TARGET* clause of the federated SPARQL query in line 1 returns a table with the binding variable *U_0* created to combine the values that the variables *C_1_0* and *C_0_*0 bind to.

5.2.4. Translated Queries with All Elements in the *WHERE* clause

The last case covers keyword-based queries for which the **Federated Translation Algorithm** generates federated SPARQL queries whose *WHERE* clause contains both *external joins* and *UNION* clauses.

The example keyword-based query K_4 = "*interaction with enzyme and metabolism*" expresses a search about drug interactions and the enzymes of these drugs. The first stage of the translation algorithm eliminates the stop words "*with*" and "*and*", matches the remaining keywords, and returns three local queries. The second stage of the algorithm generates the following federated SPARQL query.

```
1.  SELECT  ?U_0 ?U_1
2.  WHERE {
3.   { SELECT  (?C_2_0 AS ?U_0) (?sA_1_0 AS ?U_1)
4.       WHERE {
5.          SERVICE SILENT <drugbank>{
6.             ?I_C_2_1 <http://www4.wiwiss.fu-
7.  berlin.de/drugbank/resource/drugbank/enzyme> ?I_C_2_2 .
8.             ?I_C_2_0 <http://www4.wiwiss.fu-
9.  berlin.de/drugbank/resource/drugbank/interactionDrug1> ?I_C_2_1 .
10.            ?I_C_2_2 rdfs:label ?sA_1_0 .
11.            ?I_C_2_0 rdfs:label ?C_2_0 .   }
12.           SERVICE SILENT <dbpedia>{
13.            ?I_C_0_0 rdf:type <http://dbpedia.org/Enzyme> .
14.            ?I_C_0_0 <http://dbpedia.org/name> ?sA_1_0 .
15.   } } }
16.  UNION
17.   { SELECT  (?C_1_0 AS ?U_0) (?C_1_1 AS ?U_1)
18.     WHERE{
19.       SERVICE SILENT <kegg>{
20.         ?I_C_1_0 <http://bio2rdf.org/kegg:metabolism> ?I_C_1_2 .
21.         ?I_C_1_0 <http://bio2rdf.org/kegg:interaction> ?I_C_1_1 .
22.         ?I_C_1_1 rdfs:label ?C_1_0 .
23.         ?I_C_1_2 rdfs:label ?C_1_1   }
24. } } }
```

FQ 4 - Federated SPARQL query generated by responding to K_4

The query in lines 5 to 11 accesses the **DrugBank** SPARQL Endpoint, the query in lines 12 to 15 accesses the **DBpedia** SPARQL Endpoint, and the query in lines 17 to 23 accesses the **Kegg** SPARQL Endpoint, all via a *SERVICE SILENT* clause. Below, we use the terms **DrugBank** query, **DBpedia** query, and **Kegg** query to refer us the respective subqueries.

In the **DrugBank** query, the keywords "*enzyme*" and "*interaction*" respectively match the label of classes ***enzyme*** and ***drug interactions***, then the triple patterns in lines 6 to 9 represent the path leaving from ***drug interaction*** to ***enzyme*** that goes through the class ***drug***. The object property *drugbank:enzyme* joins the classes ***drug*** and ***enzyme***, which generates the triple pattern in lines 6 and 7. Note that, since the domain of the object property is the class ***drug*** and the range is the class ***enzyme***, variables *I_C_2_1* and *I_C_2_2* respectively bind to instances of these classes. Hence, it is not necessary to include triple patterns that force *I_C_2_1* to be of type ***drug***, and *I_C_2_2* to be of type ***enzyme***. The link between the classes

drug and ***drug interactions*** is through the object property *drugbank:interactions*, which is reflected in the triple pattern in lines 8 and 9. Note that, since the domain of the object property is the class ***drug interaction*** and the range is the class ***drug***, the variables *I_C_2_0* and *I_C_2_1* bind to instances of these classes, respectively. The variables *sA_1_0* and *C_2_0*, in lines 10 and 11, translate the URIs in *I_C_2_2* and *I_C_2_0* to labels. Although not reflected in the local query, the *TARGET* clause is composed of the variables *C_2_0* and *sA_1_0*, which cover the set of keywords in K_4 (excluding the stop words).

In the **DBpedia** query, the triple pattern in line 13 binds instances of class ***Enzyme*** to the variable *I_C_0_0*, since the keyword "*enzyme*" matches with the label of the class. The triple pattern in line 14 reflects the *sameAs* definition with the class ***enzyme*** in **DrugBank** query; also the variable *sA_1_0* translates the URIs in *I_C_0_0* to labels. The *TARGET* clause is composed of the variable *sA_1_0*, covering the keyword "*enzyme*".

Related to **Kegg** query, the keywords "*interaction*" and "*metabolism*" match with the label of the classes ***Interaction*** and ***Metabolism***, respectively. These classes are connected via the class ***Kegg Drug***, which is the root of the tree that they form. Then, the **Centralized Translation Algorithm** generates the triple patterns in lines 20 and 21. The triple pattern in line 20 represents the join between the classes ***Kegg Drug*** and ***Metabolism*** via the object property *kegg:metabolism*, since the domain of the object property is the class ***Kegg Drug*** and the range is the class ***Metabolism***; the variables *I_C_1_0* and *I_C_1_2* will respectively bind to instances of these classes. The object property *kegg:interaction* in lines 21 joins the domain class ***Kegg Drug*** to the range class ***Interaction***, where the variables *I_C_1_0* and *I_C_1_1* bind to instances of their respective classes. The triple patterns in lines 10 and 11 translate the URIs in *I_C_1_1* and *I_C_1_2* to labels, binding them to variables *C_1_0* and *C_1_1*. The *TARGET* clause of the local subquery in line 17 is composed of the variables *C_1_0* and *C_1_1*, which cover K_4 (excluding the stop words) and bind their values to the created variables *U_0* and *U_1*, respectively.

Let ***Q*** be the SPARQL query synthesized by the **Federated Translation Algorithm**, then ***Q*** is given by

$$\boldsymbol{Q} = (\textbf{DrugBank query} \bowtie \textbf{DBpedia query}) \cup \textbf{Kegg query}.$$

Lines 3 to 15 correspond to the join between **DrugBank** query and **DBpedia** query. These queries are joined by a *sameAs* definition between the classes ***enzyme*** (**DrugBank** query) and ***enzyme*** (**DBpedia** query), as Table 10 records, which correspond to the triple patterns in lines 10 and 14. The variable *sA_1_0* binds, with a perfect matching, the properties values of the respective properties *rdfs:label* and *http://dbpedia.org/name*, joining the instances class bound by the variable *I_C_2_2* with the instances class bound by the variable *I_C_0_0*. The *TARGET* clause of this join query in line 3 is composed of the binding variables *C_2_0* and *sA_1_0*, which cover K_4 (excluding the stop words) and bind their results to the variables *U_0* and *U_1*, respectively. The *UNION* clause to combine the queries results is shown in line 16. The *TARGET* clause of ***Q*** in line 1 returns a table with the binding variables *U_0*, that binds values about the mediated schema element **'Drug Interaction'**, and *U_1*, which binds values about the mediated schema element **'Enzyme'**. Table 12 was used to compute these maps.

5.3 Discussion of the Results

The runtime to process the selected keyword-based queries and the response structure of each of the generated SPARQL federated queries are summarized in Table 13.

The results show that all queries were successfully executed in less than 4 seconds, which is quite reasonable, considering that the system returns 750 results as limit, the size of the datasets, and that the subqueries results come from different datasets allocated in a local network. The tests were performed in accordance with each of the discussed cases and the queries were synthesized following, for each situation, the corresponding strategy. The variables in the *TARGET* clause of each SPARQL query cover the respective set of keywords. These results suggest that the algorithm performs well to respond the keyword-based search over federated RDF graphs.

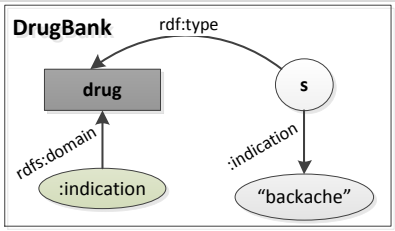
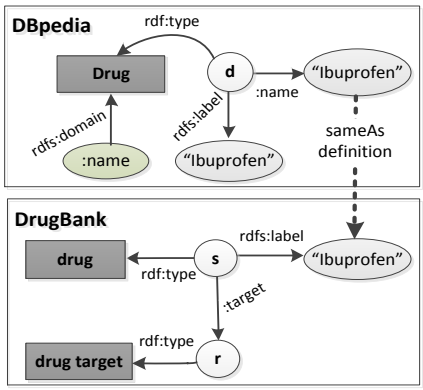
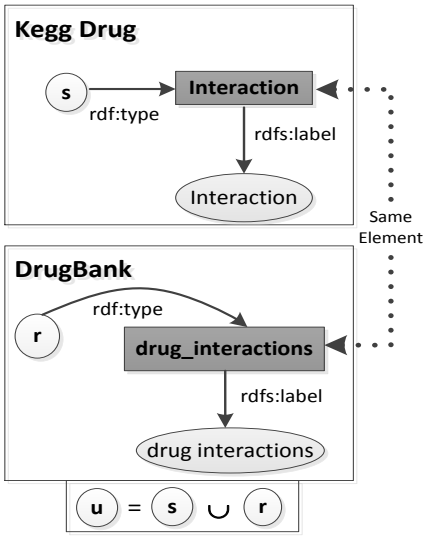
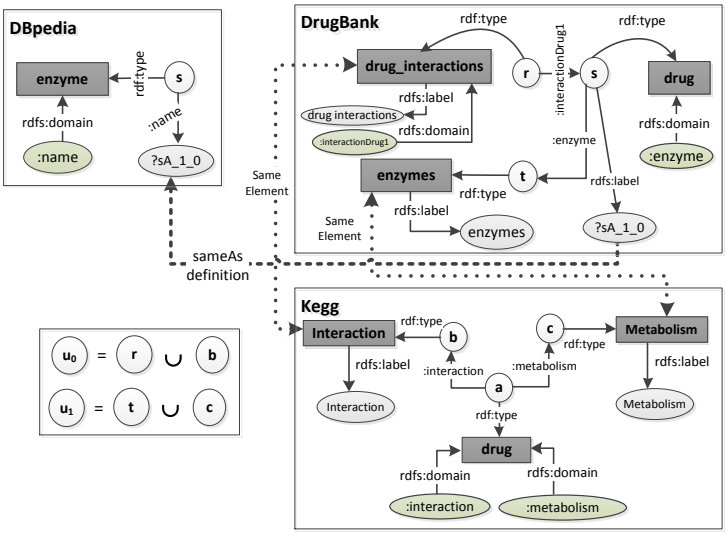
Query ID/ Keywords	Federated SPARQL Query Structure	Execution Time (s)
FQ 1 indication backache		0.34
FQ 2 'drug target' ibuprofen		1.32
FQ 3 interaction		1.54
FQ 4 interaction enzyme metabolism		3.55

Table 13 - Runtime to process sample keyword-based queries

6
Conclusions

In this work, we presented an algorithm, called **Federated Translation Algorithm**, to perform keyword search over federated RDF graphs by exploring their schemas. This algorithm extends the **Centralized Translation Algorithm** developed in (GARCÍA, *et al.*, 2017), which is used as part of Stage 1 of the federated algorithm. As the main objective was to extend the centralized algorithm to a federation of RDF datasets, we first analyzed what additional requirements would have to be incorporated to take into account the elements involved in a federation of RDF datasets. Then, we introduced an architecture to the system and described its components.

We detailed the design decisions to construct the federated SPARQL query, based on the existing relationships (called here *external joins*) between the local subqueries generated by the **Centralized Translation Algorithm**. We also defined the conditions to combine, with the help of the *UNION* clauses, the results of queries that have no external joins between them. We defined the composition of the *WHERE* clause of the federated SPARQL query and explained how the *TARGET* clause is constructed, according to the composition of the *WHERE* clause. Finally, we performed some experiments to test the performance of the proposed approach using three freely accessible RDF databases with joins between them.

The lessons learned were:

- The proposed algorithm generates queries with the following characteristics:
  - The local queries only access the data sources whose indexed data and metadata matched the keywords.
  - The variables in the *TARGET* clause of the federated SPARQL query cover a subset of the set of keywords submitted by the user.
- The experiments suggest that the proposed algorithm performs well for keyword-based search over federated RDF graphs.

As future work, we plan to test this solution in scenarios with a larger number of data sources, with RDF graphs that have more interconnections between them, and with more data and metadata. Furthermore, it is interesting to create a failure mechanism strategy to remove the *SILENT* reserved word from the federated queries and to handle exceptions in query execution, such as the timeout exceptions caused by out-of-service SPARQL endpoints or by large query answers. Finally, we plan to extend the current implementation to other federated RDF storage systems and to make the tool publicly available.

7
Bibliography

ACOSTA, M. et al. **ANAPSID An Adaptive Query Processing Engine for SPARQL Endpoints**. International Semantic Web Conference. Bonn, Germany: Springer Berlin Heidelberg. October, 2011. p. 18-34.

BIZER, C.; HEATH, T.; BERNERS-LEE, T. **Linked data: Principles and state of the art**. World wide web conference. 2008. p. 140.

BRICKLEY, D.; GUHA, R. V. **RDF Schema 1.1**, 25 February 2014. Available at: <https://www.w3.org/TR/rdf-schema/>.

BUIL-ARANDA, C. et al. **Federating queries in SPARQL 1.1 Syntax, semantics and evaluation**. Web Semantics: Science, Services and Agents on the World Wide Web, v. 18, n. 1, p. 1-17, 2013.

CYGANIAK, R.; WOOD, D.; LATHANER, M. **RDF 1.1 Concepts and Abstract Syntax**, 25 Febraury 2014. Available at: <https://www.w3.org/TR/rdf11-concepts/>.

DRAGAN, L. et al. **Converging Web and Desktop Data with Konduit**. Proc. of Scripting and Development for the Semantic Web Workshop. 2009. p. 40-51.

ELBASSUONI, S.; BLANCO, R. **Keyword search over RDF graphs**. Proceedings of the 20th ACM International Conference on Information and Knowledge Management. Glasgow, UK: ACM. 2011. p. 237-242.

GARCÍA, G. M. et al. **RDF Keyword-based Query Technology Meets a Real-World Dataset**. 20th International Conference on Extending Database Technology (EDBT). March 2017.

HARRIS, S.; SEABORNE, A. **SPARQL 1.1 Query Language**, 21 March 2013. Available at: <https://www.w3.org/TR/sparql11-query/>.

HUANG, J.; ABADI, D. J.; REN, K. **Scalable SPARQL querying of large RDF graphs**. Proceedings of the VLDB Endowment, v. 4, n. 11, p. 1123-1134, 2011.

MÖLLER, K.; DRAGAN, L.; AMBRUS, O. **A Visual Interface for Building SPARQL Queries in Konduit**. Proceedings of the 2007 International Conference on Posters and Demonstrations. 2008. p. 68-69.

MURRAY, C. **Spatial and Graph RDF Semantic Graph Developer's Guide 12c**. Oracle., p. 636. 2014. (E51611-06).

NIKOLOV, A.; SCHWARTE, A.; HÜTTER, C. **Fedsearch Efficiently combining structured queries and full-text search in a sparql federation**. International Semantic Web Conference. Sydney, Australia: Springer Berlin Heidelberg. 2013. p. 427-443.

PARR, T. **The definitive ANTLR 4 reference**. 2013.

PRUD'HOMMEAUX, E.; BUIL-ARANDA, C. **SPARQL 1.1 Federated Query**, 21 March 2013. Available at: <https://www.w3.org/TR/sparql11-federated-query/>.

QUILITZ, BASTIAN; LESER, U. **Querying distributed RDF data sources with SPARQL**. European Semantic Web Conference. Tenerife, Spain: Springer Berlin Heidelberg. 2008.

RAKHMAWATI, N. A. et al. **A comparison of federation over SPARQL endpoints frameworks**. International Conference on Knowledge Engineering and the Semantic Web. St. Petersburg, Russia: Springer Berlin Heidelberg. 2013a. p. 132-146.

RAKHMAWATI, N. A. et al. **Querying over Federated SPARQL Endpoints---A State of the Art Survey**. DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE. Galway, Ireland. 2013b. (arXiv preprint arXiv:1306.1723).

SCHWARTE, A. et al. **Fedx Optimization techniques for federated query processing on linked data**. International Semantic Web Conference. Bonn, Germany: Springer Berlin Heidelberg. 2011. p. 601-616.

VOLZ, J. et al. **Discovering and maintaining links on the web of data**. International Semantic Web Conference. Chantilly, VA, USA: Springer Berlin Heidelberg. 2009. p. 650-665.

ZENG, KAI et al. **A Distributed Graph Engine for Web Scale RDF Data**. Proceedings of the VLDB Endowment. February 2013. p. 265-276.

ZENZ, G. et al. **From keywords to semantic queries—Incremental query construction on the Semantic Web**. Web Semantics: Science, Services and Agents on the World Wide Web, v. 7, n. 3, p. 166-176, 2009.

ZHOU, Q. et al. **SPARK adapting keyword query to semantic search**. The Semantic Web. 2007. p. 694-707.

Appendix

Appendix I: Setting Up the SPARQL 1.1 Federated Query in Oracle 12c

Oracle Spatial and Graph, a native option for Oracle Database, enables to store semantic data and ontologies, with native support for World Wide Web Consortium (W3C) standards− RDF and OWL are standards for representing and defining semantic data and SPARQL is a query language designed specifically for graph analysis.

To query semantic data, use the `SEM_MATCH` table function with following specification. Their parameters are explained in (MURRAY, 2014). However, it is important to highlight that, the `query` and `models` attributes are required and the others are optional (that is, each can be a null value).

```
SEM_MATCH(
   query VARCHAR2,
    models SEM_MODELS,
    rulebases SEM_RULEBASES,
    aliases SEM_ALIASES,
    filter VARCHAR2,
    index_status VARCHAR2,
    options VARCHAR2,
    graphs SEM_GRAPHS,
    named_graphs SEM_GRAPHS
  )RETURN ANYDATASET;
```

The `SEM_MATCH` function also supports SPARQL 1.1 Federated Query. How the `SERVICE` construct can be used to retrieve results from a specified SPARQL endpoint URL, it is feasible to combine local RDF data (native RDF data or RDF views of relational data) with other, possibly remote, RDF data served by a W3C standards-compliant SPARQL endpoint.

In this way and whereas the local RDF triples are stored in the model called *family*, the example of the SPARQL query presented in Section 2.3 would be written as follows:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT name
FROM SEM_MATCH(
   'SELECT ?name
    WHERE
    {
       <http://example.org/myfoaf/I> foaf:knows ?person .
       SERVICE SILENT <http://people.example.org/sparql>
       { ?person foaf:name ?name . }
    }',
SEM_Models('family'), null, null, null, null, null));
```

The **Mediator Component** will be located in an Oracle user. In order to use the `SERVICE` construct within `SEM_MATCH` queries it needs to grant `EXECUTE` privilege on the `SPARQL_SERVICE` function in `MDSYS` user by a user with DBA privileges, it is possible running the following statement:

```
grant execute on mdsys.sparql_service to <mediator_user>;
```

Furthermore, an Access Control List (ACL) should be used to grant the CONNECT privilege to the user attempting a federated query. The following a template is presented to create a new ACL to grant the user the CONNECT privilege and assigns the domain `*` to the ACL.

```
BEGIN
dbms_network_acl_admin.create_acl (
  acl       => 'test.xml',
  description => 'Allow <USER_NAME> to query SPARQL endpoints',
  principal => '<USER_NAME>',
  is_grant  => true,
  privilege => 'connect'
);

dbms_network_acl_admin.assign_acl (
  acl  => 'test.xml',
  host => '*'
);
END;
```

Appendix II: Setting Up the SPARQL Endpoint Service in Oracle 12c

Oracle Spatial and Graph enables to set up a SPARQL web service endpoint by deploying the **joseki.war** file, available to download in http://www.oracle.com/technetwork/database/options/spatialandgraph/downloads/index-156999.html. It is possible to deploy this file in WebLogic Server or Apache Tomcat or JBoss. In this work, the Application Server used was JBoss AS 7.1.1.Final[13].

Firstly, it is mandatory that the Oracle 12 user who owns the RDF graph that will be exported to the SPARQL Endpoint has `CREATE PROCEDURE` privileges[14].

To deploy **Joseki** in *JBoss 7.1.1.Final*, we followed these steps, also available in **Oracle Database Online Documentation 12c Release 1 (12.1)[15]**.

1. Download and install JBoss Application Server 7.1.1.Final.
2. Install the JDBC driver:

```
create directory <JBOSS_file>/modules/oracle/jdbc/main/
```

3. Copy **ojdbc6.jar**[16] into this directory.
4. Create **module.xml** in this directory with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
 <module xmlns="urn:jboss:module:1.0" name="oracle.jdbc">
     <resources>
        <resource-root path="ojdbc6.jar"/>
    </resources>
    <dependencies>
        <module name="javax.api"/>
         <module name="javax.transaction.api"/>
     </dependencies>
</module>
```


[13] http://jbossas.jboss.org/downloads/
[14] https://community.oracle.com/thread/4000821
[15] https://docs.oracle.com/database/121/RDFRM/GUID-A18AD59B-10B6-41E3-8791-EF9A8DE4A1F6.htm#RDFRM745
[16] http://www.oracle.com/technetwork/apps-tech/jdbc-112010-090769.html

5. Modify **<JBOSS_file>/standalone/configuration/standalone.xml** by adding the highlighted line:

```
...
  <drivers>
     <driver name="OracleJDBCDriver" module="oracle.jdbc"/>
     <driver name="h2" module="com.h2database.h2">
        <xa-datasource-class>
            org.h2.jdbcx.JdbcDataSource
        </xa-datasource-class>
     </driver>
   </drivers>
...
```

6. Create the necessary data source.
   a. Log into the JBoss AS Administration Console:

```
http://<hostname>:9990/console/App.html#server-overview
```

   b. Click **Datasource**.
   c. Click **Profile**.
   d. Click **Add**, and enter the following:

```
Name: OracleSemDS
JNDI Name: java:jboss/datasources/OracleSemDS
```

   e. Select **OracleJDBCDirver**
   f. Click **Next**.

      The following information is displayed:

```
Connection URL: jdbc:oracle:thin:@hostname:port:sid
Username:    scott
Password:    tiger
Security Domain: (Leave empty)
```

   g. Customize the highlighted information and leave Security Domain blank, and click **Done**.
7. Highlight this new data source, click **Enable**, and then click **Confirm**.
8. Copy the **joseki.war** file in following directory:

```
<JBOSS_file>\standalone\deployments\
```

9. Deploy the **joseki.war** file using the JBoss Administration Console.
   a. Go to the following page:

```
http://<hostname>:9990/console/App.html#deployments
```

   b. Click **Deployments**.
   c. Click **Manage Deployments**.
   d. Click **Add** and specify the **joseki.war** file.

10. Verify if the deployment by using a Web browser to connect to a URL in the following format (assume that the Web application is deployed at port 8080): **http://<hostname>:8080/joseki**.

    We should see a page titled *Oracle SPARQL Service Endpoint using Joseki*, and the first text box should contain an example SPARQL query.

11. Configure the **joseki-config.ttl** file:

    By default, the **joseki-config.ttl** file contains an `oracle:Dataset` definition using a model named **M_NAMED_GRAPHS**. The following snippet shows the configuration.

```
<#oracle> rdf:type oracle:Dataset;
    joseki:poolSize     1 ;    ## Number of concurrent connections
                               ## allowed to this dataset.
    oracle:connection
    [ a oracle:OracleConnection ;
    ];
    oracle:allGraphs [ oracle:firstModel "M_NAMED_GRAPHS" ] .
```

The `oracle:allGraphs` predicate denotes that the SPARQL service endpoint will serve queries using all graphs stored in the **M_NAMED_GRAPHS** model. However, it is necessary to change this value by the real model name stored in Oracle user. Also, we recommend increase the value of `joseki:poolSize` property.

Appendix III: Common SQL Queries to Insert Metadata into Local RDF Graph

Q1. Insert the order of the classes by the cardinality

```
INSERT INTO RDF_DATA (TRIPLE)
SELECT SDO_RDF_TRIPLE_S('MODEL_NAME',
       subj$rdfterm, pred$rdfterm,  '"' || rownum || '"' )
FROM (SELECT subj$rdfterm, pred$rdfterm
FROM TABLE(SEM_MATCH(
'CONSTRUCT { ?c <PREFIX/order> ?cnt }
 WHERE{{SELECT ?c (COUNT(?r) as ?cnt)
        WHERE{ ?r rdf:type ?c .
               ?c rdf:type rdfs:Class }
        GROUP BY ?c }}',
SEM_MODELS('MODEL_NAME'), NULL, NULL, NULL))
ORDER BY to_number(obj) DESC);
```

Q2. Insert the order of the properties by the cardinality

```
INSERT INTO RDF_DATA (TRIPLE)
SELECT SDO_RDF_TRIPLE_S('MODEL_NAME',
          subj$rdfterm, pred$rdfterm,  '"' || rownum || '"' )
FROM (SELECT subj$rdfterm, pred$rdfterm
FROM TABLE(SEM_MATCH(
'CONSTRUCT { ?p <PREFIX/order> ?cnt }
 WHERE{{
        SELECT ?p (COUNT(*) as ?cnt)
        WHERE{ ?s ?p ?o .
               ?p rdf:type rdf:Property }
        GROUP BY ?p }}',
SEM_MODELS('MODEL_NAME'), NULL, NULL, NULL))
ORDER BY to_number(obj) DESC);
```

Q3. Indexing **TRUE** the properties with STRING type

```
INSERT INTO RDF_DATA(TRIPLE)
SELECT SDO_RDF_TRIPLE_S('MODEL_NAME',
                 subj$rdfterm, pred$rdfterm, obj$rdfterm)
FROM TABLE(SEM_MATCH(
'CONSTRUCT { ?p <PREFIX/indexing> "true"}
 WHERE{
        SELECT distinct ?p
        WHERE { ?p rdf:type rdf:Property .
                ?p rdfs:range ?rg .
        FILTER(?rg in
             (<http://www.w3.org/2000/01/rdf-schema#Literal>,
              <http://www.w3.org/2001/XMLSchema#string>))
        FILTER(?p != rdfs:label )
 }}',
SEM_MODELS('MODEL_NAME'), NULL, NULL, NULL));
```

Q4. Insert the default group order

```
INSERT INTO RDF_DATA (TRIPLE)
VALUES ( SDO_RDF_TRIPLE_S('MODEL_NAME',
       '<PREFIX#group_default>',
       '<PREFIX/order>',
       '"' || 1 || '"' ) );
```

Q5. Insert all properties in the “default” group

```
INSERT INTO RDF_DATA (TRIPLE)
SELECT SDO_RDF_TRIPLE_S('MODEL_NAME',
                    subj$rdfterm, pred$rdfterm, obj$rdfterm)
FROM TABLE(SEM_MATCH(
'CONSTRUCT { ?p rdfs:subPropertyOf <PREFIX#group_default> }
 WHERE { ?p rdf:type rdf:Property }',
SEM_MODELS('MODEL_NAME'),NULL, NULL, NULL));
```

Q6. Insert the triple joins `?p rdf:type owl:ObjectProperty`

```
INSERT INTO RDF_DATA (TRIPLE)
SELECT SDO_RDF_TRIPLE_S('MODEL_NAME',
                    subj$rdfterm, pred$rdfterm, obj$rdfterm)
FROM TABLE(SEM_MATCH(
'CONSTRUCT { ?p rdf:type owl:ObjectProperty }
 WHERE { SELECT distinct ?p
         WHERE { ?p rdf:type rdf:Property .
                 ?s ?p ?o .
                 FILTER ( isIRI(?o)) }
}',
SEM_MODELS('MODEL_NAME'), NULL, NULL, NULL
```