A Natural Language Query Interface to Structured Information

Valentin Tablan, Danica Damljanovic, and Kalina Bontcheva

Department of Computer Science
University of Sheffield
Regent Court, 211 Portobello Street
S1 4DP, Sheffield, UK
{v.tablan,d.damljanovic,k.bontcheva}@dcs.shef.ac.uk

Abstract. Accessing structured data such as that encoded in ontologies and knowledge bases can be done using either syntactically complex formal query languages like SPARQL or complicated form interfaces that require expensive customisation to each particular application domain. This paper presents the QuestIO system – a natural language interface for accessing structured information, that is domain independent and easy to use without training. It aims to bring the simplicity of Google's search interface to conceptual retrieval by automatically converting short conceptual queries into formal ones, which can then be executed against any semantic repository.

QuestIO was developed specifically to be robust with regard to language ambiguities, incomplete or syntactically ill-formed queries, by harnessing the structure of ontologies, fuzzy string matching, and ontology-motivated similarity metrics.

Keywords: Searching, Querying, User Interfaces, Conceptual Search.

1 Introduction

Structured information in various guises is becoming ubiquitous on today's computers. This may include a user's contacts list, their calendar of events, other structured files such as spreadsheets or databases. In addition to this, unstructured textual content may refer or add information to entities from a user's structured information space. For example a project meeting mentioned in the calendar relates to textual files such as the agenda for the meeting, the various documents relevant to the particular project, the contact information of the people who will be attending the meeting, etc. All this information can be modelled as, or mapped onto, an ontology in order bring the benefits of all the technologies developed for the semantic web to the user's desktop.

We believe that text interfaces have a role to play because they are familiar to end users, benefit from very good support both on the desktop and in web interfaces, and are easily available on all types of devices. In previous work

S. Bechhofer et al.(Eds.): ESWC 2008, LNCS 5021, pp. 361–375, 2008.
© Springer-Verlag Berlin Heidelberg 2008

[1,2] we have presented the CLOnE system (originally named CLIE) that provides a textual interface for editing a knowledge base (**KB**) through the use of an open-vocabulary, general purpose controlled language. That was designed as an interface for manual intervention in the process of generating ontological data from either structured information, through direct mapping, or from unstructured text, through semantic annotation. Continuing the same line of development, the work discussed in this paper focuses on providing access to the information stored in a KB by means of natural language queries.

Most knowledge stores provide facilities for querying through the use of some formal language such as SPARQL or SeRQL. However, these have a fairly complex syntax, require a good understanding of the data schema and are error prone due to the need to type long and complicated URIs. These languages are homologous to the use of SQL for interrogating traditional relational databases and should not be seen as an end user tool.

Different methods for user-friendly knowledge access have been developed previously. Some provide a graphical interface where users can browse an ontology, others offer a forms-based interface for performing semantic search based on an underlying ontology whilst hiding the complexity of formal languages. The most sophisticated ones provide a simple text box for a query which takes Natural Language (**NL**) queries as input.

The evaluation conducted in [3] contains a usability test with 48 end-users of Semantic Web technologies, including four types of query language interfaces. They concluded that NL interfaces were the most acceptable, being significantly preferred to menu-guided, and graphical query language interfaces. Despite being preferred by users, Natural Language Interface (**NLI**) system are not very frequent due to the high costs associated with their development and customisation to new domains, which involves both domain experts and language engineers.

We present QuestIO (**Quest**ion-based **I**nterface to **O**ntologies), a NLI system for accessing structured information from a knowledge base. The main impetus for this work is the desire to create a user-friendly way of accessing the information contained in knowledge stores, which should be easy to use, requiring little or no training.

The QuestIO application is open-domain (or customisable to new domains with very little cost), with the vocabulary not being pre-defined but rather automatically derived from the data existing in the knowledge base.The system works by converting NL queries into formal queries in SeRQL (though other query languages could be used). It was developed specifically to be robust with regard to language ambiguities, incomplete or syntactically ill-formed queries, by harnessing the structure of ontologies, fuzzy string matching, and ontology-motivated similarity metrics.

The following section presents some background information regarding user interfaces for knowledge access, putting this work into context. Next the design and implementation of the QuestIO system are presented, followed by an evaluation of its coverage, portability and scalability. Finally, we conclude and present some plans for future development.

2 Context

Tools for accessing data contained in ontologies and knowledge bases are not new, several have been implemented before using different design approaches which reach various levels of expressivity and user-friendliness.

A popular idea is adding search and browsing support to ontology editing environments. For instance Protégé [4] provides the Query Interface, where one can specify the query by selecting some options from a given list of concepts and relations. Alternatively, advanced users can type a query using a formal language such as SPARQL. Facilities of this type give the maximum level of control but are most appropriate for experienced users.

While typing queries in formal languages provides the greatest level of expressivity and control for the user, it is also the least user-friendly access interface. Query languages have complex syntax, require a good understanding of the representation schema, including knowledge of details like namespaces, class and property names. All these contribute to making formal query languages difficult to use and error prone. The obvious solution to these problems is to create some additional abstraction level that provides a user friendly way of generating formal queries, in a manner similar to the many applications that provide access to data stored in standard relational databases.

One step toward user-friendliness is creating a forms-bases graphical interface where the information request can be expressed by putting together a set of restrictions. Typically these restrictions are provided as ready-made building blocks that the user can add to create a complex query. Systems like this can either be customised for a particular application domain or general purpose. A good example of forms-based interface that offers both generic and domain specific options is provided by the KIM knowledge management platform [5].

Interfaces of this type are well suited for domain specific uses; when customised to a particular application, they provide the most efficient access path to the information. They can take advantage of good support for forms in graphical interfaces and benefit from a long tradition of forms-based interface design. Their downside, however, becomes apparent when moving to a general purpose search interface, in which case they can be either too restrictive or too complex with either too many input fields or too many alternative options.

Probably due to the extraordinary popularity of search engines such as Google, people have come to prefer search interfaces which offer a single text input field where they describe their information need and the system does the required work to find relevant results. While employing this kind of interface is straightforward for full text search systems, using it for conceptual search requires an extra step that converts the user's query into semantic restrictions like those expressed in formal search languages. A few examples of such query interfaces are discussed next.

SemSearch [6] is a concept-based system which aims to have a Google-like Query Interface. It requires a list of concepts (classes or instances) as an input query (e.g. the 'news:PhD Students' query asks for all instances of class News that are in relation with PhD Students). This approach allows the use of a simple

text field as input but it requires good knowledge of the domain ontology and provides no way of specifying the desired relation between search terms, which can reduce precision when there are several ontology properties applicable.

Another notable example, and one of the most mature from this family of systems, is AquaLog [7]. It uses a controlled language for querying ontologies with the addition of a learning mechanism, so that its performance improves over time in response to the vocabulary used by the end users. The system works by converting the natural language query into a set of ontology-compatible triples that are then used to extract information from a knowledge store. It utilises shallow parsing and WordNet, and so requires syntactically correct input. It seems geared mainly towards queries containing up to two triples and expressed as questions (e.g., who, what).

Orakel [8] is a natural language interface (**NLI**) to knowledge bases. The key advantage is support for compositional semantic construction which helps it support questions involving quantification, conjunction and negation. These advanced features come at a cost, however, as the system requires a mandatory customisation whenever it is ported to a new application domain. Due to the expertise required for performing the customisation, this is a fairly expensive process.

ONLI (Ontology Natural Language Interaction) [9] is a natural language question answering system used as front-end to the RACER reasoner and to nRQL, RACER's query language. ONLI assumes that the user is familiar with the ontology domain and works by transforming the user's natural language queries into nRQL. No details are provided regarding the effort required for re-purposing the system.

Querix [10] is another ontology-based question answering system that translates generic natural language queries into SPARQL. In case of ambiguities, Querix relies on clarification dialogues with users.

To summarise, existing language-based query interfaces either support keyword-like search or require full-blown, correctly phrased questions. In this paper, we argue that it is technologically possible and advantageous to marry both kinds of approaches into one robust system, which supports both interaction styles.

3 The QuestIO System

Like many of the systems discussed in the previous section, QuestIO works by converting natural language input into a formal semantic query. Because we use Sesame[1] as a knowledge store, the system is configured to generate SeRQL as a query language. The same architecture and most of the implementation can be used to generate queries in other formal languages.

The driving principles behind the design for the QuestIO system were that:

- it should be easy to use, requiring as little user training as possible, ideally none;

[1] Sesame is an open-source RDF repository. More details are available on its homepage at `http://www.openrdf.org/`

- it should be open domain, with no need for customisation, or with customisation being done automatically or through other inexpensive means;
- it should be robust, able to deal with all kinds of input, including ungrammatical text, sentence fragments, short queries, etc.

In order to be robust, the system needs to attempt to make sense of whatever input it gets; it cannot rely on syntax, grammatically correct queries or enough context to perform disambiguation through linguistic analysis. Unlike other similar systems, our approach puts more weight on leveraging the information encoded in the ontology, and only uses very lightweight linguistic processing of the query text. This ensures that any textual input can be ingested successfully while the bulk of the question analysis work is based on the contents of the ontology, which is a larger resource and more likely to have been well engineered.

When a query is received, some of the contained words will match ontology concepts, while the textual segments that remain unmatched can be used to predict property names and for disambiguation. The sequence of concepts and property names is then converted into a formal query that is executed against the knowledge store. Throughout the process, a series of metrics are used to score the possible query interpretations, allowing the filtering of low scoring options, thus reducing ambiguity and limiting the search space.

3.1 Initialisation of the System

When the system is initialised, it processes the domain ontology. Of great importance to the functioning of this system is the ability to recognise textual references to resources from the ontology or the knowledge base. This is done by automatically creating a *gazetteer* when initialising the system with a given knowledge base. In traditional natural language processing applications, a *gazetteer* is a large list of known words or phrases that need to be recognised in text. These typically include various types of names, such as locations, organisations, or people, and a variety of domain dependent terms. In our case we build a gazetteer by automatically extracting lexicalisations from the knowledge base.

A lexicalisation is a textual form used to refer to a particular concept or entity. Our approach is based on the observation that most ontologies and knowledge bases contain a large amount of textual data that can be used to extract a domain vocabulary. The textual elements that we use include:

- The local part of URIs.
- The values of `rdfs:label` properties.
- The values of a custom list of properties, which can be specified as a customisation option.
- The values of datatype properties that can be converted to a string. This was introduced as as a catch-all case that is intended to capture custom-named properties use to encode entity names – for example properties like 'has alias'. This is also useful because it links characteristics of entities (encoded as property values) to the entity itself, which is sometimes necessary to identify ontology instances that have no names or labels, e.g. identifying an

instance of a 'Deadline' class based on its date value. Identifying all datatype property values in the query can lead to noise but that is mitigated by giving this kind of match a lower score than the previous, more direct, matches.

Because the strings used in an ontology sometimes use different orthography conventions, we attempt to normalise them by:

- recognising and splitting words that use underscores as a form of spacing;
- recognising and splitting CamelCase words;
- finding the morphological root (i.e. noninflected form) for all constituent words;
- for complex noun phrases, we derive lexicalisations based on their constituents, e.g. from "ANNIE JAPE Transducer" we derive "JAPE transducer" and "transducer" as additional ways of referring to the same concept.

For instance, if there is an ontology instance with a local name of "*ANNIEJapeTransducer*", and with assigned property `rdfs:label` with value "*Jape Transducer*", and with assigned property `rdfs:comment` with value "*A module for executing Jape grammars*", the gazetteer will contain following the strings:

- "*ANNIEJapeTransducer*" – the value of the local name;
- "*ANNIE Jape Transducer*" – local name after camel case splitting;
- "*Jape Transducer*" – the value of `rdfs:label`;
- "*A module for execute Jape grammar*" – the value of comment, after morphological normalisation. Note that morphological normalisation is also applied to the words in the input query, thus matching is possible regardless of inflection.

All the lexicalisations thus extracted are stored in a gazetteer that is able to identify mentions of classes, properties, instances, and property values associated with instances. More details regarding the linguistic processing involved can be found in [11].

3.2 Run-Time Operation

When a query is received, the system performs the following steps:

1. linguistic analysis;
2. ontological gazetteer lookup;
3. iterative transformation until a SeRQL query is obtained;
4. execute the query against the knowledge base and display the results.

The linguistic analysis stage performs morphological analysis of the text by running a tokeniser, part-of-speech tagger and a morphological analyser. All these types of linguistic analysis are very lightweight and robust as they do not depend on grammatical structure such as syntax. Because of this, they will complete successfully on any kind of input, be it fully formed sentences or simple fragments. We use a morphological analyser to apply the same kind of normalisation as that used for the lexicalisations from the ontology.This allows us to later match query terms with concepts in the ontology, regardless of the way they are inflected.

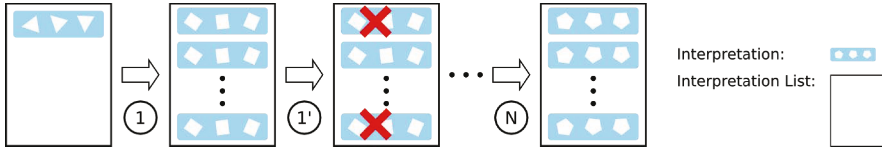

Fig. 1. Information Flow through the System

The second processing phase consists of running the ontological gazetteer built at initialisation time over the morphologically normalised query text. This creates annotations for all mentions of ontological resources that the gazetteer can identify – classes, properties, instances, and values of datatype properties.

Thirdly an iterative transformation process is started for converting the input free text into a formal query. It goes through several steps, each of which starts with a list of candidate interpretations and converts those into more complex ones. The information flow through the system is depicted in Figure 1. At the beginning of the process, the list of candidate interpretations is initialised with a simple interpretation containing the input text and the annotations created in the previous steps. In each iteration, the interpretations currently in the candidate list are transformed into more detailed ones. In cases of ambiguity, it is possible that the number of candidate interpretations grows during a transformation step (see the step numbered `(1)` in the diagram). All candidate interpretations are scored according to a set of metrics that will be detailed later and, in order to keep the number of alternatives under control, candidates that score too low can be eliminated as shown in the step `(1')` in the figure.

Expressed in pseudo-code, the top level algorithm is the following:

```
1  annotate input text with morphological data;
2  annotate input text with mentions of ontology resources;
3  create an interpretation containing the input text and annotations;
4  initialise the candidates list with the interpretation;
5  do
6    apply all possible transformations to the current candidates;
7    eliminate candidates that score too low;
8  until(no more transformation possible)
9  generate SeRQL using top scoring interpretation(s)
```

Following are some details about each of the steps, referred to using line numbers.

Step 1 includes breaking the input text into separate tokens, determining the part of speech for each token and annotating it with its morphological root.

Step 2 deals with identifying in the input text mentions of resources from the ontology. The matching is done using a normalised form that takes care of multi-word names and morphological variation.

Cycle 5 – 8 performs repeated transformations of the candidate interpretations until no more are possible.

Step 9 uses the highest-scoring interpretation to generate a SeRQL query.

In order to create a modular implementation, two abstractions are used: *Interpretation* and *Transformer*. *Interpretations* are used as a container for information – the contents of different interpretation types can be quite different. To begin with, a single Interpretation is created which holds the input text, together with all the annotations resulted from the morphological pre-processing. As the process progresses ever more complex interpretations are generated. Similarly, a *Transformer* represents an algorithm for converting a type of interpretation into another. The implementation of the top level work-flow is quite independent of the types on interpretations known or the kinds of transformers available; the cycle at lines 5-8 simply applies all known types of transformers to the current list of interpretations.

Creating a particular implementation of the system requires defining a set of interpretations and transformers (as Java classes that implement particular interfaces). In our current implementation, an interpretation is defined by the following elements:

- A list of interpretation elements which hold the actual data. The actual type of data held varies, becoming more and more detailed as the iterative process continues. We are currently using ten types of different interpretation elements of various complexity.
- A list of text tokens representing the original query text. Each interpretation element is aligned with a sequence of one or more input tokens. This is used to keep track of how each text segment was interpreted.
- A back-link to the interpretation that was used to derive the current one. This can be used to back-track through the interpretation steps and justify the reasoning that led to the final result.
- A score value – a dynamically calculated numeric value used to sort the interpretations list based on confidence.

The current implementation has seven types of transformers, which are used during the iteration steps for converting interpretations. They perform various operations to do with associating input text segments to ontology resources.

3.3 Identifying Implicit Relations

One of the main functions performed by the transformers is to identify relations which are not explicitly stated in the input query. After the ontological gazetteer is used to locate explicit references to ontology entities, the remaining text segments between those references are used to infer relations. Relations between query terms are essentially homologous with object properties in OWL, so our system attempts to match snippets from the input query to properties in the ontology. The process starts by identifying a list of possible candidates based on the definitions in the ontology: from two consecutive ontological references, we identify the ontology classes they belong to. References to classes are used directly; in the case of instance references we identify the class of the instance; for property values we first find the associated instance and then its class. If the reference is to a property name, we consider that to be an explicit relation

mention and this process does not take place. Once we have the two classes, we build a list of properties that could conceivably apply given the range and domain restrictions in the ontology.

The list of applicable properties is then used to generate candidate interpretations that are scored using the following metrics:

String Similarity Score. This score is based on the assumption that the words used for denoting a relation in the query might be quite similar to some of the lexicalisations of the candidate properties. The actual numeric value is calculated using the *Levenshtein distance metrics* which represents the distance between two strings as the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. Resulting scores are normalised to range from 0 to 1.

Specificity Score. This score is based on the sub-property relation in the ontology definition. The assumption behind using this is that more specific properties are preferable to more general ones. Specificity score is determined by the distance of a property from its farthermost super-property that has no defined super-properties. This makes more specific properties, i.e. that are placed at deeper levels in the property hierarchy, score higher. Its value is normalised by dividing it with the maximum distance calculated for the properties on the ontology level.

Distance Score. This metric was developed to compensate for the fact that in many ontologies the property hierarchy is rather flat. It is trying to infer an implicit specificity of a property based on the level of the classes that are used as its domain and range. Consider the following example: a top level class named "`Entity`" with a subclass "`GeographicalLocation`" and the properties "`partOf`" (with range and domain "`Entity`) and "`subregionOf`" (with range and domain "`GeographicalLocation`"). Although not expressed explicitly using sub-property relations, the property "`subregionOf`" can be seen as more specific than "`partOf`". Assuming that the system is trying to find the most appropriate relation between two instances of types "`City`" and "`Country`" (both subclasses of "`GeographicalLocation`") then, all other things being equal, the "`subregionOf`" property should be preferred.

The general case is illustrated in Figure 2, where classes are represented as circles, inheritance relations are shown as vertical arrows and ontological object properties are presented as horizontal arrows. Given two classes, one considered to be domain, the other range, the distance score for a candidate property represents the length of the path travelled from the domain class to the range class when choosing the property. These distances are represented by the thick dotted lines in the figure. Once calculated, the distances are normalised, through division by double the maximum depth of the ontology and inverted – as the shortest path represents the most specific property and should yield the highest score.

These similarity metrics are ontology-motivated and are largely comparable to those used in the AquaLog system. The final score associated with the candidate properties is a weighted sum of the three different atomic measures.

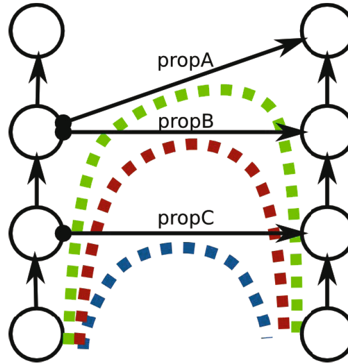

Fig. 2. Distance Score

3.4 Creating Queries

After the implicit relations have been identified, the final interpretation of the input query is presented as a list of explicit references to ontology resources interspersed with references to properties. Starting from this list of interpretation elements, a formal SeRQL query is dynamically created. In order to do this, references to instances are kept as they are, i.e. the instance URI is used directly in the query. References to classes are converted to query variables, with associated restrictions encoding the class membership. References to properties, either explicit or inferred, are used to create restrictions with regard to relations between the various query elements, either instance URIs or variables.

For example, we ran the system with the PROTON Ontology[2] populated with instances from the default knowledge base of the KIM system [12]. A textual query of "*Countries in Asia*" generated the following SeRQL formal query:

```
select c0, p1, i2
from
  {c0} rdf:type {<http://proton.semanticweb.org/2005/04/protonu#Country>},
  {c0} p1 {i2},
  {i2} rdf:type {<http://proton.semanticweb.org/2005/04/protonu#Continent>}
where
  p1=<http://proton.semanticweb.org/2005/04/protont#locatedIn> and
  i2=<http://www.ontotext.com/kim/2005/04/wkb#Continent_T.2>
```

The URI of "`http://www.ontotext.com/kim/2005/04/wkb#Continent_T.2`" is that of Asia.

For the provided query, the words "*Country*" and "*Asia*" were recognised as lexicalisations of objects in the ontology (a class and an instance respectively). In the context of the PROTON ontology, given the class "`Country`" as a

[2] The PROTON Ontology is a Base Upper-level Ontology developed within the Sekt project. More details about it can be found at its home page at `http://proton.semanticweb.org/`

domain and the class "`Continent`" (to which the instance "*Asia*" belongs) as a range, the list of candidate properties includes "`partOf`", "`subRegionOf`", and "`locatedIn`". The "`partOf`" property is defined at the very high level of the class "`Entity`", and so is very generic. The other two candidates have the same specificity, "`locatedIn`" being preferred in this case because of its closer textual similarity with the query due to the word "*in*" appearing the original input.

4 Evaluation – Coverage, Portability, Scalability

The first use case of QuestIO is in supporting developers working on open-source projects to find information from source code and user manuals, more efficiently via ontologies. More specifically, in the context of the TAO[3] project, we experimented first with learning semi-automatically a domain ontology [13] from the GATE source code and XML configuration files. It encodes the component model of GATE, the various plug-ins included in the default distribution, the types of modules included in each of the plug-ins, the parameters for for the different modules, their Java types, the associated comments, etc. The resulting ontology contains 42 classes, 23 object properties and 594 instances.

In order to evaluate QuestIO's ability to answer queries against a given ontology (i.e., its coverage and correctness), we started off by manually collecting a subset of 36 questions posted by GATE users to the project's mailing list in the past year, where they enquire about GATE plug-ins, modules and their parameters. Some example questions are: "Which PRs[4] take ontologies as a parameter?", "Which plugin is the VP Chunker in?", and "What is a processing resource?"

We divided these 36 randomly chosen postings into two groups: *answerable*, i.e., the GATE ontology contains the answer for this question; and *non-answerable*, i.e., the required information is missing in the ontology. We ran QuestIO on the subset of 22 answerable questions with the results as follows:

– The system interpreted *correctly* 12 questions and generated the appropriate SPARQL queries to get the required answer.
– *Partially-correct* answers were given to another 6 questions, where the system produced a query, but missed out one of the required constraints, so the answer was less focused. Partially correct answers are scored as 50% correct and 50% wrong.
– Lastly, the system *failed* to interpret 4 questions, so there was either no query generated or the generated query was not correct.

Overall, this means that 68% of the time QuestIO was able to interpret the question and generate the correct SPARQL queries. This result compares favourably to other ontology question-answering systems, e.g., in a similar experiment AquaLog was able to interpret around 50% of the questions posed.

[3] http://www.tao-project.eu
[4] PRs are a kind of GATE module for processing language.

Table 1. Knowledge Base Sizes

GATE Knowledge Base	
Classes	42
Object Properties	23
Instances	594
Total size (C + P + I)	659
Initialisation time	19 seconds

Travel Knowledge Base	
Classes	318
Object Properties	86
Instances	2790
Total size (C + P + I)	3194
Initialisation time	109 seconds

In another experiment, the scalability and portability of QuestIO were evaluated by using a completely different ontology with a larger knowledge base. The ontology used was Travel Guides Ontology[5] created to represent the travel guides domain, encoding geographical information for descriptions of locations as well as typical information from the tourism domain to do with hotels, room amenities, touristic attractions, etc. The ontology was created as an extension of the PROTON ontology, by adding 52 domain specific classes and 17 object properties to the existing 266 classes and 69 properties. The resulting ontology was then populated with 2790 instances [14]. A comparison between the sizes of the knowledge bases in the two experiments can be seen in Table 1. The last rows show the time spent during system initialisation in each case. It can be noted that, even for a sizeable knowledge base, this stays within reasonable limits, which is important even if it only occurs once in the system life-cycle.

Table 2 details the execution times for converting queries of varying degrees of complexity using the two knowledge bases. The complexity of the queries is measured in terms of the number of relations that are implied in the query – a query that mentions two concepts implies a relation between them, a query with three concepts implies two relations, etc. The queries shown in the last column are actual queries, as used during the experiments: note that they are mainly short fragment queries, with incomplete grammatical structure and with incorrect capitalisation. The last query used for the travel knowledge base makes use of the "`GlobalRegion`" class of PROTON, which is used to represent geographical areas smaller than a continent, e.g. "*North Asia*". This is somewhat unnatural but we used it in order to create an artificially longer query (the question having already been answered as a result of the previous query).

The two experiments show that the QuestIO system is capable of working without requiring any customisation on two knowledge bases that are quite different both semantically and in terms of size. As it can be seen from the figures in Table 2, the system scales well – the query conversion times remain within the range of a few seconds even for the more complex queries against the larger knowledge base.

Both knowledge bases used in the evaluation experiment were pre-existing ones and were in no way modified or customised for use with QuestIO. In a real application setting were the system is deployed, the results can be improved by

[5] http://goodoldai.org.yu/ns/tgproton.owl

Table 2. Query Execution Times

Query size (number of properties)	Execution time (seconds)	Number of results	Actual query
GATE Knowledge Base			
1	0.148	15	"*processing resources in ANNIE?*"
2	0.234	37	"*parameters for processing resources in ANNIE?*"
3	0.298	37	"*Java Class for parameters for processing resources in ANNIE?*"
Travel Knowledge Base			
1	1.013	52	"*countries in asia*"
2	2.030	52	"*capitals of countries in asia*"
3	3.307	52	"*capitals of countries in global regions in asia*"

ensuring a better lexicalisation of the ontology and knowledge base. This can be achieved by adding more labels or comments to the ontological resources which should help get better coverage of the domain vocabulary. Another option is to use a specialised domain dictionary that can provide synonyms for the domain terms. The use of general purpose dictionaries, such as WordNet [15,16], might not be advisable as, depending on the actual domain, they may have poor coverage or might actually introduce errors, in domains where the vocabulary is very precise.

The main purpose for developing the QuestIO system was to provide a user-friendly interface for interrogating knowledge bases. A qualitative indication of the success for this enterprise can be ascertained from comparing the textual query "*capitals of countries in global regions in asia*" with the equivalent formal query expressed in SeRQL:

```
select c0, p1, c2, p3, c4, p5, i6
from
  {c0} rdf:type{<http://proton.semanticweb.org/2005/04/protonu#Capital>},
  {c2} p1 {c0},
  {c2} rdf:type {<http://proton.../04/protonu#Country>},
  {c2} p3 {c4},
  {c4} rdf:type {<http://proton.../04/protonu#GlobalRegion>},
  {c4} p5 {i6},
  {i6} rdf:type {<http://proton.../04/protonu#Continent>}
where
  p1=<http://proton.semanticweb...04/protonu#hasCapital> and
  p3=<http://proton.semanticweb...04/protont#subRegionOf> and
  p5=<http://proton.semanticweb.../04/protont#subRegionOf> and
  i6=<http://www.ontotext.com/kim/2005/04/wkb#Continent_T.2>
```

5 Conclusion and Future Work

In this paper we presented the QuestIO system which converts a wide range of text queries into formal ones that can then be executed against a knowledge store. It works by leveraging the lexical information already present in the existing ontologies in the form of labels, comment and property values. By employing robust language processing techniques, string normalisation, and ontology-based disambiguation methods, the system is able to accept syntactically ill-formed queries or short fragments, which is what non-expert users have come to expect following their experience with popular search engines.

Work on QuestIO is continuing, and some of our plans for the future include moving from to the current single-shot query approach towards a session-based interaction, as also experimented with by AquaLog. This would make the interaction more user friendly in cases when the system can find no results for a query or the user is unhappy with the results. By making use of the session history combined with the information about the contents of the ontology, the system could guide the user toward the desired results, e.g. by suggesting relations that were missed due to the user choosing a different lexicalisation.

The current implementation already has the facility to track the process of transforming a textual query into a formal one, which was introduced to support better user interaction in the future. One example of using this would be to present, on request, a justification of how the current results are related to the query. The users who dislike black-box systems may use this to better understand how the query transformation works which, in turn, could improve the efficiency of them using the system.

We are also planning to perform a user satisfaction evaluation comparing our system with a forms-based interface, such that of KIM.

Acknowledgements. This research was partially supported by the EU Sixth Framework Program project TAO (FP6-026460).

References

1. Tablan, V., Polajnar, T., Cunningham, H., Bontcheva, K.: User-friendly ontology authoring using a controlled language. In: 5th Language Resources and Evaluation Conference (LREC), Genoa, Italy, ELRA (May 2006)
2. Funk, A., Tablan, V., Bontcheva, K., Cunningham, H., Davis, B., Handschuh, S.: Clone: Controlled language for ontology editing. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ISWC 2007. LNCS, vol. 4825, Springer, Heidelberg (2007)
3. Kaufmann, E., Bernstein, A.: How useful are natural language interfaces to the semantic web for casual end-users? In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, Springer, Heidelberg (2007)
4. Noy, N., Sintek, M., Decker, S., Crubézy, M., Fergerson, R., Musen, M.: Creating Semantic Web Contents with Protégé-2000. IEEE Intelligent Systems 16(2), 60–71 (2001)

5. Popov, B., Kiryakov, A., Ognyanoff, D., Manov, D., Kirilov, A., Goranov, M.: Towards Semantic Web Information Extraction. In: Fensel, D., Sycara, K.P., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, Springer, Heidelberg (2003)
6. Lei, Y., Uren, V., Motta, E.: Semsearch: a search engine for the semantic web. In: Managing Knowledge in a World of Networks, pp. 238–245. Springer, Heidelberg (2006)
7. Lopez, V., Motta, E.: Ontology driven question answering in AquaLog. In: Meziane, F., Métais, E. (eds.) NLDB 2004. LNCS, vol. 3136, Springer, Heidelberg (2004)
8. Cimiano, P., Haase, P., Heizmann, J.: Porting natural language interfaces between domains: an experimental user study with the orakel system. In: IUI 2007: Proceedings of the 12th international conference on Intelligent user interfaces, pp. 180–189. ACM, New York (2007)
9. Mithun, S., Kosseim, L., Haarslev, V.: Resolving quantifier and number restriction to question owl ontologies. In: Proceedings of The First International Workshop on Question Answering (QA 2007), Xian, China (October 2007)
10. Kaufmann, E., Bernstein, A., Zumstein, R.: Querix: A natural language interface to query ontologies based on clarification dialogs. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 980–981. Springer, Heidelberg (2006)
11. Damljanovic, D., Tablan, V., Bontcheva, K.: A text-based query interface to owl ontologies. In: 6th Language Resources and Evaluation Conference (LREC), Marrakech, Morocco, ELRA (May 2008)
12. Popov, B., Kiryakov, A., Kirilov, A., Manov, D., Ognyanoff, D., Goranov, M.: KIM – A semantic platform for information extraction and retrieval. Natural Language Engineering 10, 375–392 (2004)
13. Bontcheva, K., Sabou, M.: Learning Ontologies from Software Artifacts: Exploring and Combining Multiple Sources. In: Workshop on Semantic Web Enabled Software Engineering (SWESE), Athens, G.A., USA (November 2006)
14. Damljanovic, D., Devedzic, V.: Applying semantic web to e-tourism. In: Ma, Z. (ed.): The Semantic Web for Knowledge and Data Management: Technologies and Practices. IGI Global (2008)
15. Miller, G.A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K.: Introduction to WordNet: On-line. In: Distributed with the WordNet Software (1993)
16. Fellbaum, C. (ed.): WordNet - An Electronic Lexical Database. MIT Press, Cambridge (1998)