QueryGen: Semantic Interpretation of Keyword Queries over Heterogeneous Information Systems

Carlos Bobed[a,*], Eduardo Mena[a]

[a]*Dept. of Computer Science & Systems Engineering*
University of Zaragoza
50018 Zaragoza, Spain

Abstract

In the last years, users have become used to keyword-based search interfaces due to their ease of use. By matching input keywords against huge amounts of textual information and labeled multimedia files, current search engines satisfy most of users' information needs. However, the principal problem of this kind of search is the semantic gap between the input and the real user need, as keywords are a simplification of the query intended by the user. Moreover, different users could use the same set of keywords to search different information; even the same user could do it at different times. The search system, before accessing any data, should discover first the intended semantics behind the user keywords, in order to return only data fulfilling such semantics. The use of formal query languages is not an option for non-expert users, so a semantic keyword-based search based on *semantic interpretation of keyword queries* could be the solution, i.e., a search that starts discovering the semantics intended for the input user keywords, and then only data relevant to that semantics are returned as answer.

In this paper we present a system that performs semantic keyword interpretation on different data repositories. Our system 1) discovers the meaning of the input keywords by consulting a generic pool of ontologies and applying different disambiguation techniques, 2) once the meaning of each keyword has been established, the system combines them in a formal query that captures the semantics intended by the user, considering different formal query languages and possibilities that could arise, but avoiding inconsistent and semantically equivalent queries, and, finally, 3) after the user has validated the generated query that best fits her/his intended meaning, the system routes the query to the appropriate data repositories that will retrieve data according to the semantics of such a query. Experimental results show the semantic interpretation capabilities and the feasibility of our approach.

Keywords:
Semantic Keyword Interpretation, Semantic Search, Semantic Web, Ontologies, Knowledge Discovery

1. Introduction

The Web has made a huge and ever-growing amount of information available to its users. To handle and take advantage of this information, users have found in Web search engines their best allies. Most of these search engines have a keyword-based interface to allow users to express their information needs, as this is an easy way for users to define their searches. Thus, the adoption of keyword-based search interfaces has spread widely in the last few years. However, the ease of use of keyword search comes from the simplicity of its query model, whose expressivity is low compared with other more complex query models [31]. In fact, keyword queries are simplifications of the queries that really express the user's information need. On the other hand, the use of expressive formal languages (such as SQL or SPARQL) is far from being easy for common users. Moreover, to effectively use formal languages, the user must have previous knowledge of the underlying schema and data s/he is accessing. Thus, the sweet spot would be to mix the expressivity of formal languages with the ease of use of keyword queries, while making the user unaware of the data sources being accessed to solve her/his information needs. Therefore, to deal with these problems, we advocate for a semantic keyword-based search based on *semantic interpretation of keyword queries*, a keyword-based search process in which semantics of both keywords and query languages play a crucial role during the whole search process.

In any search engine which has an unstructured query language as input, the main steps performed are: query construction, data retrieval, and presentation of results. Out of these three steps, the first one is crucial because the more accurate the system is able to capture the user's information need, the more precise results it will retrieve. However, the importance of this first step is usually underestimated by adopting unstructured query models (i.e., bag of keywords), making the quick access to huge amounts of data and the ranking of results the most important steps of the whole process, while leaving the burden of processing non-relevant data to the user. In this way, these approaches might miss what the user really wanted to retrieve as they hide less promoted results, making current search engines useless when looking for certain (non-popular) infor-

*Corresponding author
Email addresses: cbobed@unizar.es (Carlos Bobed), emena@unizar.es (Eduardo Mena)

mation[1]. To enhance the search process, we aim at enhancing the capture of the user's information need by combining both the benefits of the structured query models, and the ease of use and spread of the keyword search. When it comes to keyword queries, the process to translate them into a structured query is named *keyword query interpretation* [19]. For this task, several approaches (e.g., [42, 46]) advocate starting with the discovery of the meaning of each keyword among the different possible combinations. For instance, the keyword "book" could mean "a kind of publication" or "to reserve a hotel room". These approaches consult a pool of ontologies (which offer a formal, explicit specification of a shared conceptualization [24]) and use disambiguation techniques to discover the intended meaning of each user keyword. So, plain keywords can be mapped to ontological terms (concepts, roles, or instances). However, direct interpretation might not be always possible as users tend to omit information in their keyword searches (the average number of keywords used in keyword-based search engines "is somewhere between 2 and 3" [36][2]), and therefore, relevant underlying knowledge should be used to enhance this interpretation.

In this paper, we delve into that line and present QueryGen, a system that performs semantic interpretation of keyword queries into multiple query language over different data repositories. Our system:

1. Discovers the meaning of the input keywords by consulting a generic pool of ontologies and disambiguates them taking into account their context (the rest of the keywords in the input set); i.e., each keyword in the input has influence on the rest of the keyword's meanings. In this process, it retrieves and integrates knowledge about the input terms, which will use in latter stages.
2. Then, as a given set of user keywords (even when their semantics have been properly established) could represent several queries, the system finds all the possible queries using the input keywords in order to precisely express the exact meaning intended by the user. This is done considering different formal query languages (the use of formal languages avoids ambiguities and expresses the user information in a precise way) which are made available to the system by semantically modeling them, and avoiding inconsistent and semantically equivalent queries with the help of a Description Logics (DL) reasoner [4]. During this process, our system considers the addition of *virtual terms*. These virtual terms represent missing keywords that users had in their mind but did not input[3]. This way, our system can explore further meanings when the user has given an incomplete input.
3. Finally, once the user has validated the generated query that best fits her/his intended meaning, our system routes the query to the appropriate structured data repositories that will retrieve data according to the semantics of such a query.

The architecture of our system is flexible enough to deal with different ontologies, formal query languages, and query processing capabilities of underlying data repositories. We aim at achieving the highest expressivity possible taking as starting point a plain keyword-based input as it is the most spread method to request information (using Web search engines). Moreover, our system is robust to incomplete inputs as, using the retrieved background knowledge, even in case that the user input is just a single keyword, our system is able to deal with it by exploring the implicit information description that the user had in mind when that keyword was posed.

In particular, the main contributions of this work are as follows:

- We present our approach to Semantic Keyword Interpretation, which is completely knowledge-guided. First, our system applies semantic techniques to disambiguate the input keywords and retrieve further knowledge about them; and then, it interprets the semantics of the input keywords structuring them according the semantics of the different formal query languages, obtaining a formal query that is posed to the available underlying data repositories.
- We propose a generalized keyword interpretation process based on semantic models of the query languages used to structure keyword queries (i.e. not tied to any particular query language). The semantic modelling framework proposed allows QueryGen not to be restricted to DL-based query languages, but to use any query language to interpret keyword queries as long as it is correctly modeled.
- Experiments are provided to show the semantic interpretation capabilities of QueryGen as well as the feasibility of the approach.

The rest of this paper is organized as follows. Firstly, in Section 2, we give an overview of the architecture of the whole system. The discovery of the meaning of the keywords and the disambiguation process are explained in Section 3. Then, the keyword interpretation process, in the form of query generation, is shown in Section 4. The solution adopted to handle heterogeneous query types and data is presented in Section 5. A complete example, from some sample input keywords to the retrieved data, is explained step by step in Section 6. We present experimental results on the quality and the performance of our approach in Section 7, and discuss some related work in Section 8. Finally, the conclusions and future work are drawn in Section 9.

2. Architecture of the System

In this section, we present the whole pipeline that allows us to go from plain keywords to semantic queries and, finally, access to data stored in different data repositories. We differentiate three main steps (see Figure 1):

[1] According to the different criteria adopted by the ranking schema, which can take into account other aspects apart from actual popularity.

[2] This data still hold as for April 2015, `http://www.keyworddiscovery.com/keyword-stats.html?date=2015-04-01`, last accessed May 20, 2015.

[3] For example, a user looking for movies whose genre is "horror" could enter "horror movie", omitting the keyword "genre".

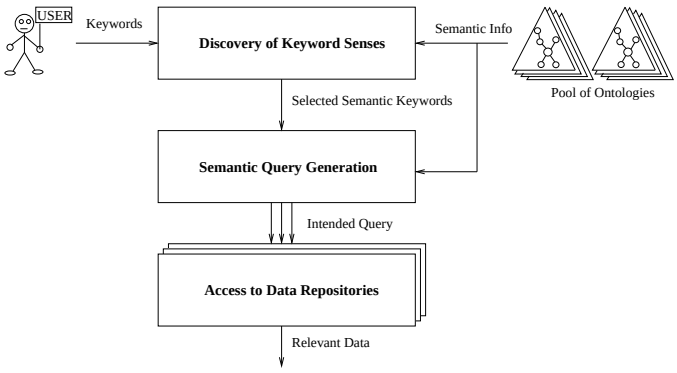


Figure 1: An overview of the whole process: from plain keywords to the data access.

1. *Discovery of Keyword Senses*: In a first step, the system obtains the exact semantics of the input keywords to transform them into *semantic keywords* (keywords with well-defined semantics). To do so, it consults a pool of ontologies to extract the possible meanings of each keyword, integrating the meanings that are similar enough in order to avoid redundancy. Then, the system applies different disambiguation techniques to finally establish the meaning for each keyword taking into account its context (the possible meanings of the rest of the keywords). However, this is only a first step towards obtaining the semantics of the input. Several queries might be behind a given set of keywords, even when their semantics have been properly established individually. For example, given the keywords "fish" and "person" meaning "a creature that lives and can breathe in water" and "a human being", respectively, the user might be asking for information about either biologists, fishermen, or even other possible interpretations based on those individual keyword meanings.
2. *Semantic Query Generation*: Once the meaning of each keyword has been established, our system automatically builds a set of formal queries which, combining all the keywords, represent the possible semantics that could be intended by the user when s/he wrote the list of plain keywords.
   To be able to interpret keyword queries into different query languages, QueryGen needs to be provided with semantic models of such query languages. Each query language is modeled using an augmented abstract grammar which comprises information about which operators are supported by that query language and their semantics. As we will see in Section 4.1, the grammars used to model the semantics of query languages are syntax-agnostic, and define how to combine the operators of such query languages in terms of *typed gaps*, i.e. they specify which kind of queries can be built using concepts, roles, and instances in the corresponding query language (e.g.: *AND concept concept*). QueryGen uses these semantic models along with the semantic keywords obtained in the previous step to build possible interpretations in different formal query languages (the expressivity of our approach is only bounded by the expressivity of the formal query languages made available to our system to precisely express the user intended query).
   The result of this generic interpretation process is a set of *abstract queries* (i.e., possible interpretations agnostic to the actual input as they are formed using typed gaps) that the system materializes into a list of actual queries by substituting the typed gaps by input keywords. Then, the set of (syntactically correct) generated queries are *semantically* filtered using a DL reasoner.
   When no query satisfies the user, our system performs a semantic enrichment of the input by adding *virtual terms*. They are generic typed gaps (to be replaced by concepts, roles, or instances) that represent the keywords that the user might have omitted, but without whom the intended query cannot be built. In a new query generation step, our system treats them as regular typed gaps but, instead of being replaced by input keywords, they are substituted by terms obtained from the ontologies which the input keywords were mapped to (during the previous discovery step). Thus any query that the user could have in mind will be generated as a candidate interpretation as long as the available query languages are expressive enough.
   This query generation process has both a syntactic and semantic dimension: it generates only syntactically correct queries according to the grammar of each of the query languages, and it takes into account the semantics of the operators of each language and the semantics of the keywords to avoid generating both duplicated and incoherent queries. This process is performed in parallel for each available query language as their expressivity can differ from each other.
3. *Access to Data Repositories*: Finally, once the user has validated the generated query that best fits her/his intended meaning, the system forwards it to the appropriate underlying structured data repositories (databases, Linked Data endpoints, etc.) that will retrieve data according to the semantics of such a query. This is far from being a trivial task, as their different query processing capabilities and data models make it necessary that our system adapt itself to their different access methods and formats of retrieved data.

In the following sections, we include a detailed description of each of these three main steps.

3. Discovery of Keyword Senses

As stated in the previous section, the first step that our system performs is the discovery of the semantics that exist behind the user keywords. This discovery is done by taking into account the individual possible semantics of each keywords as well as the possible semantics of its context (the rest of keywords), following the proposal in [46]. In particular, this process is divided into three substeps (see Figure 2):

- Extraction of Keyword Senses: The system extracts out the possible meanings of each keyword from a dynamic

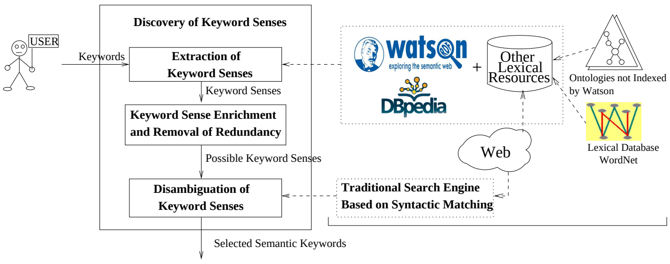


Figure 2: Discovery of keyword senses.

pool of ontologies (in particular, it queries Watson [16], DBpedia [8], WordNet [38] and other ontology repositories to find ontological terms that syntactically match the keywords - or one of their synonyms). The system builds a *sense* for each matching obtained. These *senses* represent the exact meaning of a keyword, and are composed of a list of synonym URIs for the keyword, an ontological context (defined for an ontological term in [22] as "the minimum set of other ontological terms that, belonging to its semantic description, locate the term in the ontology and characterize its meaning"), and a description in natural language of the sense. Then, the extracted senses are semantically enriched with the ontological terms of their synonyms by also searching in the ontology pool. The result is a list of candidate keyword senses for each user keyword. In Figure 3, three possible senses (two as a class and one as a property) retrieved for user keyword *star* are shown.

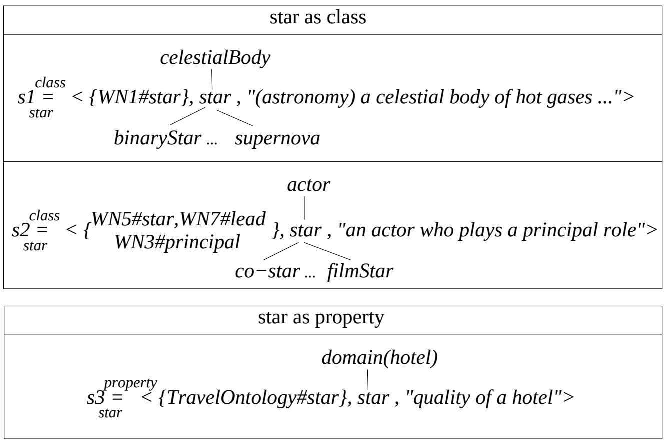


Figure 3: Possible senses for keyword *star*.

- Keyword Sense Enrichment and Removal of Redundancy: As the obtained senses were built with terms coming from different ontologies, they could represent the same semantics. An incremental algorithm is used to align the different keyword senses and merge them when they are similar enough. To assess the sense similarity, our system calculates a *synonymy probability* that considers both linguistic and structural characteristics of the source ontologies: the linguistic similarity is calculated considering as strings the different labels of each term; and the structural similarity is calculated recursively exploiting the semantics of the *semantic keywords* (their ontological context, see Figure 3) until a certain depth. Finally, both similarity values are combined to obtain the resultant synonymy measure[4].

  Senses are merged when the estimated *synonymy probability* between them exceeds a certain threshold[5]. Thus, the result is a set of *different* possible senses for each user keyword entered.

- Disambiguation of Keyword Senses: A disambiguation process is carried out to select the most probable intended sense of each user keyword by considering the possible senses of the rest of keywords. The senses are compared by combining [23]: (a) a web-based relatedness measure, that measures the co-occurrence of terms on the Web according to traditional search engines such as Google or Yahoo!, (b) the overlap between the words that appear in the context, and the words that appear in the semantic definition of the sense [5], and (c) the frequency of usage of senses (when available, as in WordNet annotated corpora). Thus, the best sense for each keyword will be selected according to its context. Note that this selection can require the user's feedback to select the most appropriate sense for each keyword in a semi-automatic way.

This discovery and disambiguation algorithm, which (due to space limitations) has been summarized here for the sake of completeness, is thoroughly described in [46], and it has been successfully applied to very different tasks such as ontology matching [21], the integration of senses in semantic repositories [20], or the construction of multi-sourced ontologies [10].

4. Semantic Query Generation

Once the meaning of each keyword has been established, our system automatically builds a set of formal queries which, combining all the keywords, represent the possible semantics that could be intended by the user when s/he wrote the list of plain keywords. The main generation steps are shown in Figure 4:

- Analysis Table Constructor: It constructs the analysis tables for the formal query languages that the generator uses to generate the possible queries. This is done off-line and just once for each language made available to the system.

- Query Generator: It builds the possible queries for each query language according to its defined operators. During the generation process, QueryGen takes into account the semantics of the different operators to avoid generating semantically equivalent queries.

[4] The formulae for the synonymy for each type of senses (concepts, roles and instances) can be found in [46].

[5] In [20], the authors proposed several strategies to obtain this threshold and validated them via thorough experimentation.

- Semantic Processor: Once the set of syntactically possible queries is obtained, the system is able to filter out the inconsistent ones with the help of a DL reasoner. During this step, it also performs a semantic enrichment to try to find possible implicit keywords that have been omitted due to the simplistic nature of the keyword query model. To do so, our system adds *virtual terms (VTs)* to the input. They are generic typed terms (they can be generic concepts, roles, or instances) that represent the keywords that the user might have omitted, but without whom the intended query cannot be built. When the system uses VTs, an extra step is carried out to substitute them with appropriate terms taken from the ontologies which the input keywords were mapped to.

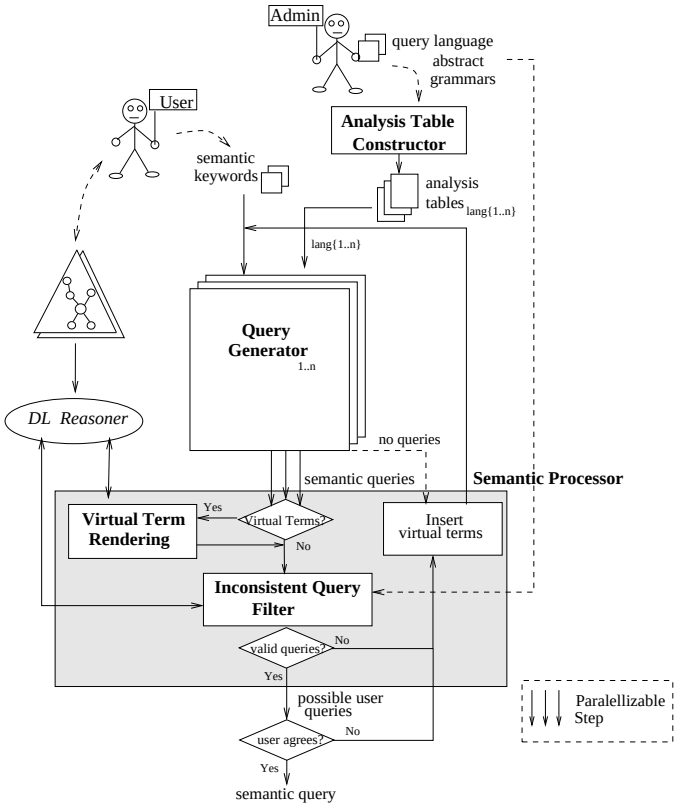


Figure 4: Multi-language query generation process.

In the following, we detail these three main query generation steps plus a presentation step to ease the intended query selection.

4.1. Analysis Table Constructor

In order to be able to interpret keyword queries into different query languages (with their different expressivities, defined by their operators), QueryGen has to be provided with a semantic model of each of such query languages, in the form of an extended abstract grammar. These grammars: 1) comprise information about the query language supported operators (and how to combine correctly their operands), 2) leave aside the syntax of the query languages they represent to make the generation process independent of any particular language, and 3) are semantically annotated with a twofold goal: to avoid generating duplicated queries, and to check the satisfiability of the query according to the knowledge retrieved by the system.

With these grammars, the system builds the analysis tables[6] that are used by the *Query Generator* (see Section 4.2) to build all the possible syntactically correct queries corresponding to the input user keywords, according to each available query language. Note that these tables are built only once for each new output query language that is made available to the system, and they are used every time a new query is posed to the system.

Specifying the Query Languages

To make a new query language available to our system, its context-free grammar G must be transformed into an abstract context-free grammar $\mathcal{G}'$, where the syntax elements of such a language have been removed. In this grammar, operators become non terminals, and the right side of their productions are the operands they accept. Instead of working with bare syntactical tokens, these grammars have three basic types of tokens: *Concept (C)*, *Role (R)*, and *Instance (I)*, which correspond to the three main types of elements in ontologies. Thus, the use of these abstract grammars makes the translation process independent of the syntax of the query languages. We define these abstract grammars as tuples $\mathcal{G}' =< \mathcal{Q}, \mathcal{N}, \mathcal{T}, \mathcal{P} >$, where:

- $\mathcal{Q}$ is the starting symbol of the grammar, and represents the root of the query.
- $\mathcal{N} = \{Op_i\} \cup \{Q_{ps}\} \cup \{R_{types}\}$, with $\{Op_i\}$ containing the set of operators of the query language; $\{Q_{ps}\}$ being a set of nonterminals to build up the different parts of the queries (if required by the query language); and $\{R_{types}\}$ being the types of the returning values of the query language operators. Each element Op_i is a tuple $< Op_{ID}, \{prop_j\} >$, with the id of the operator and its associated properties.
- $\mathcal{T}$ is the set of terminals that we work with, and is conformed by C, R, and I (corresponding to concept, role, and instance tokens, respectively), plus the empty token symbol ξ.
- $\mathcal{P} = \{< prod_i, localCond_i, globalCond_i >\}$ is the set of productions which define: a) if the left-side nonterminal is an operator, an ordered list of the types of operands it works with, and b) if it is a returning type ($\{R_{types}\}$), the operators that produce the returning values of that type; $localCond_i$ and $globalCond_i$ are expressions which define the semantic conditions that have to be checked to correctly apply the operator.

These grammars makes our system able to, once it knows whether the input keywords are concepts, roles or instances, build semantically correct interpretations expressed as formal queries in the different query languages that are available. In

[6] It builds the Goto and Action tables, as defined in [3].

Figure 5, the extended abstract grammar corresponding to a subset of BACK[7] query language [39] is shown.

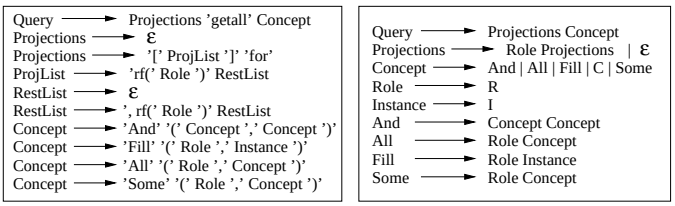


Figure 5: a) Simple BACK grammar, and b) the resulting abstract grammar (semantic annotations are not included).

In particular, in the BACK example:

- The initial symbol $\mathcal{Q}$ is *Query*.
- For $\mathcal{N}$, $\{Op_i\}$ contains *Projections, And, All, Fill,* and *Some* nonterminals; $\{Q_{ps}\}$ would be empty; and, $\{R_{types}\}$ contains *Concept, Role,* and *Instance* nonterminals.
- $\mathcal{T}$, as we have defined before, contains *C, R, I* and ξ tokens.
- $\mathcal{P}$ contains each one of the productions in Figure 5.b.

Properties of the Operators. As we have seen in the definition of the extended grammars, each of the operators comes with a list of its properties. In particular, the properties considered are *associativity*, *involution*, *symmetry*, *restrictiveness* and *inclusiveness*. The first three ones are well known properties, while *restrictiveness* and *inclusiveness* are defined in [12] as follows:

Definition 1. *Let C and R be the sets of concepts and roles in the domain discourse. A binary operator* op *is* restrictive *if there exists a function $f : K \rightarrow C$, with K being C or R, such:*

$$\forall x \in K \wedge y \in C : f(x) \sqsubseteq y \Rightarrow op(x, y) \equiv op(x, f(x))$$

Definition 2. *Let C and R be the sets of concepts and roles in the domain discourse. A binary operator* op *is* inclusive *if there exists a function $f : K \rightarrow C$, with K being C or R, such:*

$$\forall x \in K \wedge y \in C : f(x) \sqsupseteq y \Rightarrow op(x, y) \equiv op(x, f(x))$$

From these definitions, and according to the semantics of the operators in BACK, it can be shown that: 1) by choosing $f(x) = x$ with $K = C$, the *And* operator is restrictive, and the *Or* operator is inclusive; and 2) by choosing $f(x) = range(x)$ with $K = R$, the *Some* and *All* operators are restrictive. Following with the specification of the excerpt of BACK language, a summary of the properties of the considered operators is shown in Table 1.

In this example, we consider the *Projection* operator as associative as all the roles that are specified in the projections list are applied to the same concept. Thus, the order in which we apply them does not matter. The same reasoning is applied to consider it symmetric.

[7]Although it is obsolete (discontinued since 1998), we use BACK in the examples for didactic purposes as it is very concise and supports projections.

Table 1: Properties of the operators of BACK language.

Operator	Properties
And	associativity, symmetry, restrictiveness
Some	restrictiveness
All	restrictiveness
Fill	none
Projections	associativity, symmetry

Expressions for Semantic Checking. Each production in the grammar can be annotated with semantic expressions that are checked with the help of a DL reasoner. These expressions are built using the operators shown in Table 2. There are two types of conditions for each production, *local* and *global* ones, depending on the information that they comprise:

- A local condition $locCond_i$ on a production $prod_i$ details semantic constraints that the nonterminals on the right side of the production must satisfy for the production being eligible to be fired. This is used to perform an extended semantic type checking on the operands locally.
- A global condition $globalCond_i$ on a production $prod_i$ provides a semantic translation of the production that allows to translate it into a checkable DL expression (even with non-DL query languages). Using these conditions, QueryGen can build a global DL expression that comprises the semantics of the associated query, and check its consistency according to the retrieved knowledge.

Following with the simplified BACK example, there are no local conditions on the different productions due to the fact that the operators do not impose any special constraint on their operands (apart from their general type -concept, role or instance- which is explicitly stated in the right part of the production). Regarding global conditions, there are several annotated productions, as we can see in Table 3.

Table 3: Global conditions on the productions of the abstract grammar for BACK language.

Production	Global Condition
Query → Projections Concept	And($1, $2)
Projections → Role Projections	And(Dom($1), $2)
Projections → ξ	Thing
Concept → And	$1
Concept → All	$1
Concept → Fill	$1
Concept → Some	$1
Concept → C	$1
Role → R	$1
Instance → I	$1

The first condition tells the system to check the conjunction of the concepts returned by the *Projections* operator and the rest of the query (*Concept*). The concept returned by the *Projections* nonterminal is built according to the second condition: It

Table 2: Operators of the inner specification language to establish what has to be checked about the operators of the final specified language.

Operator	Meaning
$*	It refers the *-th element of the right side of the production
SubClassOf (cpt_1, cpt_2)	It checks whether cpt_1 is subsumed by cpt_2
SubPropOf ($role_1$, $role_2$)	It checks whether $role_1$ is subproperty of $role_2$
Dom (*role*)	It returns the domain of *role*
Range (*role*)	It returns the range of *role*
Class (*inst*)	It returns the class of *inst*
InstanceOf (*inst*, *cpt*)	It checks whether *inst* is instance of *cpt*
And (cpt_1, cpt_2)	It returns the concept intersection
Or (cpt_1, cpt_2)	It returns the concept union
Satisfiable (*cpt*)	It checks the satisfiability of *cpt*
$\wedge$ ($bool_1$, $bool_2$)	It calculates the boolean And of $bool_1$ and $bool_2$
$\vee$ ($bool_1$, $bool_2$)	It calculates the boolean Or of $bool_1$ and $bool_2$
$\neg$ (*bool*)	It calculates the boolean Not of *bool*
Thing	It returns the Top concept, which is the identity element for And operator
Nothing	It returns the Bottom concept, which is the identity element for Or operator
Neutral	It lets the system choose which identity element to use depending on the context

appends the domains of the different roles using *And* operators. The rest of the global conditions just tell the system not to touch anything about the returned concepts. Note that there are no global conditions for the productions with the *And, Some, All,* and *Fill* operators because they directly map to DL-expressions and their associated *Concept* can be built directly without any given expression.

With these semantic annotations, our system is able to generate only both syntactically and semantically correct queries, as we will see in the following subsections.

4.2. Query Generator

For each query language available, a different generation thread is launched to build the possible queries expressed in such a language. As example, let us assume that a user enters keywords "person fish" to find information about people devoured by fishes, which are mapped to the homonym terms in the ontology *Animals*[8], and that the simplified version of BACK defined is used as target query language. The query generation process is divided into three main steps:

- *Permutations of Keyword Types*: To decouple the generation process of any specific language, its first stage is syntax-based. So, the system firstly obtains all the possible permutations of the types of term (concept, role, or instance) corresponding to the semantics of each input keyword, to discover any syntactically possible query in latter steps. In the example, *person* and *fish* are concepts, so the output of this step would be <C,C>, as no more permutations are possible. The results of this step are shared by all the language threads.

- *Generation of abstract query trees*: For each permutation obtained in the previous step, the system generates all the syntactically possible combinations according to the grammars of the available query languages. We call these combinations *abstract queries* because they have *gaps* that will be filled later with specific concepts, roles, or instances. These abstract queries are represented as trees, where the nodes are operators and the leaves are *typed gaps* (concept, role, or instance gaps). Following the previous example, with the input <C,C> and simplified BACK as query language, *And(C,C)* would be built as an abstract query.

  In this process, the system uses bottom-up parsing techniques [3] and the tables that have been built off-line in the previous *Analysis Table Construction* step. The semantics of the operators are considered to avoid generating equivalent queries. In particular:

  - *Associativity* property is used to compact query trees such as *And(C,C,C)*, which could be the result of several abstract query trees such as *And(C, And(C,C))* or *And(And(C,C),C)*.
  - *Symmetry* is used to avoid generating duplicated trees, such as *And(Some(R, C),C)* and *And(C, Some(R,C))*.
  - *Involution* is used to avoid infinite loops in the trees. For example, when considering *Not* operator, *Not(Not(C))* will not be generated as it is equivalent to *C*.

- *Query rendering*: For each abstract query tree generated, the gaps in the leaves are filled with the user keywords matching the corresponding gap type. The result of this step for the running example would be *And(Person, Fish)*, i.e., entities which are a person and a fish, which in this case we know that do not represent what the user had in

[8] `http://www.cs.man.ac.uk/~rector/tutorials/Biomedical-Tutorial/Tutorial-Ontologies/Animals/Animals-tutorial-complete.owl`

mind although, for the system, it is a syntactically possible query to consider.

In this step, *symmetry* property is considered to avoid rendering duplicated queries. Following the example, *And(C,C)* has led to *And(Person,Fish)*, and the system has avoided to built *And(Fish,Person)* as it is equivalent due to the semantics of the operator.

Thus, at the end of this step, our system has built all the syntactically possible queries in each of the available languages. However, being syntactically correct does not imply that all of them are semantically consistent. In the following section, we present the techniques that our system applies to filter out the queries that are inconsistent according to the semantics of the keywords, and to further explore the possible query space when no query satisfies the user (e.g. due to an incomplete input).

4.3. Semantic Processor

Once the system has obtained all the *syntactically* possible queries, the Semantic Processor comes into play. As aforementioned, it has two main tasks: To check the queries semantically according to the available knowledge; and to perform a semantic enrichment of the input to suggest further interpretations when the intended query cannot be found by the user. In the rest of the section, we detail these processes.

Inconsistent Query Filtering

During the previous steps, all the user keywords have been combined into queries that are syntactically correct according to the different available query languages. However, some of these queries might not be semantically correct according to the semantics of keywords. So, the system filters out the inconsistent queries with the help of a DL reasoner using the knowledge corresponding to the keyword semantics. In our example, *And(Person, Fish)* would be removed in this step as it is classified as being inconsistent (*Person* and *Fish* are defined as disjoint classes in ontology *Animals*).

This consistency evaluation is direct when dealing with DL languages as they can be directly translated into concepts and the reasoner can be asked about their consistency. However, when it comes to non-DL languages, we have to tell the system how to check them via the specification of the language. As seen in Section 4.1, there are two types of conditions associated to each of the productions, *local* and *global ones*:

- Local conditions: They provide semantic checkings that have to be performed on the operands of the production. A query must hold all the local conditions constraints; otherwise, it must be filtered out as inconsistent one because there would be any production that should not have been fired. In Figure 6, an example of a *percentage* operator is shown.

  The definition of this operator tells the system that their operands must hold that the first one is subclass of the second one to be applicable. In this case, if *Male* was a subclass of *Person* then this part of the query would be locally consistent. All the nodes of a query have to be locally consistent for the query to be considered for global consistency. Note that, otherwise, there would be a part of the query that had been built incorrectly.

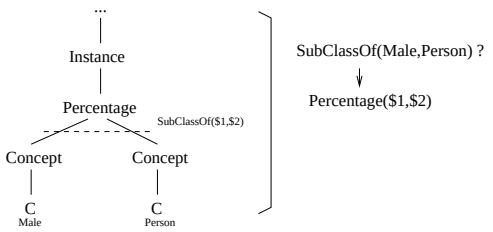


Figure 6: Example of semantic checking on the local conditions of a non-DL operator (only local conditions are shown).

- Global conditions: A query holding all the local conditions only probes that it is syntactically correct (by construction), and that all the operators have been correctly applied. However, the query might still be inconsistent due to its whole meaning. As mentioned before, the global meaning is obtained easily for DL-languages, as queries can be seen as concepts and therefore, directly translated and semantically checked. However, for non-DL languages (e.g., SQL-like languages) or for extensions of DL-languages (e.g., the projection operator of the simplified BACK language), this is not directly applicable. Global conditions provide a translation of each of the productions of the language to build a semantic expression that comprise the global meaning of the query.

  Our system translates the query into a checkable DL-expression by traversing recursively its associated query tree and applying the global conditions expressions to each node. This traversal is performed in a depth-first way. During it, our system applies the different condition operators with the help of a DL-reasoner. Following with the example in the query generation, if we added the keyword *owns* to the input, mapped to the homonym term in ontology *Animals*, the system could form the query *[owns](And(Person, Fish))*, that is, the entities that are owned by a *(Person and Fish)*. In Figure 7, an example of the global checking on this query involving a projection is shown.

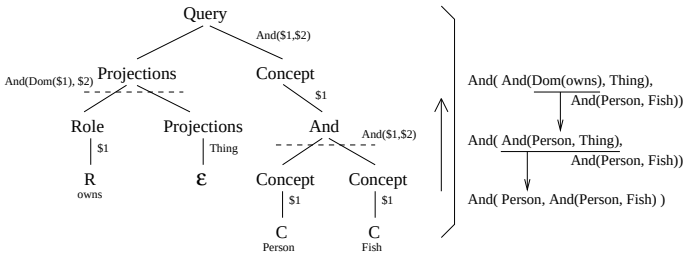


Figure 7: Example of the global semantic checking on a DL-query involving projections (only global conditions are shown).

Going from the leafs to the root node, our system is able

to form the global expression applying the global conditions on the productions. In particular, the conditions on *Projections* operator establishes that, to be able to ask for the value of a property for the instances of a particular instance, the domain of the property must be *compatible* with the concept (i.e., not disjoint, which can be checked out by evaluating their conjunction). So, the system applies the specification to translate the query tree into *And(Dom(owns), And(Person, Fish))*. Resolving the different operators with the use of the DL reasoner, this expression leads to *And(Person, And(Person, Fish))*, which is inconsistent as we cannot ask for the properties of a concept that is not satisfiable.

Note that all these checks cannot be performed before: Until the *rendering step*, the system is working just taking into account the structure of the queries. It is not until the system substitutes the typed gaps on the abstract queries with the input terms, that the actual query is built (along with its meaning). Due to the size of the query search space, we prioritized its reduction. Removing firstly all the possible abstract queries (each of which results on a set of actual queries after the *query rendering* step) pruned the search space and lead to a lower number of queries to be checked than working with the actual terms from the beginning.

Finally, the performance of this step is greatly boosted by the fact that the set of generated queries forms a *conservative extension* [15] of the original ontologies. Once an ontology has been classified, this property makes it possible to evaluate the satisfiability of the queries without reclassifying the ontology, as each query does not assert new knowledge into that ontology.

Semantic Enrichment

When no query either is generated or satisfies the user, our system considers that something could be implicit in the user input. The average number of keywords used in keyword-based search engines "is somewhere between 2 and 3" [36][9], so there is a high chance that the user might have simplified too much its information need, specially regarding complex queries.

To deal with this lack of information, our system adds *virtual terms* (*VTs*) to the original list of user keywords (*Insert Virtual Terms* step in Figure 4). These VTs represent possible keywords that the user may have omitted as part of her/his query, as a keyword query is a simplification of her/his actual information need. Then, the previous steps are executed again to generate queries considering these VTs. In our example, the extended inputs considered would be "person fish $VT_{concept}$" and "person fish VT_{role}", which allows the system to build, among others, the enriched abstract query *And(Person (Some(VT_{role}, Fish))*[10].

These queries with VTs have to be rendered again (*Virtual Term Rendering* step in Figure 4). Our system replaces any existing VT by compatible terms (i.e., terms of the same type: concept, role, or instance) extracted from the ontologies which the input keywords were mapped to in the disambiguation process (see Section 3). To build only semantically correct queries in an efficient way, the system narrows the set of candidate terms by using the ontology modularization techniques described in [30].

Moreover, in this step, *restrictiveness* and *inclusiveness* properties are considered to avoid generating equivalent queries. *Restrictiveness* property allows the system to prune for VT rendering the super classes of the other fixed operand. For example, if the system has to render the VT in *And(Person, $VT_{concept}$)*, it will only consider concepts that 1) have no subsumption relationship with Person, or 2) are subsumed by Person. The reason is that "any term that subsumes Person And Person" is always equivalent to "Person". *Inclusiveness* works the other way round, allowing the system to prune the subclasses of the fixed operand.

In the example, the previous enriched abstract query is rendered into *And(Person (Some (is_eaten_by, Fish))* (among others), which actually represents the exact semantics intended by the user when s/he entered the keywords "person fish".

Note how, in the example, our system considered only VTs for concepts and roles. At first, we did not considered instances for the semantic enrichment as ontologies themselves usually do not have them (it happens frequently [48]). Moreover, it seemed pretty safe to assume that when a user is looking for information about something very specific, the implicit keywords might be roles or concepts, rather than instances. For example, a user looking for information about terror movies (a possible query might be *And(Movie, Fill(genre,terror))*, will input "horror movies" instead of "movies genre". However, our approach can effectively deal with instances as well, but we advocate for asking the user to explicitly fill the instance value when appropriate instead of showing her/him all the possibilities (which might be even unfeasible, for instance, when we are dealing with datatypes such as strings or integers).

The query generation approach presented here extends our previous works in [11, 12] where we focused on DL query languages only, a limitation we have got rid of without losing the semantic capabilities of our system. We have extended them by proposing a semantic approach to model query languages, and applying the generation, semantic enrichment, and filtering to non-DL query languages through the use of such semantic models. Moreover, instead of selecting a target language, the system considers all the possibilities and makes it transparent for the user. Last but not least, we also consider different underlying query and data models, and as we will see in the next section.

4.4. Query Presentation

The way in which the generated queries are presented to the user also makes a difference. Users' attention is a capital resource and we have to minimize their efforts to express their information needs. Thus, to minimize and compress the information shown to the user, we adopt the pattern based techniques we introduced in [12]. These techniques could be applied along with some ranking schema, but we do not apply any ranking on

[9] This data still hold as for April 2015, `http://www.keyworddiscovery.com/keyword-stats.html?date=2015-04-01`, last accessed May 20, 2015.

[10] Here, VT_{role} is the VT to be rendered with a compatible role.

purpose to avoid hiding any possible interpretation. Moreover, it is not clear how to identify the query that a specific user had in mind when writing a set of keywords, as approaches based on semantic distances would also hide possible meanings.

Table 4: Queries and patterns generated for "person bus" using BACK.

Queries	Patterns
all(drives, person) and bus	all(R, C) and C
all(drives, bus) and person	
...	
some(drives, person) and bus	some(R, C) and C
some(drives, bus) and person	
...	
all(drives, bus and person)	all(R, C and C)
...	
some(drives, bus and person)	some(R, C and C)
...	

(C: concept, R: role)

In brief, our system takes advantage of the syntactic similarity of the generated queries and of the ontological classification of the terms that compose the queries. It analyzes the structure of the queries to extract common syntactic query patterns which lead to a very compact representation of the queries. A small example is shown in Table 4 for the input "person drives". Then, the candidates for each substitution are organized according to their taxonomy, and a DL reasoner is used to navigate through their direct subsumers and subsumees. This is especially useful when the system tries to find out the user's intention by adding VTs. A query pattern shows an expression with the VTs not substituted (i.e., with *gaps*) and the system maintains a list of potential candidates for each gap (see Figure 8).

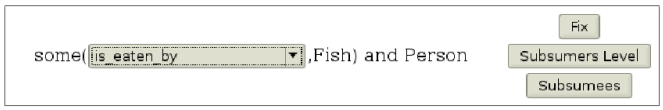


Figure 8: Example of query pattern for "person fish".

See Section 7.3 for an evaluation of the reduction effectiveness of this presentation technique. Finally, note that these patterns could be easily translated into natural language to make the usage of the different information sources and their associated formal query languages transparent to the user, although this is out of the scope of this paper.

5. Access to Data Repositories

Once QueryGen has obtained the intended semantic query, it has to access the underlying data corresponding to its semantics. These underlying data might be stored in different data repositories, with different data organizations, and with different query capabilities (query languages and formats of answers) [47]. Moreover, the accessed data repositories might support queries of different nature, e.g., some repositories process only snapshot queries, while others might be capable of processing continuous ones, refreshing the answers continually and requiring a different communication schema.

To provide QueryGen with enough flexibility to deal with this data heterogeneity, we advocate for the architecture shown in Figure 9, whose main modules are the following:

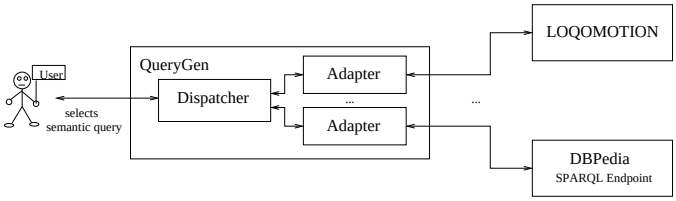


Figure 9: Our system can retrieve data from different channels and data models.

- *Dispatcher*: Once the user selects her/his intended query from those generated by QueryGen, the *Dispatcher* poses the query to the underlying data repositories that are capable to process it. It consults the characteristics of their query processing services that their *Adapters* expose, and, depending on them, creates the appropriate communication channels. For example, a snapshot query only implies one answer, while a continuous query needs answer refreshments along the time. Every underlying system that is capable of processing the selected query is accessed in a parallel way as any of them could hold the desired answer. Finally, the Dispatcher correlates the data coming from the different systems and presents them to the user.

- *Adapter*: It wraps the access to the data stored in information systems with a certain data organization (e.g., there is an Adapter for relational databases, a different one for SPARQL endpoints, etc.). It registers itself in QueryGen providing information about the querying capabilities of the accessed information system, and making itself available to the Dispatcher. There is one instance of the appropriate kind of Adapter for each system accessed by QueryGen.

Thus, once the query has been rendered and selected, it is forwarded to the Adapters that are able to process it. The effort of translating the selected query into the final query language syntax falls upon the Adapter. Note that the Adapter developer is responsible for modelling the language using an abstract grammar, and thus, QueryGen is syntax agnostic: It is completely guided by the query language models provided by the Adapters. Finally, note that the selected query will never have virtual terms: It will be a well-formed query which uses the senses of the input keywords along with ontological terms that have suggested to substitute the virtual terms (in the query rendering step).

In QueryGen, each Adapter Ad_i is characterized by a tuple

$$< \{< lang_{id}, lang_{id}^{spec} >\}, \{Q_{type}\}, \{d_{format}\} >$$

where:

- $\{< lang_{id}, lang_{id}^{spec} >\}$ is the set of languages that the Adapter Ad_i is capable to process. When the Adapter registers itself, the Dispatcher checks whether the language has been already added to the system. If it was, it adds Ad_i to the list of systems that are capable of process $lang_{id}$. Otherwise, it adds the new language to the system, requiring the Analysis Table Constructor to build the information needed to enable QueryGen to use it.
- $\{Q_{type}\}$ is type of queries that the Adapter Ad_i supports. In particular, QueryGen is able to deal with the following types:
  - Snapshot queries: they are posed and answered once, in a pull way (e.g. a SQL query against a relational DB).
  - Monitoring queries: they are posed once, but the answer is updated continually in a pull way (the *Dispatcher* is responsible for retrieving the new data). This type of queries is useful when, independently of the frequency at which the answer/data might change, the updates are not critical and, therefore, the system might relax the resources requirements, executing a snapshot periodically (e.g. a web service providing information about the weather, invoked every hour).
  - Continuous queries: they are posed once, but the answer is updated continually via update events in a push way (e.g. a location dependent query where the position of the mobile objects is continuously changing [29]). It is the underlying system, via its *Adapter*, who updates the answer at the requested query refreshment frequency.
- $\{d_{format}\}$ is the set of data formats that the Adapter Ad_i is able to offer (e.g. CSV values, RDF/XML, etc).

QueryGen relies completely on the implemented Adapters to access data, thus, the way data is actually accessed (e.g., federated query processing) is transparent to QueryGen. QueryGen routes the selected query and establishes the appropriate communication channels with each of the Adapters capable of processing it. In the developed prototype of QueryGen, we have integrated successfully the data access to two information systems of different nature: the location-dependent query processor LOQOMOTION [29] (which processes continuous queries) and DBpedia [8] (as SPARQL Endpoint)[11]. The registering tuples for their Adapters comprise the following information (for LQOOMOTION and DBpedia[12], respectively):

$$< \{< KeyLOQO, KeyLOQOGrammar >\},$$
$$\{Snapshot, Monitoring, Continuous\}, \{CSV\} >$$

$$< \{< SBack, SBackGrammar >\}, \{Snapshot\}, \{RDF\} >$$

As above mentioned, Adapters are in charge of performing the translation of the syntax-free semantic queries into the final languages that their underlying systems process. This architecture is an evolution of the *wrappers* proposed in OBSERVER [37], which adapted the queries to the different answering capabilities of the data repositories. In particular, the main capabilities that our system inherits from OBSERVER are the data integration capabilities, the query alignment capabilities (OBSERVER used static mappings, but dynamic mappings could also be considered), and the ability of processing incomplete queries, this is, accessing several sources to obtain an appropriate answer (this is done at *Adapters* level). The separation into two elements allows us to increase the flexibility and integrate systems that have different query processing capabilities, not only concerning the language expressivity, but also concerning both the query types and data models.

Finally, we want to remark that, independently of the underlying data repository accessed, QueryGen only access to data that corresponds to the semantics stated by the user, which have been maintained all along the process, thanks to the transformation of plain input keywords into a semantic query representing exactly what the user wanted to retrieve.

6. From Keywords to Data: A Complete Example

In this section, we give a complete example from the input of the user to the data retrieved from DBpedia[13] to illustrate each of the steps performed by our system. We restrict the semantics to the different ontologies used in DBpedia for the sake of simplicity in the explanations. As an illustrative example, let us assume that a user watched old cartoons starred by a dumb tall black dog many years ago. S/he does not recall its name (in fact, s/he is thinking about Goofy, the Disney character), but s/he wants to know since when this character exists. As s/he cannot provide more specific input, s/he inputs the keywords "Fictional Dog Appearance":

1. In the disambiguation process, our system offers the user several interpretations for each keyword:
   - For "Fictional" one of the proposed meanings is the concept `dbo:FictionalCharacter`.
   - For "Dog", one of the proposed meanings is an integrated sense containing the DBpedia URL `dbpedia:resource/Dog`, considered as an instance of Animal in the DBpedia ontology.
   - For "Appearance" one of the proposed meanings is the role `dbo:firstAppearance`.
2. In the generation process, the user cannot find the intended query, as no combinations of FictionalCharacter, Dog and

[11] We refer the interested reader to [9] for the details of the semantic models of both query languages.

[12] The DBpedia Adapter performs a query rewriting from the DL queries expressed in simplified BACK into SPARQL.

[13] The namespaces used in this section are *dbpedia* `http://dbpedia.org/`, *dbo* `http://dbpedia.org/ontology/`, *dbpprop* `http://dbpedia.org/property/`, and *dbresource* `http://dbpedia.org/resource/`

firstAppearance represents her/his intended query. However, by considering one VT during the semantic enrichment step, the system can try adding the role `dbpprop:species`. This allows the system to find out the query intended by the user:

$$[firstAppearance](FictionalCharacter\ and\ (Fill\ species\ Dog))$$

which has to be read as "retrieve the first appearance of the fictional characters whose species is dog".

3. The system detects that the query can be processed by the DBpedia Adapter and forwards the query to it. The adapter translates the query into the corresponding underlying query language (a SPARQL sentence):

```
SELECT  * FROM <http://dbpedia.org>
WHERE { ?x a dbo:FictionalCharacter.
        ?x dbpprop:species dbresource:Dog.
        ?x dbo:firstAppearance ?y. }
```

which retrieves the first appearance of several fictional dogs (see Table 5), among which Goofy's can be found[14].

Table 5: Results for the first appearance of fictional dogs returned by DBpedia (including Goofy's).

Character **FirstAppearance**
`dbresource:Huckleberry_Hound` Huckleberry Hound Meets Wee Willie (1958)
`dbresource:Goofy` *Mickey's Revue (1932)*
`dbresource:Scrappy-Doo` The Scarab Lives! (Scooby-Doo and Scrappy-Doo)
`dbresource:Spike_and_Tyke_(characters)` Dog Trouble (18 April 1942)
`dbresource:Spike_and_Tyke_(characters)` Love That Pup (1949)
`dbresource:Bolt_(character)` Bolt
`dbresource:Toto_(Oz)` The Wonderful Wizard of Oz (1900)
`dbresource:Max_Goof` Fathers Are People (1951) (as Goofy. Jr)
`dbresource:Max_Goof` Goof Troop (1992) (as Max Goof)
`dbresource:Puppy_(Alice's_Adventures_in_Wonderland)` Alice's Adventures in Wonderland
`dbresource:Rude_Dog` 1986
`dbresource:Odie` Garfield comic strip (August 8, 1978)

In this example, the system presented an average of five senses for each input keyword, which resulted in 14 query patterns representing 172 queries. If we search Google using the same input ("Fictional Dog Appearance"), it returns 674.000 results, without any reference to Goofy in the ten first pages[15]. The first result returned by Google links a list of famous fictional dogs in Wikipedia. But in that list there is no answer to the user query (first appearance of Goofy): s/he has to browse the whole list of 119 dogs to find Goofy (and recall its name, hopefully), and click its page to look for the information inside the text. Notice that the list returned by our system contains the first appearances of dog characters, while the list in Wikipedia links to dog characters pages (not necessarily containing their first appearance). Thus, taking the same input as starting point, our system has performed a semantic search returning the first appearance of fictional dogs, while using a search engine we can just obtain information about dogs.

[14]The amount of results returned by the public SPARQL endpoint of DBpedia depends on the current workload, the results presented are from May 20, 2015.

[15]Another possible input would be "tall black dog", for which Google does not include any reference to Goofy in the ten first pages either. QueryGen obtains the query using terms from ontologies not used by DBpedia. Thus, we have chosen this example for illustrative purposes.

7. Experimental Results

In this section, we present the evaluation of our approach from different points of view. We start presenting a qualitative evaluation of the whole process against a query set used in a search contest over Linked Data. Then, we analyze the performance of the different steps of the approach, and the reduction rate achieved by the reduction techniques we apply in the query generation process.

7.1. Evaluation of Discovery of User's Intended Query

The goal of this evaluation is to assess the semantic capabilities of QueryGen to discover the meaning of input keywords and achieve their correct interpretation comprising a user's information need. The evaluation has been done against the test query set provided by the search contest Query Answering over Linked Data (QALD)[16][34]. This contest/track focuses on assessing natural language interfaces over Linked Data; however, it also provides the same queries expressed in keywords to assess keyword-based systems. This contest considers DBpedia along with an RDF export of MusicBrainz[17] as data sets, and provides 100 queries with the expected results for each of them. We selected the set for the 2013 contest, QALD-3, to perform the evaluation, and in the following we will refer to it as the QALD query set.

In the QALD query set, for each query, they provide a SPARQL query that expresses the exact semantics corresponding to the input (natural language or keywords). In our test, we consider a success if QueryGen obtains a query that expresses the same semantics (i.e., can be considered as equivalent) as the attached SPARQL query.

To ease the explanations, despite the fact that QueryGen is able to use different query languages to interpret the input, we selected the simplified BACK language extended with: 1) the inverse version of Some and Fill operators, 2) an operator to project the value of a property of an instance, and 3) aggregation operators. We have used a controlled set of 55 ontologies to be able to trace and repeat the experiments (the test

[16]`http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/`, last accessed May 20, 2015.

[17]`http://musicbrainz.org/`, last accessed May 20, 2015.

collection OWLS-TC4[18] plus the ontologies *dbpedia_3.6.owl*[19], *schema.org*[20], *People+Pets*[21], *Koala*[22], *Animals*[23], and WordNet). Moreover, we also considered the set of properties that are automatically extracted by DBpedia, but are not currently included in its ontology (Raw Infobox Property Definitions), and the *lookup service*[24] provided by DBpedia to discover further mappings to DBpedia's resources.

In Table 6, we show the quantitative information about the results[25]: QueryGen achieves the correct interpretation in 64.64% of the cases. The reasons for not achieving a suitable query for the input keywords were: 1) the lack of proper knowledge in the provided ontologies (15 out of 35 cases), and 2) the lack of expressivity of the language (26 out of 35 cases).

Table 6: Success rate of QueryGen against the QALD test query set.

Intended Query	Number of cases	Rate
Generated with 0 VTs	34	64.64%
Generated with 1 VT	27	
Generated with 2 VTs	3	
Not Generated	35	35.36%

In the following, we detail 6 cases out of the 99 queries (see Table 7) to illustrate different issues that we have found performing the test:

- For the input keywords "trumpet player, bandleader", our system maps: *trumpet* and *bandleader* as instances in WNet, which are merged with the homonym resources obtained via the DBpedia *lookup service*; and *player* to the concept Player also in WNet, as "someone who plays a musical instrument". With two extra roles from the resources' ontological context, QueryGen can build *Player and (fill(instrument, Trumpet)) and (fill(occupation, Bandleader))* which is the intended query.
- For the input keywords "San Francisco, nickname", our system discovers *San Francisco* as the city in WNet and merges it with the DBpedia's resource; and obtains *nickname* from the set of automatically extracted properties of DBpedia. With those terms, QueryGen can build the intended query *[nickname] SanFrancisco* with no extra term.
- For the input keywords "current, national leader, Methodist", our system cannot find the intended query. Although it offers *Methodism* as a possible meaning for *Methodist*, it has not enough ontological resources to obtain all the required information to build the query. In particular, in DBpedia, this information is stated via yago:CurrentNationalLeaders, which is out of the knowledge consulted for disambiguating purposes[26].
- For the input keywords "Margaret Thatcher, chemist", *MargaretThatcher* is mapped as an instance of WNet and merged with its DBpedia's resource, and *chemist* is mapped to ontosem:chemist (among others). However, the selected query language lacks an operator to check whether an instance belongs to a particular concept.
- For the input keywords "company, Munich", our system maps *Company* to portal:Company (among others), and Munich to the city instance in WNet (among others). With one extra role (in this case protont:LocatedIn, equivalent to dbprop:location), our system is able to build *Company and fill(location, Munich)*, which is the intended query.
- For the input keywords "film, starring, direct, Clint Eastwood", our system considers *ClintEastwood* altogether as a single instance, obtaining its associated resource from the DBpedia *lookup service*. On the other hand, it maps *film* to schema.org:Movie, among other sources (merged); *starring* to its homonym term in DBpedia ontology (among others), and *direct* to DBpedia:director (merged). However, the instance *ClintEastwood* is needed twice to build the intended query. Thus, with this one extra instance, QueryGen can build the intended query *Movie and fill(director, ClintEastwood) and fill(starring, ClintEastwood)*.

QALD Contest. In Table 8, an excerpt of the results of QALD-3 contest are shown [13]. We will focus on the semantic capabilities of our system to compare its performance with the other contestants.

QALD-3 considers that a question is right when it was answered with an F-measure of 1, and that is partially right when it was answered with an F-measure strictly between 0 and 1; while our evaluation considers a right question when QueryGen has achieved a proper interpretation (regardless data access, which Adapters' different implementations are responsible for). Thus, to align the results of QALD-3 with our evaluation of QueryGen's semantic interpretation, we will consider the number of processed queries (Processed column in Table 8) as an upper bound of the number of correctly interpreted queries, and the sum of both right and partially queries (Right and Partially columns in Table 8) as a lower bound.

QueryGen has been able to process them all using as input the query expressed in keywords, instead of using the query expressed in natural language. Focusing just in the upper bounds,

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

[19] `http://downloads.dbpedia.org/3.6/dbpedia_3.6.owl`

[20] `http://schema.org/docs/schemaorg.owl`

[21] `http://www.cs.man.ac.uk/~horrocks/ISWC2003/Tutorial/people+pets.owl.rdf`

[22] `http://protege.stanford.edu/plugins/owl/owl-library/koala.owl`

[23] `http://www.cs.man.ac.uk/~rector/tutorials/Biomedical-Tutorial/Tutorial-Ontologies/Animals/Animals-tutorial-complete.owl`

[24] `http://wiki.dbpedia.org/projects/dbpedia-lookup`, last accessed May 20, 2015.

[25] QALD-3 only provides 99 queries, instead of the 100 queries provided in the previous edition.

[26] We discarded using YAGO as a disambiguation source in this test due to their complex categories, which introduce ambiguity. For example, concepts such as yago:PresidentsOfTheUnitedStates or yago:SchoolTypes are especially difficult to handle, as explicitly stated by the QALD authors in [34]. Nevertheless, we are working on how to introduce this kind of sources in QueryGen without being affected by the noise.

Table 7: Question, input keywords, generated query, and number of VTs needed for an excerpt of the QALD questions.

Question	Keywords	Query	VTs
Give me a list of all trumpet players that were bandleaders	trumpet player, bandleader	✓ Player and (fill(instrument, Trumpet)) and (fill(occupation, Bandleader))	2 roles
What are the nicknames of San Francisco?	San Francisco, nickname	✓ [nickname] SanFrancisco	0 VTs
Give me all current Methodist national leader	current, national leader, Methodist	✗ no query (due to lack of knowledge)	
Was Margaret Thatcher a chemist?	Margaret Thatcher, chemist	✗ no query (due to lack of expressivity)	
Give me all companies in Munich	company, Munich	✓ Company and fill(location, Munich)	1 role
Which films starring Clint Eastwood did he direct himself?	film, starring, direct, Clint Eastwood	✓ Movie and fill(director, ClintEastwood) and fill(starring, ClintEastwood)	1 instance

Table 8: QALD-3 Results for DBpedia (test query set).

System	Total	Processed	Right	Partially
squall2sparql	99	99	80	13
CASIA	99	52	29	8
Scalewelis	99	70	32	1
RTV	99	55	30	4
Intui2	99	99	28	4
SWIP	99	21	15	2

QueryGen would qualify in 4th position with its 64 correctly interpreted queries, behind squall2sparql [18], Intui2 [17], and close to Scalewelis [25]. If we focus on the QALD-3 lower bound, QueryGen would only be behind squall2sparql, which has been able to obtain a close interpretation to 93 queries (80 of which are perfect interpretations as QueryGen ones). However, we cannot draw more conclusions out of this data, as we cannot assume that the difference between processed and right/partially queries in QALD is due to an incorrect semantic interpretation [27]. Thus, QueryGen would qualify from 2nd to 4th regarding semantic interpretation capabilities, which we consider meritable as QueryGen is using just the plain keyword input, which is a strong simplification of the actual query.

Apart from these results, QALD-3 contest detected four queries that all systems answered, and another four ones that none did. QueryGen achieved the correct interpretation for the four former ones, while managed to obtain the intended query in 3 out of the 4 latter ones. The remaining one is the query "Does the new Battlestar Galactica series have more episodes than the old one?", expressed as "new BattleStar Galactica series, episodes, more, old Battle Star Galactica series" in keywords. QueryGen fails to disambiguate the new from the old series, and even if it could, there is still a lack of expressivity as, apart from applying aggregation to count the episodes, we would need an operator to compare the results of such aggregations.

Evaluation Conclusions. In spite of the lack of knowledge and expressivity of the query language selected, QueryGen has been able to generate the intended queries with exactly the semantics needed for most of the queries:

- The disambiguation process has been successful almost in all the cases, detecting correctly the meanings, and merging and enriching the senses of the keywords.
- The generation process reaches the user's intended query whenever the expressivity of the query language allows to build it (64.64% of the cases for the query language selected in the evaluation).
- The usage of virtual terms to discover implicit meanings is present in the 46.87% of the queries that we have been able to generate with our system. The rest of the successful queries have been generated with 0 virtual terms.
- Anyway, there are still times when QueryGen fails to generate the intended query (35 out of the 99 queries) due to two main reasons: 1) the lack of knowledge, which could be addressed by upgrading the dynamic pool of ontologies; and 2) the lack of expressivity of the query language selected for the test, which can be addressed by adding languages expressive enough to express the user's query.

Thus, the flexibility of QueryGen regarding the sources of knowledge consulted and the query models supported makes it possible to alleviate the problems detected in the evaluation. On the one hand, the more ontologies available in the pool of ontologies consulted by QueryGen, the more semantic interpretations can be considered. On the other hand, despite the fact that we have performed this test with just one selected query language, notice that QueryGen searches for the intended query taking into account all the available languages: for a given input it could express the intended query in some query language, for the next input another query language could be used, automatically.

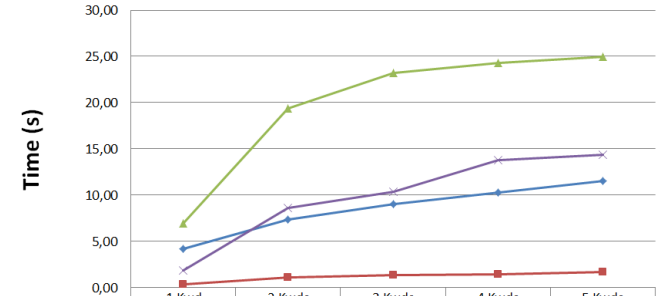
So, despite existing still a long road for improvements in our approach, we think we are on the correct way. In the following subsections, we turn our focus on different performance issues of QueryGen to show the scalability of our approach.

7.2. Keyword Disambiguation Performance

To test the feasibility of our disambiguation technique, we have performed an extensive performance test focused only on

[27] There are no further explanations of each systems's problems in the QALD-3 report.

the first step of our approach. It was executed on a Sunfire X2200 (2 x AMD Dual Core 2600 MHz, 8GB RAM). The set of ontologies used is the same one used in the previous section, a total of 55 ontologies were consulted by our prototype. In this experiment, we established an upper bound to the amount of candidate senses/ontological terms that were considered for each input keyword (i.e., QueryGen was able to consider up to 30 candidate senses for each input keyword), but it was not reached for any input keyword.



	1 Kwd	2 Kwds	3 Kwds	4 Kwds	5 Kwds
Depth 1	4,22	7,35	9,01	10,26	11,51
Depth 1 - cached	0,35	1,08	1,31	1,39	1,65
Depth 2	6,92	19,38	23,23	24,25	24,99
Depth 2 - cached	1,83	8,58	10,38	13,79	14,36

Figure 10: Keyword disambiguation performance evaluation.

The input keywords were selected randomly out from a set of actual queries proposed by students of different degrees with skills in Computer Science, and grouped according to their size in number of keywords. We considered fifty sets of input keywords to perform the tests, ten for each number of keywords. In Figure 10, the results for different sizes of the inputs are shown. As it can be seen, the disambiguation times depend on which depth (how many levels of parent and children terms in the ontological context) is considered for matching. From previous experiments, we have seen that using a depth greater than two lead to wrong results. This is due to the fact that the closer you get to the TOP concept in the ontologies, the more false positives appear, as too general subsumer terms are considered. So, a depth of two levels is considered to be semantically optimal. The cached results correspond to executions on which the extraction procedures had been already performed and stored, as at first, it was the most expensive task. However, the results of the sense alignment step can also be cached, and we expect these times to be even lower.

7.3. Query Generation Performance

We turn our focus now on the performance of the query generation step. To effectively measure the performance of the generation step, as well as the impact of the reduction techniques that QueryGen applies, we used two well-known ontologies out of the previous used set: *People+Pets* and *Koala*. Using their terms as limited vocabulary, we assured that the terms were going to be discovered in the disambiguation process, and that there were no false positives while merging and aligning the possible senses (which might introduce semantic noise in the process). The selected ontologies are two popular ones of similar size[28] to those used in well-known benchmarks such as the OAEI[29]. The DL reasoner used in the tests was Pellet 2.2.2 [43]. Due to space limitations, we only show the experimental results obtained with simplified BACK as output query language because most search approaches are based only on conjunctive queries. Nevertheless, we have also performed the experiments with another non-DL languages, and we obtained similar execution times and conclusions.

For the experiments, we considered different sample sets of input keywords (selected from the terms of the above ontologies) and measured average values grouped by the number of keywords in the set. As in the evaluation of the performance of the disambiguation process, these inputs were based on actual queries proposed by students of different degrees with skills in Computer Science. The sets were chosen according to the following distribution: 10 sets with a single keyword (5 selecting a role and 5 selecting a concept), 15 sets with two keywords (5 sets where both keywords are roles, 5 sets where both keywords are concepts, and 5 sets where one keyword is a role and the other one is a concept), 20 sets with three keywords (5 with 2 concepts and 1 role, 5 with 1 concept and 2 roles, 5 with 3 concepts, and 5 with 3 roles) and, following the same idea, 25 sets with four keywords and 30 sets with five keywords. Notice that, even though our approach can effectively deal with instances as well, we do not consider sets with instances because the selected ontologies do not have instances (as it happens frequently [48]). We set the maximum number of keywords to 5, as the average number of keywords used in keyword-based search engines "is somewhere between 2 and 3" [36], and thus we can see how our system performs with inputs below and above this average number of keywords.

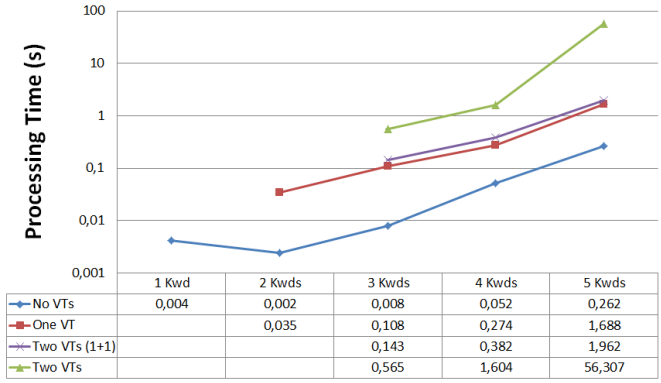
We conducted four experiments: 1) no VTs added, the system works only with the user keywords; 2) one VT added, to try to find a possible missing keyword; 3) two VTs (1+1), is the same situation as 2) with an extra refinement step once the user has selected a candidate for the first VT to be rendered; and 4) two VTs added, to find two possible missing keywords at the same time[30]. We have also considered that the user inputs at least one keyword.

The X-axis in Figures 11.a and 11.b represents the total keywords considered, i.e., the input and the VTs added by the system. Thus, considering 3 keywords, the results are for 3 user keywords (no VTs), 2 user keywords and 1 VT (one VT), and 1 user keyword and 2 VTs. As it can be seen in Figure 11.a (notice that the Y-axis is in log scale), the average times for 3 and 4 keywords are similar and really low (recall that the average number of input keywords was between 2 and 3). Thus, we can look on how they behave regarding the queries presented to the user. Figure 11.b shows the average number of generated

[28] *People+Pets* has 96 terms (60 concepts, 15 roles, and 21 instances), while *Koala* has 32 terms (21 concepts, 5 roles, and 6 instances).

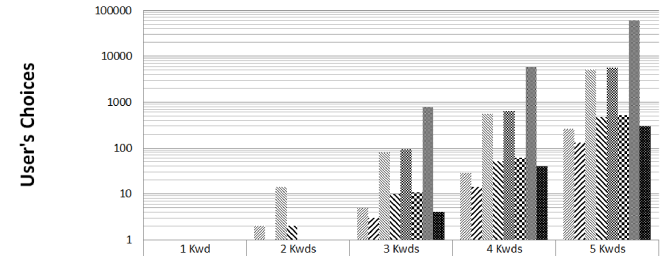
[29] `http://oaei.ontologymatching.org/`, last accessed May 20, 2015.

[30] We do not consider adding more than 2 VTs because we do not aim at discovering the user's intended query when too many keywords were missed in the input.



	1 Kwd	2 Kwds	3 Kwds	4 Kwds	5 Kwds
No VTs	0,004	0,002	0,008	0,052	0,262
One VT		0,035	0,108	0,274	1,688
Two VTs (1+1)			0,143	0,382	1,962
Two VTs			0,565	1,604	56,307

(a)



	1 Kwd	2 Kwds	3 Kwds	4 Kwds	5 Kwds
No VTs (#queries)	1	2	5	28	258
No VTs (#patterns)	1	1	3	14	129
One VT (#queries)		14	79	552	5073
One VT (#patterns)		2	10	51	471
Two VTs (1+1) (#queries)			92	630	5624
Two VTs (1+1) (#patterns)			11	60	521
Two VTs (#queries)			787	5754	61163
Two VTs (#patterns)			4	40	299

(b)

Figure 11: Performance evaluation: processing time, and average number of queries and shown patterns.

queries and the average number of patterns that are presented to the user. In these experiments, taking into account the properties of the different operators reduced the amount of generated queries up to a 70% (it depended on the amount of input keywords, and whether virtual terms where considered), while the semantic filter considered unsatisfiable (and filtered out) a mean percentage ranging from 8.5% to 17% of the resulting queries. Finally, the use of query patterns reduces up to an average 92% the final options that the user is presented with. Figure 11.b also shows that, despite generating a higher number of queries, the system compresses the queries more when it has two VTs at once than in the other situations. This may be beneficial to the user, but it might require her/him more time navigating through the candidate keywords for the VTs. The number of queries is lower for two VTs (1+1) as, in the refinement step, the user has fixed a VT and there are less options. Thus, we would advocate for using only one virtual term at a time, as adding two terms at the same time increases the amount of patterns shown at once to the user.

8. Related Work

One of the first systems whose goal is building formal queries from keywords in the area of the Semantic Web is SemSearch [33]. In this system, the input is matched to semantic entities by means of text indexes, and then a set of predefined templates is used to interpret the queries in the language SeRQL. However, not all the possible queries are considered in those templates, and therefore the system could fail in generating the user's intended query. On the contrary, our system performs a semantic disambiguation process to obtain the meaning of the keywords. Then, it considers the semantics of the languages to interpret and generate all the possible queries semantically correct that are processable by the underlying systems.

Other relevant systems in the area of semantic search are SemSearchPro [45], Q2Semantic [26], SPARK [51], and QUICK [49]. These systems find all the paths that can be derived from a RDF graph, until a predefined depth, to generate the queries. These kind of techniques have been improved and extended in several different ways. In [19] they propose a similar approach to keyword interpretation but they introduce the context of the user's search (the knowledge about previous queries) to focus the whole search process. More recently, in [44], the authors propose to interactively suggest different constraints on the different possible queries to clarify the user's intention in order to enhance the search. Finally, in [32], the authors enhance the scalability and accuracy of these keyword search techniques over graphs by partitioning the underlying RDF graph, and using a summarization of it based on calculated templates to narrow the search space. However, the query model that all these approaches support is restricted to conjunctive queries and their data models are bounded to RDF model. Regarding all these approaches, our system is able to adopt different query models through the definition of its formal language, and the data models supported can be extended via the use of *Adapters*. Besides, none of these approaches supports reasoning capabilities for query reduction and inconsistent query filtering, as opposed to our system. Our approach works at a higher semantic level, as it exploits the background knowledge not only to build new queries but also to infer which ones are satisfiable and to avoid the generation of inconsistent queries.

In the semantic search field, the most similar work to ours is the research line followed by Pound et al. [40, 41]. Firstly, in [41], the authors proposed a structured query language that had keywords as their main building blocks. They exploit this structure to obtain possible interpretations on a particular knowledge base, which has to be previously indexed. In fact, this part of their system could be used by QueryGen by adding their query language. Then, in [40], they focused on how to obtain the possible structured keyword queries from a raw keyword input. To do so, they detect the types (not the semantics) of the keywords using Natural Language techniques, and map these inputs to query schemas that are learned from an annotated query log. Thus, they advocate for finding firstly the structure of the input keywords, to then access the appropriate data. However, as they do not establish the exact semantics of the input keywords in the first step (they work just with part of the speech tags, which are similar to the tokens we use in our extended grammars), they have to run another disambiguation process when accessing the data. Moreover, they are attached to just one query language and data model, and, as they only

use query schemas learned from a log, the system could fail in generating the user's intended query. Finally, as stated above, our system is able to assess the satisfiability of the generated interpretations taking into account both the semantics of the keywords and the query languages, and filter out the inconsistent ones.

There are also some works in the area of databases to provide a keyword-based interface for databases, such as BANKS [1], DISCOVER [28] and DBXplore [2], i.e., translating a set of keywords into SQL queries. However, as emphasized in [6], most of these works rely only on extensional knowledge obtained by applying IR-retrieval techniques, and so they do not consider the intensional knowledge (the structural knowledge). Besides, most of these approaches build only conjunctive queries and require the use of a specific language by the user to express some constraints (e.g., *less than*). In [53], the authors provide a guided search which, given the user keywords, generates a list of *valid* queries (i.e., they correspond to syntactically correct SQL statements). This is done by matching the keywords against some query templates that are learned from previous query logs, and using those templates to suggest possible interpretations. Thus, they do not take into account the actual semantics of the input keywords nor the semantics of the different operators of their target query language as QueryGen does to avoid generating redundant or inconsistent queries.

In this field, Keymantic [6] and QUEST [7] are the most related works to ours. In Keymantic, authors focus on mapping the keywords on entities of the relational schema of a database and interpret them as an SQL query to provide keyword based search over databases without having to process the extensional data. More recently, in QUEST, the authors propose a keyword interpretation that maps the keywords to different terms in the database terminology by exploiting its full-text index (in fact, when they have not access to the database extension, they use Keymantic techniques to do so), and then, they build interpretations in the form of join-paths at schema level. Both approaches are oriented to keyword search over databases and the interpretation process has only one target query model. In turn, our approach is capable of obtaining the semantics of the keywords without specifying a target schema, interpreting the queries into different query models, taking into account the semantics of all the elements (keywords, query language, operators, etc.) all along the process; and, finally, accessing different underlying data models considering the previously well-established semantics.

Although Question Answering systems are traditionally more related to the processing of Natural Language (according to [27], their goal is "to allow a user to ask a question in everyday language and receive an answer quickly and succinctly, with sufficient context to validate the answer"), in essence, QueryGen shares their objectives. According to the classification given in [35], our system would fall into the category of ontology-based semantic QA systems, taking keywords as input. However, instead of having a single data source, our system is flexible enough to adapt itself to different data models by defining them according to their data access methods (query language and types of query processing). Using several different techniques, we tackle some of the traditional problems that these systems face [35]: 1) QueryGen maps the vocabulary of the user to the vocabulary of the data sources (provided that the data sources are semantically described, and the ontology is made available), 2) it disambiguates the different possible interpretations due to polysemy (and, in our case, also due to the lack of expressivity of the keyword model), 3) by consulting dynamic ontological sources, it gets rid of the domain-specific limitations that many QA approaches exhibit, 4) it works even in the absence of enough input via the semantic enrichment step, and 5) last but not least, it does not only use the consulted ontologies to interpret the query, but to filter the inconsistent queries that would make no sense with the help of a DL reasoner. However, we have to bear in mind that the premises which QA systems and QueryGen build on are quite different, and we are aware (and we are working on it) that we can introduce several techniques from these systems to improve the whole semantic keyword-search process that we have presented in this work, as, for example, the input segmentation approach used in [50].

Summing up, to our knowledge, our system is the most flexible solution in terms of query and data models supported. Moreover, our proposal takes into account the semantics of all the elements that take part of the keyword interpretation process, and is the only one that uses a DL reasoner to infer new information and remove semantically inconsistent queries (which can be expressed even in non-DL languages).

9. Conclusions and Future Work

In this paper, we have presented a system that enables semantic keyword-based search by interpreting keyword queries. This involves establishing the semantics of each of the input keywords firstly, and then interpreting them to capture the exact semantics intended by the user. During the process, our system exploits several semantic resources: a pool of third-party ontologies, formal query language specifications, and a Description Logics reasoner. This way, our system, taking a set of keywords as input, is capable of discovering their meaning and giving them a correct interpretation considering the semantics of both the keywords and the query languages (used transparently for the user) all along the process. The proposed system has the following features:

- It discovers the meaning of the input keywords by consulting a pool of ontologies. This makes it possible to handle very different domains, and not to be constrained to a fixed source of semantic information. During this process, our system merges the meanings that are considered similar enough and proposes the most probable semantics for each of the input keywords. To do so, it performs a disambiguation process that takes into account the semantics of all them as a whole.

- Our approach performs the keyword interpretation via a query generation process independent of the available

query languages. In this interpretation process, our system can handle any associated query language whenever it is specified via a semantically annotated grammar. Moreover, it takes into account the semantic properties of the query languages to avoid generating semantically equivalent queries. Finally, the user is not aware of the different underlying query models, as the system performs the generation in all the available ones (they correspond to different data repositories).

- With the help of a DL reasoner, it is capable to filter out inconsistent queries according to the knowledge retrieved. This is not only applicable to DL queries, but also to non-DL languages, which makes our approach very flexible. This filtering is greatly boosted by the fact that the set of generated queries are a conservative extension of the source ontologies, and therefore, their satisfiability can be assessed without having to reclassify the knowledge.

- It performs internally a semantic enrichment of the input to fill the possible gap between the user keywords and the user's intended query. This is done by using *virtual terms*, which allows the system to explore further meanings when the user's input is incomplete. To render them, the system considers the semantic information dynamically obtained and integrated during the disambiguation process.

- The process is independent of the underlying data models and makes the access to them transparent to the user, providing a unique point of entry to heterogeneous systems. By using the Adapters, our system provides huge flexibility regarding the systems that can be accessed. In our prototype, we have successfully integrated two very different information systems such as LOQOMOTION and DBpedia (its SPARQL Endpoint).

Moreover, we have shown the good results achieved by QueryGen regarding its semantic capability to discover the user's intended query from input keywords, and the results of the performance tests carried out have shown the feasibility of our approach.

As future work, we want to study the possibility of mapping input keywords to operators in the formal query languages. This would give us further information that could be used to further narrow the interpretation search space, as it would force the use of a specific operator in the generated queries. In this line, we would like to apply techniques such as those presented in [14] and [52] to improve the interpretation capabilities of QueryGen. Moreover, we want to study how our system could benefit from third-party NLP modules to deal with Named Entity recognition and with full natural language inputs. Finally, we want to further research in the user interaction issue. In particular, we are planning to apply visual techniques to help the user to select their intended query, and perform massive tests with different kinds of final users to measure the semantic accuracy of our prototype.

Acknowledgments

This work has been supported by the CICYT projects TIN2010-21387-C02-02 and TIN2013-46238-C4-4-R, and DGA-FSE. We want to thank the anonymous reviewers for their for their valuable comments on an earlier version of this paper

[1] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, Parag, S. Sudarshan, BANKS: Browsing and keyword searching in relational databases, in: Proc. of 28th Intl. Conf. on Very Large Data Bases (VLDB'02), China, Morgan Kauffman, 2002.

[2] S. Agrawal, S. Chaudhuri, G. Das, DBXplorer: A system for keyword-based search over relational databases, in: Proc. of 18th Intl. Conf. on Data Engineering (ICDE'02), USA, IEEE, 2002.

[3] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 2006.

[4] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Pastel-Scheneider, The Description Logic Handbook. Theory, Implementation and Applications, Cambridge University Press, 2003.

[5] S. Banerjee, T. Pedersen, Extended gloss overlaps as a measure of semantic relatedness, in: Proc. of the 18th Intl. Joint Conf. on Artificial Intelligence (IJCAI'03), Mexico, Morgan Kaufmann, 2003.

[6] S. Bergamaschi, E. Domnori, F. Guerra, R. Trillo-Lado, Y. Velegrakis, Keyword search over relational databases: A metadata approach, in: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'11), Greece, ACM, 2011.

[7] S. Bergamaschi, F. Guerra, M. Interlandi, R. Trillo-Lado, Y. Velegrakis, Combining user and database perspective for solving keyword queries over relational databases, Information Systems 55 (2016) 1–19.

[8] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, S. Hellmann, DBpedia - A crystallization point for the Web of Data, Web Semantics: Science, Services and Agents on the World Wide Web 7 (3) (2009) 154 – 165.

[9] C. Bobed, Semantic Keyword-based Search on Heterogeneous Information Systems, PhD in Computer Science, University of Zaragoza, Spain (December 2013).

[10] C. Bobed, E. Mena, R. Trillo, FirstOnt: Automatic construction of ontologies out of multiple ontological resources, in: Proc. of the 16th Intl. Conf. on Knowledge-Based and Intelligent Information and Engineering Systems (KES'12), Spain, IOS Press, 2012.

[11] C. Bobed, R. Trillo, E. Mena, J. Bernad, Semantic discovery of the user intended query in a selectable target query language, in: Proc. of 7th Intl. Conf. on Web Intelligence (WI'08), Australia, IEEE, 2008.

[12] C. Bobed, R. Trillo, E. Mena, S. Ilarri, From keywords to queries: Discovering the user's intended meaning, in: Proc. of 11th Intl. Conf. on Web Information System Engineering (WISE'10), China, Springer, 2010.

[13] E. Cabrio, P. Cimiano, V. Lopez, A.-C. N. Ngomo, C. Unger, S. Walter, QALD-3: Multilingual Question Answering over Linked Data, in: CLEF 2013 Evaluation Labs and Workshop, Online Working Notes, Spain, 2013.

[14] D. Cameron, A. P. Sheth, N. Jaykumar, K. Thirunarayan, G. Anand, G. A. Smith, A hybrid approach to finding relevant social media content for complex domain specific information needs, Web Semantics: Science, Services and Agents on the World Wide Web. Special Issue on Life Science and e-Science 29 (0) (2014) 39–52.

[15] B. Cuenca, I. Horrocks, Y. Kazakov, U. Sattler, Modular reuse of ontologies: Theory and practice, J. of Artificial Intelligence Research 31 (2008) 273–318.

[16] M. d'Aquin, C. Baldassarre, L. Gridinoc, S. Angeletou, M. Sabou, E. Motta, Characterizing knowledge on the Semantic Web with Watson, in: Proc. of 5th Intl. Workshop on Evaluation of Ontologies and Ontology-based Tools (EON'07), South Korea, CEUR-WS, 2007.

[17] C. Dima, Intui2: A prototype system for Question Answering over Linked Data, in: CLEF 2013 Evaluation Labs and Workshop, Online Working Notes, Spain, 2013.

[18] S. Ferré, SQUALL: The expressiveness of SPARQL 1.1 made available as a controlled natural language, Data & Knowledge Engineering 94, Part B (0) (2014) 163–188.

[19] H. Fu, K. Anyanwu, Effectively interpreting keyword queries on RDF databases with a rear view, in: Proc. of 10th Intl. Semantic Web Conference (ISWC'11), Germany, Springer, 2011.

[20] J. Gracia, M. d'Aquin, E. Mena, Large scale integration of senses for the Semantic Web, in: Proc. of 18th Intl. World Wide Web Conference (WWW'09), Spain, ACM, 2009.
[21] J. Gracia, E. Mena, Ontology matching with CIDER: Evaluation report for the OAEI 2008, in: Proc. of 3rd Ontology Matching Workshop (OM'08), Germany, CEUR-WS, 2008.
[22] J. Gracia, E. Mena, Web-based measure of semantic relatedness, in: Proc. of 9th Intl. Conf. on Web Information Systems Engineering (WISE'08), New Zealand, Springer, 2008.
[23] J. Gracia, E. Mena, Multiontology semantic disambiguation in unstructured Web contexts, in: Proc. of Workshop on Collective Knowledge Capturing and Representation (CKCaR'09), USA, 2009.
[24] T. R. Gruber, Towards principles for the design of ontologies used for knowledge sharing, in: N. Guarino, R. Poli (eds.), Formal Ontology in Conceptual Analysis and Knowledge Representation, Kluwer Academic Publishers, 1993.
[25] J. Guyonvarch, S. Ferré, Scalewelis: A scalable query-based faceted search system on top of SPARQL endpoints, in: CLEF 2013 Evaluation Labs and Workshop, Online Working Notes, Spain, 2013.
[26] Q. L. Haofen Wang, Kang Zhang, D. T. Tran, Y. Yu, Q2Semantic: A lightweight keyword interface to semantic search, in: Proc. of 5th European Semantic Web Conference (ESWC'08), Spain, Springer, 2008.
[27] L. Hirschman, R. Gaizauskas, Natural language Question Answering: The view from here, Natural Language Engineering 7 (4) (2001) 275–300.
[28] V. Hristidis, Y. Papakonstantinou, DISCOVER: Keyword search in relational databases, in: Proc. of 28th Intl. Conf. on Very Large Data Bases (VLDB'02), China, Morgan Kauffman, 2002.
[29] S. Ilarri, E. Mena, A. Illarramendi, Location-dependent queries in mobile contexts: Distributed processing using Mobile Agents, IEEE Trans. on Mobile Computing 5 (8) (2006) 1029–1043.
[30] E. Jimenez, B. Cuenca, U. Sattler, T. Schneider, R. Berlanga, Safe and economic re-use of ontologies: A logic-based methodology and tool support, in: Proc. of 5th European Semantic Web Conference (ESWC'08), Spain, Springer, 2008.
[31] E. Kaufmann, 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 8 (4) (2010) 377 – 393.
[32] W. Le, F. Li, A. Kementsietsidis, S. Duan, Scalable keyword search on large RDF data, IEEE Transactions on Knowledge and Data Engineering 26 (11) (2014) 2774–2788.
[33] Y. Lei, V. S. Uren, E. Motta, SemSearch: A search engine for the Semantic Web., in: Proc. of 15th Intl. Conf. on Knowledge Engineering and Knowledge Management (EKAW'06), Czech Republic, Springer, 2006.
[34] V. Lopez, C. Unger, P. Cimiano, E. Motta, Evaluating Question Answering over Linked Data, Web Semantics: Science, Services and Agents on the World Wide Web. Special Issue on Evaluation of Semantic Technologies 21 (0) (2013) 3–13.
[35] V. Lopez, V. S. Uren, M. Sabou, E. Motta, Is Question Answering fit for the Semantic Web?: A survey, Semantic Web 2 (2) (2011) 125–155.
[36] C. D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008.
[37] E. Mena, A. Illarramendi, Ontology-Based Query Processing for Global Information Systems, Kluwer Academic Publishers, 2001.
[38] G. Miller, WordNet: A Lexical Database for English, Communications of the ACM 38(11).
[39] C. Peltason, The BACK system – An overview, ACM SIGART Bulletin 2 (3) (1991) 114–119.
[40] J. Pound, A. K. Hudek, I. F. Ilyas, G. Weddell, Interpreting keyword queries over Web knowledge bases, in: Proc. of the 21st ACM Intl. Conf. on Information and Knowledge Management (CIKM'12), USA, ACM, 2012.
[41] J. Pound, I. F. Ilyas, G. Weddell, Expressive and flexible access to Web-extracted data: A keyword-based structured query language, in: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'10), USA, ACM, 2010.
[42] W. Rungworawut, T. Senivongse, Using ontology search in the design of class diagram from business process model, in: Trans. on Engineering, Computing and Technology, vol. 12, 2006, pp. 165–170.
[43] E. Sirin, B. Parsia, B. Cuenca-Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, Web Semantics: Science, Services and Agents on the World Wide Web 5 (2) (2007) 51–53.
[44] M. Teng, G. Zhu, Interactive search over Web scale RDF data using predicates as constraints, Journal of Intelligent Information Systems 44 (3) (2015) 381–395.
[45] T. Tran, D. M. Herzig, G. Ladwig, SemSearchPro: Using semantics throughout the search process, Web Semantics: Science, Services and Agents on the World Wide Web 9 (4) (2011) 349 – 364.
[46] R. Trillo, J. Gracia, M. Espinoza, E. Mena, Discovering the semantics of user keywords, J. on Universal Computer Science. Special Issue: Ontologies and their Applications (2007) 1908–1935.
[47] V. Vassalos, Y. Papakonstantinou, Describing and using query capabilities of heterogeneous sources, in: Proc. of 23rd Intl. Conf. on Very Large Data Bases (VLDB'97), Greece, Morgan Kaufmann, 1997.
[48] T. D. Wang, B. Parsia, J. A. Hendler, A survey of the Web ontology landscape, in: Proc. of 5th Intl. Semantic Web Conference (ISWC'06), USA, Springer, 2006.
[49] G. Zenz, X. Zhou, E. Minack, W. Siberski, W. Nejdl, From keywords to semantic queries–incremental query construction on the Semantic Web, Web Semantics: Science, Services and Agents on the World Wide Web 7 (3) (2009) 166–176.
[50] G. Zenz, X. Zhou, E. Minack, W. Siberski, W. Nejdl, SINA: Semantic interpretation of user queries for question answering on interlinked data, Web Semantics: Science, Services and Agents on the World Wide Web. Special Issue on Semantic Search 30 (0) (2015) 39–51.
[51] Q. Zhou, C. Wang, M. Xiong, H. Wang, Y. Yu, SPARK: Adapting keyword query to semantic search, in: Proc. of 6th Intl. Semantic Web Conference (ISWC'07), South Korea, Springer, 2007.
[52] A. J. Zitzelberger, D. W. Embley, S. W. Liddle, D. T.Scott, HyKSS: Hybrid keyword and semantic search, Journal on Data Semantics To be published, first online 7 November 2014 (2014) 1–17.
[53] A. Zouzias, M. Vlachos, V. Hristidis, Templated search over relational databases, in: Proc. of the 23rd ACM Intl. Conf. on Information and Knowledge Management (CIKM'14), China, ACM, 2014.