Undefined 1 (2009) 1–16
IOS Press

A Natural Language Interface to Semantic Web using Modular Query Patterns A complete presentation of the SWIP system

Camille Pradel [a], Ollivier Haemmerle [a] and Nathalie Hernandez [a,*]
[a] *IRIT, Universite de Toulouse - Jean Jaures, 5, Allees Antonio Machado, F-31058 Toulouse Cedex 1, France*
E-mail: {hernande}@irit.fr

Abstract. Our purpose is to provide end users with a mean to query graph-based knowledge bases using natural language queries and thus hide the complexity of formulating a query expressed in a graph query language such as SPARQL. The main originality of our approach lies in the use of generic query patterns which are presented in this article and whose exploitation is justified by the literature. We also explain how our approach is designed to be adaptable to different user languages and emphasize the implementation of the approach relying on SPARQL. Evaluations on the QALD data set have shown the relevancy of the approach.

Keywords: SPARQL, Natural Language, Query Patterns, Question Answering

1. Introduction

This document provides instructions for style and layout of a double column journal article. End users want to access the huge amount of data available through the Internet. With the development of RDF triplestores and OWL ontologies, it became necessary to interface SPARQL engines, since it is impossible 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 Web of linked data, the user must know the SPARQL query language as well as the ontologies used to express the triples he/she wants to query on. Relational databases are hidden by means of forms which lead generally to SQL queries, and several works have been done on the generation of graph queries – in formal languages such as SPARQL or Conceptual Graphs for example – from keyword queries.We now believe that the automatic translation of NL queries into formal queries is a major issue of the next few years in the field of information acces.

Our work is situated in that field of research: how can we interpret a natural language (NL) query and translate it into SPARQL. The main postulate underlying our work states that, in real applications, the submitted queries are variations of a few typical query families. Our approach differs from existing ones in that we propose to guide the interpretation process by using predefined query patterns which represent these query families. The use of patterns avoids exploring the knowledge base (KB) to link the semantic entities identified from the keywords since potential query shapes are already expressed in the patterns. Moreover, the process benefits from the pre-established families of frequently expressed queries for which we know that real information needs exist.

In [1], we proposed a way of building queries expressed in terms of conceptual graphs from user queries composed of keywords. In [2], we extended the system in order to take into account relations ex-

*Corresponding author. E-mail: ollivier.haemmerle@univ-tlse2.fr.

pressed by the user between the keywords he/she used in his/her query and we introduced the pivot language which expresses these relations in a way inspired by keyword queries. In [3], we adapted our system to the Semantic Web languages instead of Conceptual Graphs [4,5]. Such an adaptation was important for us in order to evaluate the interest of our approach on real large KBs.

This article presents our approach through the *Semantic Web Interface Using Patterns* (Swip) system, which takes a NL query as input and proposes ranked SPARQL queries and their associated descriptive sentences to the user as output. Section 2 summarizes works and systems sharing our objectives. Section 3 presents an overview of the Swip system. Section 4 introduces the pivot language which is an intermediate format between the NL query and the SPARQL query, and which allows us to adapt our system to different natural languages easily. Section 5 describes the process of translation from NL to pivot query. Section 6 explains the notion of query patterns and Section 7 their utilisation for the query graph contruction. Then, we present the implementation of our work in Section 8 and evaluation results in Section 9, before concluding and stating future work in Section 10.

2. Related work

2.1. Usability

The relevancy of NL interfaces has been reconsidered several times in the literature, without leading to any clear-cut conclusion [6,7,8,9]. In [10], the authors evaluate the usability of interfaces allowing users to access knowledge through NL. They identify three main problems in such interfaces:

- the *linguistic ambiguity* of NL, i.e. the multiple ways the same NL expression can be interpreted, is the first and most obvious one;
- the *adaptivity barrier* makes interfaces with good retrieval performance not very portable; these systems are mostly domain-specific and thus hard to adapt to other domains;
- the *habitability problem* states that end users can be quite at a loss when facing too much freedom in query expression: the user can express queries outside the systems capabilities.

The main contribution of [10] is inspired by this last problem: the *habitability hypothesis* states that an NL interface, to present the best usability performance, should impose some structure on the user in order to help him/her during the query formulation process. To support their hypothesis, the authors describe the usability study they conducted on real users. For this, they developed four query interfaces and benchmarked them against each other with 48 users.

The authors break away from the traditional dichotomic classification which separates full NL from formal query approaches. They propose to situate each query system on a *formality continuum* according to the nature of its interface with the end user. Full NL interfaces are at one end of this continuum, while formal query languages lie at the other end. Thus, according to the habitability hypothesis, the most efficient interface from the usability point of view should lie somewhere towards the middle of the formality continuum. The systems developed for the usability study are naturally situated on this continuum: (i) *NLP-Reduce* [11] is a 'naive' and domain-independent NL interface in which queries can be expressed in the form of keywords, sentence fragments or full English sentences; (ii) *Querix* [12] is a domain-independent NL interface in which queries must be expressed in the form of full English questions beginning with "Which…", linebreak"What…", "How many…", "How much…", "Give me…", or "Does…"; (iii) *Ginseng* [13] is a guided input NL search engine that makes it compulsory for the user to formulate his/her query in the form of a sentence respecting a given grammar; (iv) *Semantic Cristal* is a graphical tool which helps the user build the formal queries.

The results of the study bring out that users felt much more confortable using Querix and its query language, which tends to confirm the habitability hypothesis. However, it is important to note that this users' feedback and their perception of the correctness of answers do not represent the actual correctness of the benchmarked systems. Indeed, NLP-Reduce obtained better performance on important objective criteria (average time per query and retrieval). Some extra interaction with the user performed by Querix (explicit word disambiguation, NL feedback) seems to create more confidence.

Despite these results, we clearly consider that an important issue in the coming years will be to generate queries from NL expressed orally through mobile devices. Users are more and more used to querying their terminals as if they were asking a question to a friend. In this context, full NL interpretation will be unavoidable. We nevertheless retain from this work the need

for interaction and feedback to the user. This point of view is reinforced by the recent work presented in [14] which exploits feedback and clarification dialogue to enhance user experience.

2.2. Existing approaches

In this section, we present approaches aiming at formulating graph queries developed so far. We stick here to the classification proposed in [10], and present them following the formality continuum, from the most to the least formal, beginning with the query languages themselves.

On one end of this continuum are the formal graph languages, such as SPARQL[1]. They are the targets of the systems presented in this section. Such languages present obvious usability constraints, which make them unadapted for end users. Expressing formal queries indeed implies knowing and respecting the language syntax used, understanding a graph model and, most constraining, knowing the data schema of the queried KB. Some works aims at extending the SPARQL language and its querying mechanism in multiple ways: by taking into account keywords and wildcards when the user does not know exactly the schema he/she wants to query on [15], by using regular expression to express property path [16], or by adding navigational capabilities of nested regular expressions [17]. Here again, such approaches requires that the user knows the SPARQL language.

Very close are approaches assisting the user during the formulation of queries in such languages. Very light interfaces such as *Flint*[2] and *SparQLed*[3] implement simple features such as syntactical coloration and autocompletion. Other approaches rely on graphical interfaces such as [18] for RQL queries, [19,20] for SPARQL queries or [21] for conceptual graphs. Even if these graphical interfaces are useful and make the query formulation work less tedious, we believe they are not well suited for end users since they do not overcome the previously mentioned usability limits of formal graph query languages.

Sewelis [22,23,24] introduces *Query-based Faceted Search*, a new paradigm combining faceted search and querying, the most popular paradigms in semantic data search, while other approaches such as *squall2sparql* [25] define a controlled NL whose translation into SPARQL is straightforward. Before using these systems correctly, end users need to get used to them.

Other works aim at generating automatically – or semi-automatically – formal queries from user queries freely expressed in terms of keywords or full NL. Our work is situated in this family of approaches. 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 express formal queries in different languages such as SeREQL [26] or SPARQL [27,28,29]. In these systems, the generation of the query requires the following steps: (i) matching the keywords to semantic entities defined in the KB, (ii) building query graphs linking the entities previously detected by exploring the KB, (iii) ranking the built queries, (iv) making the user select the right one. The existing approaches focus on several main issues: optimizing the first step by using external resources (such as WordNet or Wikipedia)[26, 30], optimizing the knowledge exploration mechanism for building the query graphs [27,28], enhancing the query ranking score [30], and improving the identification of relations using textual patterns [29].

Autosparql [31] extends the previous category: after a basic interpretation of the user NL query, the system converses with the user, asking for positive and negative examples (i.e. elements which are or are not in the list of expected answers), in order to refine the initial interpretation by performing a learning algorithm. A complete survey on NL to semantic web interfaces is available in [32].

In most of the aforementioned approaches, the final query graph is obtained by browsing the KB and linking its entities which have been identified in the user query. This step is lead by some heuristics and can result in a query graph which is inconsistent or meaningless, from a user point of view. We propose to define and exploit query patterns to lead the query building step; these manually built patterns ensure that the generated graph queries are always consistent for the user.

Regarding the evaluation task, only a few works have been proposed at the moment. [33] decribes an evaluation of usability and user satisfaction for semantic search query approaches on experts and casual users points of view. We can also mention the QALD (Question Answering over Linked Data) initiative. We participated to this challenge in 2011 and 2013.

[1] http://www.w3.org/TR/rdf-sparql-query/
[2] http://openuplabs.tso.co.uk/demos/sparqleditor
[3] http://sindicetech.com/sindice-suite/sparqled/

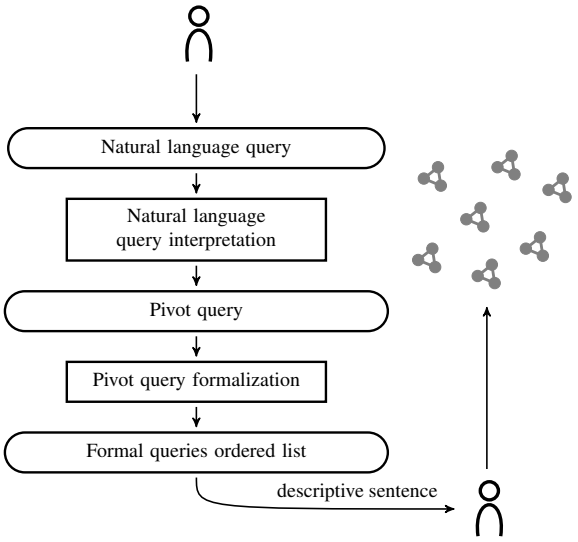

Fig. 1. Overview of the whole Swip interpretation process.

3. System overview and context

In the Swip system, the query interpretation process consists of two main steps which are illustrated in Figure 1. The first step, the *NL query interpretation*, is presented in Section 5; it roughly consists in the identification of named entities, a dependency analysis and the translation of the obtained dependency tree into a new query, called pivot query. The pivot query explicitly represents extracted relations between substrings of the NL query sentence and is presented in Section 4.

In the second step, *pivot query formalization*, presented in Section 7, predefined query patterns are mapped to the pivot query; we thus obtain a list of potential interpretations of the user query, which are then ranked according to their estimated relevance and proposed to the user in the form of reformulated NL queries.

The KB we consider in this work consists in a taxonomy of classes and properties, instances of classes, literals and relations expressed between these entities in the form of triples. It is formally defined below.

Definition 1 (Knowledge base) *A* knowledge base KB *is a tuple* (V, P, E) *where*

- V *is a finite set of vertices.* V *is conceived as the disjoint union* $\mathcal{C} \uplus \mathcal{I} \uplus \mathcal{L}$*, where* $\mathcal{C}$ *is the set of classes,* $\mathcal{I}$ *is the set of instances and* $\mathcal{L}$ *is the set of literal values. A literal value can be specified of some type;* ℓ *is the set of literal types; a function* $type : \mathcal{L} \to \ell \cup \emptyset$ *associates its type to each literal.*
- P *is a finite set of properties, subdivided by* $P = \mathcal{R}_o \uplus \mathcal{R}_d \uplus \{instanceOf, subclassOf, subpropertyOf\}$*, where* $\mathcal{R}_o$ *is the set of object properties,* $\mathcal{R}_d$ *is the set of datatype properties,* $instanceOf$ *expresses the class membership of an entity, and* $subclassOf$ *(resp.* $subpropertyOf$*) is the taxonomic relation between classes (resp. properties).* $\mathcal{R}_d$ *contains amongst other properties "*$label$*" which captures the terminological expression of an entity.*
- E *is a finite set of edges of the form* $p(v_1, v_2)$ *fulfilling one of the following conditions:*
  - $p \in \mathcal{R}_o$ *and* $v_1, v_2 \in \mathcal{I}$*,*
  - $p \in \mathcal{R}_d$ *and* $v_1 \in \mathcal{I}$ *and* $v_2 \in \mathcal{L}$*,*
  - $p = instanceOf$*,* $v_1 \in \mathcal{I}$ *and* $v_2 \in \mathcal{C}$*,*
  - $p = subclassOf$ *and* $v_1, v_2 \in \mathcal{C}$*,*
  - $p = subpropertyOf$ *and* $v_1, v_2 \in P$*,*

$\mathcal{E} = \mathcal{C} \cup \mathcal{I} \cup \ell \cup \mathcal{L} \cup \mathcal{R}_o \cup \mathcal{R}_d$ *is the set of all KB elements.*

Note that the popular KBs of the Web of linked data are mostly based on RDF schemas or OWL ontologies and can thus easily be reduced to the type of KB described above. For our evaluation purpose, we used *DBpedia*[4] and an RDF export of *Musicbrainz*[5] based on the *music ontology* [34].

In the following, we consider this shortcut function: $label : r \mapsto \{l | label(r, l) \in KB\}$.

4. The pivot query

4.1. Justification

As explained earlier, the whole process of interpreting an NL query is divided into two main steps, with an intermediate result, which is the user query translated into a new structure called the *pivot query*. This structure is half-way between the NL query and the targeted formal query, and aims at storing results from the first interpretation step. Briefly, it takes up keywords from the original NL query and expresses identified relations between those keywords; it can be expressed through a language, called pivot language. We use this intermediate result for two main reasons.

[4] http://dbpedia.org
[5] http://musicbrainz.org/

It facilitates the implementation of multilingualism by means of a common intermediate format: a specific module of translation of NL to pivot has to be written for each different language, but the pivot query formalization step remains unchanged. For the moment, we have experimented our system in English and French.

It captures relations between keywords which can be extracted from the user query dependency analysis. This information is mainly ignored by other approaches.

4.2. Formal definition of a pivot query

A pivot query is made of one or more subqueries. And a subquery is made of one or more query elements. In this section, we define these notions in bottom-up order.

The atomic element of a pivot query is called a query element. It can be a keyword (such as `"actor"`), a literal type (such as `date`), or a typed literal (such as `date<2013-04-05>`).

Definition 2 (Query element) *Let $\mathcal{K}_q$ be a dictionary containing the set of all keywords. The function val associates with each member $k \in \mathcal{K}_q$ its string value $val(k)$.*

Let ℓ_q be the set of all query elements referring to a literal type. The function val associates with each member $l \in \ell_q$ its corresponding literal type in the KB $val(l) \in \ell$.

Let $\mathcal{L}_q$ be the set of all query elements referring to a literal value. The function val associates with each member $L \in \mathcal{L}_q$ its corresponding literal in the KB $val(L) \in \mathcal{L}$, and thus $type(val(L)) \in \ell$ identifies the corresponding literal type.

$\mathcal{E}_q = \mathcal{K}_q \cup \ell_q \cup \mathcal{L}_q$ is the set of all query elements.

Query elements are ordered by sets of one, two or three, according to specific rules, to build subqueries.

Definition 3 (Subquery) *$\mathcal{S}_{q_1} = \mathcal{E}_q$ (resp. $\mathcal{S}_{q_2} = \mathcal{K}_q \times \mathcal{E}_q$, $\mathcal{S}_{q_3} = \mathcal{K}_q \times \mathcal{K}_q \times \mathcal{E}_q$) is the set of subqueries composed of one (resp. two, three) query element(s).*

$\mathcal{S}_q = \mathcal{S}_{q_1} \cup \mathcal{S}_{q_2} \cup \mathcal{S}_{q_3}$ is the set of all subqueries.

The Swip system supports three types of queries which are defined by the form of the expected result:

- *list queries* expect a list of resources fulfilling certain conditions as a result and their SPARQL translation is a "classic" `SELECT` query;
- *count queries* ask for the number of resources fulfilling certain conditions and correspond in SPARQL to a `SELECT` query using a `COUNT` aggregate as a projection attribute;
- *dichotomous queries* allow only two answers, Yes or No, and are expressed in SPARQL with an `ASK` query.

Definition 4 (Query type) *$\mathcal{T}_q = \{list, count, dichotomous\}$ is the set of query types.*

Finally, a pivot query is a finite set of subqueries associated with a set of focus elements and a given query type. Focus elements of a pivot query represent the main information required by the user and must obviously appear in this query; all query elements referring to a literal type must be focus elements and no literal value can be a focus element; a list query must contain at least one focus element.

Definition 5 (Pivot query) *$q = (S, F, T) \in 2^{\mathcal{S}_q} \times 2^{\mathcal{K}_q \cup \ell} \times \mathcal{T}_q$ is a pivot query iff: $F \subseteq \bigcup_{s \in S} s$, $F \supseteq \bigcup_{s \in S} s \cap \ell_q$, $F \cap \mathcal{L}_q = \emptyset$, and $|F| \geq 1$ if $T = list$.*

$\mathcal{Q}_q$ is the set of all pivot queries.

$\forall q = (S, F, T) \in \mathcal{Q}_q$, $focus(q) = F \in 2^{\mathcal{K}_q \cup \ell}$ denotes the set of focus elements of q, and $elem(q) = \bigcup_{s \in S} s$ denotes the set of query elements present in q.

4.3. Syntax of the pivot language

Although the pivot query is an intermediary result, we defined a language which formulates such pivot queries thereby facilitating communication and interaction during the interpretation process. The grammar of this language is defined in [3]. We give here a more intuitive presentation of its features.

The pivot language we propose is an extension of a classical keyword language, which explicitly declares relations between keywords. The optional "`?`" symbol before a query element means that this element is the focus of the query: we want to obtain specific results corresponding to this keyword. As presented in 4.2, a pivot query is composed of a conjunction of subqueries:

- unary subqueries, like `?"actor"` which asks for the list of actors in the knowledge base;

- binary subqueries which qualify a keyword with another keyword: the query `?"actor": "married."` asks for the list of married actors;
- ternary subqueries which qualify, by means of a keyword, the relationship between two other keywords: the query `?"actor": "married to"= "Penelope Cruz"` asks for the actor(s) that is/was/were married to Penelope Cruz.

The pivot language also allows a "factorization" of these subqueries, inspired by the Turtle RDF serialization language, as illustrated in the following example.

The query type is expressed at the end of a query, by a reserved uppercase keyword: `COUNT` (resp. `ASK`) implies a count (resp. dichotomous) query, no keyword implies a list query.

4.3.1. Example

A possible translation of the NL query "When was the album Abbey Road released?" is `?"Abbey Road": "album". "Abbey Road": "release"= ?date` (or in a more condensed form `?"AbbeyRoad": "album"; "release"= ?date`), whose representation using the previously introduced notations is a query q such as:

- $k_1, k_2, k_3 \in \mathcal{K}_q$, $val(k_1)$ ="Abbey Road", $val(k_2)$ ="album" $val(k_3)$ ="release",
- $l_1 \in \ell$, corresponding to type `xsd:date`,
- $q = \{\{(k_1, k_2), (k_1, k_3, l_1)\}, \{l_1\}\}$

5. From NL to pivot query

The translation of an NL query into a pivot query is based on the use of a syntactic dependency parser which produces a tree where nodes correspond to the words of the sentence and edges to the grammatical dependencies between them. As illustrated in Figure 2, the translation is done in several steps.

5.1. Named entity identification

Before parsing the query, the first stage identifies in the sentence the entities corresponding to KB 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 "who are the actors of Underworld: Awakening", "Underworld: Awakening" will be considered as a named entity as it is the label of an instance of Film in the KB. This stage is particularly crucial when querying KBs containing long labels, such as group names or film titles made up of several words or even sometimes of a full sentence. This stage relies on the use of gazetteers constructed according to the considered KB.

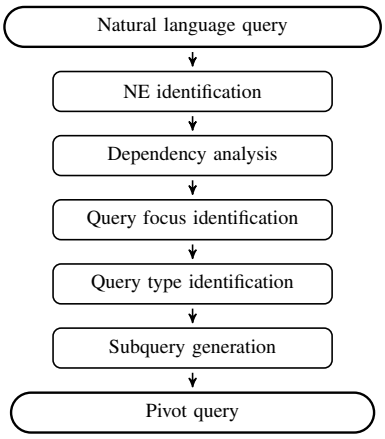

Fig. 2. Substeps of the NL query interpretation step.

5.2. Dependency analysis

Then, a dependency tree of the user NL query is processed by an external dependency parser, taking into account the previously identified named entities.

5.3. Query focus identification

This stage aims at identifying the query focus, i.e. the element of the query for which the user wants results (the element corresponding to the variable which will be attached to the SPARQL SELECT clause). The focus of the query is searched for by browsing the tree looking for an interrogative word. In the case when an interrogative word is found and an edge links it to a noun, the lemma of the noun is set to be the focus of the query. For example, for the query "Which films did Ted Hope produce?", the focus of the query will be "film". If such an edge does not exist, we set the focus query as being a keyword representing the type of answer expected by the interrogative word. For the query "When did Charles Chaplin die?", the focus will be "date". For the query "who played in the Artist?", the focus is "person". If no interrogative word is found in the tree, we look for specific phrases introducing list queries, which are queries that also expect a list of resources fulfilling certain conditions; such queries start with "List", "List all", "I am looking for", "What are","Give me"... In this case, the focus of the query

will be the lemma of the noun identified as the direct object of the verb in the clause in which the phrase is found. For example, in the query "List all the films in which Charles Chaplin acted", the focus will be "film".

5.4. Query type identification

If no query focus has been identified, we consider the query as dichotomous. This is the case for the query "Was Jean Dujardin involved in the film The Artist?" for which there will be no focus defined.

If at least one query focus was identified, we check for the presence of a specific phrase introducing count queries, such as "How many". If such a phrase is spotted, then the query is considered as a count query, otherwise it is a list query.

5.5. Subquery generation

The dependency tree is once more browsed in order to identify from the different clauses of the sentence the possible subqueries constituting the pivot query. A stop list containing typical stop words and words composing phrases introducing list queries is used. Nodes for which the lemma is in this list or for which the part of speech indicates it is a determiner or preposition are ignored.

The following rules are applied. If two nodes are part of a nominal phrase identified when their part of speech is a noun and when an edge representing a head/expansion dependency links them, a binary subquery is constructed according to the lemma of the noun playing the role of the expansion and the lemma of the noun playing the role of the head. For example, for the query "Give me all the films of Jean Dujardin", the generated binary query will be `?"film": "Jean Dujardin"`. If three nodes are part of a clause composed of a subject, a verb and an object, a ternary subquery is constructed according to the lemma of each node. For the clause "Who played in the Artist?", the corresponding ternary subquery will be `?"person": "play"= "the Artist"`. If the part of speech of a node is a relative pronoun, the keyword used for representing this node in a subquery will be the lemma of the noun it references. For example, for the query "Who played in a film in which Jean Dujardin also played?", the subquery corresponding to the second clause will be `"film": "play"= "Jean Dujardin"`, the subquery generated for the first clause is `?"person": "play"= "film"`. Note that the use of the same keyword allows us to express that the same film has to be considered in both subqueries.

For each node, each rule is analyzed. This means that several subqueries can be generated from one clause. For example, for the clause"Was Jean Dujardin involved in the film The Artist?", both subqueries `?"film": "The Artist"` and `"Jean Dujardin": "involved"= "The Artist"` will be generated.

These rules might seem simple but we have observed that the structure of queries expressed by end users is generally simple.

6. Query patterns

6.1. Justification

The main postulate underlying our work claims that queries issued by real life end users are variations of a few typical query families. The authors from [35] analyse query logs from a real Web search engine, discuss their results and compare them to previous similar studies [36,37,38]. Although first superficial observations tend to contradict our hypothesis, their conclusions after a deeper analysis confirm the relevancy of our approach.

The authors firstly point out that the vocabulary (and so potentially, for what matters to us, the semantics) of queries is highly varied. On the query set they analysed, including 926,877 queries containing at least one keyword, of the 140,279 unique terms, some 57.1% were used only once, 14.5% twice, and 6.7% three times. This represents a rather high rate of very rarely used terms, compared to "classical" text resources. These figures must be moderated, this phenomenon being partially caused by a high number of spelling errors, terms in languages other than English, and Web specific terms, such as URLs.

On the other hand, a few unique terms are used very frequently. In the queries analysed, the 67 most frequent meaningful terms (i.e. terms such as "and", "of", "the" were not taken into account) represent only 0.04% of unique terms that account for 11.5% of all terms used in all queries.

Moreover, two further analyses were carried out to complete these results. The first one addresses co-occurrence of terms; identifying the most frequently occurring pairs of terms highlights some popular and recurrent topics. The second is qualitative and is the most relevant to the semantics of queries. It consists in

the manual classification of a subset of the query set. 11 major categories were highlighted from this analysis, each one corresponding to a set of related topics.

Such observations led us to propose a mechanism for a user query to be expressed in NL and then translated into a graph query built by adapting pre-defined query patterns chosen according to the keywords.

6.2. *Intuitive presentation of patterns*

Examples 1, 2, 3 and 4 present some simple NL queries a user could ask on the cinema domain.

Example 1 *Who plays in the movie The Artist?*

Example 2 *Who is the director of the movie The Artist?*

Example 3 *Which movies were directed by Michel Hazanavicius?*

Example 4 *Which movies were directed by the Coen brothers?*

Figure 3 presents some query graphs which are formal translations of these NL query examples; these graphs target a KB on the cinema domain and the process of matching them to this KB would lead to the answers of each NL query (assuming that these answers are contained in the KB). For instance, the query in Subfigure 3(a) asks for the actors playing in the movie "The Artist". In the graph translation, the variable `?res` represents the queried resource; during the matching process of the query graph to the targeted KB, the set of all values taken by this variable will be considered as the answer to the query (see [39] for a formal definition of SPARQL and its semantics).

The pattern presented in Figure 4 is a very simple pattern covering all queries asking for the actors present in a particular movie. The node c_1 is called a qualifying element; it targets the class `cine:Movie` of the KB. This means that, to obtain the query graph corresponding to an actual query pertaining to the family identified above, we just have to substitute this qualifying element with the concerned movie instance. We call this process a *pattern instantiation*. The instantiation of the pattern 4 for the query from example 1 will lead to the query graph shown in Figure 3(a), by substituting the qualifying element with the resource `cine:TheArtist`.

Each query pattern also presents a template sentence which can be used to generate a descriptive sentence

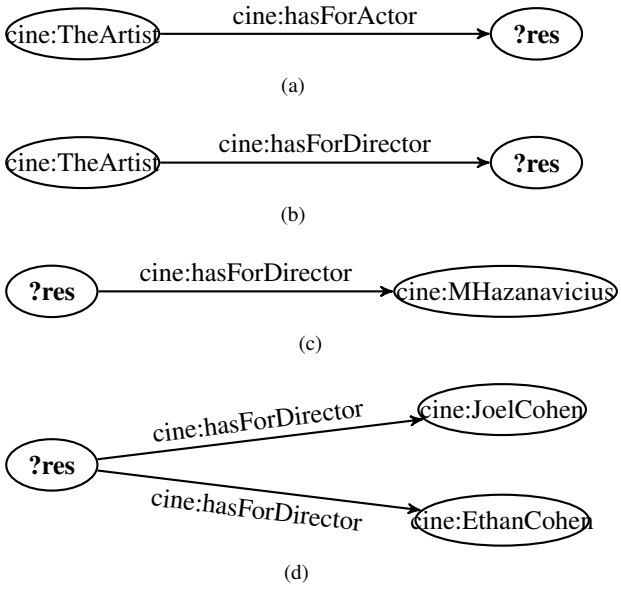

Fig. 3. Example graph queries targeting the cinema KB. Graphs 3(a), 3(b), 3(c) and 3(d) respectively correspond to NL queries from examples 1, 2, 3 and 4.

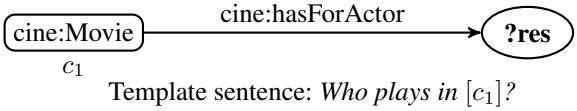

Fig. 4. Pattern issued from query 3(a).

representing the meaning of the query obtained after instantiating the pattern. This template is a succession of static strings and elements referencing the qualifying elements of the pattern graph. These referencing elements are intended to be substituted by a label of the resource which replaced the referred qualifying element during the instantiation process. For instance, the instantiation of pattern 4 for the query from example 1 will lead to the following descriptive sentence: *"Who plays in The Artist?"* (in this case, we obtain the exact same sentence because the sentence template was inspired from the example sentence). The template sentence can also be enriched with control structures in more advanced patterns (containing optional and repeatable parts), as presented later in this paper.

In the same way, we can build patterns 5(a) and 5(b), respectively inspired from query graphs 3(b) and 3(c). These patterns are quite similar, one asking for the director(s) of a particular movie, the other for the movie(s) made by a given director. Pattern 5(c) is the result of merging these two patterns; its graph thus

contains two qualifying elements c_1 and c_2, one referring to the class `cine:Movie` and the other to the class `cine:Director`. Since this pattern queries elements of different classes, the descriptive sentence template is no longer in the form of a question because we now have to anticipate both cases when either the instantiation of c_1 or the instantiation of c_2 is the focus of the query. The instantiation of this pattern for the query from example 2 will lead to the query graph in Figure 6 and the generated descriptive sentence will be: *"The Artist was directed by **some director**."* A substring of this sentence must be syntactically emphasized to inform the user on the query focus.

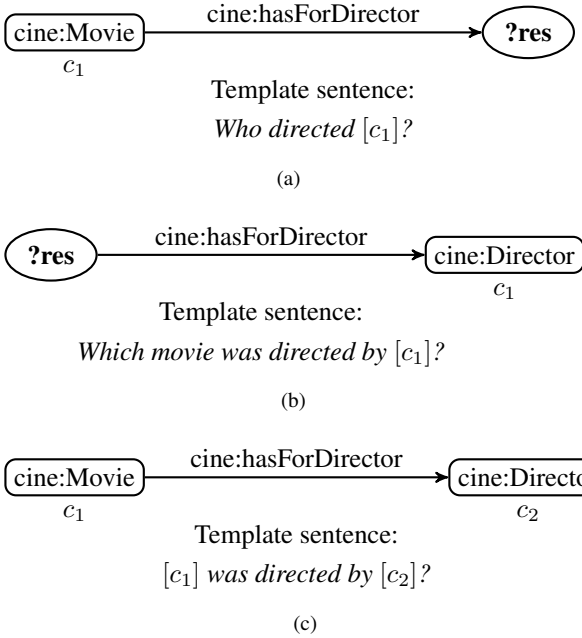

Fig. 5. Patterns 5(a) and 5(b) are respectively inspired by queries 3(b) and 3(c). Pattern 5(c) comes from merging patterns 5(a) and 5(b).

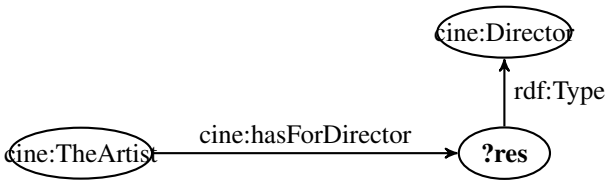

Fig. 6. Query graph issued from instantiation of pattern 5(c) for query from example 2.

Then, to allow the generation of graph query 3(d), pattern 5(c) must be instantiated by repeating the only

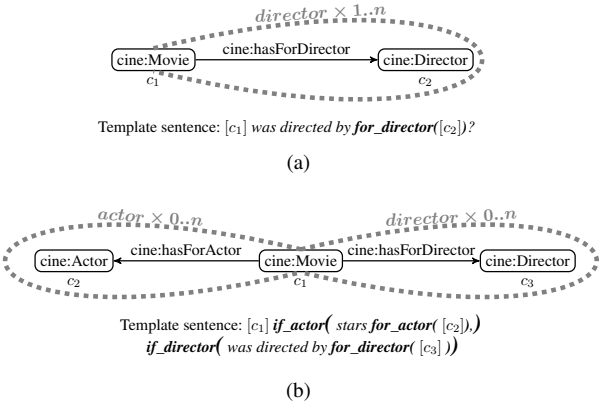

Fig. 7. Examples of complex patterns, containing optional or repeatable subpatterns.

triple constituting it, with two different instances for the Director class. This is enabled by making this triple repeatable during the pattern instantiation, assigning to it a maximal cardinality of n, as shown in Figure 7(a), which allows it to be repeated as many times as needed in a pattern instantiation. The instantiation of this pattern for the query from example 4 will lead to the query graph presented in Figure 3(d) and the generated descriptive sentence will be: *"**A movie** was directed by Joel Coen and Ethan Coen."*

In a similar way, it is possible to assign to a subpattern a minimal cardinality which, when equal to zero, makes this subpattern optional, i.e. it does not have to appear in a pattern instantiation. The pattern shown in Figure 7(b) is an evolution of pattern 7(a), containing a new optional triple to query for actors who played in a movie (or for the movies starring some actors).

6.3. Formal definition

A pattern is a special kind of subpattern. A subpattern is defined upon the notion of basic subpattern. A basic subpattern is made of pattern triples. A pattern triple is a set of three pattern elements. In this section, we define all these notions in bottom-up order.

Definition 6 (Pattern element) *Let $\mathcal{C}_P$ be the set of all class pattern elements. The function $targ$ associates with each member $c \in \mathcal{C}_P$ the class $targ(c) \in \mathcal{C}$ it targets.*

Let ℓ_P be the set of all literal type pattern elements. The function $targ$ associates with each member $l \in \ell_P$ the literal type $targ(l) \in \ell$ it targets.

Let $\mathcal{I}_P$ be the set of all instance pattern elements. The function $targ$ associates with each member $i \in \mathcal{I}_P$ the instance $targ(i) \in \mathcal{I}$ it targets.

Let $\mathcal{L}_P$ be the set of all literal value pattern elements. The function $targ$ associates with each member $L \in \mathcal{L}_P$ the literal $targ(L) \in \mathcal{L}$ it targets.

Let $\mathcal{P}_{o_P}$ be the set of all object property pattern elements. The function $targ$ associates with each member $p_o \in \mathcal{P}_{o_P}$ the object property $targ(p_o) \in \mathcal{P}_o$ it targets.

Let $\mathcal{P}_{d_P}$ be the set of all datatype property pattern elements. The function $targ$ associates with each member $p_d \in \mathcal{P}_{d_P}$ the datatype property $targ(p_d) \in \mathcal{P}_d$ it targets.

$\mathcal{E}_p = \mathcal{C}_P \cup \ell_P \cup \mathcal{I}_P \cup \mathcal{L}_P \cup \mathcal{P}_{o_P} \cup \mathcal{P}_{d_P}$ is the set of all pattern elements.

$\mathcal{E}_{qual} = \mathcal{C}_P \cup \ell_P \cup \mathcal{P}_{o_P} \cup \mathcal{P}_{d_P}$ is the set of potentially qualifying pattern elements.

A pattern triple is a list of three pattern elements respecting constraints expressed in Figure 8.

Definition 7 (Pattern triple) *$\mathcal{T}_P = (\mathcal{C}_P \cup \mathcal{I}_P) \times \Big(\big(\mathcal{P}_{o_P} \times (\mathcal{C}_P \cup \mathcal{I}_P)\big) \cup \big(\mathcal{P}_{d_P} \times (\ell_P \cup \mathcal{L}_P)\big) \Big)$ is the set of all pattern triples.*

A function $elem$ associates with each pattern triple $t \in \mathcal{T}_P$ the set of its constituting pattern elements. $\forall t = (e_{p_1}, e_{p_2}, e_{p_3}) \in \mathcal{T}_\mathcal{P}, elem(t) = \{e_{p_1}, e_{p_2}, e_{p_3}\}$

A basic subpattern is a set of pattern triples.

Definition 8 (Basic subpattern) *$\mathcal{B}_p = 2^{\mathcal{T}_P}$ is the set of all basic subpatterns.*

A function $elem$ associates with each basic subpattern $b \in \mathcal{B}_p$ the set of its constituting pattern elements. $\forall b \in \mathcal{B}_p, elem(b) = \cup_{t \in b} elem(t)$

Definition 9 (Subpattern) *Let $\mathcal{S}_p$ be the set of all subpatterns.*

If the 4-uple $s = (S, Q, c_{min}, c_{max}) \in 2^{\mathcal{S}_p \cup \mathcal{B}_p} \times 2^{\mathcal{Q}} \times \mathbb{N} \times \mathbb{N}^*$ is a subpattern, we note $cont(s) = S$ the set of (basic) subpatterns contained in s, $qual(s) = Q$ its set of qualifying elements, $card_{min}(s) = c_{min}$ and $card_{max}(s) = c_{max}$ its minimal and maximal cardinality respectively, and $elem(s) = \cup_{s' \in S} elem(s')$ the set of its constituting pattern elements.

A subpattern is defined recursively as follows:

1. *if $b \in \mathcal{B}_p$, b is connected, $Q \in 2^{\mathcal{E}_{qual}}$, $Q \subset elem(b)$, $c_{min} \in \mathbb{N}$, $c_{max} \in \mathbb{N}^*$ and $c_{min} \leq c_{max}$, then $(b, Q, c_{min}, c_{max}) \in \mathcal{S}_p$;*
2. *if $s_1, \ldots, s_n \in \mathcal{S}_p$, $n \geq 1$, $\bigcup cont(s_i)$ is connected, $Q \in 2^{\mathcal{E}_{qual}}$, $\cup qual(s_i) \cap Q = \emptyset$, $Q \subset \cup elem(s_i)$, $c_{min} \in \mathbb{N}$, $c_{max} \in \mathbb{N}^*$, $c_{min} \leq c_{max}$, $\forall i \neq j, qual(s_i) \cap elem(s_j) = \emptyset$, and $\forall i, \exists v \in elem(s_i)/v$ is a cut vertex of $\bigcup cont(s_i)$ (i.e. $\bigcup cont(s_i) \setminus v$ is non-connected and thus admits $cont(s_i) \setminus v$ as a connected component), then $(\bigcup cont(s_i), Q, c_{min}, c_{max}) \in \mathcal{S}_p$;*

In other words, a subpattern s_p is either:

- a basic subpattern attached to some of its vertices as qualifying vertices, and two integer values as minimal and maximal cardinalities, the former being logically lower than the latter,
- or a set of subpatterns attached to some vertices contained in these subpatterns as qualifying vertices, and two integer values as minimal and maximal cardinalities, such as:
  - minimal cardinality is lower than maximal cardinality,
  - the graph formed by the union of all subpattern graphs is connected,
  - for each included subpattern, its qualifying vertices do not appear in any other included subpattern nor in the set of s_p qualifying vertices,
  - each included subpattern contains a cut vertex of s_p graph; this last condition guarantees that the suppression of an optional subpattern will always lead to a connected query graph, and that the repetition of a subpattern will be made in a consistent way "around" this vertex which will be an anchor point to the rest of the query graph.

Definition 10 (Query pattern) *A query pattern is a subpattern which is not contained in any other subpattern. If p is a pattern, then $card_{min}(p) = card_{max}(p) = 1$.*

Let $\mathcal{P}_P$ be the set of all query patterns.

7. From pivot to formal query

We define here the semantics of the pivot language by presenting the interpretation that is made of a given pivot query $q \in \mathcal{Q}_q$ through a given query pattern $p \in \mathcal{P}_p$. Figure 9 shows all substeps of this process. Query patterns are mapped against the pivot query. The result of this process is a set of *pattern mappings*, each *pattern mapping* being a candidate interpretation of the pivot query (and thus the user query). In order to present to the user most relevant interpretations in priority, these *pattern mappings* are ranked according to a calculated *relevance mark*. Then, the generation for each mapping of the SPARQL query and the descrip-

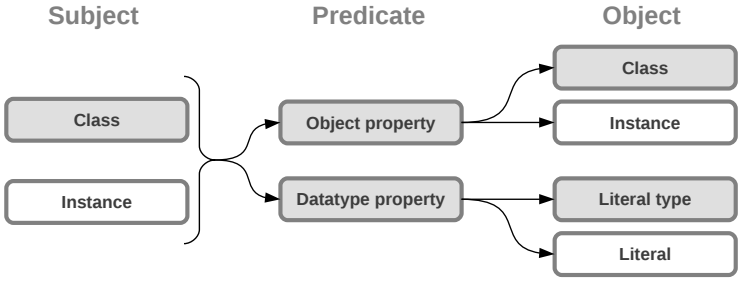

Fig. 8. Constraints on the elements constituting a pattern triple. Elements with a grey background can be qualifying elements.

tive sentence (which will be presented to the user) is straightforward thanks to the graph-shape of patterns and their sentence templates. Substeps are detailed in this section.

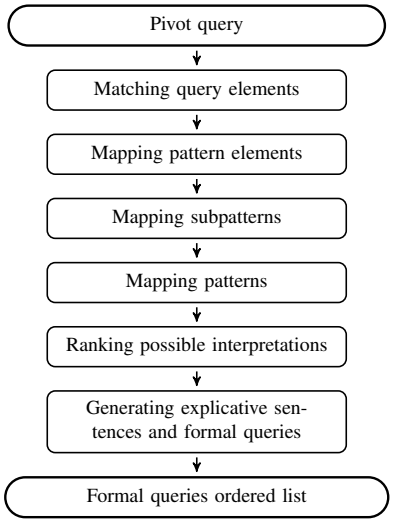

Fig. 9. Substeps of the pivot query interpretation step.

7.1. Matching query elements to KB elements

We consider a function $sim : \mathcal{K}_q^2 \rightarrow [0;1]$ defining a similarity measure between strings, such as $\forall k \in \mathcal{K}_q, sim(k,k) = 1$.

The function $sim : \mathcal{E} \times \mathcal{K}_q \rightarrow [0;1]$ returns the highest similarity measure between all labels of a considered resource $r \in \mathcal{E}$ and a string $k \in \mathcal{K}$: $\forall k \in \mathcal{K}_q, r \in \mathcal{E}, sim(r,k) = Max_{l \in label(r)} sim(l,k)$

The function $match^{KB} : \mathcal{E}_q \times]0;1] \rightarrow 2^{\mathcal{E} \times]0;1]}$ matching a query element $e_q \in \mathcal{E}_q$ to a knowledge base KB, with a threshold $\sigma \in]0;1]$ is defined as follows:

- if $e_q \in \mathcal{K}_q$, $match_\sigma^{KB}(e_q) = \{(r, sim) | sim = sim(r, val(e_q)) \geq \sigma\}$,
- if $e_q \in \mathcal{L}_q$, $match_\sigma^{KB}(e_q) = \{(val(e_q), 1)\}$,
- if $e_q \in \ell$, $match_\sigma^{KB}(e_q) = \{(val(e_q), 1)\}$.

7.2. Mapping pattern elements

The function $mapEE^{KB} : \mathcal{E}_{qual} \times \mathcal{E}_q \times]0;1] \rightarrow 2^{\mathcal{E} \times]0;1]}$ mapping a pattern element $e_p \in \mathcal{E}_{qual}$ to a query element $e_q \in \mathcal{E}_q$, with a threshold $\sigma \in]0;1]$ within the context of a knowledge base KB is defined as follows:

- if $e_p \in \mathcal{C}_P$ and $e_q \in \mathcal{K}_q$, $mapEE_\sigma^{KB}(e_p, e_q) = \{(e_p, e_q, r, t) | (r,t) \in match_\sigma(e_q, KB)$ and $(instanceOf(r, targ(e_p)) \in KB$ or $subclassOf(r, targ(e_p)) \in KB)\}$,
- if $e_p \in \mathcal{P}_{oP} \cup \mathcal{P}_{dP}$ and $e_q \in \mathcal{K}_q$, $mapEE_\sigma^{KB}(e_p, e_q) = \{(e_p, e_q, r, t) | (r,t) \in match_\sigma^{KB}(e_q)$ and $subpropertyOf(r, targ(e_p)) \in KB\}$,
- if $e_p \in \ell_P$ and $e_q \in \mathcal{L}_q$, $mapEE_\sigma^{KB}(e_p, e_q) = \{(e_p, e_q, r, t) | (r,t) \in match_\sigma^{KB}(e_q)$ and $type(r) = targ(e_p)\}$,
- if $e_p \in \ell_P$ and $e_q \in \ell$, $mapEE_\sigma^{KB}(e_p, e_q) = \{(e_p, e_q, r, t) | (r,t) \in match_\sigma^{KB}(e_q)$ and $val(r) = targ(e_p)\}$,
- otherwise $mapEE_\sigma^{KB}(e_p, e_q) = \emptyset$,

The result of this function is a set of element mappings. An element mapping maps a pattern qualifying element to a query element according to a previously established matching. When $(e_p, e_q, r, t) \in mapEE_\sigma^{KB}(e_p, e_q)$, we say that the pattern element e_p can be mapped to the query element e_q through the resource r and with the trust note t.

We also consider, for each qualifying pattern element e_p, an empty element mapping $\emptyset_M^{e_p} = (e_p, \emptyset, \emptyset, 0)$. In such a mapping, the considered pattern element is not mapped to anything.

The function $mapEQ^{KB}$ mapping a pattern element $e_p \in \mathcal{E}_{qual}$ to a pivot query $q \in \mathcal{Q}_q$, with a

$$mapEQ^{KB} : \mathcal{E}_{qual} \times \mathcal{Q}_q \times]0;1] \rightarrow 2^{\mathcal{E} \times]0;1]}$$
$$(e_p, q, \sigma) \mapsto \{mapEE_\sigma^{KB}(e_p, e_q) | e_q \in elem(q)\} \cup \emptyset_M$$

Fig. 10. Mapping a pattern element to a pivot query.

threshold $\sigma \in]0;1]$ within the context of a knowledge base KB is defined in Figure 10.

7.3. Mapping subpatterns

A mapping of a subpattern s_p consists in a set of element mappings for each of its qualifying elements $qual(s_p)$, associated with a subpattern mapping for each of its contained subpatterns $cont(s_p)$. $\mathcal{M}$ is the set of all subpattern mappings. The recursive function $mapSQ^{KB}$ mapping a subpattern $s_p \in \mathcal{S}_P$ to a pivot query $q \in \mathcal{Q}_q$, with a threshold $\sigma \in]0;1]$ within the context of a knowledge base KB is defined in Figure 11.

7.4. Mapping patterns

As query patterns are subpatterns, they are mapped in the same way. The result of this step is a set of pattern mappings. A pattern mapping m_p maps a pattern p to the considered pivot query q and constitutes a possible interpretation of the user query. We note $elemMap(m_p)$ the set of all element mappings contained in a given pattern mapping m_p.

7.5. Ranking pattern mappings

In order to present first to the user the query interpretation which seems to be the most relevant, a *relevance mark* R is processed for each pattern mapping m_p. This mark is made up of several partial marks, presented in Figure 12, each one taking into account a number of parameters determined as being relevant.

Element mapping relevance mark R_{map} (1) represents the confidence in the different element mappings involved in the considered pattern mapping. *Query coverage relevance mark* R_{Qcov} (2) takes into account the proportion of the initial user query that was used to build the mapping. *Pattern coverage relevance mark* R_{Pcov} (3) takes into account the proportion of the pattern qualifying vertices that was used to build the mapping. The final *relevance mark* R (4) is computed from previous partial marks

8. Full SPARQL implementation

A prototype of our approach has been implemented in order to evaluate its effectiveness. It is available at http://swip.univ-tlse2.fr/SwipWebClient. It has been implemented through web services and uses TreeTagger [40] for the POS tagging and MaltParser [41] for the dependency analysis of user queries. The system performs the second main process step (translating from pivot to formal query) by exploiting a SPARQL server based on the ARQ[6] query processor. In this section, we describe, in more detail, the implementation of the pivot to formal query translation step as it represents for us a real novelty: the interpretation of pivot queries is performed by exploiting SPARQL engine capabilities. Indeed, the SPARQL core feature consists in graph mapping, which is exactly the purpose of this interpretation step. This task is thus entirely carried out through SPARQL queries presented in this section. The result of each step is systematically committed into the KB, using SPARQL updates, which makes it available for subsequent steps.

Subsection 8.1 presents the OWL ontologies we developed to constitute a framework for the generated triples. Subsections 8.2, 8.3 and 8.4 give details on the first steps of the interpretation process. For the sake of brevity and simplicity, the following steps are not presented, but Figure 13 shows all implementation steps, each one corresponding to a SPARQL UPDATE or ASK (for emulating loops), and these corresponding SPARQL queries are given on the Swip presentation web page[7]. Subsection 8.5 discusses the benefits and drawbacks of this approach.

8.1. Ontologies for patterns and queries

In the rest of this paper, we express URIs with prefixes without making them explicit; they are common prefixes and must be considered with their usual value.

[6] http://openjena.org/ARQ/
[7] http://swip.univ-tlse2.fr/SwipWebClient/welcome.html

$$
\begin{aligned}
mapSQ^{KB} :& \mathcal{S}_P \times \mathcal{Q}_q \times]0;1] \rightarrow \mathcal{M} \\
& (s_p, q, \sigma) \mapsto \prod_{e_p \in qual(s_p)} mapEQ_\sigma^{KB}(e_p, q) \\
& \qquad \times \bigcup_{s'_p \in cont(s_p)} \Big(\cup_{c_{min} \leq c \leq c_{max}} \mathcal{P}_c\big(mapSQ_\sigma^{KB}(s'_p, q)\big) \Big)
\end{aligned}
$$

Fig. 11. Mapping a subpattern to a pivot query. $\prod$ and $\times$ refer to the Cartesian product and $\mathcal{P}_k(S)$ is the set of k-combinations from a set S.

$$R_{map}(m_p) = \frac{\sum_{(e_p,e_q,r,t) \in elemMap(m_q)} t}{|elemMap(m_q)|} \tag{1}$$

$$R_{Qcov}(m_q) = \frac{|\{e_q \in elem(q) \mid \exists (e_p, e_q, r, t) \in elemMap(m_q)\}|}{|elem(q)|} \tag{2}$$

$$R_{Pcov}(m_q) = \frac{|\{e_p \in qual(p) \mid \exists (e_p, e_q, r, t) \in elemMap(m_q)\}|}{|qual(p)|} \tag{3}$$

$$R(m_q) = \big(R_{map}(m_q)\big)^{\alpha_{map}} \cdot \big(R_{Qcov}(m_q)\big)^{\alpha_{Qcov}} \cdot \big(R_{Pcov}(m_q)\big)^{\alpha_{Pcov}} \tag{4}$$

$$\text{with } \alpha_{map} + \alpha_{Qcov} + \alpha_{Pcov} = 1 \tag{5}$$

Fig. 12. Computation of the partial and final relevance mark of a pattern mapping

We also define new prefixes, specific to entities defined in our ontologies: to the prefix name `patterns` is associated the URI http://swip.univ-tlse2.fr/ontologies/patterns, to `queries` is associated the URI http://swip.univ-tlse2.fr/ontologies/queries.

Our ontologies are built according to the principles of *normalization* design pattern, introduced in [42]. This method enables the development of modular and reusable ontologies, defining classes by the properties their instances should fulfill. The ontology developer does not need to care about the subsumption properties; the taxonomy can be automatically inferred by a reasoner. For this reason we present the *patterns* ontology by first introducing the main properties then the main classes composing it.

Figure 14 presents the object property hierarchy of the *patterns* ontology; properties are characterized by their domains and ranges. To each property corresponds its inverse property.

Property `patterns:makesUp` is the generic relation of meronymy; it is specialized by different subproperties which are chosen according to their range and domain, such as `patterns:isSubjectOf`, `patterns:isPropertyOf`, `patterns:isObjectOf`, `patterns:isSentenceOf` and `patterns:isSubsentenceOf`. `patterns:isSentenceOf` and `patterns:isSubsentenceOf` associates a descriptive sentence template with a query pattern. The vocabulary used to model the sentence template itself is not tackled here. The property `patterns:targets` expresses the relation between a pattern element and the resource (class, property or datatype) of the target ontology this element is referring to; it thus specifies mapping possibilities.

Data properties `patterns:hasCardinalityMin` and `patterns:hasCardinalityMax` specify cardinalities of subpatterns.

Figure 15 shows the hierarchy of the *patterns* ontology main classes, as well as their meronymy relations. It is worth noticing that, from this point of view, the ontology is not totaly faithful to the formal model given in 6.3. For instance, a pattern triple is considered as being a kind of subpattern, the concept of simple subpattern is not present, and the concept of subpattern collection (a subpattern which contains other subpatterns) appears. These slight changes where made in order to make the SPARQL interpretation simplier.

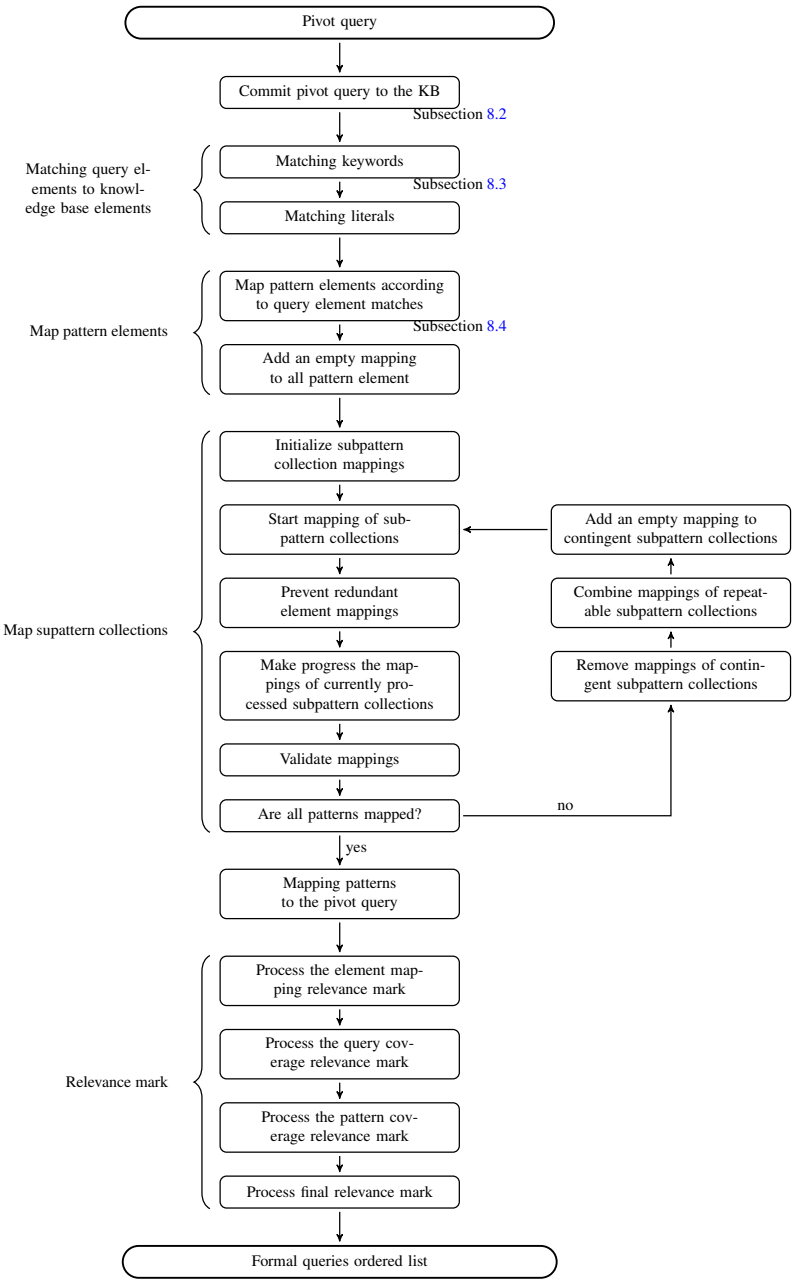

Fig. 13. Details of the pivot query formalization step.

Subpatterns are embodied by the class `patterns:Subpattern` which subsumes distinct classes `patterns:PatternTriple` and `patterns:SubpatternCollection`.

An instance of the class `patterns:PatternTriple` is a triple of the pattern graph. A triple is characterized by its subject (`patterns:hasSubject` property), its property (`patterns:hasProperty` property) and its object (`patterns:hasObject` property). Values of these properties are instances of the class `patterns:PatternElement`. A pattern element references a literal type (`patterns:LiteralPatternElement`), a class (`patterns:ClassPatternElement`) or a property (`patterns:PropertyPatternElement`) of the target ontology. For example, an instance of `patterns:ClassPat`

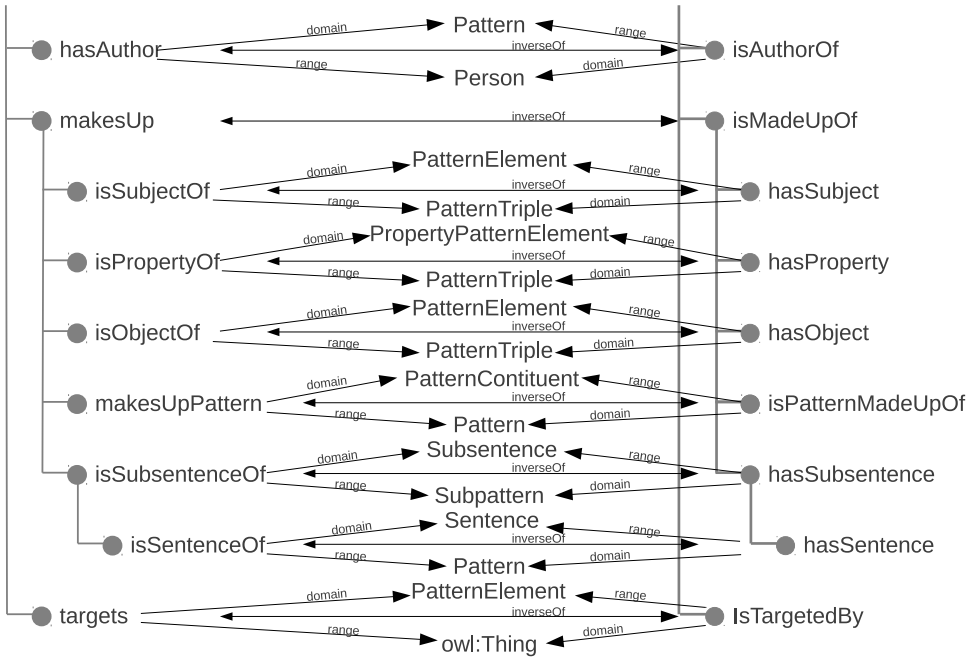

Fig. 14. Object properties of the *patterns* ontology

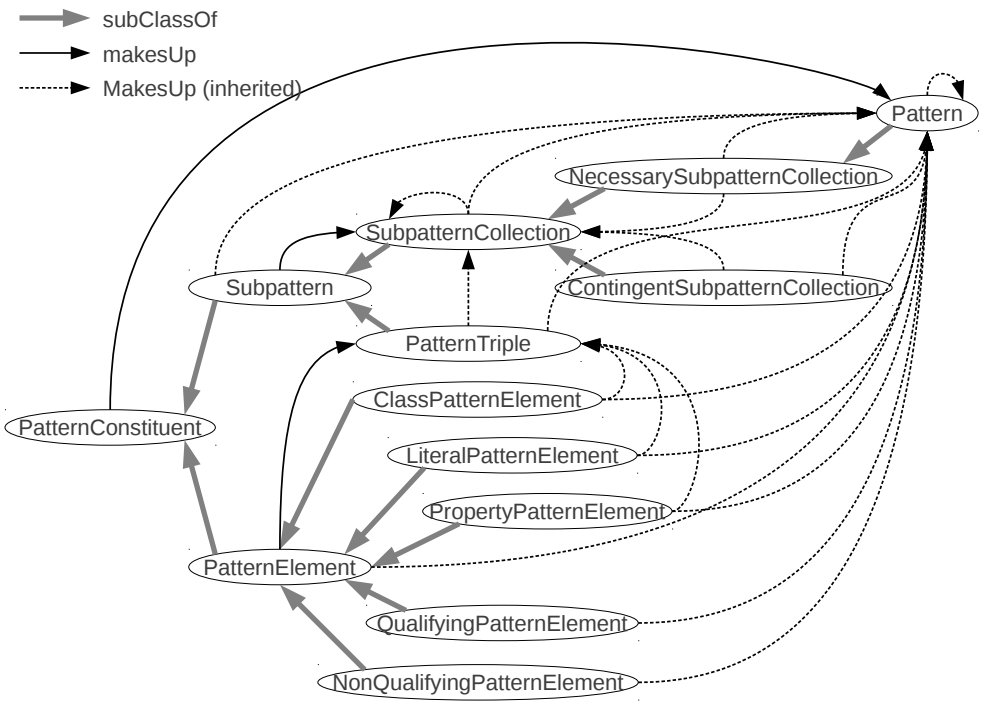

Fig. 15. The main classes describing query patterns, the taxonomic relations ordering them, as well as the possible meronymy relations between instances of these classes.

ternElement involved in a pattern will be linked by the relation patterns:targetKBElement to a class of the target ontology; this class will then, in this context, be considered as an instance. This method, called *punning*, is possible in OWL2. However, its usage cannot be modeled at the ontology level as this would require the usage of a part of reserved vocabulary (owl:Class, owl:Property, owl:Literal). owl:Thing is used instead of each of these forbidden entities, which does not differentiate between the three subclasses of patterns:Pat ternElement other than by declaring them distinct.

The *queries* ontology used to represent pivot queries and the results of the interpretation process is not detailed here as it was designed on the same principles as *patterns* and is much simpler. As explained in subsection 4.2, a pivot query is a set of subqueries, which are 1/2/3-tuples of query elements. The classes and properties of the *queries* ontology logically reflect this structure, their names are self-explaining, and an example of a pivot query and part of its interpretation results expressed in RDF and based on this ontology is used in the following subsections.

8.2. Committing pivot query to the KB

The system first processes a URI which is unique for each set of equivalent pivot queries. If this URI already exists in the KB, this query has been previously processed and the saved results can be exploited as they are. If not, the system generates an RDF graph representing the pivot query and commits it into the KB before starting the query interpretation. The RDF graph produced for the query ?"person": "produce"= "In Utero". "In Utero": "album" (pivot query translation obtained for the NL query "Who produced the album In Utero?" issued from QALD-3 training set) and based on the *Queries* ontology is shown on the left side of Figure 16. This figure shows the initial RDF data and also the results of the SPARQL updates which are processed during the matching step (cf. Subsection 8.3). RDF resources are represented in rounded nodes, literals in rectangle nodes, and properties are logically materialized by labeled edges. For the sake of readability, literal types are not shown, and resource classes are shown only when relevant.

8.3. Matching query elements to KB resources.

The first step of the pivot query interpretation consists in matching each element of the pivot query to KB entities (classes, properties and instances) or to literal types associating with each match a trust mark which represents the supposed quality of the matching and its likelihood.

Keyword matching is performed by processing a similarity measure between pivot query keywords and KB resources labels. To carry out this task, we use LARQ[8], an extension of SPARQL proposed by the ARQ[9] SPARQL engine, which exploits search functionalities of the Lucene[10] query engine. This extension introduces a new syntax, which determines both the literals matching a given string and a value representing the likeliness between them, called Lucene score: the "triple" (?lit ?score) pf:textMatch '+text' binds to ?lit all literals which are similar to the text string and to ?score the corresponding Lucene score. The SPARQL query used to perform this task is shown in Figure 17.

Figure 16 shows a subset of the bindings obtained by this query execution and the generated matching instances. The graph as it is before the update is shown in dotted lines; the part of this graph that is matched by the WHERE clause is highlighted in bold, and the committed resources and triples are shown in full lines. The following figures use the same presentation rules.

Note that although this step is performed using a (nonstandard) extension of the SPARQL language which makes it not very portable, an alternative can be implemented using standard features such as REGEX and CONTAINS string functions, or a simple string comparison.

8.4. Mapping pattern elements to query elements

Before being able to map the entire patterns to the user query, the first task consists in figuring out for each pattern element all conceivable mappings to user query elements – called *element mappings* – and their respective trust marks.

This step consists in creating a mapping between a query element and a pattern element when this query element was matched to a resource which is linked in some way to the pattern element target. We define several cases where a link is determined between a matched resource r and a pattern element target t:

[8] http://jena.apache.org/documentation/larq/
[9] http://jena.apache.org/documentation/query/
[10] http://lucene.apache.org/

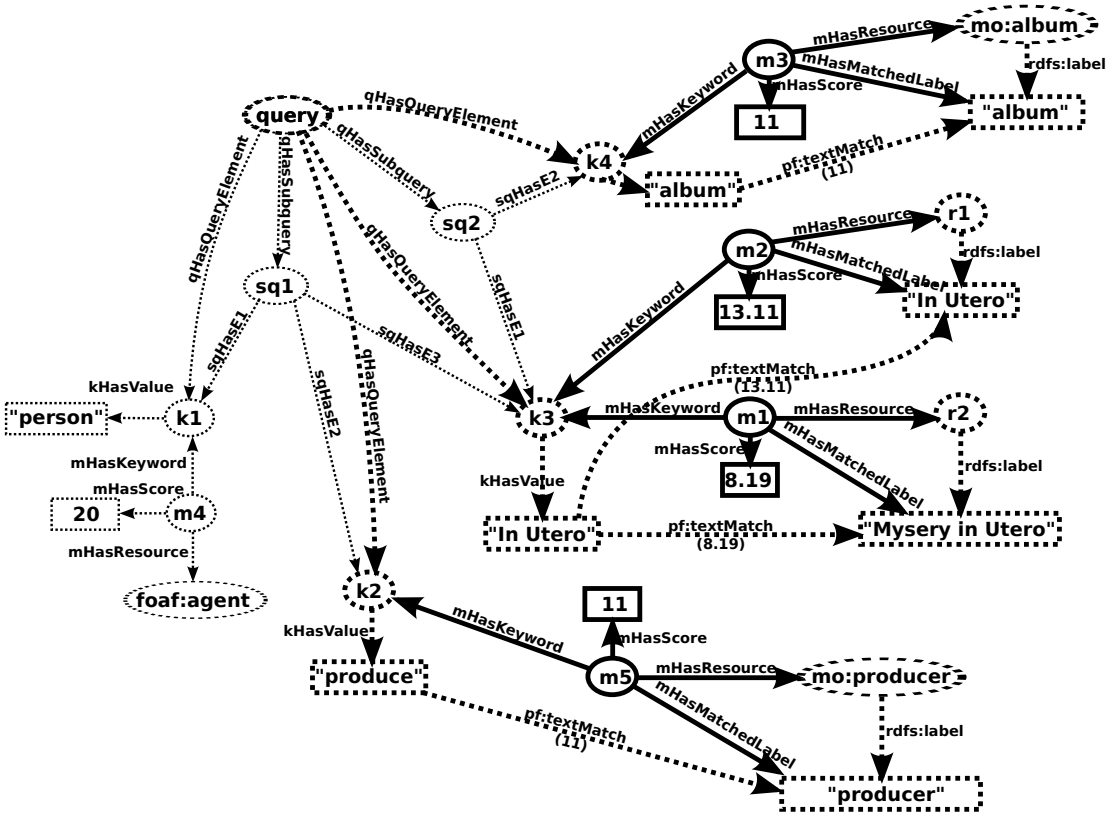

Fig. 16. The matching step on the example pivot query.

```
INSERT
{
  ?matchUri a queries:Matching;
            queries:matchingHasKeyword ?keyword;
            queries:matchingHasResource ?r;
            queries:matchingHasScore ?score;
            queries:matchingHasMatchedLabel ?l.
  ?keyword queries:keywordAlreadyMatched "true"^^xsd:boolean.
}
WHERE
{
  <[queryUri]> queries:queryHasQueryElement ?keyword.
  ?keyword a queries:KeywordQueryElement;
           queries:queryElementHasValue ?keywordValue.
  FILTER NOT EXISTS { ?keyword queries:keywordAlreadyMatched "true"^^xsd:boolean. }
  (?l ?score) pf:textMatch (?keywordValue 6.0 5).
  ?r rdfs:label ?l.
  BIND (UUID() AS ?matchUri)
}
```

Fig. 17. SPARQL update used for matching query elements.

1. r is a subclass of t (this case includes r is the class t itself),
2. r is a subproperty of t (this case includes r is the property t itself),
3. r is an instance of t (this case incorporates the special case illustrated in the upcoming paragraph on *"*Type" properties*),
4. r refers to the same literal type as t.

With each element mapping is associated a trust note whose value is the same as the involved match-

ing score. Figure 18 shows the instantiation of some element mappings through a SPARQL update. Pattern element `cd_info_element5` which targets class `foaf:agent` is mapped to keyword `k1` ("person") which matched the same class. Pattern element `cd_info_element1` which targets class `mo:Record` is mapped twice to keyword `k3` ("In Utero") which matched two instances of this same class. Pattern element `cd_info_element4` which targets property `mo:producer` is mapped to keyword `k2` ("produce") which matched the same property.

*"*Type" properties considered harmful* The instantiation of the element mapping `elemMap5` is issued from an extension of the first case stated above. This extension is due to the observation of a recurrent modelling choice made by some ontology developers, which consists in classifying instances of a class c by defining an object property with domain c, an (enumerated) class c' which represents the range of this object property and instances of c' which are the different ways to classify instances of c. For instance, in the *music ontology* [34], instances of class `mo:Record` can be involved as a subject in a triple with predicate `mo:releaseType` and with object an instance of class `mo:ReleaseType` (`mo:Album`, `mo:Single`, `mo:Live`, `mo:Soundtrack`...). It seems to us that this choice, although it must have been guided by some requirements, is not relevant, as it ignores the classification mechanism proposed by RDFS and OWL (i.e. typing instances using classes) to express a piece of knowledge that is indeed a classification. We support our thesis by pointing out two features we consider as symptomatic of this modelling flaw and are present in our example:

- in NL, the terms relative to the instances of c' are used in the same way as terms relative to classes; for instance, in the NL query (extracted from the QALD-3 challenge) "Give me all soundtracks composed by John Williams", the term "soundtracks" has the same place and function as the term "songs" in "Give me all songs by Aretha Franklin."
- the way the ontology developers themselves named the object property betrays the true nature of this property; indeed a property whose name ends with "Type" will very probably be used for typing instances and as such should be a subproperty of `rdf:type`.

Our approach overcomes this nongeneric modelling by explicitly identifying these cases. In Figure 18, the instance `wildcardReleaseType`, of type `WildcardTypeProperty` states that the resource `mo:album` should be considered during the mapping process as a subclass of `mo:Record`, which maps `cd_info_element1` to keyword `k4`; it also specifies that the typing property (which must be used while generating the final SPARQL query) is in this case `mo:release_type` instead of `rdf:type`.

Instance-class element mappings A last type of element mapping can be produced by the previously generated ones. These mappings, called *instance-class element mappings*, are issued from observing how users, when expressing themselves in NL, often specify a term referring to an instance by another term referring to the class of this instance. Some examples from the QALD-3 challenge are "the band Dover", "the album In Utero", "the song Hardcore Kids".

Such NL formulations are translated in the pivot language into a two element subquery, composed of the keyword referring to the instance qualified by the keyword referring to the class; for instance "the album In Utero" becomes "In Utero: album". Thus a particular kind of element mapping is used to handle this case. Such a mapping maps one pattern element to two keywords. Figure **??** uses the same example and illustrates the instantiation of an instance-class element mapping. `elemMap1` and `elemMap4` map `cd_info_element1` to respectively `k3` ("In Utero") and `k4` ("album"), and the resource matched by `k3` is an instance (taking into account the previous remark on "*Type" properties) of the resource matched by `k4`; this provides a new element mapping `elemMap6` mapping the considered pattern element to both query elements. Its score is the sum of trust notes of the two originally involved element mappings.

see instanceClassElementMapping.pdf

8.5. Benefits and drawbacks

The architecture described above presents several benefits. One of the most obvious is the ease of use of a cache system. As the result of each processing step is committed into the RDF repository, it is straightforward to reexploit previously generated data. For instance, as explained in Subsection 8.2, the Swip system realises that an incoming pivot query has previously been processed when its URI is already present in the repository and the result of its interpretation can

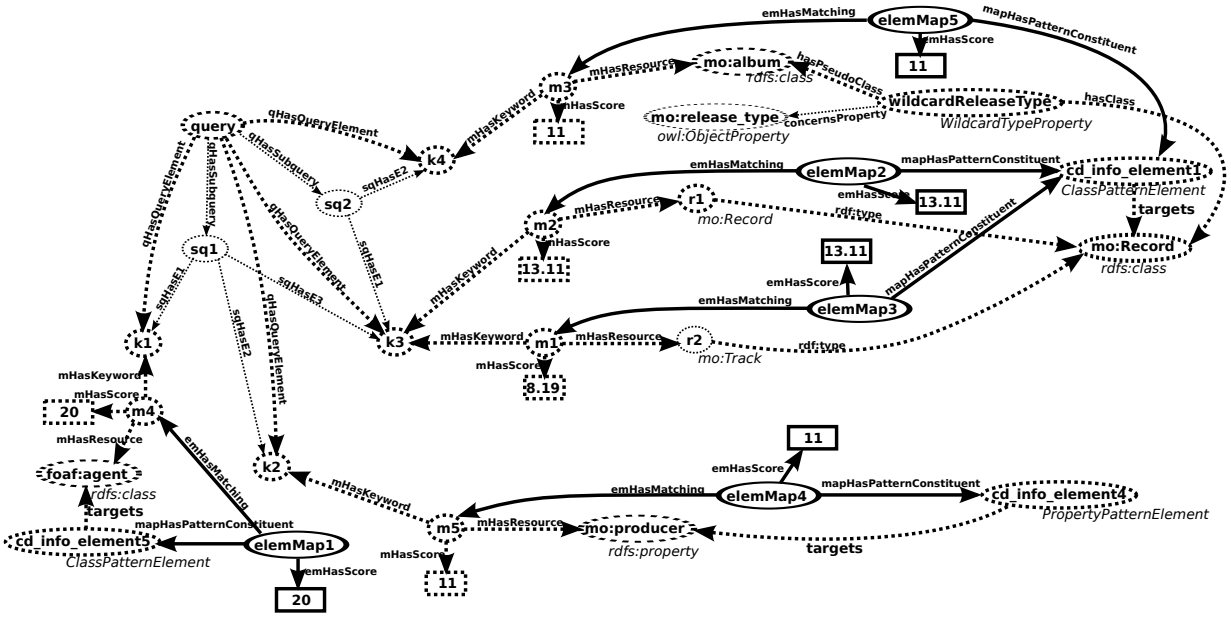

Fig. 18. The element mapping step on the example pivot query.

directly be returned. Also, the matching of a given keyword is performed only once for all queries containing this same keyword.

Moreover, with this architecture, there is a total and natural asynchronism between client and server sides: once the query URI is processed, it is returned to the client which can then update results of the interpretation progressively and independantly from the server, by directly querying the RDF repository.

This approach also ensures a total homogeneity and consistency as input, intermediary and output data is stored and manipulated through standard technologies. Even the configuration and tuning of the system can be made that way; for instance, the identified "*TypeProperties" (cf. Subsection 8.4) and some useful matchings (i.e. keyword matching at the ontology level which could not be determined by similarity measure between strings, like between keyword "husband" and property `rel:spouseOf`) are directly expressed in RDF and exploited as such during the interpretation. It is also worth noticing that, although the first part of the approach was implemented in a more "traditional" way, using web services, this could be changed exploiting the *NLP2RDF* recent initiative [43] which aims at providing results of popular NLP tools in a common format, called *NIF* (NLP Interchange Format) and integrated in the semantic web framework.

Two major drawbacks mar these qualities. The first one is the lack of control on the SPARQL queries processing and thus on the overall system performance. The SPARQL server is used as a blackbox and its efficiency directly impacts that of the interpretation process. Experience showed that ARQ server is not good with big queries (more than twenty triples in several subqueries) and that splitting these complex queries into an equivalent set of successive queries was much more efficient.

Moreover, SPARQL is still a relatively young recommendation and, despite the novelties brought by SPARQL 1.1, proposes a limited range of functions. For instance, there are very few arithmetic functions, and a degraded solution had to be found to process the final relevance mark (Figure 12, line 4), which implies the power operator.

9. Evaluation.

Experiments were carried out on the evaluation framework proposed in task 1 of the QALD-3 challenge[11]. The evaluation method was defined by the challenge organizers. It consists in calculating, for each test query, the precision, the recall and the F-measure of the SPARQL translation returned by the

[11] http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/index.php?x=task1&q=3

system, compared with a handmade gold standard query. We participated in both subtasks proposed by the challenge organizers, one targeting the DBpedia[12] KB and the other targeting an RDF export of Musicbrainz[13] based on the music ontology [34]. Although this task proposed user NL queries in several languages, we took into consideration only questions in English. The quality of the results varies with the target KB. A detailed and critical analysis of these results is presented in [44]. This section sums up and extends this analysis.

9.1. Results overview

The Musicbrainz test dataset was composed of 50 NL questions. We processed 33 of them. 24 were correctly interpreted, 2 were partially answered and the others failed. The average precision, recall and F-measure, calculated by the challenge organizers, are all equal to 0.51. It is difficult to evaluate these performances, as no other challenge participant tackled the Musicbrainz dataset. However, these results are quite good when compared to those obtained by participants on the DBpedia dataset. Indeed, among the other participants, the squall2sparql system showed the best performance by far, but this system does not accept full NL sentences as an input [25], and then, the second best F-measure is 0.36. Of course the relevance of such a direct comparison is very questionable, as both graph bases present significant differences and our system obtained poor results on DBpedia, for reasons that are explained later.

The results obtained by Swip on the DBpedia dataset, composed of 100 test questions, are not as good as our results on Musicbrainz. We processed 21 questions, of which 14 were successful, 2 were partially answered and 5 were not correct. The precision (0.16), the F-measure (0.16) and the recall (0.16) are quite low.

9.2. Pattern construction and coverage

The Musicbrainz's query patterns used for this evaluation were built by adapting (to fit the new ontology) and updating (to take into account the 50 new training queries) those created for QALD-1. We thus obtained five query patterns which can be visualized through the Swip user interface[14]. Among the set of 50 test questions, only two required query graphs which could not be generated by one of the patterns[15]. This fairly high coverage rate shows that our approach is suitable for closed domain KB, whose data is consistent with the schema.

For DBpedia, patterns were built "from scratch" with the QALD training questions. Without taking into account the "out of scope" questions, we implemented all questions in patterns; but when the QALD test questions were submitted, we noticed that only 19% of them were covered by the training patterns. In comparison, Musicbrainz patterns covered 96% of the test questions.

The main problem for us was the "tidiness" of the processed data. Indeed, most of the DBpedia KB is automatically generated from information written by thousands of contributors and not exported from a consistent database like Musicbrainz. Consequently, we had to deal with the heterogeneity and inconsistency of the target KB that Swip was not able to overcome. For instance, the nickname of an entity can be expressed in the KB by four distinct properties (`dbo:nickname`, `dbo:nicknames`, `dbp:nickname` and `dbp:nicknames`). As another example, consider the question "Which states of Germany are governed by the Social Democratic Party?" Its SPARQL translation is presented in Figure 19, the value of the property `dbp:rulingParty` can be either a URI referring to the Social Democratic Party, or an RDF plain literal.

```
SELECT DISTINCT ?uri
WHERE {
  ?uri rdf:type yago:StatesOfGermany .
  { ?uri dbp:rulingParty 'SPD'@en. }
   UNION
  { ?uri dbp:rulingParty
      res:Social_Democratic_Party_of_Germany. }
}
```

Fig. 19. Gold standard SPARQL translation of question 8 from DBpedia's training dataset.

We can deduce that our pattern based approach is very well suited for "clean" KB, with consistent data, domain specific and respecting the schema it is based on. However, in its current implementation, it is not well suited to KBs containing many inconsistencies, which are the most popular KB on the Web of linked data.

[12] http://dbpedia.org
[13] http://musicbrainz.org/
[14] http://swip.univ-tlse2.fr/SwipWebClient/patternViewer.html
[15] without taking into account the "out of scope" queries whose answer is not in the KB anyway

9.3. Unsupported queries

As explained above, 17 queries about Musicbrainz and 81 about DBpedia were not processed by the Swip system. We identified some categories whose expressivity exceeds that currently handled by our approach. Most of the non-supported queries contain aggregates, like "Which bands recorded more than 50 albums?" For now, the only aggregate function supported by Swip is the counting of the total number of results of a query, as in "How many singles did the Scorpions release?" because it is easy to spot in a NL question and it seems to be the most recurrent in user queries. Furthermore, Swip is not able to handle queries which require inferences made beforehand, such as "Which artists turned 60 on May 15, 2011?", and queries implying string comparison, such as "Give me all bands whose name starts with Play."

Of course, adding the support of these categories would significantly improve our approach's results in the challenge, and the design of these new features is the natural next step. Part of this improvement has already been carried out, as presented in the next subsection.

9.4. Improving our results

The analysis of our system performances allowed us improving our results in several directions.

9.4.1. Out of scope questions

Among questions from the QALD-3 training and test datasets, some are *out of scope*, which means they cannot be answered by exploiting exclusively the target knowledge base. Our system does not directly handle the detection of out of scope questions, but a low relevance mark for all interpretations of a given query is a strong clue for the user, whose doubt is immediately reinforced by the generated sentences which very probably do not represent the meaning of their original query.

During the evaluation, we did not try to answer the out of scope questions. But for the organisers, the correct interpretation of such a question is simply a SPARQL query which does not return any result. Considering this, it is easy for us to validate these questions on the challenge point of view (for instance by considering the first interpretation returned by the system) and thus increase our score. This rise is certainly artificial and does not change actual qualities and limits of our approach, but it allows a better comparison with other systems which have probably been adapted to take advantage of the challenge rules.

9.4.2. Supporting more aggregates

We also augmented our implementation with some features allowing to detect and take into account simple expressions implying aggregates in the SPARQL translation, as questions implying aggregate functions represent a significant proportion of the unsupported queries presented in subsection 9.3. These features, described in [45], basically consist in identifying textual patterns which are translated using aggregates in the training dataset and, if applicable, the query element which is the object of the aggregate function. These pieces of information are then added to the pivot query whose definition was slightly updated for the occasion. For instance, the question 29 from the QALD Musicbrainz test dataset *"Which bands recorded more than 50 albums?"* is translated into `"band": "record"= "album(50+)"`. The second main step of the process, mapping patterns to the pivot query, remains the same until the generation of the SPARQL query itself, where aggregate functions are added according to the information contained in the pivot query.

9.4.3. New results

Considering these improvements, we reevaluated our approach on the Musicbrainz dataset (using the evaluation tool[16] made available by the organisers). We thus validated the out of scope questions, some questions implying aggregates, and question 11 whose interpretation previously failed because of a typo in a pattern. Results are considerably improved: the newly calculated F-measure is 0.67.

10. Conclusion and future work

In this paper, we presented the approach we designed to allow end users to query graph-based KBs. This approach is implemented in the Swip system and is mainly characterized by the use of query patterns leading the interpretation of the user NL query and its translation into a formal graph query.

We justified the use of query patterns and detailed our two step interpretation process facilitating multilinguism. We also focused on our inovative imple-

[16] http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/index.php?x=evaltool&q=3

mentation which exploits SPARQL capabilities and we gave a detailed analysis of the results of our participation in the QALD challenge. These results are very encouraging and we planed to extend our work in several directions:

- experimenting the ease of adaptation to different user languages; we will participate in the *Multilingual question answering* task of the QALD challenge and we are developing a partnership with the IRSTEA (the French research institute of science and technology for environment and agriculture) in order to build a real application framework concerning French queries on observations on crop development [46];
- improving the matching process and adding the support of aggregates in order to obtain better results for the next challenge and improve the user experience;
- defining heuristics to orient the generation of mappings and accelerate the interpretation process by preventing the generation of unrelevant mappings;
- experimenting methods to automate or assist the conception of query patterns; we first want to automatically determine the pattern structures by analysing graph query examples, and then compare the developed method(s) to an approach based on NL query learning;
- exploring new leads for the approach to evolve and stick more to the data itself than to the ontology, in order to obtain better results on datasets from the Web of linked data, such as DBpedia.

References

[1] C. Comparot, O. Haemmerlé, and N. Hernandez, "An easy way of expressing conceptual graph queries from keywords and query patterns," in *ICCS*, pp. 84–96, 2010.

[2] C. Pradel, O. Haemmerlé, and N. Hernandez, "Expressing conceptual graph queries from patterns: how to take into account the relations," in *Proceedings of the 19th International Conference on Conceptual Structures, ICCS'11, Lecture Notes in Artificial Intelligence # 6828*, (Derby, GB), pp. 234–247, Springer, July 2011.

[3] C. Pradel, O. Haemmerlé, and N. Hernandez, "A semantic web interface using patterns: The swip system (regular paper)," in *IJCAI-GKR Workshop, Barcelona, Spain, 16/07/2011-16/07/2011* (M. Croitoru, S. Rudolph, N. Wilson, J. Howse, and O. Corby, eds.), no. 7205 in LNAI, (http://www.springerlink.com), pp. 172–187, Springer, mai 2012.

[4] J. F. Sowa, "Conceptual structures: information processing in mind and machine," 1983.

[5] M. CHEIN and M.-L. MUGNIER, "Conceptual graphs: fundamental notions," *Revue d'intelligence artificielle*, vol. 6, no. 4, pp. 365–406, 1992.

[6] S. Chakrabarti, "Breaking through the syntax barrier: Searching with entities and relations," *Knowledge Discovery in Databases: PKDD 2004*, pp. 9–16, 2004.

[7] S. Dekleva, "Is natural language querying practical?," *ACM SIGMIS Database*, vol. 25, no. 2, pp. 24–36, 1994.

[8] S. Desert, "Westlaw is natural v. boolean searching: A performance study," *Law Libr. J.*, vol. 85, p. 713, 1993.

[9] C. Thompson, P. Pazandak, and H. Tennant, "Talk to your semantic web," *Internet Computing, IEEE*, vol. 9, no. 6, pp. 75–78, 2005.

[10] E. Kaufmann and A. Bernstein, "Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 8, no. 4, pp. 377–393, 2010.

[11] E. Kaufmann, A. Bernstein, and L. Fischer, "Nlp-reduce: A" naïve" but domain-independent natural language interface for querying ontologies," in *4th European Semantic Web Conference ESWC 2007*, pp. 1–2, 2007.

[12] E. Kaufmann, A. Bernstein, and R. Zumstein, "Querix: A natural language interface to query ontologies based on clarification dialogs," in *5th International Semantic Web Conference ISWC 2006*, no. November, pp. 5–6, Citeseer, 2006.

[13] A. Bernstein, E. Kaufmann, and C. Kaiser, "Querying the semantic web with ginseng: A guided input natural language search engine," in *In 15th Workshop on Information Technologies and Systems Las Vegas NV*, no. December, pp. 45–50, MV-Wissenschaft, Münster, 2005.

[14] D. Damljanovic, M. Agatonovic, H. Cunningham, and K. Bontcheva, "Improving habitability of natural language interfaces for querying ontologies with feedback and clarifcation dialogues," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 19, no. 2, 2013.

[15] S. Elbassuoni, M. Ramanath, R. Schenkel, and G. Weikum, "Searching rdf graphs with sparql and keywords," *IEEE Data Eng. Bull.*, vol. 33, no. 1, pp. 16–24, 2010.

[16] F. Alkhateeb, J.-F. Baget, and J. Euzenat, "Extending sparql with regular expression patterns (for querying rdf)," *J. Web Sem.*, vol. 7, no. 2, pp. 57–73, 2009.

[17] J. Pérez, M. Arenas, and C. Gutierrez, "nsparql: A navigational language for rdf," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 8, no. 4, pp. 255–270, 2010.

[18] N. Athanasis, V. Christophides, and D. Kotzinos, "Generating on the fly queries for the semantic web: The ics-forth graphical rql interface (grql)," in *International Semantic Web Conference*, pp. 486–501, 2004.

[19] A. Russell and P. R. Smart, "Nitelight: A graphical editor for sparql queries," in *International Semantic Web Conference (Posters & Demos)*, 2008.

[20] A. Clemmer and S. Davies, "Smeagol: a "specific-to-general" semantic web query interface paradigm for novices," in *Proceedings of the 22nd international conference on Database and expert systems applications - Volume Part I*, DEXA'11, (Berlin, Heidelberg), pp. 288–302, Springer-Verlag, 2011.

[21] CoGui, "A conceptual graph editor." Web site, 2009. http://www.lirmm.fr/cogui/.

[22] S. Ferré and A. Hermann, "Semantic search: reconciling expressive querying and exploratory search," *The Semantic Web–ISWC 2011*, pp. 177–192, 2011.

[23] S. Ferré, A. Hermann, M. Ducassé, *et al.*, "Combining faceted search and query languages for the semantic web," in *Semantic Search over the Web (SSW)-Advanced Information Systems Engineering Workshops-CAiSE Int. Workshops*, vol. 83, pp. 554–563, 2011.

[24] S. Ferré and A. Hermann, "Reconciling faceted search and query languages for the semantic web," *International Journal of Metadata, Semantics and Ontologies*, vol. 7, no. 1, pp. 37–54, 2012.

[25] S. Ferré, "Squall: A controlled natural language for querying and updating rdf graphs," in *Controlled Natural Language*, pp. 11–25, Springer, 2012.

[26] Y. Lei, V. S. Uren, and E. Motta, "Semsearch: A search engine for the semantic web," in *EKAW*, pp. 238–245, 2006.

[27] Q. Zhou, C. Wang, M. Xiong, H. Wang, and Y. Yu, "Spark: Adapting keyword query to semantic search," in *ISWC/ASWC*, pp. 694–707, 2007.

[28] T. Tran, H. Wang, S. Rudolph, and P. Cimiano, "Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data," in *ICDE*, pp. 405–416, 2009.

[29] E. Cabrio, J. Cojan, A. Aprosio, B. Magnini, A. Lavelli, and F. Gandon, "Qakis: an open domain qa system based on relational patterns," in *International Semantic Web Conference (Posters & Demos)*, 2012.

[30] H. Wang, K. Zhang, Q. Liu, T. Tran, and Y. Yu, "Q2semantic: a lightweight keyword interface to semantic search," in *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, pp. 584–598, Springer-Verlag, 2008.

[31] J. Lehmann and L. Bühmann, "Autosparql: let users query your knowledge base," in *Proceedings of the 8th extended semantic web conference on The semantic web: research and applications-Volume Part I*, pp. 63–79, Springer-Verlag, 2011.

[32] V. Lopez, V. Uren, M. Sabou, and E. Motta, "Is question answering fit for the semantic web?: a survey," *Semantic Web*, vol. 2, no. 2, pp. 125–155, 2011.

[33] K. Elbedweihy, S. N. Wrigley, and F. Ciravegna, "Evaluating semantic search query approaches with expert and casual users," in *Proceedings of the 11th International Conference on The Semantic Web - Volume Part II*, ISWC'12, pp. 274–286, Springer-Verlag, 2012.

[34] Y. Raimond, S. Abdallah, M. Sandler, and F. Giasson, "The music ontology.," 2007.

[35] A. Spink, D. Wolfram, M. Jansen, and T. Saracevic, "Searching the web: The public and their queries," *Journal of the American society for information science and technology*, vol. 52, no. 3, pp. 226–234, 2001.

[36] B. Jansen, A. Spink, J. Bateman, and T. Saracevic, "Real life information retrieval: a study of user queries on the web," in *ACM SIGIR Forum*, vol. 32, pp. 5–17, ACM, 1998.

[37] B. Jansen, A. Spink, and T. Saracevic, "Real life, real users, and real needs: a study and analysis of user queries on the web," *Information processing & management*, vol. 36, no. 2, pp. 207–227, 2000.

[38] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz, "Analysis of a very large web search engine query log," in *ACm SIGIR Forum*, vol. 33, pp. 6–12, ACM, 1999.

[39] J. Perez, M. Arenas, and C. Gutierrez, "Semantics of SPARQL," tech. rep., Technical Report TR/DCC-2006-17, Universidad de Chile, 2006.

[40] H. Schmid, "Probabilistic part-of-speech tagging using decision trees," 1994.

[41] J. Nivre, J. Hall, and J. Nilsson, "Maltparser: A data-driven parser-generator for dependency parsing," in *Proceedings of the fifth international conference on Language Resources and Evaluation (LREC2006), May 24-26, 2006, Genoa, Italy*, pp. 2216–2219, European Language Resource Association, Paris, 2006.

[42] A. Rector, "Modularisation of domain ontologies implemented in description logics and related formalisms including owl," in *Proceedings of the 2nd international conference on Knowledge capture*, pp. 121–128, ACM, 2003.

[43] S. Hellmann, J. Lehmann, S. Auer, and M. Brümmer, "Integrating nlp using linked data,"

[44] C. Pradel, G. Peyet, O. Haemmerlé, and N. Hernandez, "Swip at qald-3: results, criticisms and lesson learned (working notes)," in *CLEF 2013, Valencia, Spain, 23/09/2013-26/09/2013*, 2013.

[45] F. Amarger, O. HaemmerlÃl', N. Hernandez, and C. Pradel, "Taking SPARQL 1.1 extensions into account in the SWIP system," in *Conceptual Structures for STEM Research and Education*, (Bombay, Inde), p. 75âĂŞ89, Springer, 2013.

[46] C. Roussey, J.-P. Chanet, V. Cellier, and F. Amarger, "Agronomic taxon," in *Proceedings of second international Workshop on Open Data (WOD 2013)*, BNF Paris, 2010.