http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 9 (1): 67-78, 2010
ISSN 1812-5638

Querying Ontology using Keywords and Quantitative Restriction Phrases

[1]N. Hasany, [1]A.B. Jantan, [2]M.H.B. Selamat and [1]M.I. Saripan
[1]Department of Computer and Communications System Engineering,
Faculty of Engineering, University Putra Malaysia, Malaysia
[2]Department of Computer Science and Information Technology, University Putra Malaysia, Malaysia

Abstract: Many approaches for converting keyword queries to formal query languages are presented for natural language interfaces to ontologies. Some approaches present fixed formal query templates, so they lack in providing support with increasing number of words in the user query. Other approaches work on constructing and manipulating subgraphs from RDF graphs so their processing is complex with respect to time and space. Techniques are presented to perform operations by obtaining a reduced RDF graph but they limit the input to some type of resources so their complete complexity with all type of input resources is unknown. For formal query generation, we present a variable query template whose computation is facilitated by less complex and distributed RDF property and relation graphs. A prototype QuriOnto is developed to evaluate our design. The user can query QuriOnto with any number of words and resource types. Also, to the best of our knowledge, it is the first system that can handle quantitative restrictions with keyword queries. As QuriOnto has no support for semantic similarity at this time except for rdfs labels so its recall is low but high precision shows that the approach is promising for the generation of corresponding formal queries.

Key words: Natural language interfaces, ontology, keyword search, SPARQL, query, semantic web

INTRODUCTION

Many systems providing natural language interfaces to ontologies have been presented, based on either syntactic (Kaufmann *et al*., 2006; Mithun *et al*., 2007; Wang *et al*., 2007) or keyword or unrestricted input (Lei *et al*., 2006; Kaufmann *et al*., 2007; Tran *et al*., 2007; Damljanovic *et al*., 2008; Tablan *et al*., 2008; Wang *et al*., 2008; Ramachandran and Krishnamurthi, 2009). However, search engine users feel more comfortable with keyword input because it did not restrict users to follow grammatical rules hence require less time and effort in formulating a query. From the systems mentioned earlier, it can easily be observed that the latest trained is towards keyword or unrestricted input approach.

We discuss here the two major differences for keyword search interface for web pages and for ontologies. The first difference lies in the expectation of the output or result generated by the interface. The search engine for web pages return the links to pages and documents containing the user words based on some ranking criteria. However, posting a keyword query to a search interface for ontology, the user expects the specific and precise result to be returned.

The second difference is in the processing mechanism for the user query. For finding the terms in a web page, the general information retrieval techniques are applied to count the occurrences of the word or their synonyms. But, as ontology posses a formal structure with well-defined concepts and relations related to a domain, the keywords query needs an understanding to be explored for the supplied user terms. This interpretation is then needed to be converted in a formal query language e.g., SPARQL (Prud and Seaborne, 2006) to access the facts from the ontology. This difference in processing a user request for HTML search and for ontology is shown in Fig. 1. The transformation logic mentioned is responsible for the two issues of processing the user query i.e., to provide the possible interpretation for the user query and to deal with the issues for converting it into the formal query language for the retrieval of facts from the ontology. Present system QuriOnto discusses the mechanism for this transformation logic with both aspects.

On one side, QuriOnto provides the user to get to the specific information on the privilege of writing natural language keyword queries and on the other side, the domain independent architecture of QuriOnto verifies its portability to any domain. QuriOnto uses variable template to support any number of resources from the user query. Its processing is based on two simple property graphs. With the combination of property

Corresponding Author: Noman Hasany, Department of Computer and Communications System Engineering, Faculty of Engineering, University Putra Malaysia, Malaysia

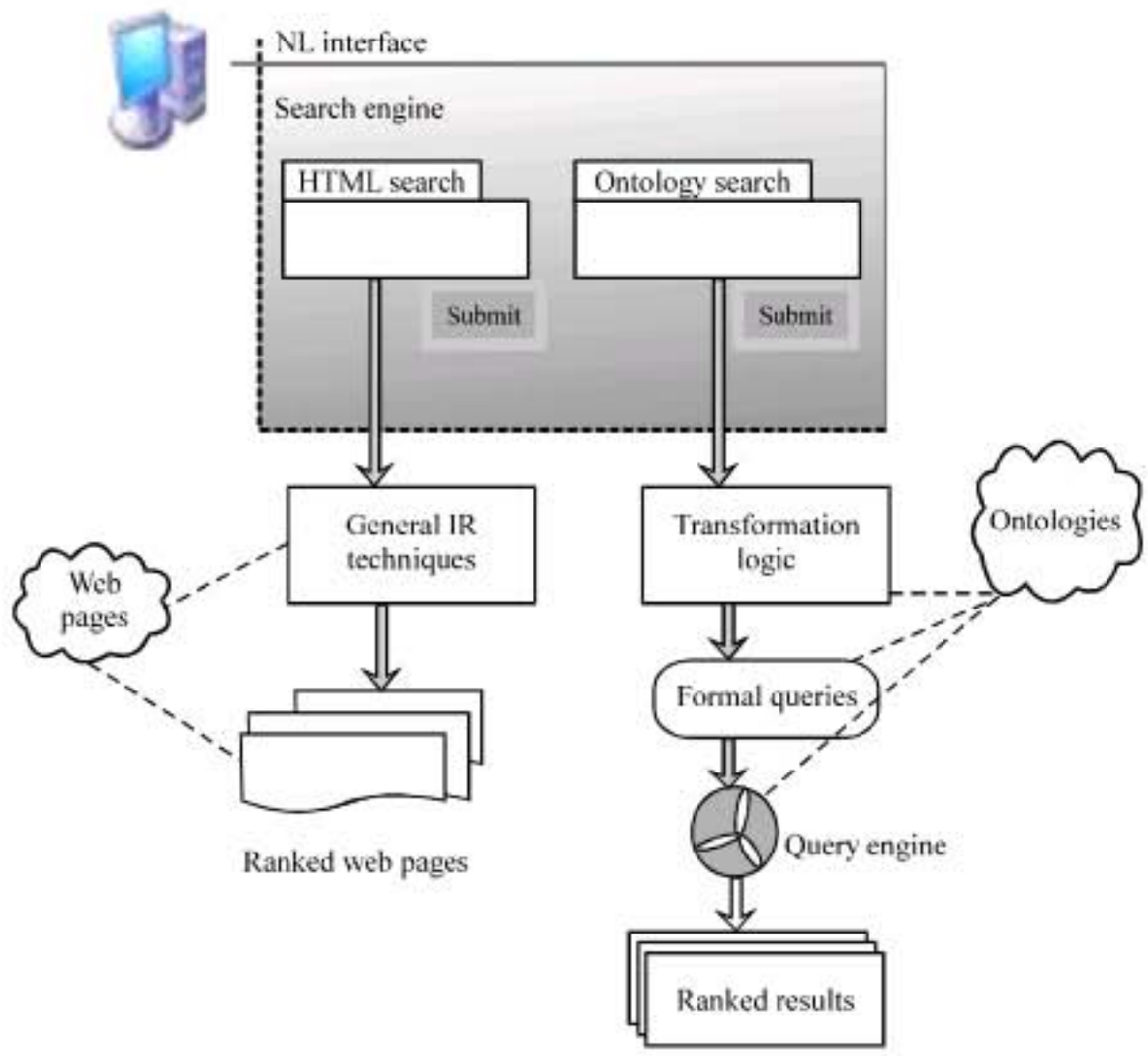

Fig. 1: Overview of an ontology search supported search engine

graphs and variable template, QuriOnto supports better coverage for SPARQL, as to the best of our knowledge; QuriOnto is the first system that handles quantifier restrictions with keyword queries.

THE PROBLEM MODEL

Before discussing the actual problem, we discuss the ontology model O_M on which we apply the user query elements. For our problem model, we define the ontology model O_M as:

$$O_M : < C_S, I_S, PD_S, PO_S, T_s, L_S, S_{PD}, S_{PO} >$$

Where:

C_S = Set of classes
I_S = Set of individuals
PO_S = Set of object properties that maps C_i to C_j, where $C_i, C_j \in C_S$
T_S = Set of literal datatypes
PD_S = Set of datatype properties that maps C_i to T_i, where $C_i \in C_s$, $T_i \in T_s$
L_S = Set of literal strings used in the ontology as values of T_i, where $T_i \in T_s$
S_{PO} = Set of statements, where each statement is a triple of type:

$$< C_s, I_s > x\ PO_s\ x < C_s, I_s >$$

where, S_{PD} is the set of statements of, where each statement is a triple of type:

$$< C_S, I_S > x\ PD_S\ x < L_S >$$

where, O_R is the ontological resource model contains the elements that can be distinguished using URIs, hence:

$$O_R: < C_S, I_S, PD_S, PO_S >$$

The elements of O_R defines the formal vocabulary of the ontology.

Generally, the problem can be viewed as converting a natural language query Nq to all possible formal ranked queries, $Fq_1 \ldots Fq_n$. This implies a two step mapping:

$$f_1: Nq \rightarrow Pq_1 \ldots Pq_n$$
$$f_2: Pq_i \rightarrow Fq_i \mid null$$

where, f_1 provides a mapping of user words to formal ontology vocabulary O_R. The collection of all mapped terms with their ontological category is a pattern Pq. f_2 applies rules to interpret the combination in Pq_i and to obtain a formal query Fq_i representative of Nq.

THE QURIONTO

QuriOnto takes the user input as free-form natural language keywords, maps it to ontological resources,

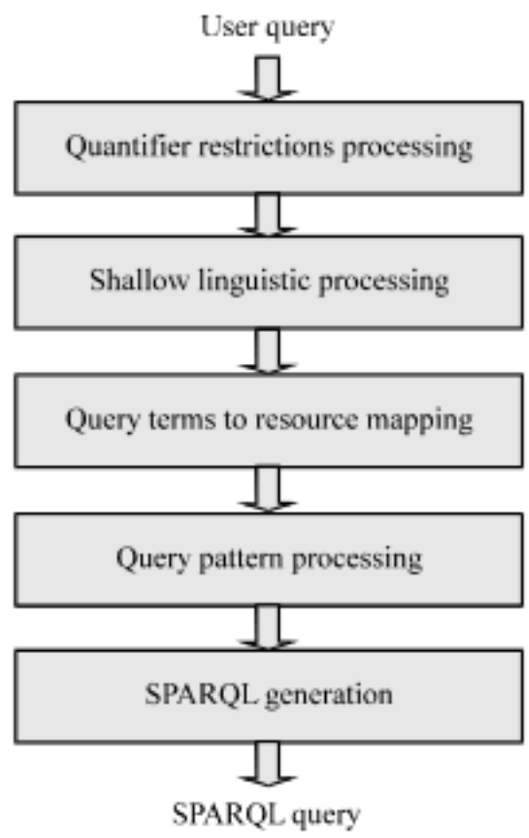

Fig. 2: Phases of QuriOnto

provides an interpretation to it and compute candidates for the elements of the variable template for construction of the SPARQL query which is then presented to Jena access layer for execution to access the facts from the selected ontology.

QuriOnto undergoes five processing steps to achieve the target query, which are termed as shallow linguistic processing or preprocessing, filter expression processing, resource matching, pattern identification and processing and formal query generation.

The steps mentioned in Fig. 2 are repeated for each user query for the ontology. But the preprocessing is performed once when an ontology is selected for querying.

Preprocessing: The preprocessing of QuriOnto involves a list of ontological resources and two matrices representing graphs. List of ontological resources i.e., classes, individuals, object property and datatype property are maintained separately to facilitate the matching in the mapping process. A matrix representation of the object property triples with each domain and range extended to the subclasses also is defined to determine the relation between classes. We term that a PO-C-C graph. Another matrix that we call a C-PD graph is maintained to store datatype properties along with classes on which they are defined.

Quantitative restrictions processing: This phase of QuriOnto is responsible for dealing with the quantifier restriction phrases. We consider 19 modifiers shown in Table 1 and two heuristic cases discussed below as quantifier phrases.

Table 1: Quantitative Restriction phrases for QuriOnto and equivalent operators

Modifier phrases	Relational operator
Less than/below	<
Less than or equal/equal to or less than/equal or less than/below or equal/equal or below	≤
Greater than/above/more than	>
Greater than or equal/equal or greater than/above or equal/equal or above/more than or equal to/at least/at least	≥
Equal to/exactly	=

The syntactic input based ONLI+ (Mithun *et al*., 2007) has considered only 10 modifiers without any heuristics. Besides some new modifiers for QuriOnto we also consider some options which are usually uncommon in written or spoken english but can occur while writing in a sequence without going back to correct the form due to the confidence on keyword interface. For example, observe the phrase "above or equal", however the commonly used form is "equal or above". This increased list makes an option to tolerate up to some extent the ungrammatical input regarding the modifier phrases.

The phrases listed in Table 1 are replaced directly by the corresponding relational operators in the query and the tag is changed to FO i.e., filter operator to exclude it from any tagging or morphological processing. In addition to these modifiers, the two rules are listed below that can yield to filter expressions also.

Rule 1: when quantifier restriction is not a single value but a range specified using terms "between", "in the range of" or simply the word range. Such cases need an intermediate step before operator conversion. As the user is free to formulate the query by providing the larger value first, range values are needed to be checked. Consider the queries, "salary between 10000 to 20000" or "salary in the range of 20000 to 10000" then these values are parsed and converted to float to determine the lesser and greater values. This expression is then converted into the following filter expression:

(?salary ≤ lesser_value & & ?salary ≥greater_value)

Rule 2: If the user specifies a value besides a datatype property with or without a proposition has two possibilities:

- It can be a value of a datatype property, for example in phrases like "students with GPa 3" or "students with 3 GPa" or "students with GPa of 3"
- It can be a part of an ontological resource name, for example in our extended university ontology we have

instances like 'Batch2006,' 'Semester_1' etc. But it is difficult to come up with a final decision in this phase as ontological resources are not explored. So, two interpretations are marked and carried forward to the next phase. If the concatenation of the numerical value with the adjacent term corresponds to an individual or class, the second interpretation is discarded, otherwise the equal to operator interpretation is carried forward

Shallow linguistic processing: The shallow linguistic processing phase helps QuriOnto in two ways. First, it removes stop words and noisy words from the query. Second, it makes the remaining query words capable for the processing in mapping phase discussed next.

Stop words like 'a', 'an' and 'the' are first removed from the query. Noisy words or phrases are those which have no positive impact from the target query point of view but rather have a drastic negative impact. These words will be propagated to the later phases and the system will try to accommodate them in each processing phase. Some noisy words and phrase are "show", "tell me", "please I want to know", "find for me" etc. QuriOnto has taken the list of these words/phrases from the questions of the Mooney data (Tang and Mooney, 2001).

The remaining query words are then passed from morphological processing and tagging. Morphological root provides help for matching of inflected words e.g., if user query is 'professors' and ontological label or resource tag is 'professor' then morphological root for both elements make the comparison successful. The POS tagging helps in making rules for consideration and concatenation of words for an ontological resource. e.g., if the user query is "assistant professors and lecturers" then the equivalent tag for the query is "JJ NN CC NN". JJ NN words can be combined and searched in the ontology for a possible single match as in this case it is matched to the label of a single resource class 'AssistantProf'. All these shallow linguistic processing steps are helpful for the matching component.

Mapping query terms to ontological resources: Matching of user terms to ontological resources is performed. Single or a combination of user terms can be mapped to single or multiple ontological resources in O_R. We first try to match multiple terms from the user query to map to a single ontological resource. This process select terms based on some tagging combination; e.g., "assistant professor" have tag pattern "JJ NN" can be considered for a try. If a group of terms map to a single ontological resource, the

```
Template = Prefix + "Select "+ select_vars + " where  {" + relation_triples
+ " " + class_association_triples + " " + property_specification_triples +
" " + individual_specification_filter + " " + Quantifier_restriction_filter +
" " + label_triples + " } "
```

Fig. 3: Variable template used in QuriOnto

terms under the group are considered as covered. If a term is completely matched with an ontological resource, this is considered as the best match for that user term and other options which partly cover the term(s) are discarded. We term it a maximum match principle. The terms are stored with their ontological resource identifiers and category in a structure discussed next.

Query pattern processing: This is the main phase of QuriOnto. In this phase, the categories marked by the previous phases are combined as a string of pattern and the algorithm working on pattern decides the candidates for elements of variable query template. The variable template of QuriOnto is shown in Fig. 3.

For storing all the necessary information required for formal query generation, QuriOnto uses a list of lists of strings, termed as a query structure for reference. Each row of query structure corresponds to the details regarding a single ontological resource identified in the mapping phase.

Instead of defining fixed templates as in SemSearch, we identified six primitives or constituents for the where clause of SPARQL. Each constituent is either a triple or a filter, listed as follows:

- **Relation triple:** This type of triple is used to determine the connection between classes. For example, the triple '{?i1 ?p1 ?i2}' indicates that individuals represented by ?i1 are connected to individuals represented by ?i2 with the help of property represented by ?p1. We require the exploration of the relation graphs to determine the relation triple
- **Class association triple:** This type of triple associates a variable to the ontology class resource to access the individuals of the class. For example, in triple '{?i1 rdf:type :C1}' or '{?i1 a :C1}', the variable ?c1 is used to represent the individuals of the ontology class C1
- **Property specification triple:** It is used when a property resource is determined from mapping. For example, in the triple '{?i1 :supervises ?i2}' ?i1 and ?i2 act for domain and range individuals for which the object property 'supervises' is defined. Or in triple,

'{?i1 :has salary ?pvar}' ?pvar is used for accessing the value stored in the datatype property range for individuals ?i1

- **Individual specification filter:** This filter is used to restrict the class individuals to a particular individual or individuals. It is used when an individual is found from mapping. For example 'filter (?i1= :I1)', where ?i1 individuals is restricted to individual I1
- **Quantifier restriction filter:** The quantifier restriction is implemented using a filter expression to put a restriction on data type property values. For example, 'filter (?salary >= 20000)'. The operator of this filter is determined on the basis of phrases discussed in section of Quantitative phrase processing
- **Label triple:** It is used to access the linguistic names associated with an ontology resource. For example, '{:AssistantProf rdfs:label ?li1 }' or '{ ?p1 rdfs:label ?lp1}'

The algorithms listed below compute all the necessary information required for the generation of the triples and filters for the final generation of the SPARQL query.

The pattern_processing algorithm

```
Query structure= Resource Category + Resource term + filter operator and
                 filter value for datatype property
if (one (object property or datatype Property) in Query structure){
      assign arbitrary labels for domain and range classes;
      return;// to generate SPARQL
  }
for (individual in Query structure)
    find and store the class of Individual in Query structure;
if (more than one class in Query structure){
      find_Class_Relation for all distinct classes;
      if (not (successPath or SuccessConnection)) {
                check and merge common category classes;
                if (classes do not belong to same category)
                      print "No relation between concepts exists";
                      return (-1); }
      else {
            select one from PathList and ConnectionList which
                      has minimum edge count;
                      //if equal prefer PathList
            store in FinalPath; }
   }
check_class_for_instance;
for (each edge in the FinalPath) {
      insert a property row in the Query structure with
              property marked as variable;
       for (object property in Query structure)
       {
         fetch property statements from PO-C-C;
         find appropriate property row by matching
             domain and range with PO-C-C triple;
         if (domain and range matched in Query structure) {
             replace property variable by object property
              for that row;
             delete the object property row from Query structure;}
         }
      search domain class in Query structure and
              replace with class variable;
      search range class in Query structure and
              replace with class variable;
   }
for (datatype property in Query structure) {
      fetch property statements from PD-C;
      find appropriate class in Query Structure
      replace class with class variable;      }
```

Find_class_relation algorithm

```
Create subgraph for the classes in Query structure;
//select all triples that contain any of the class as domain or range
put all classes in class_list;
consider the class columns of PO-C-C graph;
while ( the length of class_list)
  {
place one class in Ci;
        put all other in remaining_class_list;
        Boolean Connection possible=true;

        for (the length of remaining_class_list)
        {
            put current indexed class from remaining_class_list in Cj;
            path=calculate shortest path from Ci to Cj;
            if (path is null) //means no connection between Ci and Cj
                Connection possible=false; //no need to check the remaining
                        //nodes for connection path
      if (path is not null)
             if path contains all classes of class_list
              {
                  place the path in PathList;
                  if (edges count = class_list_length – 1)
                      return with successPath=1; //optimal solution found
              }
            else if (Connection possible)
                  place path in temp Connection;
      } //end of for the length of remaining class
      if (Connection possible) {
            place temp Connection in ConnectionList;
             increment index of ConnectionList;
             if (number of edges in temp Connection = class_list_length)
                  return with success Connection=1;
      }
 initialize temp Connection;
 increment index in remaining_list;
}
```

Check_class_for_instance algorithm

```
while (class C as first element in Query structure)
        if (no instances found for C)
            substitute C with subclasses having instances;
```

DESCRIPTION OF THE ALGORITHM

Now, we explain the algorithm by dividing it into two major cases. The first case is the single resource case where the user query is mapped to a single ontological resource in O_R and the other one is when the query is mapped to multiple resources in O_R.

Single resource case: In this case, the resource can be any of the four, a class, an individual, a datatype property or an object property.

- **Case 1a:** Identified resource is a class
- **Example query#1:** BS students

In this case, these two query words is converted to a single ontological resource by keywords to mapping resource module and identified as a class. Class BSstudents is checked for instances, if found the instances are displayed otherwise the class is replaced by subclasses having instances.

- **Case 1b:** Identified resource is an individual
- **Example query#2:** Noman Hasany

In this case, all the object and datatype property values related to the individual are displayed.

- Case 1c: Identified resource is an object property or datatype property

This pattern alone is rarest to happen but we have the provision for it in QuriOnto. If the query term leads to a datatype or object property, it means the user wants to see the domain and range for which the property is defined. Often the terms(s) resulted in object property are defined in the domain or in the range of the object property. For example, for the user query "Departments" can result in the class 'Department' or object property 'has department'. In such a case, where this sort of ambiguity occurs, QuriOnto treats it a class and case 1a is applied.

- **Example query#3:** Recruiter (in job KB of Mooney data)

Here, is an issue. If some similarity technique is implied in the system and it finds company as a synonym for 'recruiter' and reported it as a class resource, which is the range of 'has recruiter' in the job ontology of Mooney data then the results are the instances of the companies. Usually, object property occurs in multiple resource case that will be discussed later.

- **Example query#4:** Salaries

The datatype property pattern alone can be a case that has a little bit higher chance than object property but lesser than C and I. The terms resulted to datatype property have lesser chances to be found in the domain or range of the triples. For example, Salaries become salary after Morphological analysis and results in datatype property 'has salary'. May be here the user wants to see the salaries of the employees, so salaries with respect to each individual for which the property is defined are displayed.

Both types of properties are considered in a single case because the resultant SPARQL for them is:

Select ?i1 ?lprop ?i2 where {?i1 :Prop ?i2. :Prop rdfs:label ?lprop}

Multiple resources case: The main processing of QuriOnto lies in the exploration of relations for the classes, directly or indirectly referred in the query. As finding the relation between classes involves at least two classes, it is discussed in the multiple resource case. Here we explain the process mentioned in the find_Class_Relation pseudo code.

- **Case 2:** All identified resources are classes
- **Case 2a:** When the classes are linked
- **Example query#5:** Teaching staff course semester

Output: The query means that the teaching staff taking a course in the semester be displayed.

The semantics are supported by ontology that the classes are found interconnected. To explore whether or not they have relations among them, QuriOnto implies three scenarios.

Scenario 1: Whether the nodes lies on a path. As for the example query#5, Fig. 4 shows the segment of the ontology where the three classes are connected on a path (Teaching staff, Course), (Course, Semester).

Scenario 2: Whether there is a node among them which has connection with other nodes. Suppose there is no back edge from Teaching staff to Course, then the example query#5 still has a meaningful interpretation in the ontology. In QuriOnto, we term it as a connection. So, the connection will be (Course, Teaching staff), (Course, Semester). Contrary to it, suppose the user enters example query#4.

Scenario 3: When the mentioned classes have indirect relations among them.

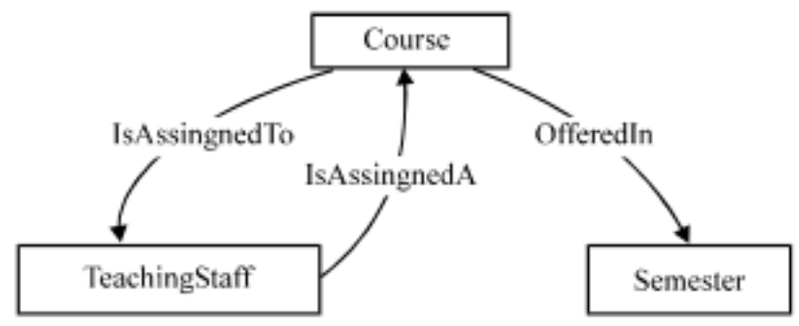

Fig. 4: Subgraph from the university ontology showing the accessibility of all classes from Teaching staff and Course

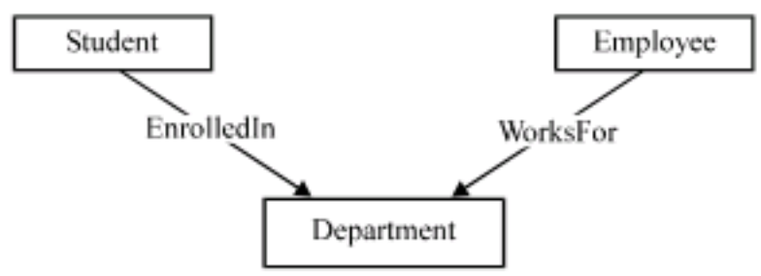

Fig. 5: Subgraph from University ontology showing no connection between student and employee

- **Example query#6:** Teaching staff, semester

Figure 4 shows that Teaching_Staff and Semester have an indirect relation via course class. When Dijkstra shortest path is applied it explores the intermediate nodes on the path and corresponding relation triples are formed. This is because the user is not familiar with the underline ontology model, so QuriOnto explores the indirect relations also. The situation contrary to it is explained for query example#7.

- **Example query#7:** Student, employee

Neither there is a direct link between student and employee, nor there is a class connecting them as shown in Fig. 5. Such a query will be considered as meaningless in QuriOnto as it does not find a path or a connection. A connection is established if there is a node whose outgoing edges provide links for the classes.

These scenarios not only help QuriOnto to check whether the query is meaningful, but it provides the edges that are used to form the relation triples for the formal query. In SemSearch, such condition is handled by making combinations; out of which may be one is successful.

- **Case 2b:** When the classes are subclasses of a common class
- **Example query#8:** Associate professors and professors

A case when the mentioned classes have no link between them can have a possible interpretation that the user wants to see the instances of the classes. But in this case it is necessary that the classes must have some common semantic category. As currently, QuriOnto has no semantic similarity measures, another technique is used. In future, this technique can be combined with semantic similarity procedure to make it more effective. QuriOnto has two conditions for this case to save the user from any misinterpretation of the results.

Condition#1: There must be a class that implies Is-A relation with the mapped classes and the class must be other than owl: Thing.

Condition#2: The mentioned classes have some common concept word in their labels or they must possess instances. Otherwise, the user has to enter the words separately to see the individuals, as mentioned in case 1a.

After the relations between classes are determined, the check_class_for_instance algorithm ensures that the classes going to be used in the formal query must possess instances. If a class having no individuals is mentioned in the formal query, nothing is retuned by the query engine. To overcome this problem, the main class reference is replaced by subclasses which have instances. Sometimes the user mentions the general class which differs from the ontological arrangement e.g., the user uses the term 'teaching staff' in the query which has professors, associate professors, senior lecturers etc. as subclasses categorizing individuals. QuriOnto checks if the class has no direct instances, it is replaced by union of all its subclasses that have individuals. For example, if the class association triple is {?c1 rdf:type :Teaching staff} then it is replaced by the union of its subclasses i.e., {{?c1 rdf:type :Lecturer} union {?c1 rdf:type :AssistantProf} union {?c1 rdf:type :Professor}}

- **Case 3:** Individuals with classes
- **Example query#9:** Ph.D. students of Computer Engineering

Individuals referred in the query are stored in the query structure as individuals with their immediate parent class. The find relation method is called after the individuals are mapped to the parent classes as mentioned in the Pseudo code.

- **Case 4:** Object property with classes and/or individual

The find_Class_Relation helps in formulating the relation triples with a property variable between the classes. But if a property is mapped from the user query term(s), it specializes the relation. In that case, the general relation triple having a property variable in between is replaced by the property resource. To ensure that the object property is applied on the appropriate relation triple, the related property statement are fetched from the PO-C-C graph and the domain and range classes are checked in the query structure. The matched general property row is updated and object property row is removed from the query structure.

- **Case 5:** Datatype property with other resources

Same procedure is applied for the datatype property using the PD-C graph where only the domain may need substitution with appropriate class variable on which the property is defined.

SPARQL GENERATION

This phase is responsible for determining the select variables and triples and filters for the where clause from the query structure for the variable template. As all the necessary information is computed by the pattern processing phase, the SPARQL generation is straight forward, as shown in SPARQL-Generation algorithm.

SPARQL-Generation algorithm

```
for each class resource
    form-class-triples;
for each individual
    form-individual-filter;
for each edge in FinalPath
    {form-Object-property-triples;
     add-Select-vars-with-order;}
for each datatype-property d
    {
    form-datatype-property-triples;
    if operator associated with d
     form-filter-expression;
    append-in-Select-vars;
    }
form each resource in Query structure
    {
    form-label-triples;
    add-select-vars;
    }
substitute select variables, triples and filters
    //in the query template
```

The illustrative example below explains the working of both the query pattern processing and SPARQL generation phases.

An illustrative example: Here, we explain the process by taking a query that contains all ontological resources with quantifier restrictions. Consider the user query is:

- Students computer engineering electronic engineering enrolled in MS program age below 24, GPa above or equal to 3

After query terms to resource mapping phase, the following are the identified ontological resources for the user query of Fig. 5:

Class: Student
Individuals: Computer engineering, Electronic engineering, MS_program
Object property: EnrolledIn
Datatype property: has Age, has GPA
Filter targets: <24, ≥3

The filter targets are stored with the respective datatype properties.

Find class for individuals: Three individuals are found in the query, ':Computer Eng.' and ':Electronic Eng.' are mapped to class 'Teaching Department' and ':MS_program' is mapped to ':Academic program'.

Find class relation: This function takes the three classes as input and results the edges depicting the relation between them. For this function, we use the heuristic to start the search from the class which is mentioned first in the query as it may provide the focus of the query. In the above case, student class is found to have relations with the two classes. The edges returned are: (Student, Teaching department), (Student, Academic program).

On returning from class relations, two property rows are inserted in the Query structure with variables ?p1 and ? p2.

?p1 ?i1 ?i2
?p2 ?i1 ?i3

As there is an object property 'enrolledIn', the Pattern_Processing method retrieves all the triples for the property from the PO-C-C table, among which the domain and range of the triple "Student enrolledIn Academic program" is found in the Query structure. The method retrieves the variables associated with 'Student' and 'Academic program' classes which are found to be ?i1 and ?i3, so, the general property ?p2 is replaced by 'enrolledIn'.

The entries in Query structure now contain:

:EnrolledIn ?i1 ?i3
?p1 ?i1 ?i2

This method checks the domain classes for the datatype variables, for example ':hasAge' and ':has GPA' both has domain class ':Student', which has variable ?i1 assigned. For range of datatype new variables are assigned per datatype property. so the triples formed are:

{:has Age ?i1 ?d1}
{:has GPA ?i1 ?d2}

Check_class_for_instance: Student class has no direct instances. The method searches for subclasses with instances. In the first iteration it searched BSstudent and Graduate student. BSstudent got instances while Graduate student is further divided into MSstudent and Ph.Dstudent which have instances. So, the student class is replaced by BSstudent, MSstudent and Ph.Dstudent. Classes 'Teaching department' and 'Academic program' are not checked for instances as they are mapped from individuals.

Form_class_triples: As the Query Structure contains three main classes, three class association triples are formed.

{ {?i1 a :BSstudent} union {?i1 a :MSstudent} union {?i1 a :PhDstudent } }
{?i2 a Teaching department}
{?i3 a Academic program}

Form_individual_filters: As Computer engineering and Electronic engineering belong to the same class Teaching department, they are combined using OR operator in a single filter.

Filter(?i2= :Computer engineering || ?i2= :Electronic Engineering)
Filter(?i3 = :MSprogram)

Form_ObjectProperty_triples: The triples arrangement from property rows are:

?i1 :enrolledIn ?i3
?i1 ?p1?i2

Form_DataTypeProperty_triples:

?i1 :has Age ?d1
?i1 :hasGPA ?d2

Form_filter_exprs:

Filter (?d1 < 20)
Filter (?d2 >=3)

Form_label_triples: Labels are created for each class variable, object property, datatype property using rdfs:label. For each specific property a variable is created in select vars.

The class relation triples and object triples are used to decide the arrangement of variables but the datatype variables are placed in the last.

```
SELECT ?i1 ?li1 ?lp1 ?i2 ?li2 ?lpo1 ?i3 ?li3 ?ld1 ?d1 ?ld2 ?d2 WHERE
{ { {?i1 a p1:BS_student} union

{?i1 a p1:MSstudent} union {?i1 a p1:PhDstudent} } ?i2 a p1:Teaching
Department.
?i3 a p1:AcademicProgram. filte r(?i2= p1:ComputerEngg || ?i2= p1:
ElectronicEngg).
?i1 p1:enrolled_In ?i3.?i1 ?p1 ?i2. p1:has Age ?d1. ?i1 p1:has GPA ?d2.
filter (?d1 > 20). filter (?d2 >= 3). ?i1 rdfs:label ?li1. ?i2 rdfs:label ?l12.
?i3 rdfs:label ?li3. ?p1 rdfs:label ?lp1. p1:enrolled_In rdfs:label ?lpo1.
p1:has GPA rdfs:label ?ld2. ?i1 p1:has Age rdfs:label ?ld1. }
```

Fig. 6: Query generated by QuriOnto for the user query mentioned earlier

?i1 ?li1 ?lp1 ?i2 ?li2 ?lop1 ?i3 ?li3 ?ld1 ?d1 ?ld2 ?d2

The final query submitted to the SPARQL query engine by QuriOnto is shown in Fig. 6.

IMPLEMENTATION

For accessing the ontology and for SPARQL execution, we used the Jena API for RDF. As Jena is a Java API, we used Java to program our prototype. For the design and testing of QuriOnto, we extended the University ontology to incorporate courses, projects and student data, using the Protégé GUI. The GATE morphological analyzer is used in the shallow linguistic processing phase.

RESULTS

We have tested present approach on two different domain ontologies. The first ontology is extended from the university ontology available as OWL test data (http://www.ifi.uzh.ch/ddis/research/ semweb/talking-to-the-semantic-web/owl-test-data/), we call it extended University Ontology, which consists of 34 classes, 89 individuals, 20 object properties and 9 datatype properties. The second ontology we selected is the job ontology constructed for the Mooney job data (Tang and Mooney, 2001), used in the evaluation of many NLIs to ontologies; consisting of 8 classes, 218 individuals, 7 object properties and 12 datatype properties. We have not made any changes to the job ontology except for the rdfs labels which are defined for each ontological resource. While naming the labels, we use the resource names and added some expansions for abbreviations.

We have prepared 56 questions for university ontology out of which 18 questions were separated for testing and 38 were used during design. For the job

Table 2: Results obtained after running queries with QuriOnto for two different ontologies

Ontology	Total queries	Answers returned	Answer not returned	Answer correct	Answer partially correct	Answer incorrect
University extended	18	14	4	13	0	1
Job	60	46	14	38	3	5

ontology, 60 questions were selected from the job queries excluding the negation questions and duplicated questions. So, after the noise removal phase, all questions were distinct. Table 2 has shown the results when the queries are submitted to our QuriOnto prototype.

DISCUSSION

More number of correct answers from the extended university ontology is due to the fact that we had formulated the questions while designing the QuriOnto. The questions did not pose any similarity failures. The failure and 'not answered' cases were due to many duplicate entries for 'MS' in the ontology e.g., MS_Program, MS, ME, MSstudent, all have the word masters in the label. Due to strictness of maximum match principle, the word 'ms' is always matched with 'MS', without considering other options.

The queries that are failed to return correct answers or partially correct answers are due to one of the fact that keywords queries lack in understanding complex and dependent linguistic structures. The 'answers not returned' is due to two reasons:

- QuriOnto similarity scope is only limited to rdfs labels
- QuriOnto has heuristics and strict rules to eliminate multiple options for user words, which can be overcome by ranking

Without considering the partially correct answers, the average precision of 0.87 of the two ontologies with individual precisions of 0.92 and 0.82 for university and job ontology respectively shows that if the user words are correctly mapped to ontological resources and correct resources are selected then the QuriOnto approach for the generation of correct SPARQL queries is quite promising.

The idea of NLIs to ontologies is carry-forwarded from NLIs to databases in which the users submit a natural language query instead of a formal database query to access the information. Some of the systems adapted the techniques used in DB NLIs for NLIs to ontologies (Minack *et al*., 2008; Zhou *et al*., 2007). One of the major advantages that ontologies possess over databases is that they have a linguistically rich structure that makes them more attractive for NLIs.

The NLIs to ontologies developed so far can be distinguished from many points of views. One of the major distinguishing features is the type of input they accept. The first category is the one that accepts the syntactic input and parse the user question to find out the RDF statement matches for the question. Some systems for this category are ONLI+ (Mithun *et al*., 2007) and Panto (Wang *et al*., 2007).

ONLI+ (Mithun *et al*., 2007) has handled quantifiers and number restrictions for syntactically correct sentences while QuriOnto approach is for query keywords. ONLI+ reported some sentences that were not converted to formal query due to the incorrect parse generated by Minipar. Present approach works in all situations whether the user write correct sentences or enter keywords with number restrictions and quantifier restrictions.

PANTO is a remarkable system as it covers most of the SPARQL functionalities for the user query with a recall and precision of 86.12 and 89.17% for the job ontology (Wang *et al*., 2007). The syntactic questions are beneficial as they give extra information to understand dependencies and other linguistic features. But on the other side, parsing lengthy questions can be time consuming and such type of input cannot be restricted for search engines.

QuriOnto prototype introduced in this study is based on the keyword input. The recent systems based on keywords are SemSearch (Lei *et al*., 2006), Spark (Zhou *et al*., 2007), QuestIO (Tablan *et al*., 2008; Damljanovic *et al*., 2008), Q2Semantic (Wang *et al*., 2008) and an approach by Tran *et al*. (2007). SemSearch (Lei *et al*., 2006) has introduced fixed templates of formal query. The cases are discussed for two resources with a suggestion of rules to combat more than two resource cases. With the increasing length of user query, there will be requirements for many templates out of which one will be successful for the user query. QuriOnto has a variable template and elements of which are computed by an interpretation process based on the resources explored from the user query. SemSearch (Lei *et al*., 2006) restricts the user to separate subject and other keywords with a colon. QuriOnto has no input format restriction user neither the user is required to give any additional information regarding the terms used. Unlike SemSearch, QuriOnto variable template also contains the provision for filters to accommodate quantifier restriction from user query.

QuestIO dealt with the unrestricted natural language (Tablan *et al*., 2008; Damljanovic *et al*., 2008). Its main focus is in determining the most appropriate relations between the classes. QuestIO uses other closed category

language words to disambiguate among different identified relations to find out the most appropriate one. QuestIO has not discussed the situation in which a class does not possess instances. QuestIO tries to find out a property when there are two classes while QuriOnto assumes a variable for property when missing or sometimes no property is specified as the user wants to see the list of instances. It means QuriOnto template has a more general structure to accommodate a lot of user queries. QuriOnto considers intermediate classes to build the link even if they are not mentioned in the user query. Also quantifier restrictions are not handled in QuestIO.

Some complete graph based processing approaches (Tran *et al*., 2007; Zhou *et al*., 2007; Wang *et al*., 2008) are introduced. These approaches construct a graph for the keywords mapped resources and then convert the graph to the formal query. As constructing and manipulating the complete graph for all resources is time consuming, Spark has reported best performance for 2 to 6 keywords query. Tran *et al*. (2007) places limit on the depth of the graph to avoid complexity. Q2Semantic has presented a time effective approach over the two by first trying to obtain a subgraph for the mapped resources, apply a process to compact it and then perform rest of the actions on that compact graph (Wang *et al*., 2008). But Q2Semantic accepts certain ontological resources i.e., classes and literals from the user input for processing so the real complexity is unknown until a solution incorporating all options from the user query have been considered. QuriOnto technique is a combination of both graph and template. The graphs in QuriOnto are based on classes and properties which are not as large as the whole ontology graph and which do not change as frequently as individuals and literals do. QuriOnto deals with all ontological resources for any number of terms in the user query. QuriOnto offers better coverage for SPARQL as SPARQL filters are supported for quantifier restrictions.

CONCLUSION AND FUTURE WORK

In this study, we present an approach for converting keyword queries to formal ontology queries that uses property and relation graphs only with a variable template. The property graphs reduce the RDF graph scope while the variable template can accommodate any number of concepts from the user query. The SPARQL coverage by QuriOnto is more as compared to the existing keyword based systems as it can handle quantifier restriction also.

The similarity issues of QuriOnto are currently limited to the ontology terms and to the terms defined using rdfs labels. The incorporation of semantic matching capability before the terminology mapping phase can result in an increase in recall. The future work lies in discovering similarity issues for both domain dependent ontology terms and general linguistic terms for QuriOnto.

PANTO has shown handling of negation with syntactic approach (Wang *et al*., 2007). Handling negation and other linguistic features with keywords input is a potential future work.

ACKNOWLEDGMENT

This research is supported by eScience Fund SF0704, Ministry of Science Technology and Innovation, Malaysia.

REFERENCES

Damljanovic, D., V. Tablan and K. Bontcheva, 2008. A text-based query interface to owl ontologies. Proceedings of the 6th International Language Resources and Evaluation, May 2008, Marrakech, Morocco, ELRA, pp: 1-8.

Kaufmann, E., A. Bernstein and R. Zumstein, 2006. Querix: A natural language interface to query ontologies based on clarification dialogs. Proceedings of the 5th International Semantic Web Conference, November 2006, Springer, Athens, GA., pp: 980-981.

Kaufmann, E., A. Bernstein and L. Fischer, 2007. Nlp-reduce: A naive but domain independent natural language interface for querying ontologies. Proceedings of the 4th European Semantic Web Conference, June 2007, Innsbruck, Austria, pp: 1-3.

Lei, Y., V. Uren and E. Motta, 2006. SemSearch: A Search Engine for the Semantic Web. In: Managing Knowledge in a World of Networks, Staab, S. and V. Svatek (Eds.). LNAI 4248, Springer Verlag, Berlin Heidelberg, ISBN-13: 978-3-540-46363-4 pp: 238-245.

Minack, E., W. Siberski, G. Zenz and X. Zhou, 2008. SUITS4RDF: Incremental query construction for the semantic web. Proceedings of the International Semantic Web Conference (Posters and Demos), 2008. http://www.l3s.de/web/page25g.do?kcond12g.att1=1116.

Mithun, S., L. Kosseim and V. Haarslev, 2007. Resolving quantifier and number restriction to question OWL ontologies. Proceedings of the 3rd International Conference on Semantics, Knowledge and Grid, Oct. 29-31, IEEE Computer Society, Washington, DC., pp: 218-223.

Prud, E. and A. Seaborne, 2006. SPARQL query language for RDF. W3C Working Draft. http://www.w3.org/TR/2006/WD-rdf-sparql-query-20060220/.

Ramachandran, V.A. and I. Krishnamurthi, 2009. NLION: Natural language interface for querying ontologies. Proceedings of the Bangalore Compute Conference, Bangalore, India, ACM. http://doi.acm.org/10.1145/1517303.1517322.

Tablan, V., D. Damljanovic and K. Bontcheva, 2008. A Natural Language Query Interface to Structured Information. In: The Semantic Web: Research and Applications, LNCS, Bechhofer, S. (Ed.). Springer, Berlin/Heidelberg, pp: 361-375.

Tang, L.R. and R.J. Mooney, 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. Proceedings of the 12th European Conference on Machine Learning, (ECML'01), Springer-Verlag, London, UK., pp: 466-477.

Tran, T., P. Cimiano, S. Rudolph and R. Studer, 2007. Ontology-Based Interpretation of Keywords for Semantic Search. LNCS, Springer, Berlin/Heidelberg, pp: 523-536.

Wang, C., M. Xiong, Q. Zhou and Y. Yu, 2007. PANTO: A portable natural language interface to ontologies. Proceedings of the 4th European Conference on the Semantic Web, (ECSW'07), Springer-Verlag, Berlin, Heidelberg, pp: 473-487.

Wang, H., K. Zhang, Q. Liu, T. Tran and Y. Yu, 2008. Qisemantic: A lightweight keyword interface to semantic search. Proceedings of the European Semantic Web Conference, (ESWC'08), UK., pp: 584-598.

Zhou, Q., C. Wang, M. Xiong, H. Wang and Y. Yu, 2007. Spark: Adapting keyword query to semantic search. Proceedings of the Semantic Web, LNCS, (SWL'07), Springer, Berlin/Heidelberg, pp: 694-707.