Keyword Search over Federated RDF Datasets

Yenier T. Izquierdo[1], Marco A. Casanova[1,2], Grettel M. García[1], Frederic Dartayre[2], Carlos H. Levy[2]

[1]Department of Informatics – Pontifical Catholic University of Rio de Janeiro, RJ, Brazil
[2]Instituto TecGraf – Pontifícia Universidade Católica do Rio de Janeiro
{yizquierdo,casanova,ggarcia}@inf.puc-rio.br,
{fdartayre,levy}@tecgraf.puc-rio.br

Abstract. This paper describes an algorithm to perform keyword search over federated RDF datasets. The algorithm compiles keyword-based queries into federated SPARQL queries, without user intervention, under the assumption that the RDF datasets and the federation have a schema. The compilation process is explained in detail, including how to synthesize external joins between local queries, how to combine local queries with *UNION* clauses, and how to construct the *WHERE* and *TARGET* clauses. The paper then presents the architecture of a system that implements the algorithm. Finally, the paper describes experiments with three real-world datasets to validate the implementation and to help understand the different situations faced by the compilation process.

Keywords: Keyword search; federated SPARQL query; RDF.

1 Introduction

The Resource Description Framework (RDF) was adopted as a W3C recommendation in 1999 and today is a standard for exchanging data on the Web. Indeed, a huge amount of data has been converted to RDF [15] and published openly on the Web, following the Linked Data principles [2]. These datasets frequently have an RDF schema, often under the form of a standard ontology, such as the Music Ontology, and are interconnected.

The SPARQL Protocol and RDF Query Language (SPARQL) was officially introduced in 2008 to specify queries over RDF datasets. The SPARQL 1.1 Federated Query extension offers services for executing queries distributed over different RDF datasets. SPARQL is a sophisticated language, which is difficult to master, though.

Keyword-based queries offer an alternative way to access databases, which is attractive when users are unaware of the way data is organized, or do not know the syntax of the query language. Specifically, keyword-based queries for RDF datasets has been extensively investigated, but most approaches assume that the RDF triples are stored in a centralized repository [6,8,10,20].

By contrast, this paper addresses the problem of processing keyword-based queries over federated RDF datasets. The main contributions of this paper are: (i) An algorithm to compile keyword-based queries into federated SPARQL queries, without user intervention, under the assumption that the RDF datasets and the federation have a schema;

(ii) An implementation of the algorithm; (iii) Experiments with the implementation to assess its usability.

The reminder of this paper is organized as follows. Section 2 summarizes related work. Section 3 contains basic definitions and introduces the example used throughout the paper. Section 4 details the algorithm to compile keyword-based queries into federated SPARQL queries. Section 5 covers experiments with an implementation of the algorithm. Finally, Section 6 presents the conclusions and proposes future work.

2 Related Work

Keyword Search over RDF Graphs in Centralized Environments. Several approaches have been developed to help solve the keyword search problem over RDF datasets. The main challenge addressed in many of these approaches has been to synthesize SPARQL queries from a set of keywords, since users are generally unaware of the query language and the RDF schema to be queried.

For example, the *Konduit* tool [4, 10] provides non-expert users with a way to visually specify SPARQL queries. However, this tool has the disadvantage that the user must have a basic knowledge of SPARQL.

The *QUICK* (QUery Intent Constructor for Keywords) [20] helps users construct queries in a given domain. The tool combines the convenience of keyword search with the expressiveness of semantic queries. Users start with a keyword query and are then guided through an incremental refinement process to specify the query intention. The intermediate queries are listed and ranked. *QUICK* has the drawback that the user must have some knowledge of the RDF schema.

SPARKLIS [7] helps users explore and query RDF datasets by interactively exploring the RDF schema, in a questions and answers process. SPARKLIS combines the fine-grained guidance of faceted search, most of the expressiveness of SPARQL, and the readability of natural languages. It has the advantage that no specific endpoint configuration is required, since the schema is discovered on the fly. However, unlike our proposal, in SPARKLIS, the query is built by selecting query elements, and only a single RDF dataset is accessed.

The tools described in [8, 20] synthesize SPARQL queries by exploring the underlying RDF schema. Unlike [20], the tool described in [8] is fully automatic, and allows users to specify a keyword-based query with *filters*, that involve comparison operators.

Frameworks supporting federated SPARQL queries. Research on querying distributed RDF datasets with SPARQL-like queries typically explores optimizations based on structural information (i.e., graph partitioning) [9,13,19]. Furthermore, according to [15], the tools and systems designed to address federated queries focus mostly on dataset selection and join optimization during federated SPARQL query execution.

Well-known federation systems are *ANAPSID* [1], *FedSearch* [11], and *FedX* [17]. But all these systems focus on improving the performance of the SPARQL query execution, and not on the construction of the queries.

A comparison of existing SPARQL federation frameworks can be found in [14,16]. 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. The existing frameworks for SPARQL 1.1 Federation Extension support the *SERVICE* keyword, but not all of them support the *BIDING* and *VALUES* operators. For instance, ARQ is a query engine processor for Jena that supports federated queries, providing *SERVICE* and *VALUES* operators. The framework implements nested loop joins to retrieve and combine result from multiple RDF datasets. Also, it provides a set of Java packages to build SPARQL queries programmatically.

The work reported in this paper extends the centralized algorithm developed in [8] to compile keyword-based queries into federated SPARQL queries. The implementation reported in the paper follows the architecture for federations of SPARQL endpoints described in [15], uses the Java packages that ARQ provides, and stores the RDF schema of each available RDF dataset in auxiliary tables.

3 Definitions and Examples

3.1 Basic Definitions

A *link* between RDF datasets T_i and T_j, with $i \neq j$, is an RDF triple of the form (s_i,p,s_j), where s_i and s_j are URIs occurring in T_i and T_j. A *linkset* between T_i and T_j is simply a set of links between T_i and T_j.

Assume that T_i and T_j have RDF schemas. A *linkset schema* for a linkset L_k between T_i and T_j is a triple (c_i,p,c_j), where c_i and c_j are classes of the RDF schemas of T_i and T_j. It restricts the links in L_k to be of the form (s_i,p,s_j), where s_i and s_j are instances of the classes c_i and c_j in T_i and T_j, respectively.

An *RDF dataset federation* is a collection $T = \{T_1, \dots, T_m\}$ of RDF datasets and a collection $L = \{L_1, \dots, L_n\}$ of linksets between the datasets in T, denoted simply as $T \cup L$. The RDF graph of the RDF dataset federation $T \cup L$ is the RDF graph induced by the set of triples $T_1 \cup \ldots \cup T_m \cup L_1 \cup \ldots \cup L_n$.

A *keyword-based query K* is a set of literals, or *keywords*. An *answer* for K over $T \cup L$ is defined as in [8], except that the RDF graph is that induced by the set of triples $T_1 \cup \ldots \cup T_m \cup L_1 \cup \ldots \cup L_n$. A detailed definition is omitted for brevity. The problem of finding answers for keyword-based queries over RDF dataset federations (or, briefly, the *RDF-FKwS problem*) is defined as: *"Given an RDF dataset federation $T \cup L$ and a keyword-based query K, find an answer for K over $T \cup L$".*

In this paper, we assume that each RDF dataset T_i follows an RDF schema S_i, for $1 \leq i \leq m$, and that each linkset L_j follows a linkset schema M_j, for $1 \leq j \leq n$. We also assume that the RDF dataset federation $T \cup L$ has a *federated schema*, which consists of the schemas of its datasets and linksets, a set of *metadata matches*, and a set of *instance match definitions*. Section 3.2 provides an example of a federated schema.

More precisely, a *federated schema* for the *RDF* dataset federation $T \cup L$ is a quadruple $\Phi = (\sigma,\lambda,\mu,\delta)$, where:

1) $\sigma = (S_1,\ldots,S_m)$ and $\lambda = (M_1,\ldots,M_n)$.

2) μ is a set of *metadata matches*, that is, a set of pairs of the form (x_i,x_j) such that x_i is a class (datatype property or object property) defined in S_i and x_j is a class (respectively, datatype property or object property) defined in S_j, for $1 \leq i \neq j \leq m$; in this case, we say that x_i and x_j *match*.
3) δ is a set of *instance match definitions*, that is, a set of pairs of the form $(q,(c_i,(p_{i1},\ldots, p_{ik}),c_j,(p_{j1},\ldots,p_{jk})))$ such that:
   a) q is an object property;
   b) c_i and c_j are classes defined in S_i and S_j, respectively;
   c) $(p_{i1},\ldots,p_{ik})$ is a list of datatype properties defined in S_i whose domain is c_i, and likewise for $(p_{j1},\ldots,p_{jk})$.

By (1), the federated schema simply contains all dataset and linkset schemas. By (2), it admits just one-to-one metadata matches (also called schema mappings).

An instance match definition $(q,(c_i,(p_{i1},\ldots,p_{ik}),c_j, (p_{j1},\ldots,p_{jk})))$ specifies a set of links L of the form (s_i,q,s_j) such that s_i is an instance of class c_i in T_i, s_j is an instance of class c_j in T_j, and there are triples in T_i of the form (s_i,p_{iu},v_u) and triples in T_j of the form (s_j,p_{ju},v_u), for $1 \leq u \leq k$. That is, (s_i,q,s_j) is a link in L iff the value of property p_{iu} for s_i is equal to the value of property p_{ju} for s_i, for $1 \leq u \leq k$. A common type of inter-dataset property definition specifies that s_i and s_j denote the same real-world entity iff the values of their properties $(p_{i1},\ldots,p_{ik})$ and $(p_{j1},\ldots,p_{jk})$ are equal; these definitions are usually called *sameAs* definitions.

We could account for more complex metadata matches or instance match definitions at the expenses of a more complex keyword federated translation algorithm, which we leave for future work. In particular, *sameAs* definitions could incorporate transformations (e.g. lower case to upper case) and similarity measures (e.g. Levenshtein distance), as in tools such as Silk [18].

3.2 An Example

Consider the following RDF datasets: the drug and enzyme data in *DBpedia*, with 11.198 triples; *DrugBank*, with 765.936 triples; and the *Kegg Drug*, with 713.737 triples. We assume that these datasets have the RDF schemas described in Figure 1.

Consider a federation of these three datasets with the federated schema described in Figure 2, where dotted arrows represent metadata matches and dashed arrows indicate instance match definitions. In more detail, we have:

- RDF schemas: (as described in Figure 1).
- linkset schemas: (the federation has no linksets).
- metadata matches: *drug* in *DBpedia* with *drug* in *DrugBank*; *enzyme* in *DrugBank* with *Metabolism* in *Kegg Drug*; and *drugInteraction* in *DrugBank* with *Interaction* in *Kegg Drug*.
- instance match definitions: a *sameAs* definition that relates *enzyme* instances in *DrugBank* and *enzyme* instances in *DBpedia*; a *sameAs* definition that relates *Drug* instances in *DBpedia* and *drug* instances in *DrugBank*; a definition for a new object property that relates *drug* instances in *DrugBank* with *drug* instances in *Kegg Drug*.

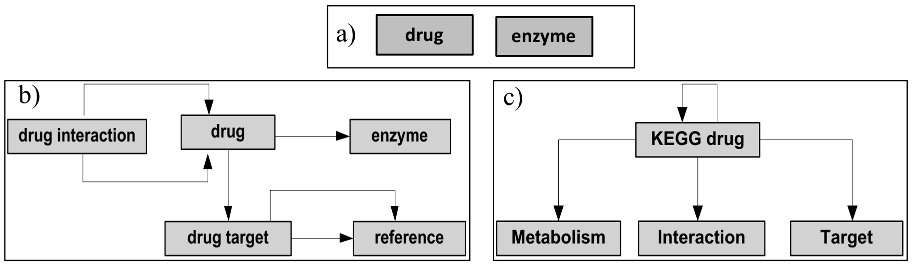


Figure 1 - RDF Schema of: (a) *DBpedia*; (b) *DrugBank*; (c) *Kegg Drug*.

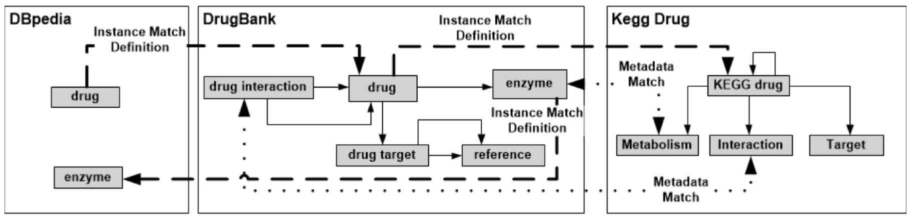


Figure 2 - Federated Schema of *DBpedia*, *DrugBank* and *Kegg Drug*.

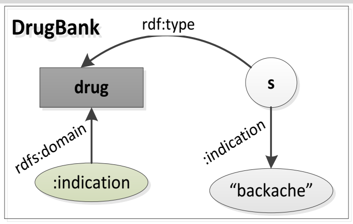
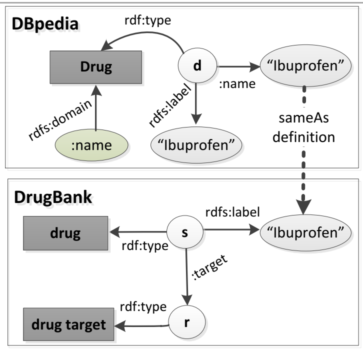
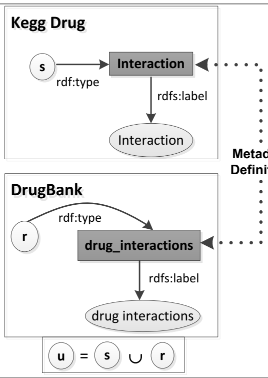
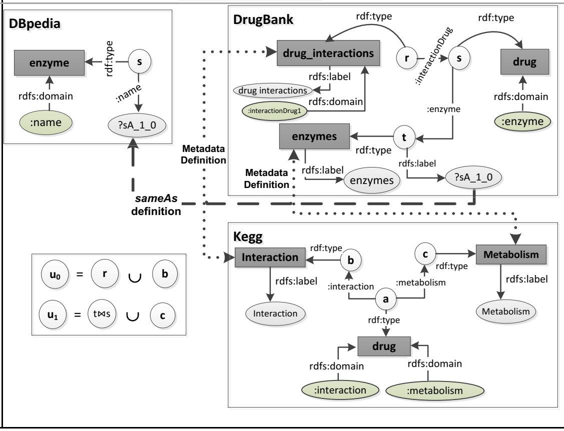
Table 1 illustrates the types of federated queries the federated translation algorithm generates. The table is organized as follows:

- Column 1: examples of keyword-based queries with the time to compile and run each query (see Section 5).
- Column 2: SPARQL queries over the federation that translate the keyword-based queries.
- Column 3: a schematic description of each query.

Section 4 discusses how to compile the keyword-based query in each row into the corresponding federated SPARQL query. The rows of the table show:

- Row 1: a query over *DrugBank* only.
- Row 2: a federated query with two local queries over *DBpedia* and *Drugbank* that are combined with the help of a *sameAs* definition that relates the *DBpedia* instance for "ibuprofen" with the *DrugBank* instance for the same drug.
- Row 3: a federated query with two local queries over *Kegg Drug* and *Drugbank* that are combined with the help of a *UNION* clause; the synthesis of the *UNION* clause is possible since *drugInteraction* in *DrugBank* matches *Interaction* in *Kegg Drug*.
- Row 4: a more complex federated query with three local queries over *DBpedia*, *Kegg Drug* and *DrugBank*; the local queries over *DBpedia* and *DrugBank* are combined with the help of a *sameAs* definition that relates the *DBpedia* instances of type enzyme with *DrugBank* instances also of type enzyme; the local query over *Kegg Drug* and the query formed by the external join between the local queries over *DBpedia* and *DrugBank* are combined with the help of a *UNION* clause.

Table 1 – Sample keyword-based queries and their translations.

Keywords	Runtime (s)	Generated Query	Federated SPARQL Query Structure
indication backache	0.34s	`SELECT ?C_0_0 ?P_0_0 WHERE{ SERVICE SILENT <drugbank> { ?I_C_0_0 rdf:type drugbank:drugs . ?I_C_0_0 drugbank:indication ?P_0_0 FILTER oracle:textContains (?P_0_0, "fuzzy({backache}, 70, 1)", 1) ?I_C_0_0 rdfs:label ?C_0_0 } }`	
'drug target' ibuprofen	1.32s	`SELECT ?C_0_0 ?C_1_1 WHERE{ SERVICE SILENT <dbpedia>{ ?I_C_0_0 rdf:type dbpedia:Drug . ?I_C_0_0 dbpedia:name ?sA_1_0 FILTER oracle:textContains (?C_0_0, "{\"ibuprofen\"}", 0) ?I_C_0_0 rdfs:label ?C_0_0 } SERVICE SILENT <drugbank>{ ?I_C_1_0 drugbank:target ?I_C_1_1 . ?I_C_1_0 rdfs:label ?sA_1_0 FILTER oracle:textContains (?sA_1_0, "{\"ibuprofen\"}", 0) ?I_C_1_1 rdfs:label ?C_1_1 } }`	
interaction	1.54s	`SELECT ?U_0 WHERE{ { SELECT (?C_1_0 AS ?U_0) WHERE{ SERVICE SILENT <drugbank>{ ?I_C_1_0 rdf:type drugbank:drug_interactions. ?I_C_1_0 rdfs:label ?C_1_0 } } UNION{ SELECT (?C_0_0 AS ?U_0) WHERE{ SERVICE SILENT <kegg>{ ?I_C_0_0 rdf:type kegg:Interaction . ?I_C_0_0 rdfs:label ?C_0_0 } } } }`	
interaction enzyme metabolism	3.55s	`SELECT ?U_0 ?U_1 WHERE { { SELECT (?C_2_0 AS ?U_0) (?sA_1_0 AS ?U_1) WHERE { SERVICE SILENT <drugbank>{ ?I_C_2_1 drugbank:enzyme ?I_C_2_2 . ?I_C_2_0 drugbank:interactionDrug ?I_C_2_1 . ?I_C_2_2 rdfs:label ?sA_1_0 . ?I_C_2_0 rdfs:label ?C_2_0 . } SERVICE SILENT <dbpedia>{ ?I_C_0_0 rdf:type dbpedia:Enzyme . ?I_C_0_0 dbpedia:name ?sA_1_0 . } } } UNION { SELECT (?C_1_0 AS ?U_0) (?C_1_1 AS ?U_1) WHERE{ SERVICE SILENT <kegg>{ ?I_C_1_0 kegg:metabolism ?I_C_1_2 . ?I_C_1_0 kegg:interaction ?I_C_1_1 . ?I_C_1_1 rdfs:label ?C_1_0 . ?I_C_1_2 rdfs:label ?C_1_1 } } } }`	

4 The Federated Translation Algorithm

The key problems to compile keywords into federated queries are: (1) How to generate local queries that cover the input keywords as much as possible; (2) How to combine the local queries, using the metadata matches and instance match definitions of the federated schema. The first problem was addressed in [8] and the second problem is the focus of this section.

4.1 Overview of the Federated Translation Algorithm

This section details the *federated translation algorithm* that compiles keyword based-queries into federated SPARQL queries (see Figure 3). It has as input a keyword-based query K and a federated schema Φ and returns a federated RDF query Q.

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. It also returns, for each variable v_k, with $1 \leq k \leq n$:

1. the set of keywords K_k that v_k covers;
2. the URIs of the elements that v_k binds to, as follows:
   - if v_k binds to instances of a class c_i, then the URI of c_i is returned;
   - if v_k binds to property values of a property p_i, then the URIs of p_i and c_i, are returned, where c_i is the domain of p_i.

The set of keywords that Q_i covers is then $K_i = K_1 \cup \ldots \cup K_n$. By construction of the centralized translation algorithm, we have $K_i \subseteq K$.

The following sections detail how Stage 2 synthesizes a federated SPARQL query Q from the local queries. Very briefly, Step 2.1 finds the set of *external joins* that are candidates to link the set of local queries. Step 2.2 creates the *federated query multigraph* G_F=<V_F, E_F, W_F> such that the nodes are the local queries, the edges are the

```
FEDERATED TRANSLATION ALGORITHM
Input:   A keyword-based query $K$
         A federated RDF schema $\Phi$
Output:  A federated SPARQL query $Q$
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 $Q$:
  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 the UNION clauses.
  2.5. Synthesize and return $Q$.
```

Figure 3 - Outline of the *Federated Translation Algorithm.*

candidate external joins, and the weights are as in Section 4.3; then, Step 2.2 computes a maximum directed spanning forest T_F of G_F. Step 2.3 inserts into the local queries the triple patterns corresponding to the arcs of T_F. If T_F is not connected, then Step 2.4 tries to combine the local queries represented by the connected components of T_F through *UNION* clauses. Step 2.5 synthesizes the federated SPARQL query *Q*.

4.2 Overview of the Centralized Translation Algorithm

This section briefly overviews the centralized translation algorithm, introduced in [8]. The algorithm accepts a keyword-based query *K* and an RDF dataset *T*, and outputs a SPARQL query *Q*, which is a correct interpretation for *K*, in the sense that any result of *Q* is an answer for *K* over *T*. It assumes that *T* follows an RDF schema *S*.

The algorithm starts by computing a set of *metadata matches* and a set of *property value matches* between the keywords and elements of *T*. It organizes the matches into *nucleuses*, which have a class *c*, together with a list of properties whose domain is *c* and whose values match keywords.

The centralized translation algorithm implements two heuristics, called the *scoring* and the *minimization* heuristics. Briefly, the scoring heuristic: (1) considers how good a match is, say "city" matches "Cities" better than "Sin City"; (2) assigns a higher score to metadata matches, on the grounds that, if the user specifies a keyword, say "city", that matches both a class label, say, "Cities", and the property value of an instance, say the film title "Sin City", then the user is probably more interested in the class labelled "Cities" than the specific film "Sin City"; (3) assigns a higher score to nucleuses that cover a larger number of keywords. The heuristic is formalized by defining a *score function* for the nucleuses.

The minimization heuristic tries to generate minimal answers, in two stages. Ideally, we should try to find the smallest set of nucleuses that covers the largest set of keywords and that has the largest combined score. However, this is an NP-complete problem (by a reduction to the bin packing problem). The first stage of the minimization heuristic then implements a greedy algorithm that prioritizes the nucleuses with the largest scores that cover a large subset of *K*. The second stage of the minimization heuristic then connects the classes in such nucleuses, using a small number of equijoins. This is equivalent to generating a Steiner tree *ST* of the graph of the RDF schema of *T* that covers the classes in the prioritized nucleuses. Then, the algorithm uses the edges of *ST* to generate equijoin clauses of the SPARQL query *Q*.

4.3 Computing External Joins

Recall that the result of executing the centralized translation 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 Stage 1 returns queries Q_1, Q_2, …, Q_k, with $k \leq n$. Let $\boldsymbol{C_i}$ denote the set of classes present in Q_i, for $1 \leq i \leq k$. Also, for each $c_i \in \boldsymbol{C_i}$, let $score(c_i)$ be the score of the nucleus that contains c_i, as computed by the centralized translation algorithm.

For each pair of local queries Q_i and Q_j, with $1 \leq i \neq j \leq n$, Step 2.1 creates a *candidate external join*, denoted (Q_i,Q_j), between Q_i and Q_j iff the federated schema has a linkset schema of the form (c_i,p,c_j) or an instance match definition of the form $(q,(c_i,(p_{i1},\ldots,p_{in}),c_j,(p_{j1},\ldots,p_{jn})))$ such that $c_i \in \boldsymbol{C_i}$ and $c_j \in \boldsymbol{C_j}$. We say that the external join is *generated by* the linkset schema or by the instance match definition. Step 2.1 computes the *score* of (Q_i,Q_j) as the sum of the scores of c_i and c_j.

Step 2.2 first creates the *federated query multigraph* $G_F = <V_F, E_F, W_F>$, where:

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

Note that G_F is indeed a multigraph, since there might be more than one candidate external join between the same pair of local queries. Also, G_F may not be connected.

Step 2.2 then computes a maximum spanning forest T_F of G_F. The arcs in T_F represent the *selected external joins* that will be used to combine the local queries.

After selecting the external joins, Step 2.2 computes the *score* of a local query Q_i as:

$$score(Q_i) = max\{\ W_F((Q_i,Q_j))\ /\ (Q_i,Q_j) \in T_F\ \}$$

Then, for each selected external join (Q_i,Q_j), Step 2.3 synthesizes *external join triple patterns* as follows. There are two cases to consider. If the external join is generated by a linkset schema (c_i,p,c_j), then a triple pattern of the form $(s_i\ p\ s_j)$ is included in Q_i. If the external join is generated by an instance match definition of the form $(q,(c_i,(p_{i1},\ldots,p_{in}),c_j,(p_{j1},\ldots,p_{jn})))$, then triple patterns of the forms $(s_i\ p_{ik}\ v_k)$ and $(s_j\ p_{jk}\ v_k)$ are included in Q_i and Q_j. Note that the use of the same variable v_k in both triple patterns forces the property values to be equal.

For example, row 2 of Table 1 shows a federated query with two local queries, Q_1 and Q_2, respectively over *DBpedia* and *DrugBank*. These queries are combined by external join triple patterns, which are in turn generated by a *sameAs* definition. Let $\boldsymbol{C_1}$ and $\boldsymbol{C_2}$ be the sets of classes present in Q_1 and Q_2, respectively. So, corresponding to the *sameAs* definition, the triple pattern `(?I_C_0_0 dbpedia:name ?sA_1_0)` is added to Q_1 and `(?I_C_1_0 rdfs:label ?sA_1_0)` to Q_2. The score of Q_1 is given by

$$score(Q_1) = W_F((Q_1,Q_2)) = score(Drug)+score(drugs)$$

where we recall $Drug \in \boldsymbol{C_1}$ and $drugs \in \boldsymbol{C_2}$ and the values of $score(Drug)$ and $score(drugs)$ are computed by the centralized translation algorithm in Stage 1.

4.4 Computing *UNIONs*

Recall that Step 2.2 creates a federated graph G_F and computes a maximum spanning forest T_F of G_F. However, T_F may not be connected, that is, T_F may consist of several trees, in which case Step 2.4 tries to compute *UNION* clauses, under certain conditions.

Let $\boldsymbol{Q_1}$ and $\boldsymbol{Q_2}$ be two SPARQL queries in different trees of T_F. Assume that $S_1 = \{v_1, v_2, \ldots, v_m\}$ is the set of variables in the *TARGET* clause of $\boldsymbol{Q_1}$, $S_2 = \{w_1, w_2, \ldots, w_n\}$ is the set of variables in the *TARGET* clause of $\boldsymbol{Q_2}$, S_1 covers the set of keywords K_1 and S_2 covers the set of keywords K_2. It is possible to combine $\boldsymbol{Q_1}$ and $\boldsymbol{Q_2}$ with the help of a *UNION* clause iff

(1) S_1 and S_2 have the same number of variables, that is, $m=n$;

(2) There is a permutation π of $1,\ldots,m$ such that each pair of variables v_i and $w_{\pi(i)}$ are bound to instances of classes (or properties) that match; in this case, we also say that v_i and $w_{\pi(i)}$ *match*.

To generate a federated SPARQL query with a *UNION* clause, a bind variable u_s, for s=0,…,m, is created.

For example, row 3 of Table 1 shows a federated query with a *UNION* clause that combines two local queries, Q_1 and Q_2, respectively over *DrugBank* and *Kegg Drug*. Note that, the number of variables in the *TARGET* clause of Q_1 is equal to that of Q_2. Variable `?C_1_0` in Q_1 binds to instances of the class *drug_interactions* and variable `?C_0_0` in Q_2 binds to instances of the class *Interaction*. According to the federated RDF schema depicts in Figure 2, *drug_interactions* and *Interaction* match. Therefore, variables `?C_1_0` and `?C_0_0` also match. Thus, a variable `?U_0` is created, which binds to the values in `?C_1_0` and `?C_0_0`, and Q_1 and Q_2 are combined via a *UNION* clause.

4.5 Defining the *WHERE* clause

This section schematically describes how the *WHERE* clause of the federated queries are synthesized.

Let "$Q_1 \bowtie Q_2$" represent an external join between two queries Q_1 and Q_2. Assume that Stage 1 returns queries Q_1, Q_2, …, Q_k, with $k \leq n$. Let $\boldsymbol{Q}$ be the federated SPARQL query to be constructed, and $\boldsymbol{W_Q}$ be the *WHERE* clause of $\boldsymbol{Q}$. In general, $\boldsymbol{W_Q}$ is a union of queries combined by external join patterns, which is denoted as $\boldsymbol{W_Q} = \bigcup_{k=1}^{m}(\bowtie_{j=1}^{n_k} Q_{k,j})$.

For a better understanding of the above definition, consider the following case. Suppose that the local SPARQL queries are Q_1, Q_2, Q_3 and Q_4. Assume that Q_1 and Q_2 are combined by external join triple patterns, and likewise for Q_3 and Q_4. Let $P_1 = Q_1 \bowtie Q_2$ and $P_2 = Q_3 \bowtie Q_4$. Assume that P_1 and P_2 can be combined by a *UNION* clause. Then, the *WHERE* clause of the federated query would be:

$$\boldsymbol{W_Q} = P_1 \cup P_2 = (Q_1 \bowtie Q_2) \cup (Q_3 \bowtie Q_4)$$

Note that, when P_1 and P_2 cannot be combined by a *UNION* clause, that is, when they do not meet the conditions defined in Section 4.4, then the federated translation algorithm will generate only one of the queries, P_1 or P_2.

For example, row 4 of Table 1 shows a federated query whose *WHERE* clause corresponds to a union of queries combined by external join triple patterns. Indeed, let Q_1, Q_2, and Q_3 denote the local queries over *DBpedia*, *Kegg Drug* and *DrugBank*, respectively. Observe from row 4 of Table 1 that Q_1 and Q_2 are combined by external join triple patterns generated by a *sameAs* definition, denoted $P_1 = Q_1 \bowtie Q_2$. Note that:

(1) Corresponding to the *sameAs* definition, the triple patterns `(?I_C_2_2 rdfs:label ?sA_1_0)` and `(?I_C_0_0 dbpedia:name ?sA_1_0)` are included in Q_1 and Q_2, respectively;

(2) Variables `?C_2_0` and `?sA_1_0` cover the keyword set K_1 = {"interaction", "enzyme" };

(3) The results in variable `?C_2_0` represent instances of the class *drug_interaction* in *DrugBank*;

(4) The results in variable `?sA_1_0` represent instances of the class *enzyme* that occurs in *DBpedia* and *DrugBank*.

As the example shows, the *TARGET* clause of Q_3 contains variables `?C_1_0` and `?C_1_1`, which cover the set of keywords K_2 = {"interaction", "metabolism"}. Variable `?C_1_0` represents instances of the class *Interaction*, and variable `?C_1_1` binds to the labels of the instances of the class *Metabolism*. Then, the *TARGET* clauses of both P_1 and Q_3 have the same number of variables and, according the federated schema defined in Figure 2, variables `?C_2_0` and `?C_1_0` are bound to classes that match. Then, the results of both variables are bound to the new variable `?U_0`. Likewise, variables `?sA_1_0` and `?C_1_1` match, and the results are bound to a variable, `?U_1`. So, the *WHERE* clause of the federated query is given by $\boldsymbol{W_Q} = P_1 \cup Q_3 = (Q_1 \bowtie Q_2) \cup Q_3$.

4.6 Defining the *TARGET* clause

Let $\boldsymbol{Q}$ be a 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 $\boldsymbol{Var}(\boldsymbol{S_Q})$ of the set of variables $\boldsymbol{Var}(\boldsymbol{W_Q})$ present in $\boldsymbol{W_Q}$. The computation of $\boldsymbol{Var}(\boldsymbol{S_Q})$ depends on $\boldsymbol{W_Q}$ and the coverage of the keywords set K. The different cases are explained in what follows.

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

Suppose that the centralized translation algorithm outputs a single local query Q_1. Then, the federated SPARQL query $\boldsymbol{Q}$ will be Q_1, with an additional *"SERVICE SILENT"* clause to query the target dataset, and $\boldsymbol{Var}(\boldsymbol{S_Q})=\boldsymbol{Var}(\boldsymbol{S_{Q1}})$. Row 1 of Table 1 illustrates this case.

Federated Query with a WHERE clause with only external join triple patterns

Suppose that the *WHERE* clause of the federated query $\boldsymbol{Q}$ is of the form $\boldsymbol{W_Q} = Q_1 \bowtie \ldots \bowtie Q_n$.

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

Let $\boldsymbol{CQ} = \{Q_1, \ldots, Q_m\}$ be the set of local queries. The strategy starts with $\boldsymbol{Var}(\boldsymbol{S_Q})=\varnothing$ and a set of *covered keywords* $K' = \varnothing$. Assume that the local query Q_i, with $1 \le i \le m$, has the highest value score. Then, the variables in $\boldsymbol{Var}(\boldsymbol{S_{Qi}})$ are added to $\boldsymbol{Var}(\boldsymbol{S_Q})$, and the keywords covered by $\boldsymbol{Var}(\boldsymbol{S_{Qi}})$ are added to K'. If $K = K'$ or all local queries have been analyzed, the process stops. Otherwise, the next local query Q_j in decreasing score order is analyzed and, if there is a variable $v_j \in \boldsymbol{Var}(\boldsymbol{S_{Qj}})$ such that v_j covers a set of keywords $K_j \subseteq K$, and there is a keyword $k \in K_j$ such that $k \notin K'$, then v_j is added to $\boldsymbol{Var}(\boldsymbol{S_Q})$ and k is added to K'.

For example, row 2 of Table 1 shows a federated query $\boldsymbol{Q}$ such that the set of variables $\boldsymbol{Var}(\boldsymbol{S_Q})$ is equal to $\boldsymbol{Var}(\boldsymbol{S_{Q1}})$, because Q_1 is the query with the highest score and the variables in $\boldsymbol{Var}(\boldsymbol{S_{Q1}})$ cover the set of keyword K_3.

Federated Query with a WHERE clause with only UNION clauses

Suppose that the *WHERE* clause $\boldsymbol{W_Q}$ of the federated query $\boldsymbol{Q}$ is of the form $\boldsymbol{W_Q} = Q_1 \cup \ldots \cup Q_n$.

For brevity, consider only the case of two local queries, that is, $\boldsymbol{Q} = Q_1 \cup Q_2$. 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 $1,\ldots,m$ such that each pair of variables v_i and $w_{\pi(i)}$ match. Then, a new variable u_i is created to bind the results of variables v_i and $w_{\pi(i)}$ and the *TARGET* clause $\boldsymbol{S_Q}$ of $\boldsymbol{Q}$ is composed of the bind variables $u_1,\ldots,u_m$.

For example, row 3 of Table 1 shows a federated query $\boldsymbol{Q}$ such that the set of variables $\boldsymbol{Var}(\boldsymbol{S_Q})$ consists of a new bind variable `?U_0`.

Federated Query with a WHERE clause with external join triple patterns and UNION clauses

Suppose that the *WHERE* clause $\boldsymbol{W_Q}$ of the federated query $\boldsymbol{Q}$ is of the form $\boldsymbol{W_Q} = \bigcup_{k=1}^{m}(\bowtie_{j=1}^{n_k} Q_{k,j})$.

The strategy for choosing the variables in the *TARGET* clause of $\boldsymbol{Q}$ is similar to the previous case and is based on the structure of $\boldsymbol{W_Q}$. As an example, suppose that $\boldsymbol{Q} = (Q_1 \bowtie Q_2) \cup (Q_3 \bowtie Q_4)$ and that the sets of variables of $Q_1 \bowtie Q_2$ and $Q_3 \bowtie Q_4$ are $\boldsymbol{Var}(\boldsymbol{S_{Q1\bowtie Q2}})=\{v_1,\dots,v_m\}$ and $\boldsymbol{Var}(\boldsymbol{S_{Q3\bowtie Q4}})=\{w_1,\dots,w_m\}$, respectively. Assume that these sets satisfy Conditions (1) and (2) defined in Section 4.4 and that they cover the keyword set K. Then, the results of $Q_1 \bowtie Q_2$ and $Q_3 \bowtie Q_4$ can be combined via a *UNION* clause, and a new variable u_i is created to bind the results of variables v_i and $w_{\pi(i)}$, for $i=1,\dots,m$. The set of variables $\boldsymbol{Var}(\boldsymbol{Q})$ is composed of the new bind variables $u_1,\dots,u_m$.

For example, Row 4 of Table 1 shows a federated query $\boldsymbol{Q}$ such that $\boldsymbol{W_Q} = (Q_1 \bowtie Q_2) \cup Q_3$ and $\boldsymbol{Var}(\boldsymbol{Q})=\{$`?U_0, ?U_1`$\}$.

5 Implementation and Experiments

We implemented a complete *federated keyword search system*, which incorporates the federated translation algorithm. The architecture of the system has the following components: (1) a *Web interface*, which allows users to submit keyword-based queries to the federation; (2) a *Mediator*, which orchestrates the processing of the keyword-based queries; (3) a *Storage Component*, which stores data and metadata about the RDF datasets of the federation; (4) a *Federated Schema Component*, which saves the federated schema of the federation; and (5) the RDF datasets that compose the federation.

Briefly, the processing of a keyword-based query goes as follows. The *Mediator* receives the set of keywords specified by the user and executes the federated translation algorithm, which: (1) uses the *Storage Component* to find the data and metadata that matches the keywords; (2) uses the *Federated Schema Component* to find the external joins between the computed subqueries; (3) if necessary, it creates *UNION* clauses to combine the result of queries that are not linked by inter-dataset property definitions; (4) synthesizes the federated SPARQL query. Then, the *Mediator* executes the federated SPARQL query and returns the response to the user.

We ran a suite of keyword-based queries to assess the performance of the federated translation algorithm. All experiments were conducted using the Web interface of the system, develop in Java. The APP ran on a desktop machine with OS Windows 7 Ultimate, a quad-core processor Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz, 4 GB of RAM. In the same desktop machine, all system components were allocated to an Oracle Database version 12c, configured with a PGA size of 324 MB and an SGA size of 612 MB with 148 MB of cache size and 296 MB of buffer cache.

As introduced in Section 3.2, three RDF datasets were selected for the experiments: the drug and enzyme data in *DBpedia*, with 11.198 triples; *DrugBank*, with 765.936 triples; and the *Kegg Drug*, with 713.737 triples. These datasets were stored in different station nodes, located in the same local network, configured using the Oracle Spatial and Graph for Semantic Technologies, available in Oracle 12c. Data and metadata stored in the *Storage Component* were indexed using Oracle Text to facilitate full text search and each remote RDF graph was indexed using the B-Tree indexing for RDF models and entailments supported by the Semantic Network feature of Oracle.

The keyword-based queries were selected to show the different configurations of the *WHERE* clause of the federated query, and the coverage of the set of keywords by the variables in the *TARGET* clause. Table 1 shows some of the queries run in the test suite. In particular, Column 1 of Table 1 shows that all queries were successfully executed in less than 4 seconds (the time reported is the average of 10 executions for each sample query). This is quite reasonable, considering: the size of the datasets; the fact that the system returns 750 results as a limit; and the fact that the subquery results come from different datasets stored in a local network.

Indeed, the results obtained by running the complete test suit suggest that the algorithm performs well for the keyword-based search over federated RDF graphs.

6 Conclusions and Future Work

We presented an algorithm, called *federated translation algorithm*, to perform keyword search over federated RDF datasets by exploring their schemas. The algorithm extends the centralized translation algorithm described in [8] and generates a federated SPARQL query such that: (1) the local queries only access the datasets whose indexed data and metadata match the keywords; and (2) the variables in the *TARGET* clause of the federated SPARQL query cover a subset of the set of keywords submitted.

We detailed the design decisions that support the construction of the federated SPARQL query, based on the information of the federated schema and on the local queries generated by the centralized translation algorithm. We defined the conditions to combine, with the help of external join patterns and *UNION* clauses, the local queries. We explained how the *TARGET* clause is constructed, according to the composition of the *WHERE* clause.

Finally, we briefly described the implementation of a federated keyword search system, which includes the federated translation algorithm, and conducted experiments to test the performance of the system, using three freely accessible RDF databases, with metadata matches and instance match definitions specified between them. The experiments suggest that the proposed algorithm performs well for keyword-based search over federated RDF datasets.

As future work, we are developing a benchmark for RDF federated keyword search systems, which we plan to use to further test the system described in this paper. The benchmark will include scenarios with a larger number of RDF datasets and more complex federated schemas. Finally, we plan to extend the current implementation to other federated RDF storage systems and to make the tool publicly available.

References

1. Acosta, M., Vidal, M. E., Lampo, T., Castillo, J., & Ruckhaus, E. (2011). ANAPSID: an adaptive query processing engine for SPARQL endpoints. ISWC 2011, pp. 18-34.
2. Bizer, C., Heath, T., & Berners-Lee, T. (2008). Linked data: Principles and state of the art. WWW 2008, pp. 1-40.
3. Buil-Aranda, C., Arenas, M., Corcho, O., & Polleres, A. (2013). Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. Web Semantics: Science, Services and Agents on the World Wide Web, 18(1), 1-17.
4. Dragan, L., Möller, K., Handschuh, S., Ambrus, O., & Trüg, S. (2009). Converging Web and Desktop Data with Konduit. In Proc. of Scripting and Development for the Semantic Web Workshop.
5. Cyganiak, R., Wood, D., & Lanthaler, M. (2014). RDF 1.1 concepts and abstract syntax. W3C Recommendation, 25, 1-8.
6. Elbassuoni, S., & Blanco, R. (2011). Keyword search over RDF graphs. ACM CIKM 2011, pp. 237-242.
7. Ferré, S. (2017). Sparklis: an expressive query builder for SPARQL endpoints with guidance in natural language. Semantic Web, 8(3), 405-418.
8. García, G. M., Izquierdo, Y. T., Menendez, E. S., Dartayre, F., & Casanova, M. A. (2017). RDF Keyword-based Query Technology Meets a Real-World Dataset. EDBT 2017 – Industrial and Applications Paper, pp. 656-667.
9. Huang, J., Abadi, D. J., & Ren, K. (2011). Scalable SPARQL querying of large RDF graphs. Proceedings of the VLDB Endowment, 4(11), 1123-1134.
10. Möller, K., Dragan, L., Ambrus, O., & Handschuh, S. (2008). A visual interface for building SPARQL queries in Konduit. ISWC-PD 2008, pp. 68-69.
11. Nikolov, A., Schwarte, A., & Hütter, C. (2013). Fedsearch: Efficiently combining structured queries and full-text search in a sparql federation. ISWC 2013, pp. 427-443.
12. Prud'hommeaux, E., & Buil-Aranda, C. (2013). SPARQL 1.1 federated query. W3C Recommendation, 21.
13. Quilitz, B., & Leser, U. (2008). Querying distributed RDF data sources with SPARQL. ESWC 2008, pp. 524-538.
14. Rakhmawati, N. A., Umbrich, J., Karnstedt, M., Hasnain, A., & Hausenblas, M. (2013). A comparison of federation over SPARQL endpoints frameworks. In International Conference on Knowledge Engineering and the Semantic Web (pp. 132-146).
15. Rakhmawati, N. A., Umbrich, J., Karnstedt, M., Hasnain, A., & Hausenblas, M. (2013). Querying over Federated SPARQL Endpoints---A State of the Art Survey. arXiv preprint arXiv:1306.1723.
16. Saleem, M., Khan, Y., Hasnain, A., Ermilov, I., & Ngonga Ngomo, A. C. (2016). A fine-grained evaluation of SPARQL endpoint federation systems. Semantic Web, 7(5), 493-518.
17. Schwarte, A., Haase, P., Hose, K., Schenkel, R., & Schmidt, M. (2011). Fedx: Optimization techniques for federated query processing on linked data. ISWC 2011, pp. 601-616.
18. Volz, J., Bizer, C., Gaedke, M., & Kobilarov, G. (2009). Discovering and maintaining links on the web of data. ISWC 2009, pp. 650-665.
19. Zeng, K., Yang, J., Wang, H., Shao, B., & Wang, Z. (2013). A distributed graph engine for web scale RDF data. Proceedings of the VLDB Endowment 6(4), pp. 265-276).
20. Zenz, G., Zhou, X., Minack, E., Siberski, W., & Nejdl, W. (2009). From keywords to semantic queries—Incremental query construction on the Semantic Web. Web Semantics: Science, Services and Agents on the World Wide Web, 7(3), 166-176.