Taking SPARQL 1.1 extensions into account in the SWIP system

Fabien Amarger, Ollivier Haemmerlé, Nathalie Hernandez, Camille Pradel

► **To cite this version:**

Fabien Amarger, Ollivier Haemmerlé, Nathalie Hernandez, Camille Pradel. Taking SPARQL 1.1 extensions into account in the SWIP system. 20th International Conference on Conceptual Structures (ICCS 2013), Jan 2013, Bombay, India. pp. 75-89, 2013. <hal-01193096>

HAL Id: hal-01193096

https://hal.archives-ouvertes.fr/hal-01193096

Submitted on 4 Sep 2015

Taking SPARQL 1.1 Extensions into Account in the SWIP System

Fabien Amarger, Ollivier Haemmerlé, Nathalie Hernandez, and Camille Pradel

IRIT, Université de Toulouse le Mirail,
Département de Mathématiques-Informatique, 5 allées Antonio Machado,
F-31058 Toulouse Cedex
fabien.amarger@gmail.com,
{ollivier.haemmerle,nathalie.hernandez,camille.pradel}@univ-tlse2.fr

Abstract. The SWIP system aims at hiding the complexity of expressing a query in a graph query language such as SPARQL. We propose a mechanism by which a query expressed in natural language is translated into a SPARQL query. Our system analyses the sentence in order to exhibit concepts, instances and relations. Then it generates a query in an internal format called the pivot language. Finally, it selects pre-written query patterns and instantiates them with regard to the keywords of the initial query. These queries are presented by means of explicative natural language sentences among which the user can select the query he/she is actually interested in. We are currently focusing on new kinds of queries which are handled by the new version of our system, which is now based on the 1.1 version of SPARQL.

1 Introduction

The amount of knowledge available on the semantic web increases everyday. Many OWL ontologies and RDF triplestores are put online, especially in the context of the linked open data initiative [1]. Accessing this knowledge is a real challenge since it is difficult for an end-user to handle the complexity of the "schemata" of these pieces of knowledge: in order to express a valid query on the knowledge of the semantic web, the user needs to know the SPARQL query language, the ontologies used to express the triples he/she wants to query on as well as the "shape" of the considered RDF graphs.

Extensive work has been carried out in order to help users express queries in graph formalisms (CGs, SPARQL...) during the recent period. The help provided for the user can rely on graphical interfaces such as [2] for RQL queries, [3] and [4] for SPARQL queries or [5] for conceptual graph queries. But such graphical interfaces need the end-user to be familiar with and, moreover, to understand the semantics of the expression of a query expressed in terms of graphs. The work presented in [6] aims at extending the SPARQL language and its querying mechanism in order to take into account keywords and jokers when the user does not exactly know the schema he/she wants to query on. Here again, such an approach requires that the user knows the SPARQL language.

Other works aim at the automatic or semi-automatic translation of formal queries from user queries expressed in terms of keywords. The user expresses his/her information need in an intuitive way, without having to know the query language or the knowledge representation formalism used by the system. Some works have already been proposed to generate formal queries from keywords, resulting in different languages such as SeREQL [7], SPARQL [8,9] or conceptual graphs [10].

Our work belongs to this family of approaches. In [10], we proposed a way of building queries expressed in terms of conceptual graphs from user queries composed of keywords. In [11] we extended the system in order to take into account relations expressed by the user between the keywords he/she used in his/her query. In [12], we adapted our system to the Semantic Web languages instead of Conceptual Graphs. Such an adaptation was important for us in order to evaluate the interest of our approach on large knowledge bases. Since then, our system has taked into account queries expressed in natural language. Our work is based on two observations. First, end-users want simple query languages since they are used to this kind of querying on classic search engines on the Web. Second, in the main real applications, the submitted queries are variations of a few typical query families. We believe that each family can be prototyped and represented with a pattern. These observations led us to propose a mechanism allowing a user query expressed in terms of natural language to be translated into a SPARQL query built by adapting pre-defined query patterns chosen according to the natural language query.

The use of patterns allows us to avoid the step which consists in parsing the ontology in order to find potential relations which can be used to link the classes and instances identified in the natural language query, since the relevant relations appear in the pre-defined patterns. The process takes advantage of the relevant query families, which correspond to an actual information need. One of the main issues of our approach is therefore to select the pattern which best fits user needs. For the moment, patterns are built manually by domain experts.

The SWIP system presented in [12] was based on the 1.0 version of the SPARQL semantic web query language. At the beginning of 2012, SPARQL 1.1 [13] was released by the W3C. This new version features several improvements. We studied the new version and considered that the aggregates – which are more or less the same as in SQL – offer the possibility of handling new kinds of queries that were difficult to process in the previous version of SWIP. This article presents the extension of the SWIP system by taking into account the SPARQL 1.1 aggregates. For example, we are now capable of dealing with queries such as "How many artists are involved in a given film?", "The number of awards an artist won in a competition?" or "What is the average length of a film produced in 2012?".

In section 2, we present the SWIP system briefly. Section 3 introduces the aggregates in SPARQL 1.1. Section 4 details the extension of SWIP in order to implement the aggregates. Finally, section 5 describes the implementation of our system and presents a first experimentation.

2 The SWIP System

2.1 Overview

In order to allow end-users to express queries on knowledge bases expressed in the Semantic Web languages (RDF triples built on OWL ontologies), we propose a system by which a query expressed in natural language is translated into a SPARQL query. The SWIP[1] system was first presented in [11] in its preliminary Conceptual Graph based version, then in [12] in its SPARQL version. We do not present the SWIP system exhaustively in this article but briefly recall in this section the main features of the system and how it works, before focusing on the extensions of SWIP in section 4.

An overview of the process is presented in Figure 1. The global process of the system is as follows: (i) the user expresses his/her query in terms of a natural language query; (ii) the natural language query is then transmitted to a syntactic dependency analyzer which produces a dependency graph. This graph provides the main keywords of the query as well as the relations linking them ; they are represented by means of what we call a pivot query; (iii) the keywords and relations represented in the pivot query are matched to the elements of the ontology – concepts, instances and relations which seem to correspond to them ; (iv) the concepts, relations and instances obtained are mapped with the available query patterns; (v) explicative sentences corresponding to each possible SPARQL query are generated and proposed to the user, so that he/she can select the final query which best fits his/her information need; (vi) finally, the actual SPARQL query is generated and processed. In the following paragraphs, we present the process of the SWIP system in more details.

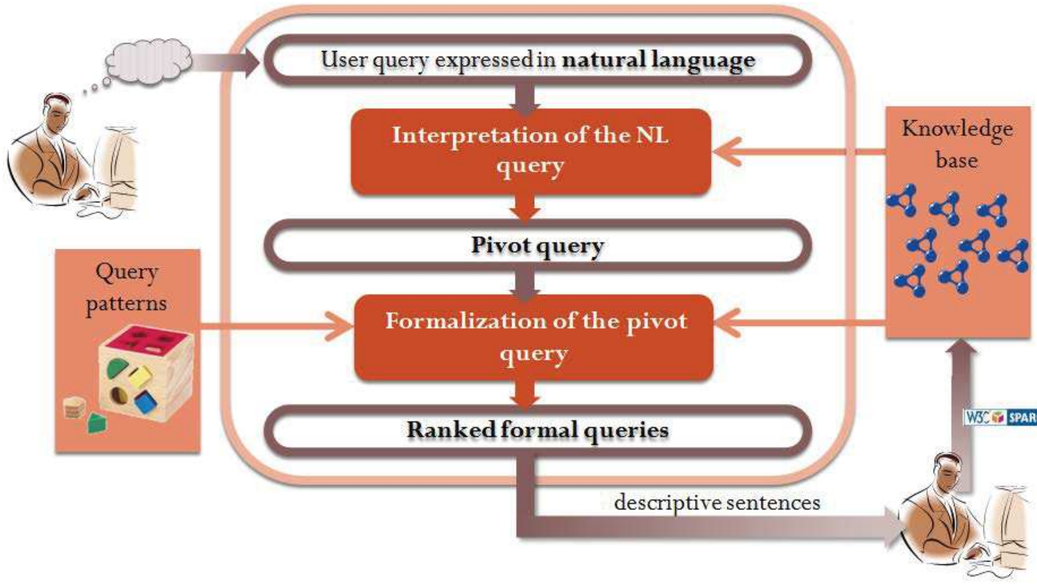
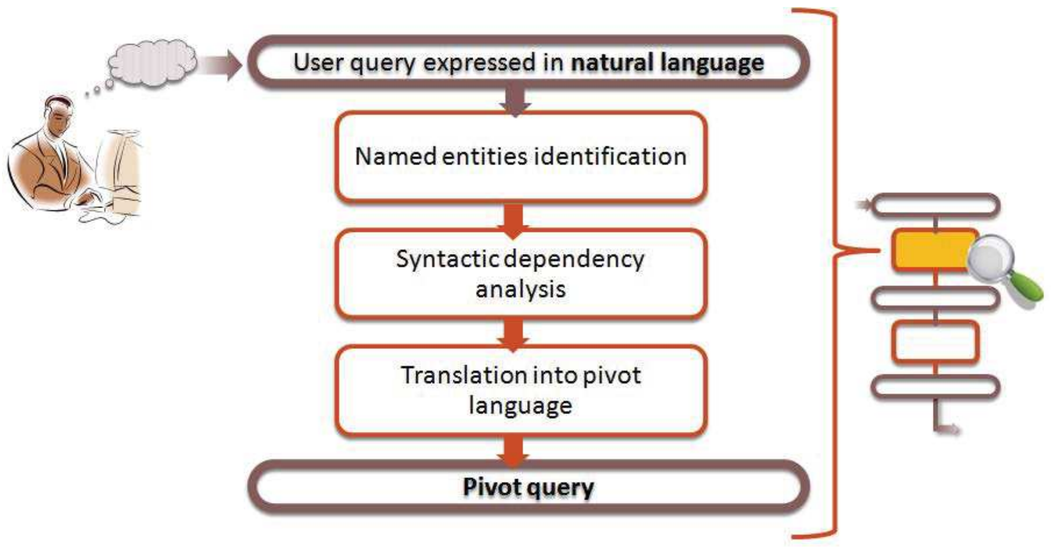

User query expressed in natural language
Interpretation of the NL query
Knowledge base
Pivot query
Query patterns
Formalization of the pivot query
W3C SPARQL
Ranked formal queries
descriptive sentences

Fig. 1. Overview of the process

[1] Semantic Web Interface using Patterns.

2.2 From Natural Language Queries to Pivot Queries

Here we describe the first main step of the SWIP process. This step is illustrated in Figure 2. The user enters his/her query in natural language into the system. The first translation is performed to generate the query in a simplified and synthetic form that we call *pivot language*. This language is based on keywords connected with relationships which are more or less explicit. The detailed grammar of the pivot language is presented in [12]. We use this pivot language in order to facilitate the implementation of multilingualism by means of a common intermediate format.



User query expressed in natural language
Named entities identification
Syntactic dependency analysis
Translation into pivot language
Pivot query

Fig. 2. Interpretation of the natural language query

The pivot language we propose is an extension of the language composed of keywords. The optional "?" symbol before a keyword means that this keyword is the focus of the query: we want to obtain specific results corresponding to this keyword.

A pivot query expressed in the pivot language is composed of a conjunction of subqueries:

- unary subqueries, like `?"singer"` which asks for the list of singers in the knowledge base;
- binary subqueries which qualify a keyword with another keyword: the query `?"singer": "married"` asks for the list of married singers;
- ternary subqueries which qualify, by means of a keyword, the relationship between two other keywords: the query `?"singer": "married to"= "Madonna"` asks for the singer(s) that is/are/was/were married to Madonna.

The translation of a natural language query into a pivot query is based on the use of a syntactic dependance analyzer which produces a graph where nodes correspond to the words of the sentence and edges to the grammatical dependencies between them. Before parsing the query with the analyzer, a first stage identifies in the sentence the named entities corresponding to knowledge base resources. These entities are then considered as a whole and will not be separated by the parser in the next stage. For example, in the sentence “what are the films of Jean Dujardin”, “Jean Dujardin” will be considered as a named entity as it is the label of an instance of Actor in the knowledge base. This stage is particularly crucial when querying knowledge bases containing long labels, such as group names or film titles made up of several words or even sometimes of a full sentence.

Once the named entities are identified and the dependency graph is generated, a set of rules are applied to construct the pivot query. The different clauses of the sentence are considered. First, the head of the expression playing the role of the subject in the main clause is identified as the keyword corresponding to the object of the query. Then, if there is one, the expansion of the expression is used to complete a binary subquery. For example, for the query “what are the films of Jean Dujardin”, the generated pivot query will be `?"film": "Jean Dujardin"`. If the clause is composed of a verb and a complement, a ternary subquery is constructed. For the query “what films were awarded prizes in Cannes?”, the corresponding pivot query will be `?"movie": "awarded prizes in"= "Cannes"`. If the query contains relative clauses, the same rules are applied. The entity referenced by the relative pronoun is expressed in the subquery with the keyword used in the previous subquery. For example for the query “What are the awarded films in which Jean Dujardin played?”, the corresponding pivot query will be `?"film": "awarded". "Jean Dujardin": "played in"= ?"film".` These rules might seem simple but we have observed that the structure of queries expressed by end-users is generally simple. Note that we use a specific syntactic analyzer, Maltparser [14], trained for each language we consider.

2.3 From Pivot to SPARQL

The second part of the process, which consists in the formalization of the pivot query, is illustrated in Figure 3. It is divided in four substeps which are described below. A more precise description of this step is given in [12].

Matching Keywords to Knowledge Base Entities. First, the ontology is used to determine which elements (concepts, instances or relations) are closest to the keywords appearing in the pivot query. This notion of closeness is based on the similarity measure between strings, as the one presented in [15]. However, in our system implementation, we do not use a classic method such as Levenshtein distance, but the Lucene score function, saving us the task of implementing a similarity measure and allowing us to benefit from the powerful Lucene indexation and “fuzzy matching” features, to handle different forms of lemmas and mistypings.

For example, with the pivot query `?"film": "Jean Dujardin’"`, SWIP searches for the labels corresponding to `"film"`, then for the labels corresponding to `"Jean Dujardin"`. For each element of the pivot query, SWIP returns a list of possible ontology entities weighted with respect to their relevance. The keyword `"film"` is associated with the concept “Film” with a weight of 1 since one of

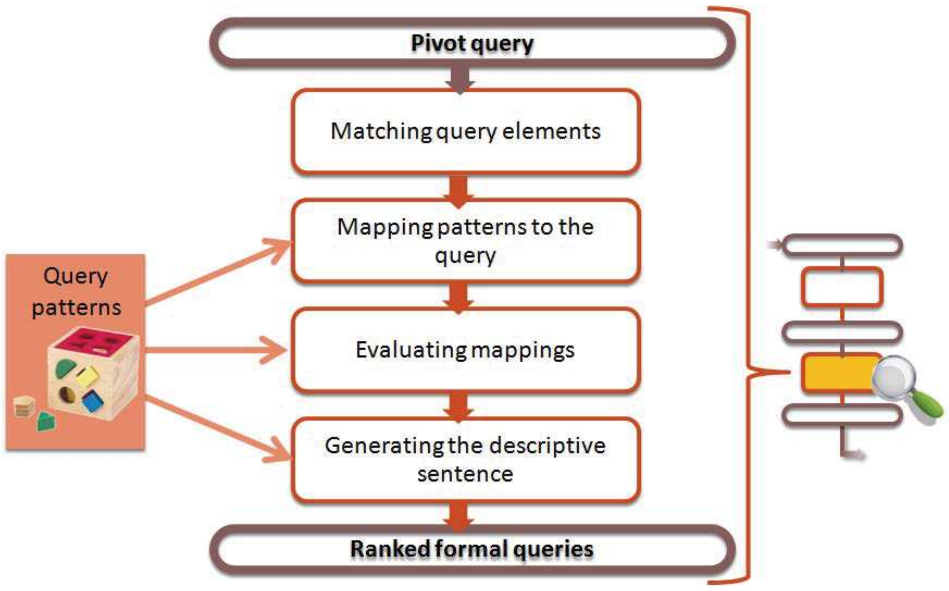


Fig. 3. Formalization of the pivot query

the labels of the concept "Film" is the string "Film". It is also matched with the concept "disaster film" because its label "disaster film' obtained a similarity measure of 0.625.

Mapping the Query Patterns to the Pivot Query. Once the ontology elements are identified, SWIP has to find one or more query patterns which fit these elements. The patterns are formally defined in [11,12]. Each pattern corresponds to a family of queries which can be asked on the knowledge base. Roughly speaking, each pattern is composed of: (i) a query graph which is a pre-written generic query expressed in SPARQL corresponding to an identified typical family of queries; (ii) a set of concepts and relations which belong to the query graph and correspond to the characteristic elements of the pattern; (iii) a model of an explicative sentence which will be used to generate the sentence in natural language allowing the user to understand the meaning of the SPARQL query. After the matching step, SWIP tries to associate the highest-weighted concepts with the different patterns in order to exhibit the patterns which seem to be used as a basis of relevant final queries.

Evaluating the Generated Mappings. The mapping step leads to a set of query mappings, each one corresponding to a possible query interpretation. These candidate interpretations must be ranked in order to present first to the user the queries which seem to be the most relevant. To this end, this step will allocate to each query mapping a *relevance mark* R, made up of several partial marks, each one taking into account a number of parameters that seem important to us:

– *Element mapping relevance mark* R_{map} represents how much we trust the different element mappings involved in the considered query mapping.

– *Query coverage relevance mark* R_{Qcov} takes into account the proportion of the initial user query that was used to build the mapping.
– *Pattern coverage relevance mark* R_{Pcov} takes into account the proportion of the pattern qualifying vertices that was used to build the mapping.

Generating the Query and the Explicative Sentence. The last step of the SWIP process consists in presenting the results to the user, and allowing him/her to query the knowledge base. For this, for each mapping between a set of keywords and a pattern we generate a sentence in natural language explaining the query represented by the mapping. We then present all the sentences to the user in decreasing relevance order. Thus, reading the explicative sentences, the user can easily understand the meaning of each query and choose the one matching his/her need. The system then formulates from the chosen mapping the final query expressed in SPARQL. Both operations, generating explicative sentences and formulating the query graph, are trivial, thanks to the explicative sentence attached to each pattern and to the graph architecture of each pattern.

For each mapping, the generation of an explicative sentence is carried out by taking the generic sentence attached to the mapped pattern and personalizing it, i.e. for each element mapping, the substring associated with the mapped pattern element is replaced by an appropriated string that comes from the matched label in the case of ontology elements or that is the string representation of the literal value in the case of literals. As regards pattern elements which are not involved in any element mapping, we keep the default associated substrings in the sentence.

The mapping generation is made more dynamic by adding the possibility of using regular expressions on parts of patterns, drawing ideas from [16]; some parts are omitted in the final mapping. This improves the readability of the explicative sentences of each mapping and makes patterns more generic (therefore less numerous).

The query graph of the selected query mapping is the pattern graph, apart from the odd detail; it is generated using the procedure presented in [10], except that relation vertices can be modified by specialization or generalization, in the same way as that for concept vertices.

3 SPARQL 1.1 Evolutions for Dealing with End-User Queries

3.1 SPARQL 1.1 Update

The W3C SPARQL 1.1 recommendation aims at improving the initial version of the language with incremental updates offering ascending compatibility with the first version [17]. The main issue is to ease the generation of queries by expanding the language syntax.

The most significant update for dealing with end-user queries is the possibility of using aggregate functions. In the initial version of SPARQL, it was possible to retrieve a solution set of triples according to a specific query pattern. With the use of aggregates, it is now possible to partition this set according to specified criteria and to compute a new solution using an aggregate on these partitions. When needed, aggregates were previously calculated by the applications. Now, however, they are calculated by the SPARQL engine.

The aggregate functions available in SPARQL1.1 are COUNT, SUM, MIN, MAX, AVG, GROUP_CONCAT and SAMPLE. As in SQL queries, aggregate functions can be used either as projection or as selection attributes.

COUNT
The COUNT aggregate enables the counting of the triples contained in the solution set as in the following query.

```
SELECT COUNT(?film)
WHERE
{
    ?film rdf:type cine:Film
}
```

The result of this query will be the integer corresponding to the number of instances of the class Film in the triplestore.

The aggregate can also be used in the condition defined to select the partitions to be considered as in the following query.

```
SELECT ?actor
WHERE
{
    ?film rdf:type cine:Film.
    ?actor rdf:type cine:Actor.
    ?film cine:hasForActor ?actor.
}
GROUP BY ?actor
HAVING COUNT(?film) > 5
```

This query will give all the actors who have played in more than 5 films. The "GROUP BY" clause specifies on which attribute the partitions are built and the "HAVING" clause specifies the condition that has to be satisfied by the partition.

SUM - AVG - MIN - MAX
The aggregates ("SUM" for sum, "AVG" for average, "MIN" for minimum and "MAX" for maximum) can be used in the same way as COUNT except that they have to be applied on numerical attributes.

GROUP_CONCAT - SAMPLE
These two aggregates are more specific to SPARQL 1.1. "GROUP_CONCAT" enables the generation of a string from all the triples contained in the solution

set. SAMPLE will randomly select an element in the solution set. The query below will return the concatenation of the URI of each film contained in the dataset, separated with the character “|”.

```
SELECT GROUP_CONCAT(?film ; separator="|")
WHERE
{
    ?film rdf:type cine:Film
}
```

3.2 Aggregates in End-User Queries

When generating SPARQL queries from natural language queries, considering aggregates can be interesting for two reasons. End-users may want to retrieve data that are not explicitly stated in the triplestore but can be calculated. Examples of queries can be: (i) “How many artists are involved in a given film?”, (ii) “The number of awards an artist won in a competition?”, (iii) “What is the average length of a film produced in 2012?”. As this information may be calculated, they will most probably not be stated explicitly with an RDF triple linking the considered resource to a property and to the corresponding value. In its current state, this is what our system will look for. This kind of information will have to be retrieved by generating the corresponding SPARQL query using the aggregate function. For example, in the triplestore we consider for our experiments, the SPARQL query corresponding to query i) will have to count all the artists involved in a given film. However, for some queries the same question words will not always lead to the use of aggregate functions. For example, for the query “How many members are there in the jury of a given competition”, the use of the aggregate “count” may not be relevant as a property can be explicitly stated. It is the case for this query as there is often no point in naming all the jury members, but only the number of members. As there are two ways of translating question words corresponding to aggregates in natural language into SPARQL, both possibilities will have to be considered when generating the SPARQL query. Aggregates will also be needed when the end-user wishes to select data according to comparison with other data. Examples of queries can be “What is the longest film ever made?”, “Who are the actors that have obtained more than 2 awards”, ...

In order to evaluate the added value of considering aggregates in the SWIP system, we asked 20 people to propose queries in natural language in the field of the cinema. The queries were manually translated into SPARQL1.1 to query a triplestore[2]. Among the 98 queries we collected, 9% of them implied the use of aggregates in the SPARQL query. Moreover, in all of these queries, aggregates are used as projection attributes. We thus decided to focus our work on this kind of query. By considering aggregates as projection attributes, our system is now able to deal with 99% of the queries asked by users. The remaining 1% corresponds

[2] The triplestore is described at `http://ontologies.alwaysdata.net/cinema`

to queries that contain terms that are not yet in the considered ontology (for example, SWIP cannot deal with the query “What is the filmography of Jean Dujardin?” as filmography is not in the ontology).

4 Evolution of SWIP 1.0 into SWIP 1.1

Several extensions to the SWIP system have been proposed in order to generate SPARQL queries with aggregate functions. The language proposed to represent pivot queries has been updated. Note that by considering pivot queries it is possible to dissociate treatments that are language-dependent from those that transform the pivot query into the SPARQL query. As the interpretation of the natural language query relies on the use of syntactic analyzers, we have improved the rules leading to the generation of the pivot query by taking into consideration typical words that may correspond to aggregates. The process generating the SPARQL query has also been improved.

Pivot Query Language. The language defined to represent pivot queries has been extended to take into account aggregate functions.

We propose to extend the pivot language by adding an optional subquery to the initial grammar described in this section.

This subquery corresponds to the case when an aggregate is used as a projection attribute. The subquery will be composed of one of the following keywords: COUNT, SUM, MIN, MAX, AVG. The semantics associated with the new subquery is that the aggregate function is applied on the projection attribute represented by the character `?` in the query.

Definition 1. *q is a well formed query in the pivot language if it conforms to the following grammar, given in Backus-Naur form.*

```
query ::= subquerySet ( "." subquerySet )* (".")? ( ("COUNT"
| "MAX" | "MIN" | "AVG" | "SUM" ) (".")? )?

subquerySet ::= keyword ( ":" keyword ( "="keyword ( ",
"keyword)*)? )?

keyword ::= ('a'..'z' | 'A'..'Z' | '_') ( ( 'a'..'z' |
'A'..'Z' | '0'..'9' | "_" ) )*
```

For example, the query “how many artists are involved in The Artist” corresponds to the pivot query : `?"artist":"involved in"="The Artist".COUNT`

From the Natural Language Query to the Pivot Query. The pivot query is built automatically from the query in natural language with rules defined on the output of a syntactic analyzer.

To detect queries which could need the use of an aggregate once transformed in SPARQL, we have built a dictionary for each language we consider (French and English) stating, for each aggregate and condition, terms that may refer to them. The dictionary has been constructed by analyzing manually a corpus of approximately 100 queries in each language. A sample of the dictionary for English is presented in the following table:

Aggregate	Corresponding words
COUNT	"The number of", "How many", ...
SUM	"The sum", "add", "adding", ...
MIN	"The minimum", "minimum",
MAX	"The maximum", "maximum",
AVG	"The average", "average", ...

The words defined in the dictionary are searched in the syntactic dependencies graph given by the analyzer.

If one of them is found to be depending directly on the head of the expression playing the role of the subject in the sentence (often found after the question word), a subquery composed of the corresponding aggregate is added to the pivot query. Dictionary words can be found in the direct context of the subject in the following queries : "*How many* artists are involved in a given film?", "What is *the number of* awards an artist won in a competition?", "What is *the average* length of a film produced in 2012?". The remaining parts of the sentences are analyzed as in the previous version of SWIP.

In order to deal with the case when the data does not need to be calculated with an aggregate function because it is explicitly stated in the data set with a property and its corresponding value, a pivot query is also constructed without paying any specific attention to words from the dictionary. The pivot query is thus generated as in the previous version of SWIP. For example, for the query "What is the number of members in the Oscar jury?", the second pivot query will be `"member":"?number","member":"jury"="Oscar"`.

From the Pivot Query to the SPARQL Query. To generate the SPARQL query, the two pivot queries are mapped to the patterns. For the query containing the aggregate subquery, the mapping process is the same as in the previous version of the system as we have chosen to ignore the possible aggregate subqueries composing the pivot query during this phase. The main reason is that we consider than aggregate functions can be used on any of the qualifying concepts of the patterns and that the need for an aggregate will not be an indication for discriminating one pattern from another. Patterns represent information needs that can be expressed differently. The same pattern can map a query which may or may not need an aggregate in its interpretation.

Once the patterns are ranked according to the pivot query, the eventual aggregate subqueries are considered for completing the generated descriptive sentences. The goal is to show the user how the aggregate has been interpreted. The name of the aggregate is added in brackets before the term corresponding to the subject of the aggregate.

The SPARQL query is generated according to the sentence chosen by the user. The aggregate is added in the SELECT clause.

5 Implementation and Experimentation

5.1 Implementation

The improvements presented in the previous section have been implemented in the SWIP system. A graphical user interface has been added to the system, developed with the JQuery technology. We have implemented a simple interface with two text fields and very few buttons, as can be seen in Figure 4. The first text field is used so that the user can enter his/her query in natural language. The "translate" button generates the pivot language query related to the natural language query. The "search" button generates all the possible SPARQL queries.

When the user clicks on the search button, SWIP searches all the possible mappings to generate the best SPARQL queries, which are displayed on a dynamic table, as can be seen on Figure 5. All the results are displayed sorted by score (here, the "rel" column) as is the generated sentence associated with the SPARQL queries directly. To indicate the searched element on the query on the sentence, it is preceded by the "?" character. There are some selections available directly on the sentence to allow the user to modify the query. For example, if he/she wants to generalize a specific element. On each row there is a "+" button to display all the details of the query, the SPARQL query associated with it

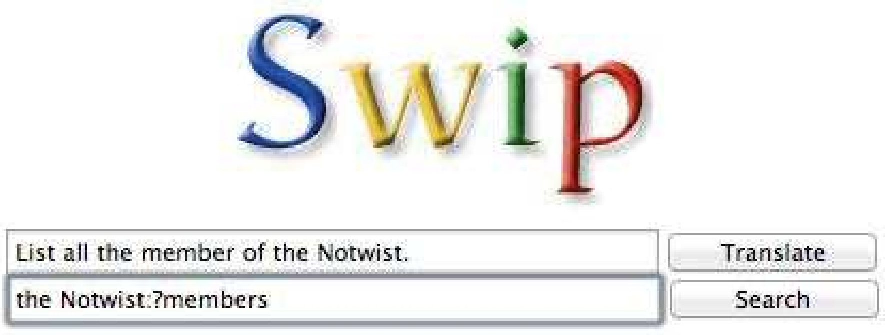
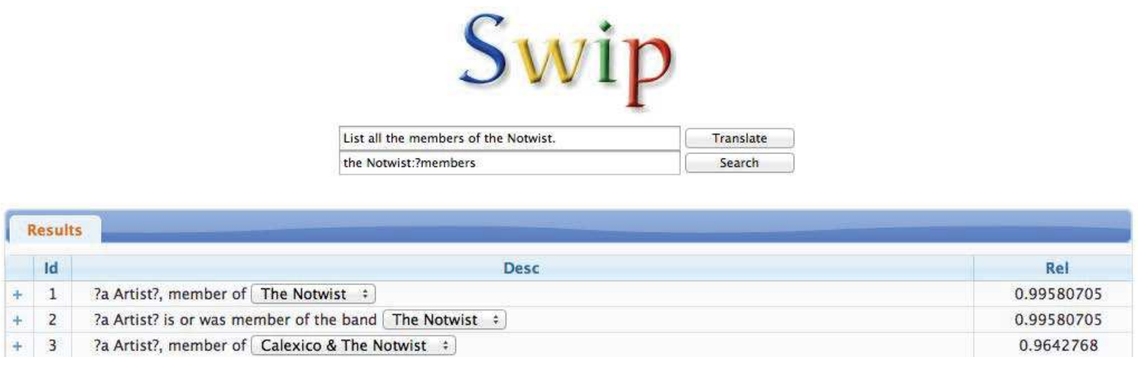


Fig. 4. SWIP interface



	Id	Desc	Rel
+	1	?a Artist?, member of The Notwist	0.99580705
+	2	?a Artist? is or was member of the band The Notwist	0.99580705
+	3	?a Artist?, member of Calexico & The Notwist	0.9642768

Fig. 5. SWIP interface - Generated SPARQL queries

and the mappings generated by SWIP. A double click on a row will execute the SPARQL query directly on the SPARQL endpoint to finally obtain the answer to the initial question. The answer is displayed on another table, next to the "Results" tab.

5.2 Experimentation

As explained in section 3, 9% of the queries collected on the field of the cinema implied the use of aggregates as projection attributes when translated manually into SPARQL.

In order to evaluate our approach, we compared the SPARQL queries generated by the new version of SWIP with the manually written query. For 100%, the "right" query appears in the first three propositions. This means that the generated pivot query is mapped relatively correctly with the right pattern. Moreover, the descriptive sentence obviously shows the right interpretation. This means that it is not difficult for the end-user to select the right query to generate.

We also considered the generated queries in which a dictionary word suggests that an aggregate function needs to be used but for which we have not used an aggregate in the manually built SPARQL query. This kind of query represents 4% of the total queries. For 50% of them, the first proposition made by SWIP is the correct one. When SWIP gives a wrong interpretation (i.e. an interpretation considering an aggregate), it is because the label of property that must be used in the query has not been identified correctly in the query. For example, in the query "how many members are there in the jury of the oscars" the property "numberOfMember" is mapped with too low a score to "how many members".

The evaluation is encouraging but we are currently looking for a larger set of queries to expand it.

6 Conclusion and Future Work

In this paper, we proposed a development of the SWIP system introduced in [10,11,12], for the system to take advantage of an enhancement of the SPARQL 1.1 query language: the aggregates. For this, we proposed an evolution of the pivot query language, and adapted the query interpretation process to make it take the aggregates into account.

The first evaluation results are very encouraging. We can now handle some queries that previously could not be processed.

The system has been provided with a graphical user interface which is very simple: the users are able to express their queries, and then choose the best query and modify it if necessary.

The preliminary step of building query patterns is done manually and thus requires a large amount of work. Moreover, this step must be repeated each time we want to address a new knowledge base. This is why the automatic or assisted pattern generation is the last important task we need to carry out to obtain a fully functional system. We have two potential leads for this purpose: building patterns covering a set of graph queries, or learning these patterns from a set of natural language queries. At first glance, the first method seems to be the easier to implement, since the input of this method consists of formal structures which are easy to handle. However, end user queries expressed in a graph formalism could be costly to obtain in practice, when natural language queries on a domain should be easy to find, looking on forums, FAQs, or simply asking users.

Moreover, we plan to extend our work in two other directions:

– we are currently looking for a larger set to evaluate our approach more precisely in order to identify its drawbacks; we are developing a partnership with the IRSTEA (a French institute on ecology and agriculture) in order to build a real application framework concerning French queries on organic farming.
– in the near future we intend to consider aggregate functions as selection attributes by means of nested subqueries which are also an evolution of SPARQL1.1.

References

1. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semantic Web Inf. Syst. 5(3), 1–22 (2009)
2. Athanasis, N., Christophides, V., Kotzinos, D.: Generating On the Fly Queries for the Semantic Web: The ICS-FORTH Graphical RQL Interface (GRQL). In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 486–501. Springer, Heidelberg (2004)
3. Russell, A., Smart, P.R.: Nitelight: A graphical editor for sparql queries. In: Bizer, C., Joshi, A. (eds.) International Semantic Web Conference (Posters & Demos). CEUR Workshop Proceedings, vol. 401. CEUR-WS.org (2008)
4. Ferré, S., Hermann, A.: Semantic Search: Reconciling Expressive Querying and Exploratory Search. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 177–192. Springer, Heidelberg (2011)
5. CoGui. A conceptual graph editor. Web site (2009), `http://www.lirmm.fr/cogui/`
6. Elbassuoni, S., Ramanath, M., Schenkel, R., Weikum, G.: Searching rdf graphs with sparql and keywords. IEEE Data Eng. Bull. 33(1), 16–24 (2010)
7. Lei, Y., Uren, V.S., Motta, E.: SemSearch: A Search Engine for the Semantic Web. In: Staab, S., Svátek, V. (eds.) EKAW 2006. LNCS (LNAI), vol. 4248, pp. 238–245. Springer, Heidelberg (2006)
8. Zhou, Q., Wang, C., Xiong, M., Wang, H., Yu, Y.: SPARK: Adapting Keyword Query to Semantic Search. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 694–707. Springer, Heidelberg (2007)
9. Tran, T., Wang, H., Rudolph, S., Cimiano, P.: Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In: ICDE, pp. 405–416. IEEE (2009)
10. Comparot, C., Haemmerlé, O., Hernandez, N.: An Easy Way of Expressing Conceptual Graph Queries from Keywords and Query Patterns. In: Croitoru, M., Ferré, S., Lukose, D. (eds.) ICCS 2010. LNCS, vol. 6208, pp. 84–96. Springer, Heidelberg (2010)

11. Pradel, C., Haemmerlé, O., Hernandez, N.: Expressing Conceptual Graph Queries from Patterns: How to Take into Account the Relations. In: Andrews, S., Polovina, S., Hill, R., Akhgar, B. (eds.) ICCS 2011. LNCS (LNAI), vol. 6828, pp. 229–242. Springer, Heidelberg (2011)
12. Pradel, C., Haemmerlé, O., Hernandez, N.: A Semantic Web Interface Using Patterns: The SWIP System. In: Croitoru, M., Rudolph, S., Wilson, N., Howse, J., Corby, O. (eds.) GKR 2011. LNCS, vol. 7205, pp. 172–187. Springer, Heidelberg (2012)
13. Harris, S., Seaborne, A.: Sparql 1.1 query language. w3c working draft (July 24, 2012), World Wide Web Consortium, `http://www.w3.org/TR/sparql11-query`
14. Nivre, J., Hall, J., Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., Marinov, S., Marsi, E.: Maltparser: A language-independent system for data-driven dependency parsing. Natural Language Engineering 13(02), 95–135 (2007)
15. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 707–710 (1966)
16. Alkhateeb, F., Baget, J.-F., Euzenat, J.: Extending sparql with regular expression patterns (for querying rdf). J. Web Sem. 7(2), 57–73 (2009)
17. Kjernsmo, K., Passant, A.: Sparql new features and rationale. World Wide Web Consortium, Working Draft WD-sparql-features-20090702 (2009)