Q2Semantic: A Lightweight Keyword Interface to Semantic Search

Haofen Wang[1], Kang Zhang[1], Qiaoling Liu[1], Thanh Tran[2], and Yong Yu[1]

[1] Department of Computer Science & Engineering
Shanghai Jiao Tong University, Shanghai, 200240, China
{whfcarter,jobo,lql,yyu}@apex.sjtu.edu.cn
[2] Institute AIFB, Universität Karlsruhe, Germany
{dtr}@aifb.uni-karlsruhe.de

Abstract. The increasing amount of data on the Semantic Web offers opportunities for semantic search. However, formal query hinders the casual users in expressing their information need as they might be not familiar with the query's syntax or the underlying ontology. Because keyword interfaces are easier to handle for casual users, many approaches aim to translate keywords to formal queries. However, these approaches yet feature only very basic query ranking and do not scale to large repositories. We tackle the scalability problem by proposing a novel clustered-graph structure that corresponds to only a summary of the original ontology. The so reduced data space is then used in the exploration for the computation of top-k queries. Additionally, we adopt several mechanisms for query ranking, which can consider many factors such as the query length, the relevance of ontology elements w.r.t. the query and the importance of ontology elements. The experimental results performed against our implemented system Q2Semantic show that we achieve good performance on many datasets of different sizes.

1 Introduction

The Semantic Web can be seen as an ever growing web of structured and interlinked data. Examples for large repositories of such data available in RDF are DBpedia[1], TAP[2] and DBLP[3]. A snippet of RDF data contained in TAP is shown in Fig. 1. It describes the entity SVGMobile (a W3CSpecification) in terms of its relations to the other entities and its attribute values.

The increasing availability of this semantic data offers opportunities for semantic search engines, which can support more expressive queries that address complex information needs [1]. However, query interfaces in current semantic search engines [2, 3] only support formal queries e.g. SPARQL[4]. For example, when a person wants to find specifications about "SVG" whose author's name

[1] http://dbpedia.org
[2] http://tap.stanford.edu
[3] http://dblp.uni-trier.de/
[4] http:// www.w3.org/TR/rdf-sparql-query

```
<rdf:Description rdf:about="SVGMobile">
<rdf:type> W3CSpecification</rdf:type>
<tap:hasAuthor rdf:resource="Capin,_Tolga"/>
<rdfs:label xml:lang="en">Mobile SVG Profiles: SVG Tiny and SVG Basic</rdfs:label>
</rdf:Description>
```

```
PREFIX tap: <http://tap.stanford.edu/tap#>
SELECT ?spec
WHERE {
    ?spec tap:hasAuthor ?person.
    ?spec tap:label "SVG".
    ?person tap:name "Capin".
}
```

Fig. 1. a) Sample RDF snippet. b) Sample SPARQL query.

is "Capin", he needs to type in the SPARQL query shown in Fig.1. The user thus needs to learn the complex syntax of the formal query. Moreover, the user also needs to know the underlying schema and the literals expressed in the RDF data.

Keyword interfaces is one solution to this problem. User's are very familiar with these interfaces due to their widespread usage. Compared with formal queries, keyword queries have the following advantages: (1) *Simple Syntax*: they are simply lists of keyword phrases (2) *Open Vocabularies*: the users can use their own words when expressing their information needs. In the above example, the user would have to type in only "Capin" and "SVG".

Since keyword interfaces seem to be suitable for casual users, many studies have been carried out to bridge the gap between keyword queries and formal queries, notably in the information retrieval and database communities [4–7]. There also exist approaches that specifically deal with keywords interfaces for semantic search engines. The template-based approach discussed in [8] fixes the possible interpretations and thus, cannot always capture the meaning intended by the users. This problem has been tackled recently by [9, 10]. In [10], a more generic graph-based approach has been proposed to explore the connections between nodes that correspond to keywords in the query. This way, all interpretations that can be derived from the underlying RDF graph can be computed.

However, three main challenges still remain: (1) How to deal with keyword phrases which are expressed in the user's own words which do not appear in the RDF data? (2) How to find the relevant query when keywords are ambiguous (ranking)? For instance, [10] exploits only the query length for ranking. (3) How to return the relevant queries as quickly as possible (scalability)? Both [9, 10] require the exploration of a possibly large amount of RDF data, and thus, cannot efficiently deal with large repositories.

In this paper, we address the above challenges as follows:

- (1) We leverage terms extracted from Wikipedia to enrich literals described in the original RDF data. This way, users need not use keywords that exactly match the RDF data.
- (2) We adopt several mechanisms for query ranking, which can consider many relevant factors such as the query length, the relevance of ontology elements w.r.t. the query as well as the importance of ontology elements.
- (3) We propose an exploration algorithm and a novel graph data structure called clustered graph, which represents only a summary of the original RDF

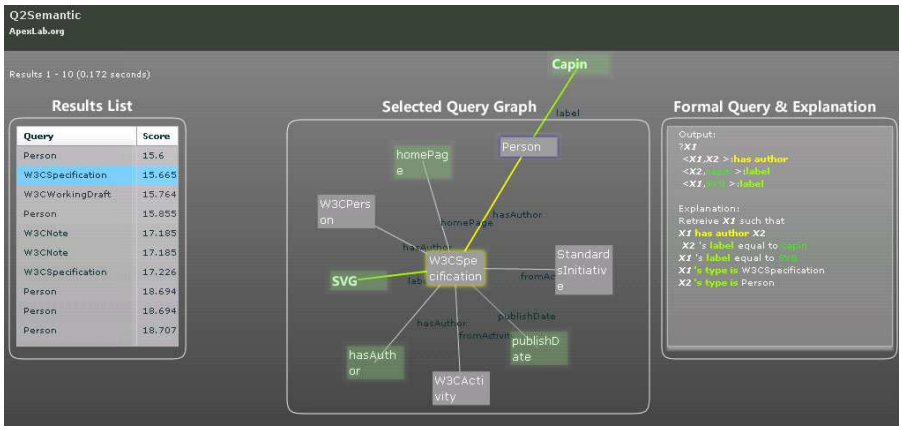

Fig. 2. The result view of Q2Semantic.

data. This improves scalability particularly because the data space relevant for exploration becomes smaller in size. Additionally, the exploration algorithm also allows for the construction of the top-k queries, which can help to terminate the interpretation process more quickly.

Also, we have implemented a keyword interface called Q2Semantic to evaluate our approach. The experiments performed on several large datasets show that our solution achieves high effectiveness and efficiency.

The rest of the paper is organized as follows. We will start in section 2 with an overview of Q2Semantic. Section 3 shows how the underlying data models are preprocessed. Section 4 elaborates on how these models are used in the main steps involved in the translation process. Section 5 presents several mechanisms for query ranking. The experimental results are given in section 6. Section 7 contains information on related work. Finally, we conclude the paper with discussions of current limitations and future work in section 8.

2 Q2Semantic

2.1 Feature Overview of Q2Semantic

Q2Semantic is equipped with a keywords-based search interface. In order to facilitate usage, this interface supports auto-completion. This feature exploits the underlying RDF literals enriched with Wiki terms to assist the user in typing keywords. This is extended to "phrase completion" such that when the first keyword has been entered, Q2Semantic will automatically generate a list of phrases containing these keywords from which the user can choose from.

After submitting the keyword query, the user sees the results as shown in the screenshot of our AJAX interface in Fig. 2 (corresponds to our example query "Capin" and "SVG"). On the left, the query results are listed in an ascending

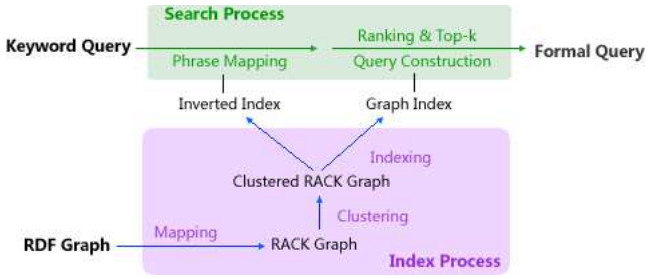

Fig. 3. Workflow of Q2Semantic.

order according to the ranking scores of their corresponding queries. For the selected result, the corresponding formal query and its natural language explanation are presented on the right. In the middle, the data space that is explored to compute the queries is visualized for the user to understand and explore the queries. For the selected query, the relevant path in this data space is highlighted (in yellow and green). The user explores the data by double clicking on a node to see (further) neighbors. These and other features such as query refinement can be tested at http://q2semantic.apexlab.org/UI.html.

2.2 Query Translation in Q2Semantic

Q2Semantic supports the translation of keyword queries to formal queries. In particular, a keyword query K is composed of keyword phrases $\{k_1, k_2, \cdots, k_n\}$. Each phrase k_i has correspondence (i.e. can be mapped) to literals contained in the underlying RDF graph. A formal query F can be represented as a tree of the form $\langle r, \{p_1, p_2, \cdots, p_n\}\rangle$, where r is the root node of F and p_i is a path in F, which starts from r and ends at leaf nodes that correspond to k_i. The root node of F represents the target variable of the query. So basically, we restrict our definition of formal queries to a particular type of tree-shaped conjunctive queries [11] where the leaf nodes correspond to keywords entered by the user. In our example, K includes k_1 = "Capin" and k_2 = "SVG", and $F = \langle r, \{p_1, p_2\}\rangle$, where r = W3CSpecification, $p_1 = \langle x1, label, SVG\rangle$ and $p_2 = \langle x1, hasAuthor, x2, name, Capin\rangle$. Since SPARQL is essentially, conjunctive query plus additional features, our formal query can be directly rewritten as triple patterns to obtain a SPARQL query like the one presented in section 1.

The translation process is illustrated in Fig. 3, which includes two main steps: (1) *Phrase Mapping*: Retrieve terms stored in an inverted index using the keyword phrases entered by the user (2) *Query Construction and Ranking*: Search the clustered graph to construct potential formal queries and assign costs to them. Meanwhile, top-k queries are returned based on the costs. Note that these online activities are performed on the inverted and the graph index. There is more pre-processing required to build these two data structures, including mapping, clustering and indexing. We will continue with a detailed elaboration

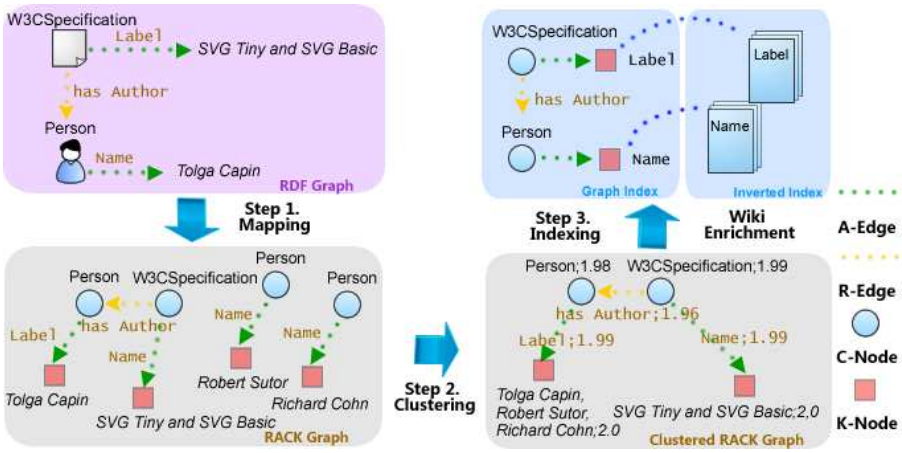

Fig. 4. Index process.

on these pre-processing steps and then, discuss the online activities required to translate the keywords.

3 Data Pre-Processing in Q2Semantic

3.1 Graph Construction via Mapping and Clustering

Graph exploration as done in other approaches is expensive due to the large size of the A-Box (RDF graph) [9, 10]. As observed in [12], similar instances always share similar attributes and relations. Adopting this idea, we propose a clustered RACK graph which corresponds to a summary of the original RDF graph. This reduction in size enables faster query construction and ranking especially for RDF graph containing a large number of instances. In the following, we will describe our notion of RACK graph and the rules for clustering.

A RACK graph consists of the four elements R-Edge, A-Edge, C-Node and K-Node, obtained from the original RDF graph through the following mappings:

- Every instance of the RDF graph is mapped to a *C-Node* labelled by the concept name that the instance belongs to.
- Every attribute value is mapped to a *K-Node* labelled by the value literal.
- Every relation is mapped to a *R-Edge* that is labelled by the relation name and connects two C-Nodes.
- Every attribute is mapped to an *A-Edge* that is labelled by the attribute name and connects a C-Node with a K-Node.

As shown in Fig. 4, the mapping process results in a RACK graph. Note that each instance is mapped to the most special concepts if it belongs to multiple concepts. We also do not consider any axioms (e.g. subsumption between

concepts) in the RACK graph as it does not support reasoning capability for query interpretation. A clustered RACK graph can be further obtained by the iterative application of the following four rules.

- Two C-Nodes are clustered to one if they have the same label.
- Two R-Edges are clustered to one if they have the same label and connect the same pair of C-Nodes.
- Two A-Edges are clustered to one if they have the same label and is connected to the same C-Node.
- Two K-Nodes are clustered to one if they are connected to the same A-Edge. The resulting node inherits the labels of both these K-Nodes.

For each clustered node, we track and store the number of original nodes that collapsed to it during the clustering. Also, for each clustered edge, we store the number of node pairs that were connected by the original edges collapsed to it. These numbers stored in nodes and edges are used to compute their costs on the basis of cost functions discussed in section 5. The costs are shown in Fig. 4. They will be used later in the construction and ranking of the query.

3.2 Clustered Graph Indexing

The clustered RACK graph computed in the previous step can be stored in a graph index as discussed in [13]. In our current experiments, we directly load the clustered RACK graph model into the memory for fast query construction since it is very small. However, the graph index will be used when the clustered graph is too big to be loaded into memory.

3.3 Phrase Indexing

We make use of an inverted index to store the labels of K-Nodes. This index is used to locate relevant K-Nodes for a given keyword phrase faster. In particular, we create a document for each K-Node and take its labels as the document content. This document is further enriched with terms extracted from Wikipedia.

This enrichment is performed to support keywords that are expressed in the user's own words that do not match the literals of RDF data. In fact, we adopt the idea in [14] to leverage Wikipedia. Namely, for each K-Node label, we search the Wikipedia database to see whether it matches the title of any article. If so, several semantic features of the article as introduced in [15] are added as additional labels of the K-Node. These features include the title, the anchor texts that link to the article, and the titles of other articles that redirect to the article. Therefore, user keywords might be mapped to the actual labels of the K-Nodes or any of these extracted features added to the K-Nodes.

4 Query Interpretation in Q2Semantic

The query interpretation begins with the mapping of user keywords to the labels of K-Nodes in the inverted index. Starting from the matched K-Nodes, an

exploration on the clustered graph is performed, which is similar to the single-level search algorithm discussed in [16]. It expands the current nodes to their neighbors iteratively until reaching a common root. In this process, the edge with the lowest cost is selected for traversal. The process terminates until the top-k queries have been found. In the following subsections, we will describe these steps in detail.

4.1 Phrase Mapping

Each keyword phrase k_i in K entered by the user is submitted as a query to the index, resulting in hits that represent the matching K-Nodes. They are returned in a ranked list as $KL_i = \{k\text{-}node_{i1}, k\text{-}node_{i2}, \cdots, k\text{-}node_{im_i}\}$, associated by the retrieval engine with the matching score $S_i = \{s_{i1}, s_{i2}, \cdots, s_{im_i}\}$. Each s_{ij} is used as the dynamic weight of the respective $k\text{-}node_{ij}$ with respect to k_i. For instance, KL_1 contains one K-Node that matches "Capin" while KL_2 contains three K-Nodes matching "SVG", as illustrated in Fig. 5.

4.2 Query Construction

After obtaining these K-Nodes, we construct the potential queries by exploring the clustered RACK graph. The process is as follows: For each keyword phrase, we create a thread. Then we do traversal in these threads until all the threads converge at a same node. This way, the traversal paths correspond to a tree, from which we construct a tree-shaped formal query. In the following, we first define the thread and the expansion operations required to traverse the graph. Then we will present the detailed algorithm.

A *thread* maintains cursors that haven't been expanded yet. A *cursor* is defined on a node, which traces the expansion track in a thread. Each cursor has four fields (c; n; p; k), where c represents the cost for the track, n is the node where the cursor locates in, p is the parent cursor of the current cursor, and k is the keyword phrase corresponding to the thread that the cursor is in. Note that all cursors in the same thread share the same keyword phrase.

Given a thread, a *thread expansion* (*T-Expansion*) selects a cursor in it, executes cursor expansion, and then removes the cursor from it. Given a cursor C_{cur}, a *cursor expansion* (*C-Expansion*) includes a validation step and an exploration step. In the validation, we check whether a new formal query rooted at the node $C_{cur}.n$ has been found. It is accomplished by checking whether cursors in other threads have arrived at this node. In the exploration, new cursors (e.g. C_{new}) are created for all neighbors of the node $C_{cur}.n$ and added to the current thread, i.e. $C_{new}.k = C_{cur}.k$. The current cursor then becomes parent cursor of these new cursors, i.e. $C_{new}.p = C_{cur}$. The costs of the new cursors are calculated using the formula $C_{new}.c = C_{cur}.c + dist(C_{cur}.n, C_{new}.n)$, where $dist()$ is a distance function between two nodes in the graph. By default, it is the cost of the edge which connects the two nodes.

The sequence of doing T-Expansions has an impact on the speed of query construction. This speed is also influenced by the sequence of C-Expansions

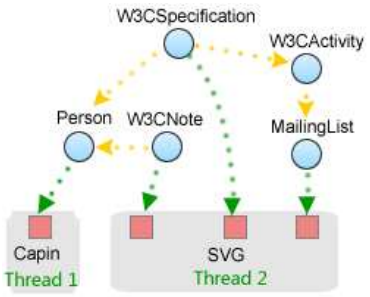

Fig. 5. Exploring the clustered graph.

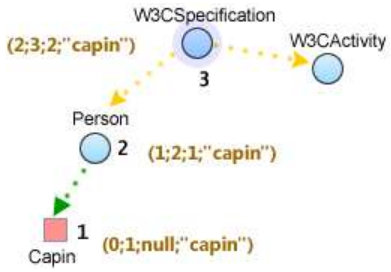

Fig. 6. Example on repeated expansion.

Input: $K = \{k_1, k_2, ..., k_n\}$, where k_i hits the K-Nodes
$KL_i = \{k\text{-}node_{i1}, k\text{-}node_{i2}, \cdots, k\text{-}node_{im_i}\}$ with the matching relevance as $S_i = \{s_{i1}, s_{i2}, \cdots, s_{im_i}\}$;
Output: A: result set, initially $\emptyset$;
Data: τ_{prune}: pruning threshold, initially τ_0;

```
1  for i ∈ [1, n] do
2  |  t_i = new Thread();
3  |  for j ∈ [1, m_i] do
4  |  |  t_i.add( new Cursor(s_ij, k-node_ij, NULL, k_i));
5  |  end
6  end
7  while ∃i ∈ [1, n] : t_i.peekCost() ≠ ∞ do
8  |  j ← pick from [1, n] in a round-robin fashion;
9  |  c ← t_j.popMin();
10 |  C-Expansion(c); // A and τ_prune will be updated here;
11 |  if t_j.peekCost() > τ_prune then
12 |  |  Output the top k answers in A;
13 |  end
14 end
```

Algorithm 1: Query Interpretation Process

performed during the T-Expansions. Considering that, we use the following two strategies when choosing what to expand next: 1) *Intra-Thread Strategy*: In a T-Expansion, we choose the cursor with the lowest cost for the next C-Expansion. 2) *Inter-Thread Strategy*: Within different threads, we choose the thread with the lowest number of expanded cursors for the next T-Expansion in order for a round robin fasion. These two strategies have been proven optimal in the single-level search algorithm [16].

This query construction process is described in Algorithm 1. We first initialize thread t_i for each keyword phrase k_i in K (Line 2), and fill t_i with cursors for the K-Nodes in KL_i (Line 4). Then we do T-Expansions on the threads in a round-robin fashion (Line 8). In each T-Expansion, we do C-Expansion on the cursor which has the lowest cost (Line 9). Note that for each thread, $popMin()$ pops out the cursor with the minimal cost, whereas $peekCost()$ just returns the

minimal cost. Line 11 to Line 13 is the optimization for top-k termination, which will be discussed in the next subsections.

As shown in Fig. 5, after the initialization, t_1's cursor locates in the K-Node labelled "Capin", and t_2's cursors point to three K-Nodes. When we expand the cursor in t_1 to the C-Node Person, and assuming cursors in t_2 have already reached this node (e.g. a cursor starts from "SVG", expands through W3CNote and reaches Person), we get a formal query rooted at Person. One path of the query is from Person to the K-Node labelled "Capin", and the other is from Person to the most left K-Node labelled "SVG".

4.3 Optimization for Top-k Termination

In order to find out the top-k queries only, we maintain a pruning threshold called τ_{prune}, which is the current kth minimal cost of the already computed queries. τ_{prune} will be initialized to τ_0. When we find a valid formal query in C-Expansion, the cost of the query is calculated by the ranking mechanism, which will be discussed in Section 5. For a new formal query to be in a top k position, its cost should be no greater than τ_{prune}. When such a query is found, it will be added to the answer set A and τ_{prune} will be updated accordingly. Since a cursor actually indicates a path in query, if all cursors' costs are larger than τ_{prune}, new queries including these paths will have even larger costs. Therefore, we can stop the query interpretation process and output the top-k formal queries.

4.4 Optimization for Repeated Expansion

We assume that it rarely happens for people to propose a query which contains the same relations several times (e.g. "find Tom's friends' friends' friend, who is Spanish"). Based on this assumption, we adopt a mechanism to avoid redundant exploration of the same elements, which can speed up the construction process. Namely, we add penalty to the cursor whose track contains repeated nodes. This is done by using a different $dist()$ function for C-Expansion, namely

$$dist^*(n_1, n_2) = \begin{cases} P & \text{If } n_2 \text{ has been visited} \\ dist(n_1, n_2) \end{cases} \quad (1)$$

where P is set to a large number as the predefined penalty parameter.

In Fig. 6, there is a cursor on W3CSpecification. Its track is indicated by 1, 2 and 3. Assume that the cost of the current cursor is two, every edge has one as weight, and P is set to five. Then the cost of the new cursor on W3CActivity gets three, while the one on Person gets seven as it has been visited already at 2. This way, repeated expansion on Person is still allowed but with a higher cost.

5 Query Ranking in Q2Semantic

Since the query construction process can result in many queries, i.e. possible interpretations of the keywords, a ranking scheme is required to return the queries

that most likely match the user intended meaning. Ranking has been dealt with in other approaches. For ranking ontologies, [17] returns the relevant ontologies based on the matching score of the keywords w.r.t. the ontology elements. It also considers the importance of nodes and edges in the ontology graph as a static score similar to Google's PageRank. For ranking complex relationships, [18, 19] employ the length of the relation paths. Besides these approaches for ranking ontology (answers) and relations, work has been done for ranking queries. [10] uses the length of the formal query and [9] considers also the keywords' matching score.

We define three ranking schemes from simple to complex, which adopt ideas from other approaches mentioned above, to extend existing work on ranking queries. They compute the cost for a query. The most complex scheme leverages all the above factors including the query length, the keyword matching score and the importance of nodes and edges.

Path Only: The basic ranking scheme R_1 considers the query length only, which is as follows:

$$R_1 = \sum_{1 \leq i \leq n} (\sum_{e \in p_i} 1) \tag{2}$$

This formula computes the total length of paths in the formal query, where p_i is a path and e is an edge in p_i. Each p_i represents a connection between the root of the formal query and a matched K-Node. Lower cost queries are preferred over higher cost queries. Since the cost of every edge is defaulted to exactly one, in effect, shorter queries are preferred over longer ones. As discussed in [10], shorter queries tend to capture stronger connections between keyword phrases.

Adding matching relevance: When further considering the matching distance between the user's keyword phrases and the literals in the RDF graph, a ranking scheme R_2 can be defined as

$$R_2 = \sum_{1 \leq i \leq n} (\frac{1}{D_i} \sum_{e \in p_i} 1) \tag{3}$$

where D_i is the score stored in the p_i's starting K-Node, which has been computed in the phrase mapping. In this case, R_2 prefers shorter queries with higher matching score of keyword phrases w.r.t. K-Nodes labels.

Adding Importance of Edges and Nodes: This ranking scheme assumes that users prefer to find entities with types and relations that are more "important". Ranking scheme R_3 considers also the importance of query elements. In particular, specific cost functions are defined for nodes and edges, which reflect their importance for the RDF graph. R_3 and these cost functions are defined as

$$R_3 = cost_r \sum_{1 \leq i \leq n} (\frac{1}{D_i} \sum_{e \in p_i} cost_e) \tag{4}$$

$$cost_{node} = 2 - \log_2(\frac{|node|}{N} + 1) \tag{5}$$

$$cost_{edge} = 2 - \log_2(\frac{|edge|}{M} + 1) \tag{6}$$

Table 1. Table of TAP sample queries

Query	Keywords	Potential information need
Q3	Supergirl	Who is called "supergirl"
Q5	Strip, Las Vegas	What is the well-known "Strip" in Las Vegas
Q9	Web Accessibility Initiative, www-rdf-perllib	Find persons who work for Web Accessibility Initiative and involve in the activity with mailing list "www-rdf-perllib"

where N is the total number of nodes in the original RACK graph, $|node|$ is the number of original nodes clustered to the node (as discussed for clustering in section 3), M is the total number of edges in the original RACK graph, $|edge|$ is the number of original edges clustered to the edge, and $cost_r$ is the cost function of the query root. The cost functions guarantee the cost of each node and edge to be in the interval (1,2). Since both local frequencies, i.e. the number of original elements clustered to an element, and total number of nodes and edges are incorporated, these function compute the importance of nodes and edges in a manner similar to TF/IDF used in information retrieval. Note that the higher its frequency is, the more important a node is considered to be because it will obtain a lower cost. As the cost is lower for elements with high importance, they have more positive impact on the rank of the query.

6 Evaluation

6.1 Experiment Setup

As there is yet no standard benchmark for evaluating the performance of translating keyword queries to formal queries, we use TAP, DBLP and LUBM [20] for the experiment. For TAP, we manually construct nine scenarios where the keywords and the corresponding potential information needs are listed in table 1 for three scenarios. The experiments are conducted on a Intel PC with 2.6GHz Pentium processor and 2GB memory. Note that the following presentation will focus on results performed on TAP. The proposed queries and their results for DBLP and LUBM as well as the extended presentation of our experiments can be found in the technical report [21] at http://q2semantic.apexlab.org/Pub/Q2Semantic-TR.pdf.

6.2 Effectiveness Evaluation

For ranking query, precision and recall as applied for information retrieval can not be used directly because only one of the computed query matches the meaning of the keywords intended by the user. Hence, we introduce a new metric called *Target Query Position (TQP)* to evaluate the effectiveness of query ranking. Namely, $TQP = 11 - P_{target}$, where P_{target} means the position of the intended query in the ranked list. Note the higher the rank of the intended query,

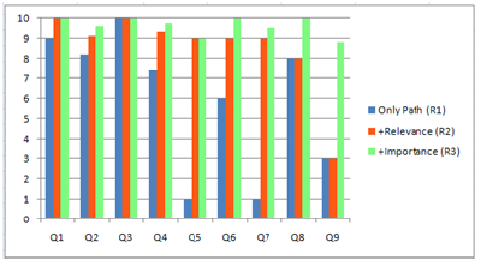

Fig. 7. TQPs of different ranking schemes on TAP.

the higher its TQP score. If the rank of a query is greater than ten, its TQP is simply 0. Thus, the TQP score of a query range from 0 to 10.

Since this metric is sensitive to the query rank, it can be used to evaluate our approach for query construction and the different ranking schemes. For this, We invite twelve graduate students to identify the query from a ranked list computed by Q2Semantic, which corresponds to their interpretations of the given keyword query. For each keyword query, we compute the final TQP score as an average of the scores obtained for each participant. Fig. 7 illustrates results of our experiments performed on TAP using the three different ranking schemes.

Note that the performance of R_1 is relatively good for Q1-Q4. This is because keywords in these queries have little ambiguity, i.e. can be mapped exactly one or two K-Nodes (e.g. "supergirl" in Q3). In these cases, the query length is very effective in ranking the queries. When applying R_2, significant improvements can be obtained for Q5-Q7. Keyword phrases in these queries are ambiguous, i.e. mapped to many K-Nodes, resulting in a lot of queries having the same path length. As R_2 also considers the matching score of the keyword phrases to K-Nodes, it helps to resolve this ambiguity. For instance, the query containing K-Nodes with highest matching score for "Strip" and "Las Vegas" (Q5) is indeed the one intended by the user. Finally, another improvement is obtained for Q8 and Q9 when using R_3 to consider also the importance of nodes and edges. This improvement comes from the usage of costs for nodes and edges, which guide the traversal and the selection of the root note. Note that elements with higher importance are preferred during expansions. For instance, in Q9, the author is preferred over the book because it has higher importance (lower cost).

In summary, the results show that our approach offers high quality translation of keywords to formal queries, especially when using R_3 as the ranking scheme. Our technical report also shows that the overall performance on all keyword queries for LUBM is promising and the average TQP reaches 9.125 by using R_3.

6.3 Efficiency Evaluation

Table 2 compares the statistical information of the original RACK graphs and the clustered RACK graphs (Bold numbers). The number of K-Node is the same as that of A-Edge according to the fourth clustering rule. It is observed that the

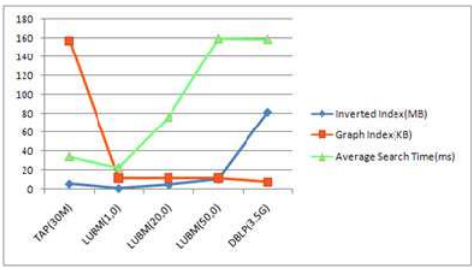

Fig. 8. Index size and search time on different datasets.

Table 2. RACK graph versus Clustered RACK graph

	R-Edge		A-Edge		C-Node		K-Node	
TAP	41914	**158**	87796	**666**	167656	**314**	87796	**666**
LUBM(1,0)	41763	**43**	30230	**39**	16221	**13**	30230	**39**
LUBM(20,0)	1127823	**43**	815511	**39**	411815	**13**	815511	**39**
LUBM(50,0)	2788382	**43**	2015672	**39**	1018501	**13**	2015672	**39**
DBLP	5619110	**19**	12129200	**23**	1366535	**5**	12129200	**23**

sizes are largely reduced after clustering. That is, the relevant data space to be explored in the query interpretation process is much smaller, which leads faster query construction. This is indicated by the average time in Fig. 8, which also shows the size of the inverted and the graph index for TAP, LUBM(1,0), LUBM(20,0), LUBM(50,0) and DBLP. The average time ranges from 20ms to 160ms on all datasets. Since no evaluation has been carried out to measure performance in previous approaches, we cannot make any comparative analysis. However, the reduction in data space must have a positive effect on performance.

In this regard, we found out that the size of the clustered graph index depends heavily on the schema structure of the original RDF graph. Namely, the simpler the schema (number of T-Box axioms), the smaller the index size. For example, the graph index size of DBLP and LUBM is smaller than that of TAP as TAP contain much more concepts. Also the performance depends on the size of the inverted index. This mainly depends on the number of literals in the ontology.

In summary, the experiments show promising performance. Besides the reduction of the original RDF graph, top-k query answering helps to terminate even more quickly, i.e. avoid the calculation of all possible queries. Our technical report provides more details on the impacts of the ranking mechanism, the top-k parameter, and the penalty parameter on the performance.

7 Related Work

Translating keywords to formal queries is a line of research that has been carried out in both the information retrieval and the database communities. Notably,[4,

5, 22] support keyword queries over databases while [6, 7] specifically tackle XML data by translating keyword queries to XQuery expressions. However, none of them can be directly applied to semantic search on RDF data since the underlying data model is a graph rather than relational or tree-shaped XML data.

[8] represents an attempt that specifically deals with keyword queries in semantic search engines. There, keywords are map to elements of triple patterns of predefined query templates. These templates fix the structure of the resulting queries a priori. However, only some but not all interpretations of the keywords can be captured by such templates. Also, since queries with more than two keywords lead to a combinatorial explosion of different possible interpretations, a large number of templates would be needed. These problems have been tackled recently by [9, 10]. In [10], a more generic graph-based approach has been proposed to explore all possible connections between nodes that correspond to keywords in the query. This way, all interpretations that can be derived from the underlying RDF graph can be computed.

With respect to these recent works [9, 10], our approach is distinct in three aspects. Firstly, we enrich RDF data with terms extracted from Wikipedia. Thus, users can also use their own words because keywords can map also to Wikipedia terms. Secondly, we extend the ranking mechanism in [10] to a more general framework for query ranking, which can incorporate many factors besides the query length. Most importantly, query construction has been relied on a large number of A-Box queries that are performed on the original RDF graph. Our approach greatly reduces this space to a summary graph, and thus scales to large repositories. The additional support for top-k queries can further help to terminate the translation even more quickly.

8 Conclusions and Future Work

In this paper, we propose a solution to translate keyword queries to formal queries that can address drawbacks in current approaches. RDF Data is enriched with terms from Wikipedia to support also keywords specified in the user own words. The RDF graph used for exploration is clustered down to a summary graph. Combined with top-k query answering, this increases scalability and efficiency of the translation process. To improve effectiveness, a more general ranking scheme is proposed that considers the query length, the element matching score and the importance of the elements. Evaluation of the implemented system Q2Semantic shows high quality translation of keyword queries processed against datasets of different sizes and domains.

Currently, our approach support keywords that match literals and concepts contained in the RDF data (where concepts are treated as special K-Nodes in the current implementation). We will extend the current query capability to support also keywords in the form of relations and attributes. Another future work is to integrate query interpretation with query answering in a unified graph index as one still need to use the original graph instead of the clustered RACK graph for answering the translated queries from keywords.

References

1. Tran, D.T., Bloehdorn, S., Cimiano, P., Haase, P.: Expressive resource descriptions for ontology-based information retrieval. In: Proceedings of the 1st International Conference on the Theory of Information Retrieval (ICTIR'07), 18th - 20th October 2007, Budapest, Hungary. (2007) 55–68
2. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying rdf and rdf schema. In: ISWC. (2002) 54–68
3. Lu, J., Ma, L., Zhang, L., Brunner, J.S., Wang, C., Pan, Y., Yu, Y.: Sor: A practical system for ontology storage, reasoning and search. In: VLDB. (2007) 1402–1405
4. Hristidis, V., Papakonstantinou, Y.: Discover: Keyword search in relational databases. In: VLDB. (2002) 670–681
5. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using banks. In: ICDE. (2002) 431–440
6. Hristidis, V., Papakonstantinou, Y., Balmin, A.: Keyword proximity search on xml graphs. In: ICDE. (2003) 367–378
7. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: Xrank: Ranked keyword search over xml documents. In: SIGMOD Conference. (2003) 16–27
8. Lei, Y., Uren, V.S., Motta, E.: Semsearch: A search engine for the semantic web. In: EKAW. (2006) 238–245
9. Zhou, Q., Wang, C., Xiong, M., Wang, H., Yu, Y.: Spark: Adapting keyword query to semantic search. In: ISWC/ASWC. (2007) 694–707
10. Tran, T., Cimiano, P., Rudolph, S., Studer, R.: Ontology-based interpretation of keywords for semantic search. In: ISWC/ASWC. (2007) 523–536
11. Horrocks, I., Tessaris, S.: Querying the semantic web: a formal approach. In: Proceedings of the ISWC 2002, Chia, Sardinia, Italy (2002) 177–191
12. Fokoue, A., Kershenbaum, A., Ma, L., Schonberg, E., Srinivas, K.: The summary abox: Cutting ontologies down to size. In: ISWC. (2006) 343–356
13. Zhang, L., Liu, Q., Zhang, J., Wang, H., Pan, Y., Yu, Y.: Semplore: An ir approach to scalable hybrid query of semantic web data. In: ISWC/ASWC. (2007) 652–665
14. Milne, D., Witten, I.H., Nichols, D.: A knowledge-based search engine powered by wikipedia. In: Proc. of CIKM. (2007)
15. Fu, L., Wang, H., Zhu, H., Zhang, H., Wang, Y., Yu, Y.: Making more wikipedians: Facilitating semantics reuse for wikipedia authoring. In: ISWC/ASWC. (2007) 128–141
16. He, H., Wang, H., Yang, J., Yu, P.S.: Blinks: ranked keyword searches on graphs. In: SIGMOD Conference. (2007) 305–316
17. Ding, L., Pan, R., Finin, T.W., Joshi, A., Peng, Y., Kolari, P.: Finding and ranking knowledge on the semantic web. In: ISWC. (2005) 156–170
18. Anyanwu, K., Maduko, A., Sheth, A.P.: Semrank: ranking complex relationship search results on the semantic web. In: WWW. (2005) 117–127
19. Lehmann, J., Schüppel, J., Auer, S.: Discovering unknown connections - the dbpedia relationship finder. In: Proc. of the 1st SABRE Conference on Social Semantic Web (CSSW). (2007)
20. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. J. Web Sem. **3**(2-3) (2005) 158–182
21. Wang, H., Zhang, K., Liu, Q., Yu, Y.: Q2semantic: Adapting keywords to semantic search. Technical report, APEX Data & Knowledge Management Lab (2007)
22. Balmin, A., Hristidis, V., Papakonstantinou, Y.: Objectrank: Authority-based keyword search in databases. In: VLDB. (2004) 564–575