Journal of Database Management
Volume 29 • Issue 4 • October-December 2018

RDF Keyword Search by Query Computation

Zongmin Ma, Nanjing University of Aeronautics and Astronautics, Nanjing, China
https://orcid.org/0000-0001-7780-6473

Xiaoqing Lin, Eastern Liaoning University, Dandong, China

Li Yan, Nanjing University of Aeronautics and Astronautics, Nanjing, China

Zhen Zhao, Bohai University, Jinzhou, China

ABSTRACT

Keyword searches based on the keywords-to-SPARQL translation is attracting more attention because of a growing number of excellent SPARQL search engines. Current approaches for keyword search based on the keywords-to-SPARQL translation suffer from returning incomplete answers or wrong answers due to a lack of underlying schema information. To overcome these difficulties, in this article, we propose a new keyword search paradigm by translating keyword queries into SPARQL queries for exploring RDF data. An inter-entity relationship summary with complete schema information is distilled from the RDF data graph for composing SPARQL queries. To avoid potentially wasteful summary graph expansion, we develop a new search prioritization scheme by combining the degree of a vertex with the distance from the original keyword element. Starting from the ordered priority list that is built in advance, we apply the forward path index to faster find the top-k subgraphs, which are relevant to the conjunction of the entering keywords. The experimental results show that our approach is efficient and scalable.

KEYWORDS

Inter-Entity Relationship Summary, Keyword Search, Property Paths, SPARQL, Subgraph Search

INTRODUCTION

RDF (Resource Description Framework) is a standard model for data interchange on the Web. By this model, more and more structured and semi-structured data have been mixed, exposed, and shared across different applications (Klyne, Carroll, and McBride, 2004). As a result, available RDF data rapidly increases. RDF data is a collection of triples, the form (subject, predicate, object). Such a collection of triples can be represented as a directed graph, in which vertices represent subjects or objects, and edges represent predicates that connect subjects and objects. Several important issues of RDF data management are recently reviewed in (Ma, Capretz, Miriam, and Yan, 2016; Wylot,

DOI: 10.4018/JDM.2018100101

Hauswirth, Philippe, and Sakr, 2018), including RDF data storage techniques, indexing strategies, and query execution mechanisms.

Since SPARQL (Prud'hommeaux and Seaborne, 2013) has been recommended to be the standard query language for RDF data by the World Wide Web Consortium (W3C), there has been a rapid increase in the number of users who want to access RDF data (Yan, Ma, Li, and Cheng, 2017). SPARQL allows the specification of triple and graph patterns to be matched over RDF data graphs (Ma, Jia, Cheng, and AngryK, 2016). And then we can access RDF data correctly and efficiently by SPARQL. But it is still infeasible for non-expert users to master the RDF schema and SPARQL query language. Let us look at an example of SPARQL query for the DBLP dataset shown in Example 1 by the data in Figure 2.

Example 1: Find the published articles and proceedings by author "*Coles:Drue*" for the DBLP dataset.

```
SELECT ?x ?y
WHERE {?x isIncludedIn ?y.
       ?x type Article_in_Proceedings.
       ?y type Proceedings.
       ?x author "Coles:Drue".}
```

It is shown that, to construct a SPARQL query like Example 1, non-expert users are required not only to master SPARQL but also to know the schema information of RDF data such as "*isIncludedIn*", "*Article in Proceedings*" and "*Proceedings*". So, it is not easy for non-expert users to construct SPARQL queries. At this point, keyword search has been a popular tool for exploring RDF data for non-expert users (Izquierdo et al., 2018; García, Izquierdo, Menendez, Dartayre, and Marco, 2017). Users only need to enter keywords and then top-*k* query results answered can be directly returned to the user. Currently, the number of excellent SPARQL search engines is growing rapidly (Broekstra, Kampman, and Harmelen, 2002; Garrison, Stevens, and Jocuns, 2004; Neumann and Weikum, 2008; Neumann and Weikum, 2010). To make it easy for non-expert users to compose SPARQL queries, in this paper, we concentrate on the approach for translating keyword queries into SPARQL queries. As mentioned in (Gkirtzou, Papastefanatos, and Dalamagas, 2015; Ladwig and Tran, 2010; Tran, Wang, Rudolph, and Cimiano, 2009; Lin, Ma, and Yan, 2018; Wen, Jin, and Yuan, 2018), keyword search based on translation has its advantages. It can provide users with a friendly interface for querying RDF data on the one hand and, on the other hand, we can obtain a better query performance by using the existing SPARQL search engines on the condition of guaranteeing the correctness.

Generally speaking, two categories of keyword searches can be identified according to different query processing ways. The first one aims to find smaller substructures that contain all keywords over the original RDF datasets. The substructures, which can be *graphs* (Li, Ooi, Feng, Wang, and Zhou, 2008; Sinha, Lu, and Theodoratos, 2010; Han, Zou, Yu, and Zhao, 2017), *cliques* (Kargar and An, 2011), *trees* (Chen, Wang, Liu, and Lin, 2009; Fakas, 2011; He, Wang, Yang, and Yu, 2007; Kacholia, Pandit, Chakrabarti, Sudarshan, and Karambelkar, 2005; Hulgeri and Nakhe, 2002; Kasneci, Ramanath, Sozio, Suchanek, and Weikum, 2009; Le, Li, Kementsietsidis, and Duan, 2014; Liu, Yu, Meng, and Chowdhury, 2006; Sun, Chan, and Goenka, 2007; Wang, Zou, Pan, and Zhao, 2012) and other patterns defined over RDF graphs (Dass, Aksoy, Dimitriou, Theodoratos, and Wu, 2016; Li, Yan, and Ma, 2019; Peng, Zou, and Qin, 2017), are computed by a scoring mechanism. A semantic search engine is developed in (Taha and Elmasri, 2009) to answer XML keyword-based queries as well as loosely structured queries using object-oriented techniques. In (Lian, Hoyos, Chebotko, Fu, and Reilly, 2013), *k*-nearest keyword search in RDF graphs is investigated by utilizing an index for *k* closest pairs of vertices to prune search space. The second one interprets keyword queries into SPARQL queries (Fu, Gao, and Anyanwu, 2011; Gkirtzou et al., 2015; Ladwig et al., 2010; Tran et al., 2009; Zenz, Zhou, Minack, Siberski, and Nejdl, 2009; Zhou, Wang, Xiong, Wang, and Yu, 2007).

In (Fu et al., 2011; Gkirtzou et al, 2015; Zenz et al., 2009; Zhou et al., 2009), for example, keyword queries are automatically translated into SPARQL queries and then the translated SPARQL queries are performed by existing SPARQL search engines. Following this direction, natural language queries are transformed to SPARQL queries in (Cabrio, et al., 2012; Zheng, et al., 2015; Zou, et al., 2014). Also, there are some efforts that try to combine keywords and SPARQL queries together (Elbassuoni, Ramanath, Schenkel, and Weikum, 2010; Peng, Lei, and Zheng, 2017; Wang, Tran, Liu, and Fu, 2011).

There are few works on keyword search based on keywords-to-SPARQL translation. In (Zenz et al., 2009), users are required to provide a series of incremental refinement steps for constructing SPARQL queries. Zhou et al. (2007) presents a statistical approach, which selects the most likely SPARQL query from a ranked list of SPARQL by a probabilistic query ranking model. Being similar to the most statistical methods, the returned results are affected by the probabilistic model greatly. To generate the most likely intended interpretation, query history is exploited in (Fu et al., 2011) to interpret a new query. The context of each input keyword is taken into account in (Wen, Jin and Yuan, 2018) and user intention in differential graph queries is considered in (Vasilyeva, Thiele, Bornhövd, and Lehner, 2015). In (Fu et al., 2011; Gkirtzou et al., 2015; Zenz et al., 2009), keyword search based on translation is performed well on the DBLP dataset. But these approaches cannot guarantee that the query results are correct with respect to certain RDF datasets. More importantly, there is a common limitation on the summary defined in (Fu et al., 2011; Gkirtzou et al., 2015; Zenz et al., 2009). That is, there may exist more than one relationship between one type of entities and other types of entities.

For the LUBM dataset, for instance, "*FullProfessor0*" is mapped to a type vertex "*FullProfessor*" in the summary, and "*University389*", "*University942*" and "*University643*" are all mapped to the same type vertex "*University*". However, the properties "*mastersDegreeFrom*", "*undergraduateDegreeFrom*" and "*doctoralDegreeFrom*" cannot be all kept by the summary defined in the literatures (Fu et al., 2011; Gkirtzou et al., 2015; Zenz et al., 2009). For example, for the keyword search to find the universities that "*FullProfessor0*" professor graduated from, the SPARQL queries produced may be Figure 1 (a), Figure 1 (b) or Figure 1 (c). Hence, whatever which query is produced in Figure 1, we cannot obtain the complete answers.

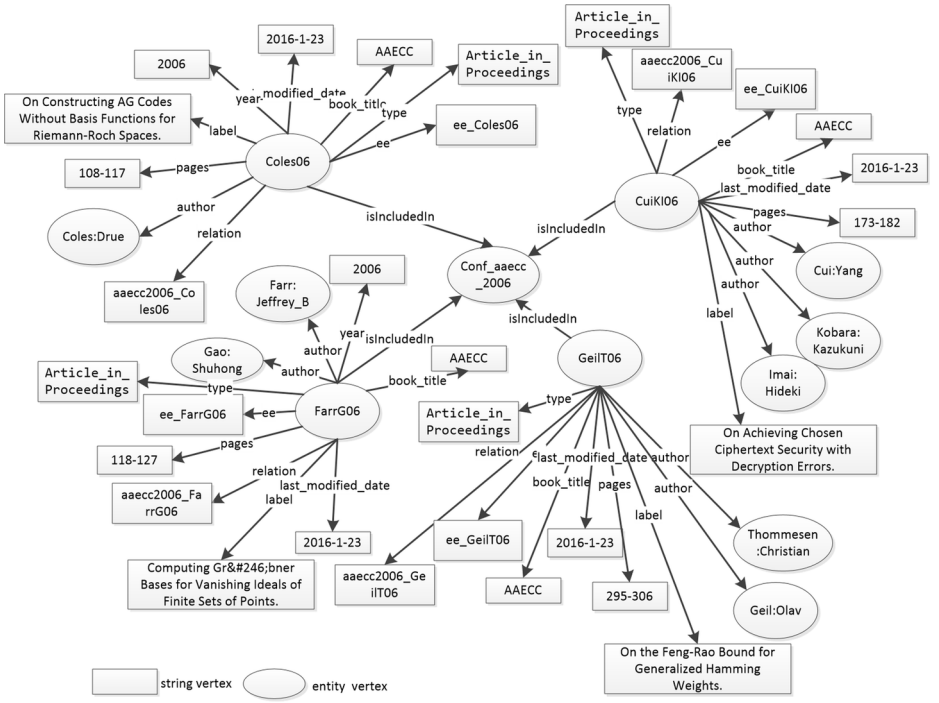
In this paper, we devote to RDF keyword search based on keywords-to-SPARQL translation. Our work is different from the works in (Fu et al., 2011; Gkirtzou et al., 2015; Zenz et al., 2009) mainly in two aspects. First, we distill a complete inter-entity summary from the original RDF data graph instead of using the summary defined in (Fu et al., 2011; Gkirtzou et al., 2015; Zenz et al., 2009) to tackle the problem that the query answers may be incomplete. Second, to reduce the search space, we devise a prioritization scheme to directly search the top-*k* subgraphs with inter-entity relationships over the inter-entity summary. Unlike the Backward search in (Bhalotia, et al., 2002) that explores the backward paths from the matched keyword elements toward the answer roots, we explore the path ordered by a prioritization mechanism, which combines a degree of a vertex with the distance from the starting keyword elements. Our prioritization scheme allows a preferential expansion to the paths with less branching and closer distance from the starting keyword element. For the vertices that have large fan-in, the Backward search has to explore a large number of vertices backward.

Let us look at an example. In Figure 2, many authors "*Cui:Yang*", "*Kobara:Kazukuni*" and "*lmai:Hideki*" refer to a particular conference "*CuiKl06*" and many conferences "*Coles06*",

Figure 1. Possible SPARQL queries produced by the work in (Tran, et al., 2009)

SELECT ?a ?b WHERE { ?a type FullProfessor. ?b type University. ?a undergraduateDegreeFrom ?b. ?a name "FullProfessor0"}	SELECT ?a ?b WHERE { ?a type FullProfessor. ?b type University. ?a mastersDegreeFrom ?b. ?a name "FullProfessor0"}	SELECT ?a ?b WHERE { ?a type FullProfessor. ?b type University. ?a doctoralDegreeFrom ?b. ?a name "FullProfessor0"}
(a)	(b)	(c)

Figure 2. An RDF data graph for DBLP



"*CuiKl06*", "*GeilT06*" and "*FarrG06*" refer to the conference vertex "*Conf_aaecc_2006*". In this case, a Backward search may lead to a poor performance. We consider the distance from the starting keyword element. In general, the content keyword elements are very close to each other. And the vertices that are far away from the starting keyword element are not the useful and informative answer roots. Here we use the concept of *r*-cliques, which can improve system performance significantly and is more efficient to explore only the content keyword elements rather than the whole graph (Kargar and An, 2011).

In summary, we propose a new paradigm for RDF keyword search by translating keyword queries into SPARQL queries. Being a little similar to the work in (Rivero, Hernández, Ruiz, and Corchuelo, 2015), we distill an inter-entity relationship summary, but our approach is schema agnostic. This is crucial because even when there is a schema, it is a common case in RDF data that not all relations and attributes of entities can be captured. The summary graph is derived from the data to capture the "schema" information that is necessary for query computation. Our exploration does not operate directly on the data, but on the summary. Moreover, a prioritization scheme is used to search top-*k* subgraphs that can collect all keyword elements over the summary as quickly as possible. Then, the top-*k* subgraphs are transformed into top-*k* SPARQL queries. We exploit the property paths of SPARQL 1.1, which are denoted by "|", "/" or "^", to compose SPARQL queries from the top-*k* subgraphs. Finally, the produced SPARQL queries are executed by a SPARQL search engine. Being different from that in the literature (Virgilio, Cappellari, Maccioni, and Torlone, 2012), all paths are indexed off-line, starting from a source and ending into a sink that represents the main flow of information in the graph. To the best of our knowledge, this is the first effort that composes SPARQL queries with the property path operators of SPARQL 1.1 for the translation of keywords-to-SPARQL. The main contributions of this paper are summarized as follows:

- We devise a new prioritization mechanism. We utilize the degree of the vertex to search less branching on the one hand and on the other hand, we exploit the distance from the keyword element to find the answer, where vertices are close one another. With the priority list, we search paths forward to keyword elements through the forward search index;
- We pre-compute the shortest distance index over the summary by using the breadth-first algorithm. Considering that the overhead is enormous if all the entity pairs of the summary are indexed, we only pre-compute the shortest distance index and the predicate path index in the given steps;
- We utilize the property paths of SPARQL 1.1 to transform top-*k* subgraphs to SPARQL queries and this makes the transformation process easy.

The rest of the paper is organized as follows. The next section outlines the approach proposed in the paper. The third section describes the indexing graph data. The fourth section presents the top-*k* subgraph search. The fifth section proposes the translation of top-*k* subgraphs with the forward path search. Experiments are given in the sixth section. The last section 7 concludes this paper.

OVERVIEW OF THE PROPOSED APPROACH

This section outlines our approach for querying RDF data by using keywords. Our approach consists of the following four phases:

- **Keywords Mapping:** In this phase, the keywords issued by the user are mapped to the elements over the summary graph. Here a keyword can be a data value or a property. We construct a keyword index which can index all string literals together with the types of RDF data. We call the matched elements of the summary graph keyword elements in our approach. At this phase, we calculate the top-*k* possible combinations of keyword mappings;
- **Summary Graph Exploration:** The second phase is to explore the summary graph for finding top-*k* subgraphs that connect the keyword elements. Here the keyword elements are sorted according to the combined distance from the connecting vertex *root* to each keyword element v_i. A novel prioritization scheme is utilized and then top-*k* subgraphs can be found faster by the forward path index;
- **Translating Keyword Queries to SPARQL Queries:** At this phase, the top-*k* subgraphs produced by the last phase are translated into top-*k* SPARQL queries. To transform a subgraph to a SPARQL query, the property paths of SPARQL 1.1 is utilized to construct SPARQL queries;
- **Performing SPARQL Queries:** At the final phase, the produced top-*k* SPARQL queries are executed by a SPARQL search engine and the query results are returned to the user.

Indexing Graph Data

This section describes the off-line indexing process, in which RDF data is preprocessed and stored in the data structures of an inter-entity relationship summary and a keyword index opst.

Inter-Entity Relationship Summary

We aim at translating keyword queries to SPARQL queries. But it costs too much for the exploration of the whole RDF data graph. We need to know the relationships among entities that correspond to query patterns. For this purpose, we distill a summary only with entity relationships from the original RDF data graph and then we can perform exploration over the summary graph. This can reduce the search space greatly. We partition the entire RDF data graph into two parts, which are *the summary graph* and *the keyword index*, respectively. The summary is defined in Definition 2.

Definition 1: An RDF data graph G is a tuple (V, L, E). Here V is a finite set of vertices and the union of disjoint sets V_E, V_C and V_V, in which V_E represents the set of entity vertices (i.e., IRIs), V_C represents the set of type vertices, and V_V represents a set of value vertices. L is a finite set of edge labels and the union of disjoint sets of L_R, L_A and {type, subclass}, in which L_R represents the set of the edges connecting entity vertices, and L_A represents the set of the edges connecting entity vertices and value vertices. E is a finite set of edges with the form of $e(v_1,v_2)$, in which v_1, $v_2 \in V_E$ and $e \in L$. Then we have $e \in L_R$ if v_1, $v_2 \in V_E$, $e \in L_A$ if and only if $v_1 \in V_E$ and $v_2 \in V_V$, e = type if and only if $v_1 \in V_E$ and $v_2 \in Vc$, and e = subclass if and only if v_1, $v_2 \in V_C$.

Definition 2: An inter-entity relationship summary G' of an RDF data graph $G = (V, L, E)$ is a tuple (V', L', E') with vertices $V' = V_E$, inter-entity edge labels $L' = L_R$, and $E' \subset E$. Here:

(1) E' connects two entity vertices;
(2) V_E represents the set of entity vertices (i.e., IRIs);
(3) *Edge* $e(v_1, v_2)$ exists in the summary if and only if there is an edge $e(v_1, v_2) \in E$ and $v_1, v_2 \in V_E$.

It is shown in Definition 2 that there are not any edges with $e \in L_A$ and $e = type$ in the summary. Such edges are stored in the keyword index.

Though the size of our summary is larger than the one in (Than et al., 2009), some inter-entity relationships are lost. Our summary can summarize the complete data structures of the original RDF data graph. In addition, given a subject and an object in the summary (Than et al., 2009), there may exist more than one relationship between them, that is, multiple-edges many exist between two vertices. This case cannot be dealt with. In our summary, for example, there are two relationships "*headOf*" and "*worksFor*" between the vertices "*Fullprofessor*" and "*Department*". But by the work in (Than et al., 2009), only one relationship can be kept and other relationships are lost. We distill an updatable inter-entity relationship summary from the original RDF data so that the property paths and the path lengths of any two entity vertices can be effectively estimated by the summary.

A summary graph example for the DBLP dataset is illustrated in Figure 3, which is refined from Figure 2. We note from the summary that only entity vertices and inter-entity relationships remain.

Keyword Index

Opst index is used to locate the keyword for the designated entity, where o, p, s, and t are respectively the object, the predicate, the subject of the triple and the type that the subject belongs to. Unlike the posc index in (Harth and Decker, 2005), which contains the subject, the object, the predicate and the context of triple (s, p, o), our approach needs the corresponding type for the SPARQL query translation (we detail this in Section Translation of top-k subgraphs). The ops of our opst index are the same to the one in (Weiss, Karras, and Bernstein, 2008), in which each object key o_i is associated with a sorted list $\{p_1^i, p_2^i, \ldots, p_{ni}^i\}$ of n_i predicate keys, and each predicate key p_j^i is linked to an associated sorted vector of $k_{i,j}$ subject keys in its turn. The quad in Table 1 is extracted from the RDF data graph for the keyword index.

It is shown in Table 1 that the object "*MIT*" includes a predicate vector with two entries "*bachelorFrom*" and "*worksFor*". Each of these predicate entries is appended with a list of associated subjects and the subjects' types. In this particular example, each list contains one item only, that is, "*ID1*" for the "*bachelorFrom*" predicate and "*ID2*" for the "*worksFor*" predicate. For a given keyword (data value or a property), a list of the summary graph elements (entity IDs) is returned. At the same time, we can get the predicate, the subject and the type matching the keyword (object) by the opst index.

SEARCH FOR TOP-*K* SUBGRAPHS

Now we describe the top-k subgraphs exploration for the inter-entity summary in order to find the inter-entity relationships. A prioritized scheme is presented by combing the degree of a vertex with

Figure 3. An inter-entity relationship summary for DBLP

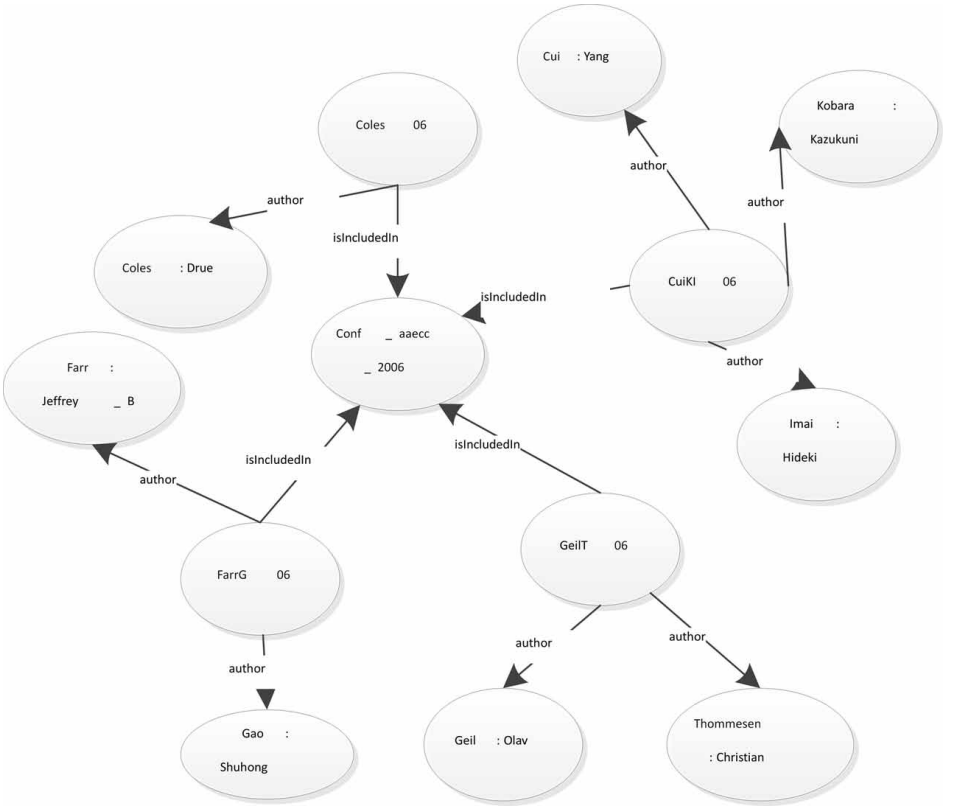


Table 1. RDF triples for opst index

Object	Predicate	Subject	Type
'*AI*'	teacherOf	ID1	FullProfessor
'*MIT*'	BachelorsFrom	ID1	FullProfessor
'*Cambridge*'	mastersFrom	ID1	FullProfessor
'*Yale*'	phdFrom	ID1	FullProfessor
'*MIT*'	worksFor	ID2	AssocProfessor
'*DataBases*'	teacherOf	ID2	AssocProfessor
'*Yale*'	bachelorsFrom	ID2	AssocProfessor
'*Standford*'	phdFrom	ID2	AssocProfessor
'*AI*'	teachingAssist	ID3	GraduateStudent
'*Princeton*'	mastersFrom	ID3	GraduateStudent
'*DataBases*'	tackesCourse	ID4	GraduateStudent
'*Columbia*'	bachelorsFrom	ID4	GraduateStudent

the vertex distance from the keyword element. Meanwhile, the forward path index is applied to explore the summary forward and thus top-*k* subgraphs can be found faster.

Prioritization for Backward Search

In this subsection, we motivate and describe our approach for prioritizing on the summary graph. We aim to find the potential answers by examining as a small part of the summary graph as much as possible.

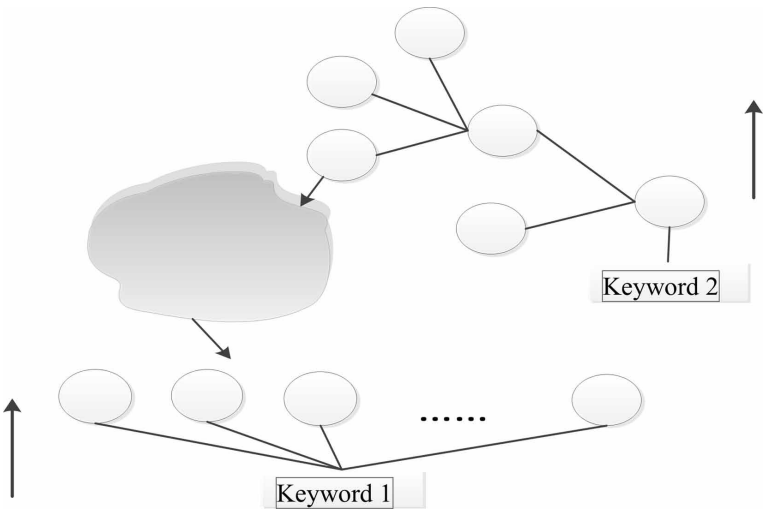
Motivation for Prioritizing Backward Search

A large number of graph vertices have to be explored by the Backward search in some scenarios. For example, to hit a "hub" vertex with a very large of fan-in or fan-out (called the degree in short) is hit, we need to explore a large number of vertices. For example, a department vertex in a university has many teachers, students and other vertices connected to the department vertex. Our top-k answer is a sorted list of subgraphs that connect all keyword elements. This list is ordered by the combined distance from the connected vertex to all keyword elements.

In the case of the above scenario, a Backward search may visit a large number of vertices before finding the relevant answers. As shown in Figure 4, the Backward search must hurt performance as a large number of vertices match "keyword1". Unlike the search algorithm in (He et al., 2007), the vertices to be backward expanded are ordered by the distance from the keyword element. A list of priorities defines the order of our backward search. The priority scores are calculated by the following Formula 1, in which Deg(*u*) is the degree of vertex *u* and dist(*u*, *keyword*) denotes the shortest path length from *u* to the keyword element. A factor μ controls their relative weight. Here $\mu = 0.7$ is set by experiments. Both the degree of each vertex and the distance from the keyword element are embodied in Formula 1:

$$P_u = \begin{cases} \dfrac{\mu}{Deg(u)} + \dfrac{(1-\mu)}{Dis(u, keyword)}, & if\ u \neq keyword \\ \dfrac{1}{Deg(u)}, & if\ u\ is\ the\ keyword \end{cases} \quad (1)$$

Figure 4. The backward search with forward path index



The prioritization mechanism can reduce the search space and time cost of finding a potential answer root. The priority score of a vertex is bigger and its scheduling priority is higher. In Formula 1, the priority of vertex is inversely proportional to the distance from the keyword element as well as the degree of the vertex. In (Hulgeri and Nakhe, 2002), however, the vertices with higher degrees will get higher prestige. With Formula 1, the vertices that are closer to the potential root or that have lower degrees can get higher priority. The degree of vertex reflects the size of the frontier to be expanded. Therefore, when there are vertices with high degrees, a Backward search may explore a large portion of the graph and this may lead to a longer search time.

Our prioritization mechanism is a little similar to the heuristic activation factors that are used to estimate the potential answer roots (Kacholia et al., 2005). The vertices to be expanded are ordered by a prioritization mechanism. Note that their approach does not include the distance from the original keyword element. Any worst-case performance guarantee cannot be provided because the activation factors are derived from the general topology and the explored elements. Unlike that the forward search in (Kacholia et al., 2005) is aimless, the forward paths in our approach are followed by the forward search index, which is built based on the distances to keyword elements (we detail this in Section Forward path index). A depth value cutoff d_{max} is applied to prevent the generation of answers that are not intuitive due to excessive path lengths. To ensures the termination, we adopt a generous default of $d_{max} = 8$ by experiments. The examples of priority lists are shown in Figure 5, in which two elements are included: the vertex id and the priority score calculated by Formula 1.

Forward Path Index

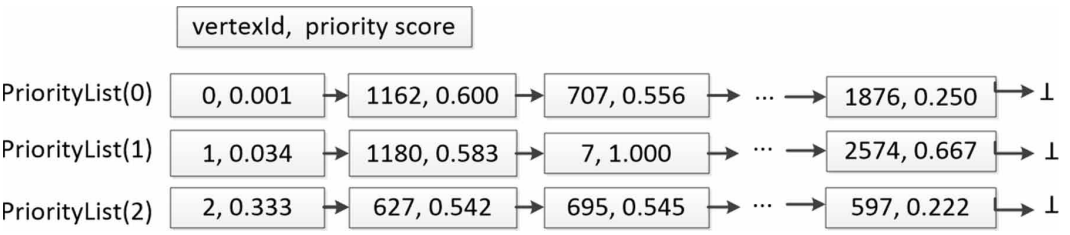
In this subsection, we describe the forward path index that guides the forward expansion when retrieving top-k subgraphs. Here we apply the breadth-first search algorithm to generate the forward search index over the summary. The weight of each edge of the summary graph is set to be 1. As there exists a large scale of entities over the summary, too much overhead of time and space will be required if we index all the entity pairs over the summary. In our approach, only entity pairs with d_{max} steps from the starting vertex are indexed.

The algorithm of generating forward search index is shown in Algorithm 1.

For the summary, the shortest graph distance of any two vertices is precomputed and organized in a hash table Map_{dist}. Given two vertices u and v, Map_{dist} $(u \rightarrow v)$ returns the shortest distance from u to v, or ∞ if u cannot reach v within a step of d_{max}. We assume that the answers out of d_{max} steps aren't good ones even if they are found because the answers in which keyword elements are too far away are not useful and informative (Kargar et al., 2011). The Map_{dist} has the form as shown in Table 2.

The adjacency list *AdjList* stores the entity summary graph illustrated in Table 3. The array *Visited* is used to indicate whether the *ith* entry of the *AdjList* is visited. The queue Q is a temporary queue for the breadth-first search. The map Map_{dist} returns pairs with the form of $(v_1 \rightarrow v_2, distance)$, which is actually the forward search index. A separate entity table is to store long URIs shown in Table 4. Thus, there are not long URIs but there are the entity IDs in the summary graph and the keyword index.

Figure 5. Sample priority lists for the keyword element to the vertex



Algorithm 1. Building a forward search index with d_{max} steps

Input: entity-relationship summary *AdjList*
Output: the shortest path index with d_{max} steps Map_{dist}

```
1  Variables: adjacent list AdjList, storing the entity summary graph; queue Q, each element of which
   contains two parts, the vId, and the step; array V isited, indicating whether the vertex is visited;
2  for i ← 0 to AdjList.size () do
3      Visited [i] ← 0;
4      Q.add(< AdjList .get(i).vId ,0 > );
5      while Qnotempty do
6          u ← Q.poll();
7          if u.step>d max then
8              Q.clear();
9              break;
10         if !Visited[u.vId] then
11             Mapdist .put(AdjList.get (i).vId →
               u.vId,u.step );
12             Visited [u.vId ] ← 1 ;
13             u.step++ ;
14             for j ← 0 to
               AdjList .get(u.vId ).firstEdge .size()
               do
15                 t ⇐ (AdjList .get(u.vId )
16                 .firstEdge .get(j).vId , u.step);
17                 Q.add(t);
```

Table 2. The shortest path distance of any two vertices

From V → to V	**Distance**
$v_1 \rightarrow a$	0
$v_1 \rightarrow b$	1
$v_1 \rightarrow c$	1
$v_1 \rightarrow d$	2

Table 3. The adjacency list for the entity summary graph

EntityId	AdjList(predicate, EntityId)
0	(^ub:memberOf,707),(^ub:worksFor,766),(^ub:suborgnizationOf,784),...
1	(^ub:memberOf,1162),(^ub:memberOf,707),(^ub:subOrganizationOf,784),...
2	(ub:publicationAuthor,1),(ub:publicationAuthor,627),(ub:publicationAuthor,695),...
3	(ub:publicationAuthor,1)
4	(ub:publicationAuthor,1),(ub:publicationAuthor,641),...
5	(ub:publicationAuthor,1),(ub:publicationAuthor,647),(ub:publicationAuthor,712)
6	(ub:publicationAuthor,1),(ub:publicationAuthor,610)
7	(ub:publicationAuthor,1)

Table 4. Entity table

Id	Entity URI
0	http://www.Department0.University0.edu
1	http://www.Department0.University0.edu/AssistantProfessor0
2	http://www.Department0.University0.edu/AssistantProfessor0/Publication0
3	http://www.Department0.University0.edu/AssistantProfessor0/Publication1
4	http://www.Department0.University0.edu/AssistantProfessor0/Publication2
5	http://www.Department0.University0.edu/AssistantProfessor0/Publication3
6	http://www.Department0.University0.edu/AssistantProfessor0/Publication4
7	http://www.Department0.University0.edu/AssistantProfessor0/Publication5
8	http://www.Department0.University0.edu/AssistantProfessor1
9	http://www.Department0.University0.edu/AssistantProfessor1/Publication0

In the first iteration of Algorithm 1 (line 2-17), for each i with $i <$*AdjList.size* (), we insert the *ith* entry of the *AdjList* into the queue Q (line 4), in which *AdjList* is an adjacency list that stores the inter-entity relationship summary graph. Then, we use the breath first search algorithm to traverse all vertices of which steps are not greater than d_{max} steps from the *ith* head vertex of *AdjList*. The pairs with the form of (*headvId* → *v, distance*) are stored in a map Map_{dist} (line 5-17). Then the iteration doesn't stop until i reaches the size of *AdjList*. Each time when the queue Q is not empty, we get the frontest entry u from the queue Q (line 6). If *u.step* is greater than d_{max}, the *while* loop is terminated (line 7-9) and the pair (*headvId* → *u.vId, u.step*) is inserted into the map Map_{dist} (line 11). Then, the array *V isited*[*vId*] is set to be 1. It means that the *vId-th* entry of the *AdjList* has been visited (line 12). Finally, all the vertices that are adjacent to the *u.vId* are inserted into the queue Q (line 14-17).

The worst time complexity of the algorithm is O (n*(n+e)), where n is the number of vertices for the inter-entity relationship summary and e is the number of the edges in the summary graph.

Search for Top-k Subgraphs With the Forward Path Index

In this subsection, we deal with top-k subgraph search with the forward path index. Let $c = \{w_1, w_2, ..., w_m\}$ be a hint of keyword element combination that matches query $q = \{k_1, k_2, ..., k_m\}$. A set M of elements is used to keep the state of the connecting vertex. In other words, what keywords are reachable to the vertex and their best-known property paths in d_{max} steps. $M[u]$ is used to indicate the bookkeeping for the connecting vertex u. Especially, each element of M is a list of m pairs with form (*vertexId*, *propertypath*). A (*vertexId*, *propertyPath*) pair in the *jth* entry of $M[u]$ denotes the property path from u to keyword w_j with the shortest distance.

In Figure 1, we have an element M[Department0] = {(University0,subOrganizationOf), (AssistantProfessor0, ^worksFor), (GraduateCourse39, ^worksFor/teacherOf), ...} in M. The entry denotes that "Department0" is reachable by keywords "University0", "AssistantProfessor0", "GraduateCourse39" and etc., and the corresponding property paths are subOrganizationOf, ^worksFor, ^worksFor/teacherOf and etc., where "^" is the reverse path operator of SPARQL property path. These property paths are used for composing SPARQL queries later.

In the following, we propose the top-k subgraph search algorithm illustrated in Algorithm 2. We precompute the property paths in d_{max} steps and organize them in a hashMap Map_{path} form of ($u \rightarrow v$, *propertypath*). The computation is similar to the forward path index.

Algorithm 2 consists of two main steps. Step 1 explores the summary graph to find the answer roots of top-k subgraphs by collecting all keyword elements (line 1-19). Step 2 returns top-k subgraphs

Algorithm 2. Subgraph searching with the forward path index

Input: priority list *PriorityList*; forward path index map Map_{dist}; property path index Map_{path}; all possible keyword element combinations $C = W_1 \times W_2 \times ... \times W_m = \{c = (w_1, ..., w_m) | w_i \in W_i, i = 1,...,m\}$; V, a set form of (< *u,sumDist* >) pairs, *u* representing the connecting vertex, *sumDist* representing the combined distance from *m* keywords
Output: top-*k* subgraphs *M*

```
1  M ← ∅;
2  for each c=(w_1, ..., w_m) ∈ C and i ∈ [1,m] do
3      cursor_i ← new Cursor(PriorityList(w_i));
4      for i ← 0 to PriorityList(w_i).size() do
5          u ← cursor_i.next().vId;
6          sumDist ← 0;
7          for j ∈ [1,m] do
8              if Map_dist(u → w_j) < ∞ then
9                  sumDist ← sumDist +
                   Map_dist(u → w_j);
10             else
11                 sumDist ← 0;
12                 break;
13         if sumDist < τ_prune and sumDist ≠ 0
           then
14             if u is not in V then
15                 V.add(<u, sumDist >);
16             if |V| >= k then
17                 τ_prune ← the k-th largest of
                   {sumDist(v)|v ∈ V};
18         if |V| >= k and sumDist > τ_prune then
19             exit;
20 for u ∈ V do
21     for j ∈ [1,m] do
22         M[u][j] ← (w_j, Map_path.get(u → w_j)) ;
23 return M (if found) or nil (if not);
```

with property paths (line 20-22). Now we elaborate on these two steps. Given a query (k_1, k_2, ..., k_m), a cursor is used for the designated priority list for each possible matching combination c = (w_1, w_2, ..., w_m) (line 3). Cursor $cursor_i$ advances on list *PriorityList*(w_i) by calling next, which returns the next vertex in the list (line 5). The forward path index is used to expand the search forward. When a vertex is visited, its distance to the other keyword elements is checked by the map Map_{dist} (line 7-12). Using the information in Map_{dist}, the root of an answer is immediately determined if the distances from *u* to other keywords are all less than ∞. If one of the distances from the root to keyword elements is ∞, the root cannot be the root of an answer and the loop is terminated (line 10-12). A structured form of a pair (*root*, *sumDist*) is stored (line 14-15), where the root is the visited vertex and *sumDist* is the combined distances from the root vertex to the keyword elements. The pair is used to find the vertices with the top-k combined distance from the keyword elements. At last, we store *M* with the answer roots in *V*, and the property paths from the answer root to the keyword elements are calculated by Map_{path} (line 20-22). A threshold for pruning is τ_{prune}, which is the current *k*th shortest combined distance among all known answer roots (line 17). For a new answer to be in the top *k*, its root must have a combined distance that is not greater than τ_{prune} (line 13).

The time complexity of Algorithm 2 is O (|*C*|×*m*×|*PriorityList'*|), where |*C*| is the number of possible keyword element combinations, *m* is the number of the matched keyword elements and |*PriorityList'*| is the average size of list PriorityList.

Termination. The search terminates when one of the following conditions is satisfied: a) the root of an answer must have combined distance greater than τ_{prune} and |*V* | >*k*; b) the paths with a given length d_{max} have been explored for all designated priority lists that contain the keyword elements.

TRANSLATION OF TOP-*K* SUBGRAPHS

In this section, the derived subgraphs by the last section are mapped to SPARQL queries. The property path of SPARQL 1.1 is used to construct the query.

Property Path of SPARQL 1.1

In this subsection, we describe the property paths of SPARQL 1.1 (Prud'hommeaux et al., 2013). Property paths are a new feature introduced in SPARQL 1.1 as a way of adding navigational power for querying over RDF graphs. A property path is a possible route through a graph between two graph vertices. A trivial case is a property path with exact length 1, which is actually a triple pattern. The ends of the paths may be RDF terms or variables. Note that variables cannot be used as part of a path itself, which can only be the ends. Property paths enable more concise expressions for some SPARQL basic graph patterns. Also, they enable to match connectivity of two resources by an arbitrary length path. If there is a pair (*a, b*) of vertices in the graph such that there is a path from *a* to *b*, the sequence of edge labels belongs (as a string) to the regular language defined by the expression.

Let us look at an example. Given a query to find the university that "AssistantProfessor0" works for, we have the following comparative SPARQL queries (Than et al., 2009):

```
select ?x ?y ?z {
       ?x ub:worksFor ?y.
       ?y ub:subOrganizationOf ?z.
       ?x ub:name "AssistantProfessor0".
       ?x rdf:type ub:AssistantProfessor.
       ?y rdf:type ub:Department.
       ?z rdf:type ub:University.}
```

Though there are no direct path from "AssistantProfessor0" to the university, we can utilize the sequence path of SPARQL property paths. Thus, we have the following query:

```
select ?x ?z {
       ?x ub:worksFor/ub:subOrganizationOf ?z.
       ?x ub:name "AssistantProfessor0".
       ?z rdf:type ub:Univeristy.
       ?x rdf:type ub:AssistantProfessor.}
```

Here "*ub*:*worksFor/ub*:*subOrganizationOf*" is the property path of SPARQL. As far as we are concerned, the property paths of SPARQL 1.1 make the translation of keyword queries to SPARQL queries easier and more efficient. With the SPARQL 1.1 specification (Prud'hommeaux et al., 2013), we have the following definition of property path expressions:

Definition 3: Let e:= (iri)|(^e)|(e_1/e_2)|(e_1|e_2)|(e^+)|(e^*)|(e?)| (!{iri_1|...|iri_k})|(!{^iri_1|...|^iri_k}). Here *iri, iri_1, ..., iri_k* are IRIs (Internationalized Resource Identifiers); the expressions starting with "!" are called negated property sets; "^" e is the reverse path construction; e_1/e_2 is a sequence path of e_1 followed by e_2; e_1|e_2 is an alternative path of e_1 or e_2.

Note that we do not consider e^+, e^*, $e^?$, !{iri_1|...|iri_k} and !{^iri_1|...|^iri_k} because these cases are seldom encountered in the keyword search with our approach.

Top-K Subgraph Mapping

A match of SPARQL query on a subgraph is a mapping from edges and vertices of the subgraph to property paths and variables of the SPARQL query. Such a mapping is illustrated in Algorithm 3. Algorithm 3 consists of three main steps:

- Step 1, for each element of *M*, defines a variable *q* initialized with ¨select distinct * where {¨;
- Step 2 translates the vertices and edges from the connecting vertex in *M* to *m* key-words; and
- Step 3 combines *q* with } and inserts *q* into answers *A*.

Here *M* is a set of (subject, type) pairs. The time complexity of this algorithm is *O* (*k***m*), where *k* is the number of generated subgraphs and *m* is the number of keywords. With Algorithm 3, a complete mapping of such a subgraph to a SPARQL query can be obtained:

- **Processing Vertices:** All the vertices in top-*k* subgraphs obtained are taken as distinguished variables. Then the select clause is "select distinct *" to avoid generating duplicated results by adding "distinct" (line 2). Var(*v*) returns the SPARQL query variable denoted by "?*v*" and also denoted by "$*v*". We use "?" for marking the query variable;
- **Mapping Subgraph Paths:** *M*[*u*] stores a list of pairs (*vId*, *path*) from the connecting vertex to keyword elements, where *vId* indicates the id of vertex and the path indicates the property path. Each subgraph path stored in *M*[*u*] is equivalent to the property path of the SPARQL query. When translating top-*k* subgraphs into SPARQL queries, we transform the subgraph path to the corresponding triple pattern expression of SPARQL by adding query variables (line 4);
- **Mapping Edges with E_A Relations:** Edge E_A is used to connect the keyword to the entity. Each edge like this relationship is translated into two triple pattern expressions of SPARQL: (var(*x*) property keyword) and (var(*x*) rdf:type $name_{type}$) (line 5-6). The property can be derived from a hashMap $Map_{property}$ by the opst index, and $name_{type}$ can be derived from Map_{type}. The items in $Map_{property}$ are in the form of (*keyword* → *subject*, *property*). Therefore, we can get the property if the keyword (object) and the subject are given. Suppose that there exists a keyword "Stanford" and its subject "ID2" as shown in Table 1. Then, by $Map_{property}$ and Map_{type}, we obtain the property "dhdFrom". Thus, the triple pattern expressions for the SPARQL query are '*var(x) phdFrom* "*Stanford*". 'and '*var(x) rdf:type AssistantProfessor.*' . Var (*x*) generates SPARQL query variables (line 10-11).

Algorithm 3. Translating the subgraph to the SPARQL query

Input: *M*; a set of (*subject*, *type*) pairs Map_{type}; a set of (*keyword* → *subject*, *property*) pairs $Map_{property}$
Output: top-*k* SPARQL queries *A*

```
1   foreach u ∈ M do
2       q ← select distinct * where {;
3       for j ∈ [1, m] do
4           q←q+var (u)  M[u][j].path        var (M[u][j].vId).;
5           q← q+var (M[u][j].vId) Map_property.get(k_j →M[u][j].vId)        k_j.;
6           q← q+var (M[u][j].vId)  rdf:type  Map_type.get(M[u][j].vId).;
7       q ← q+};
8       A.add(q);
9   return A;
10  var(v) begin
11      return ?v
```

Query Pattern Graph

In this subsection, we present the query pattern graphs of SPARQL with the property paths by our approach. Two example query pattern graphs for "*Q*: Research5 FullProfessor9 Publication17" are shown in Figure 6 (a) and (b), respectively. The property path connecting query variable *?523* and query variable *?533* in Figure 6 (a) is "^*ub*:*publicationAuthor*", which is a reverse path with exact length 1. Here *?523* is the connecting vertex of Figure 6 (a) and (b). Property path "^*ub:advisor/^ub:publicationAuthor*" from a variable *?523* to *?438* is a sequence path of SPARQL query, in which "^" is a reverse property path operator of SPARQL.

With our approach, the pattern graphs of Figure 6 (a) and (b) are respectively mapped to the two SPARQL queries Qg_1 and Qg_2 as follows. Of course, the query *Q* can generate many query graphs. Here, we only illustrate the examples Qg_1 and Qg_2.

Qg_1:

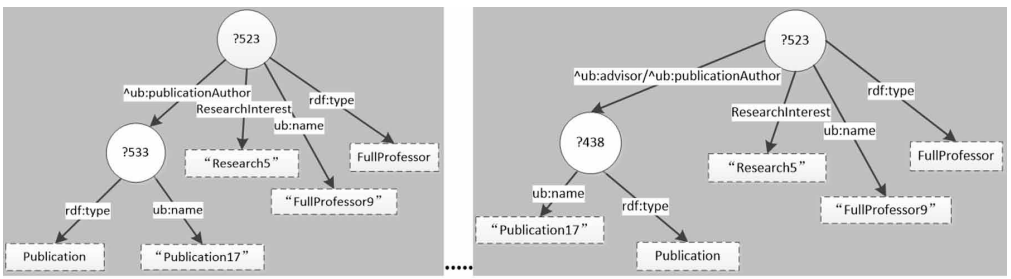
```
prefix ub:<http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select distinct * where {
?523 ub:researchInterest "Research5".
?523 rdf:type ub:FullProfessor.
?523 ub:name "FullProfessor9".
?523 ^ub:publicationAuthor ?533.
?533 ub:name "Publication17".
?533 rdf:type ub:Publication.}
```

Qg_2:

```
prefix ub:<http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select distinct * where {
?523 rdf:type ub:FullProfessor.
?523 ub:researchInterest "Research5".
?523 rdf:type ub:FullProfessor.
?523 ub:name "FullProfessor9".
?523 ^ub:advisor/^ub:publicationAuthor ?438.
?438 ub:name "Publication17".
?438 rdf:type ub:Publication.}
```

In (Than et al., 2009), a conjunctive query is simply a conjunction of all the predicates and the variables generated from a given subgraph. However, we exploit property paths of SPARQL,

Figure 6. Query pattern graph examples for Q2



(a) query graph Q2-1 ..... (b) query graph Q2-2¶

combining property path pattern vectors from the connecting vertex to keyword elements without the need to translate every vertex and every edge of the subgraph.

EXPERIMENTS

In this section, we use two datasets DBLP and LUBM to assess the efficiency, effectiveness, and usability of our approach proposed in this paper. DBLP (http://lsdis.cs.uga.edu/projects/semdis/swetodblp), which contains 26M triples about computer science publications, has been commonly used for keyword search evaluation. In addition, LUBM (Lehigh University benchmark) (Guo, Pan, and Heflin, 2005) is applied to ensure the validity of our experimental results. Using the data generator, we create LUBM 50. We conduct experiments on an SMP machine with 2.93GHz Intel Dual Core processors and 4GB memory. All experiments are implemented in Java. We utilize the SPARQL search engine (Broekstra, Kampman, and Frank, 2002) to preprocess RDF data and execute SPARQL queries produced in the end.

Table 5 shows the sample queries from LUBM and DBLP datasets. For the LUBM dataset, all keywords are chosen from the first university except for keywords "Publication17" and "Publication18". For these two indicated keywords, we select one copy of each publication from the first university and then randomly pick the rest of them from other universities to simulate the cases in a real dataset, in which not all the keywords in a query are close to each other.

Index Performance

We would investigate the size and the time of the index. Here the index includes the forward path index and the inter-entity relationship summary index. It is shown in Figure 7 that DBLP dataset contains more entities than LUBM dataset. In addition, it is shown in Figure 8, with the size of two datasets LUBM and DBLP varying, the size of inter-entity relationship summary for LUBM is becoming bigger than that for DBLP because LUBM dataset has more structures than DBLP dataset (e.g., more relations between inter-entities). For the dataset DBLP, Figure 9 shows that comparing to the size of the forward path index, the inter-entity relationship summary results in at least two orders with less size. Figure 9 also shows the most overheads of our index come from the forward

Table 5. Query examples

Query	Keywords
Q_1	Lecturer6 Publication19
Q_2	Research5 Fullprofessor9 Publication17
Q_3	FullProfessor9 GraduateStudent0 Publication18 Lecturer6
Q_4	Department0 GraduateStudent1 Publication18 AssociateProfessor0
Q_5	Department12 FullProfessor0
Q_6	GraduateStudent0 Department0 GraduateCourse16
Q_7	1999 springer
Q_8	1995 dynamic optimal
Q_9	ADMA Genetic multiple sequences
Q_{10}	2005-09-08, text classification
Q_{11}	semantic tagging aaai
Q_{12}	Gaussian Weighted Histogram IEEE

Figure 7. Number of entities: LUBM versus DBLP

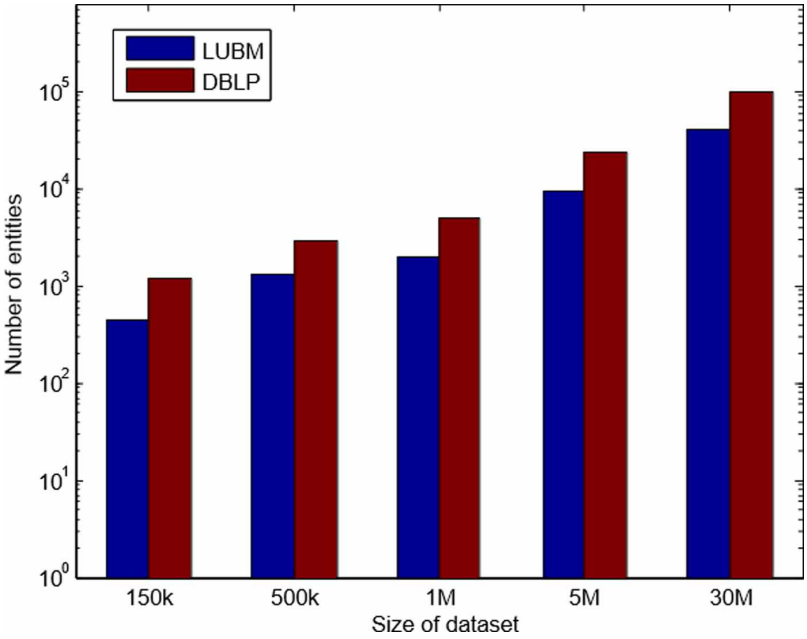
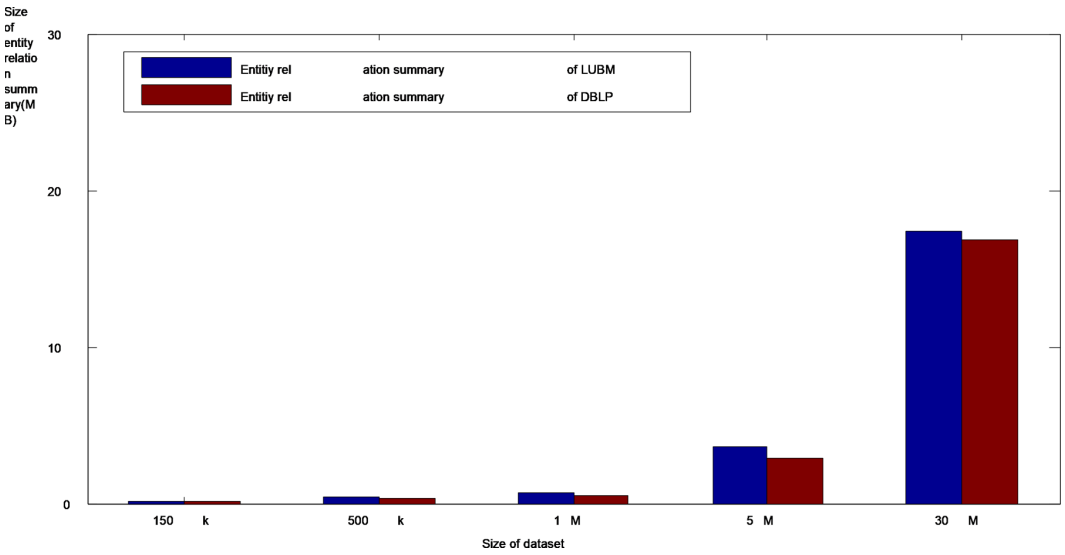


Figure 8. Size for inter-entity relationship summary: LUBM versus DBLP



path index. Figure 10 validates that building the forward path index takes more time than building the inter-entity relationship summary.

The overheads will be enormous if we index all entity pairs. So, we only index the shortest paths which are less than d_{max} steps. Then the size and time of the index can be controlled by adjusting d_{max} value. Figure 11 shows the time of building the forward path index with different d_{max}. Note that the query response time is affected by some other factors, such as the loading time of the forward path

Figure 9. Size: Inter-entity relationship summary versus forward path index for DBLP

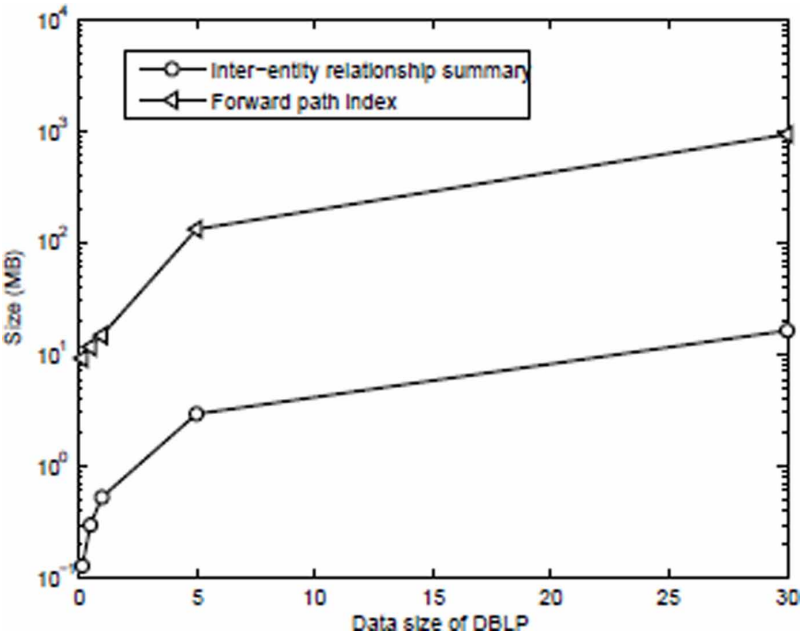
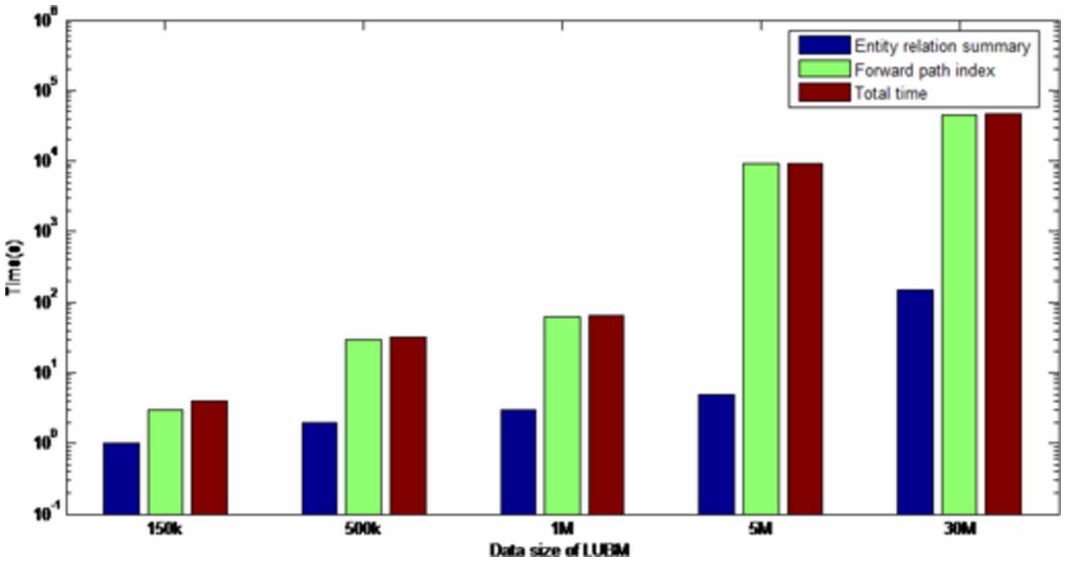


Figure 10. Time for index construction

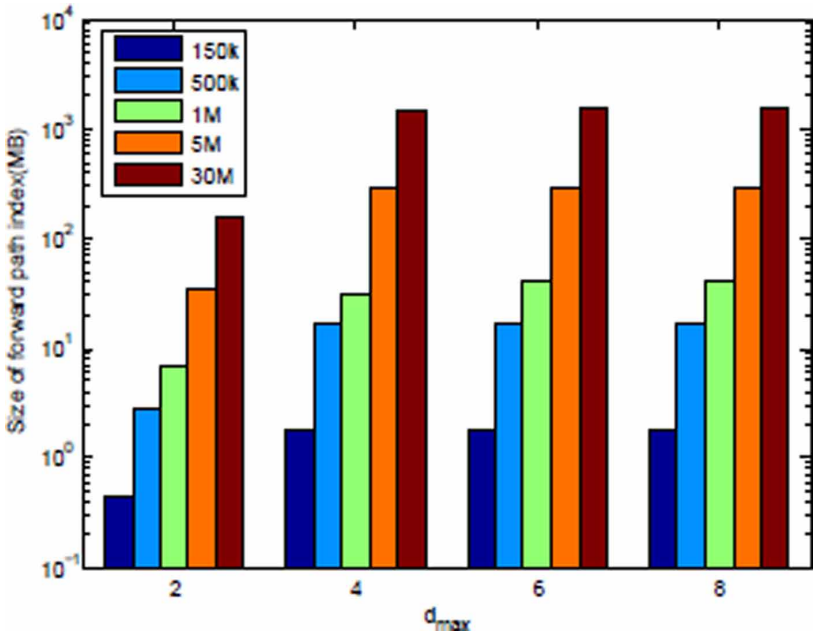


index and the inter-entity relationship summary, the time of constructing SPARQL queries, the time of executing the SPARQL queries produced, and etc.

Prioritization Mechanism Performance

To ensure the validity of our prioritization mechanism on search over the inter-entity relationship summary, we use two metrics shown in Table 6 for comparison, which are respectively the number

Figure 11. Impact of *dmax* to the size of the forward path index



of vertices explored and the time taken when top-*k* subgraphs are generated by the backward search algorithm and our approach.

In Table 6, Column 2 shows the number of occurrences for the keywords being queried in our respective dataset. We compare the Backward search with our approach and present two columns of numbers that give performance measure ratios. As shown in Column 5, our approach visits significantly

Table 6. Our approach vs. Backward search on the sample queries

Query	#Keyword Vertices	Nodes Explored			Gen Time		
		Backward	*Ours*	*Bkwd/Ours Ratios*	*Backward*	*Ours*	*Bkwd/Ours Ratios*
Q_1	(20, 13)	1,138	764	1.49	0.07	0.06	1.17
Q_2	(9, 4, 83)	7,203	1,510	4.77	0.62	0.13	4.77
Q_3	(4, 15, 40, 5)	992	670	1.48	0.43	0.37	1.16
Q_4	(1, 15, 40, 15)	3,652	920	3.97	1.21	0.41	2.95
Q_5	(1, 16)	7,207	364	19.80	0.16	0.05	3.2
Q_6	(1, 1, 16)	3360	703	4.78	0.73	0.72	1.01
Q_7	(1150, 2584)	442,780	65,890	6.72	2.72	1.23	2.21
Q_8	(976, 61, 18)	487,167	74,490	6.54	4.30	1.37	3.14
Q_9	(220, 183, 220, 15)	340,100	122,338	2.78	4.61	2.25	2.05
Q_{10}	(97, 5)	706	450	1.57	0.10	0.09	1.11
Q_{11}	(40, 1, 4011)	2,367,424	98,030	24.15	2.48	2.03	1.22
Q_{12}	(17, 43, 13, 3156)	1,939,430	120,163	16.14	5.30	2.21	2.40

fewer vertices explored. In addition, as shown in Column 8, our approach takes less time than the Backward search when the top-*k* subgraphs are obtained. It is shown in Table 6 that, comparing the Backward search with our approach, query Q_5 has higher ratios of vertices explored and generating time. The hub vertex "*department*12" has a large number of fan-in vertices. Then the backward search algorithm explores a substantial number of vertices in order to find answers. Queries Q_{11} and Q_{12} have the same cases as Q_5.

Figure 12 shows the number of nodes explored of the Backward search versus our approach. Figure 13 shows the generating time of the Backward search versus our approach. To clearly show the contrast to the nodes explored and generating time ratios of the Backward search versus our approach, a broken line graph as shown in Figure 14 compares our approach with the Backward search.

Query Performance

First, we investigate the query response time for queries from Q_1 to Q_{12} on the datasets LUBM and DBLP. Computing the query time is similar to that in (Le, Li, Kementsietsidis, and Duan, 2014). The query time starts from entering keywords until getting top-*k* query results. The query time contains the time of producing top-*k* SPARQL queries and the time of executing top-*k* SPARQL queries on a SPARQL search engine. The query time for queries Q_1-Q_{12} is shown in Figure 15. From Figure 15, we observe that the query response time for each keyword search consists of not only the time of generating top-*k* SPARQL queries but also the time of executing top-*k* SPARQL queries. And the time of executing top-*k* SPARQL queries is very shorter than the time of generating top-*k* SPARQL queries by the existing SPARQL search engine. It is also shown in Figure 15 that, with our approach, the difference in the time of executing top-*k* SPARQL queries is relatively moderate. So, the query response performance is mainly determined by the time of generating top-*k* SPARQL queries. The efficiency of a keyword search is determined by a collection of factors (He et al., 2007) and there is

Figure 12. Number of nodes explored: Backward search versus our approach

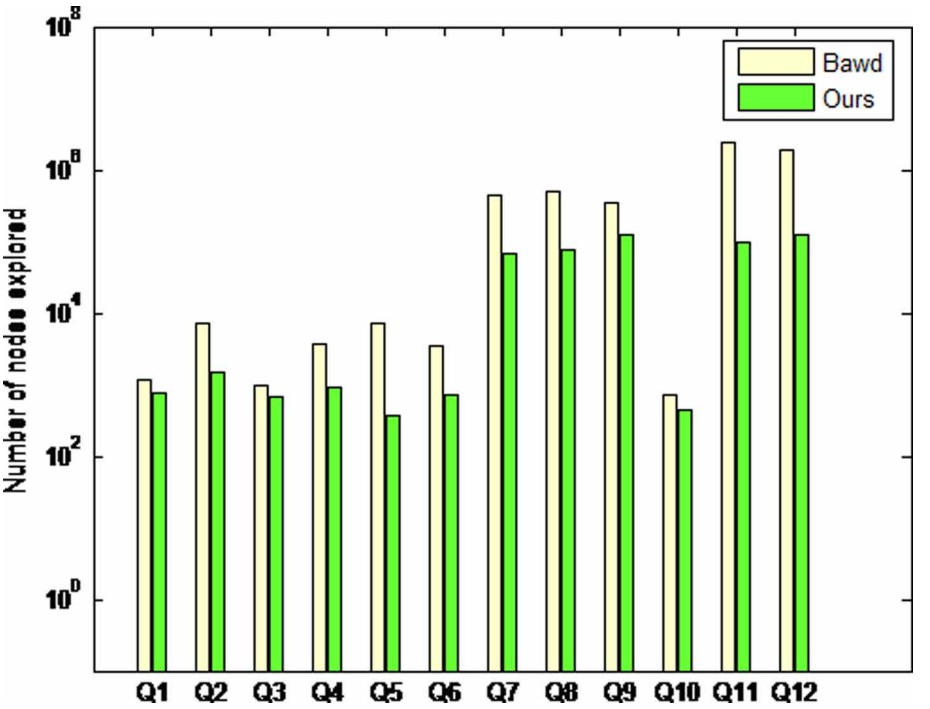


Figure 13. Generating time: Backward search versus our approach

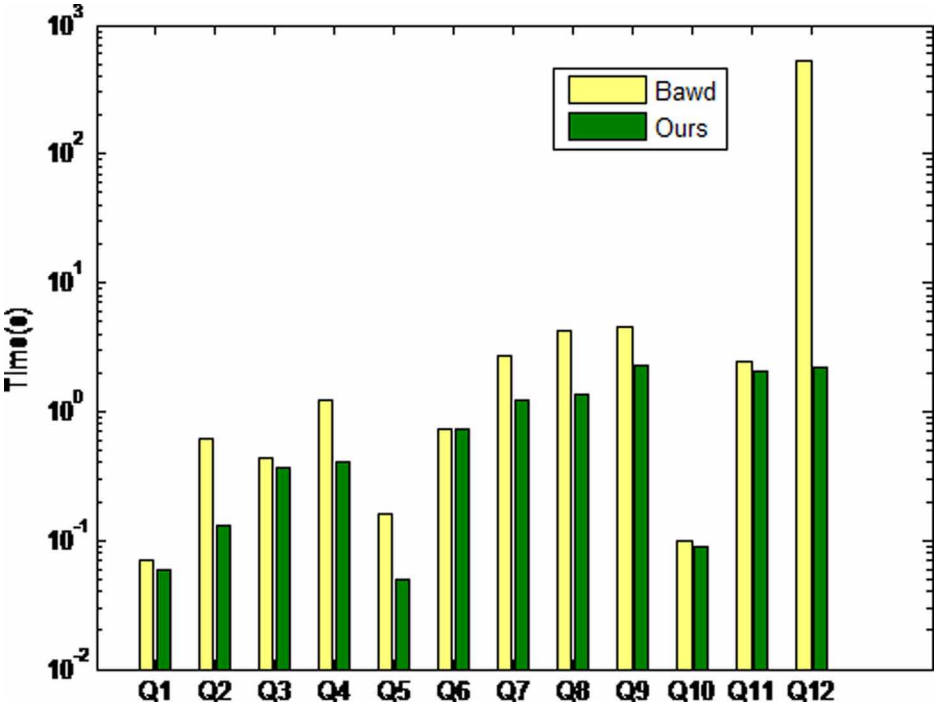


Figure 14. Ratio: Backward search versus our approach

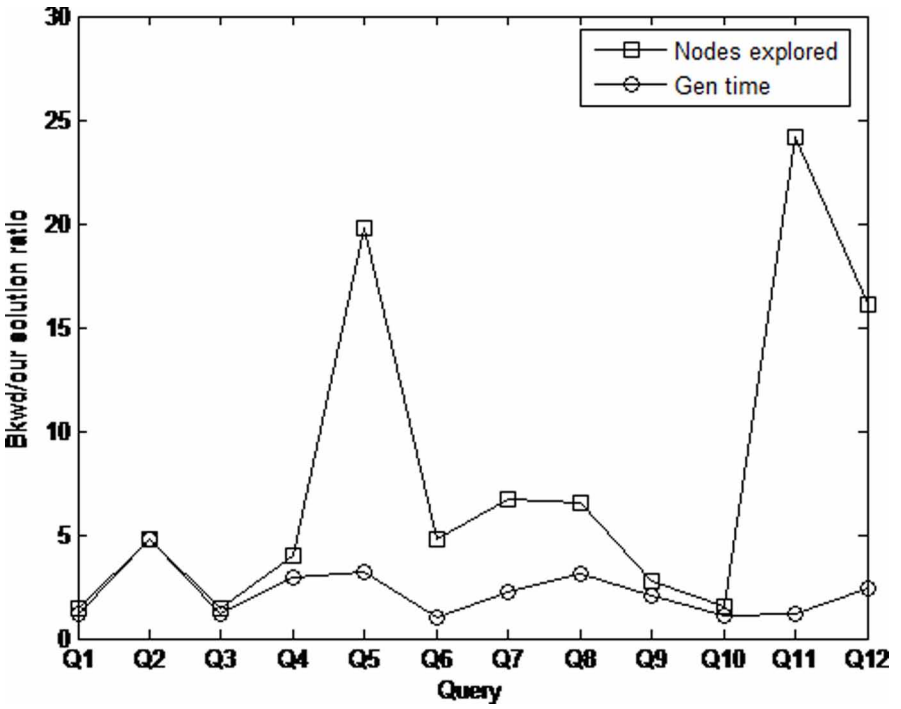
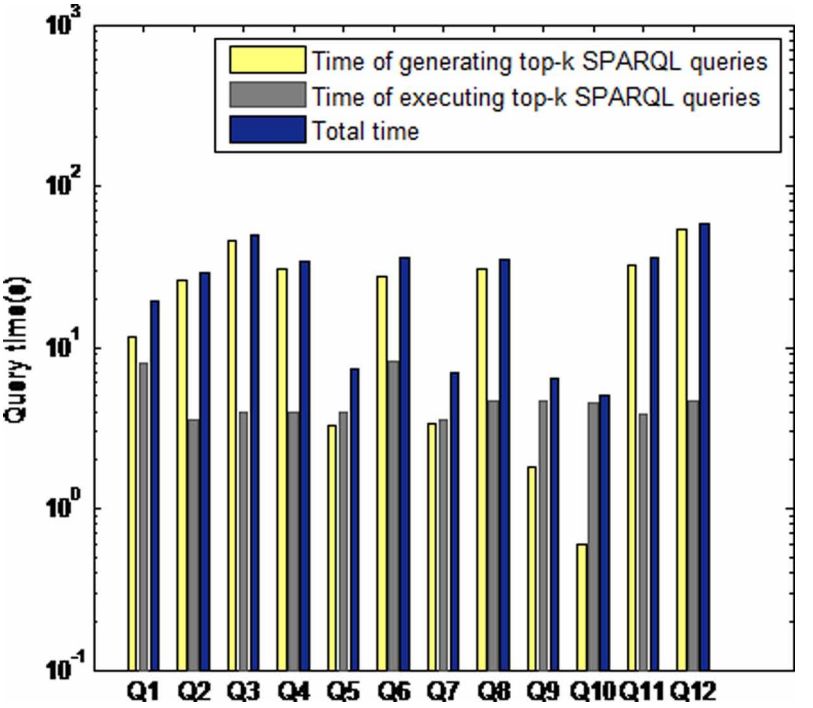


Figure 15. Query time



no single factor being the most deterministic one. With the approach for RDF keyword search based on the keywords-to-SPARQL translation, we can exploit the optimization capabilities of existing search engines to improve the query performance. It has been proved in (Than et al., 2009; Lin et al., 2018) that keyword search based on translation is faster than direct keyword query answering. Furthermore, to reduce the time of generating top-k SPARQL queries, we exploit the prioritization mechanism to prune the search space.

Now we investigate the impact of k on query performance. Figure 16 shows the average response time of 30 queries with length 2-4 on the LUBM dataset, in which $k = 10$, $k = 15$ and $k = 20$ are used. Note that the query time increases linearly when k becomes larger because more time of generating query and more time of executing query are required. In addition, the impact of query length on query performance is minimal when k is 10. The impact of query length on query performance gets larger when a bigger k is used. Hence, we set k=10 in our approach.

Finally, we investigate the search performance of our approach by comparing our approach with the existing approaches proposed in (Le et al., 2014) (denoted as SUMM), in which we perform the same queries and the time is in log scale. Both SUMM and our approach have provable guarantees on the correctness of query results. It is shown in Figure 17 that the query time of our approach has a tiny variation when we vary queries from Q_1 to Q_{12}. That is, our query performance is very steady even if the query time of our approach for Q_1, Q_4 and Q_5 are longer than that in (Le et al., 2014). For Q_2, Q_3 and Q_6-Q_{12}, we observe that our approach performs better than SUMM since SUMM uses a lower bound to decide when to terminate. As a result, SUMM must access more data than what is necessary to correctly answer queries. In addition, our approach can be done almost completely in memory as the summary index for expansion are lightweight and can be cached in memory for query evaluation. We also note from Figure 17 that the query length has little influence on the search performance of our approach. In other words, the query with shorter length does not always take shorter query time for our approach.

Figure 16. Impact of k to query time

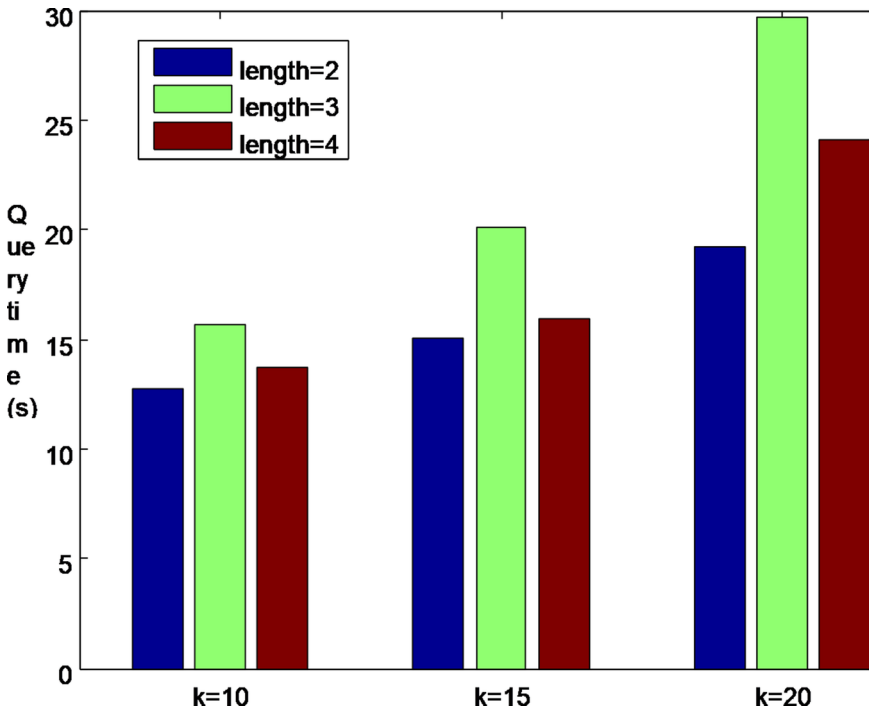
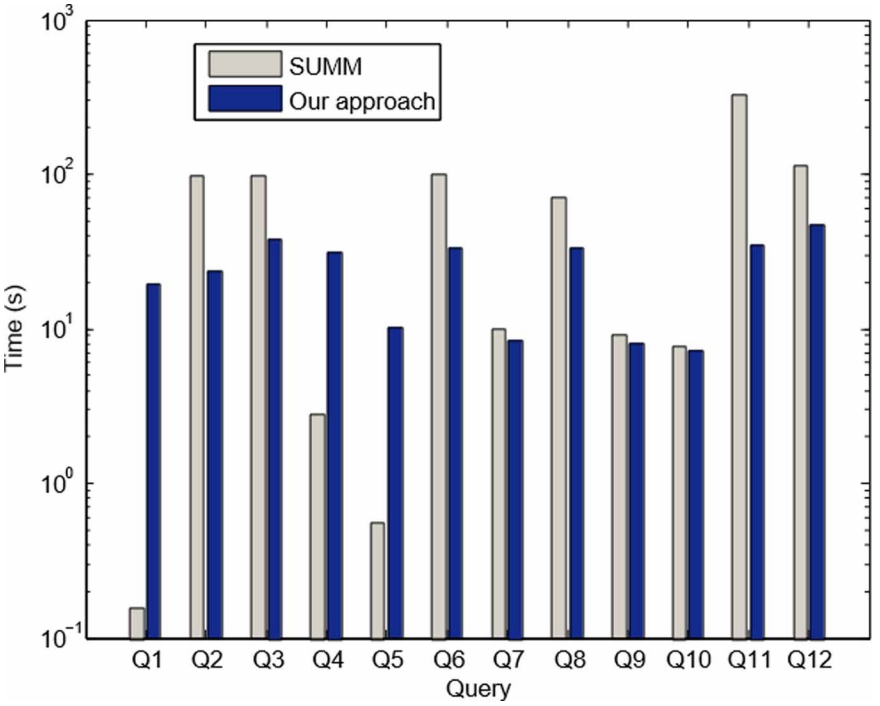


Figure 17. Query performance -- Our approach versus SUMM



CONCLUSION

Keyword search has been a crucial means for exploring massive RDF data. In this paper, we propose a new approach for RDF keyword search based on the keywords-to-SPARQL translation. Aiming to explore the minimum number of vertices in finding the top-*k* subgraphs that connect all keyword elements, we apply a prioritization mechanism to reduce the search space over the summary, which considers the degree of the vertex to be visited as well as the distance from the keyword element. Starting from the priority list that contains the keyword elements and is built in advance, we use the forward path index to search forward and derive the top-*k* subgraphs. The top-*k* subgraphs are then translated to top-*k* SPARQL queries with the property paths of SPARQL 1.1. Finally, the top-*k* SPARQL queries are executed by a SPARQL search engine and the query results are returned to the user. Our approach is portable, efficient and scalable by experiments on both RDF benchmark and real RDF datasets. Comparing our approach to other prior approaches for RDF keyword search based on keywords-to-SPARQL translation, our approach applies a complete inter-entity summary for getting complete query answers and a prioritization scheme for reducing the search space. In addition, our approach utilizes SPARQL 1.1 in the translated SPARQL queries for easy translation. In the future, we are interested in optimizing the translated SPARQL queries by property path indexes in order to improve SPARQL query efficiency.

ACKNOWLEDGMENT

This work was supported by the *National Natural Science Foundation of China* (61772269 and 61370075).

REFERENCES

Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S., & Bombay, I. I. T. (2002). Keyword searching and browsing in databases using banks. In *IEEE International Conference on Data Engineering* (pp. 431-440). doi:10.1109/ICDE.2002.994756

Broekstra, J., Kampman, A., & Harmelen, F. V. (2002). Sesame: A generic architecture for storing and querying RDF and RDF schema. In *International Semantic Web Conference* (pp. 54-68).

Cabrio, E., Cojann, J., Aprosio, A. P., Magnini, B., Lavelli, A., & Gandon, F. (2012). Qakis: An open domain qa system based on relational patterns. In *International Semantic Web Conference (Posters & Demos)* (pp. 9-12).

Chen, Y., Wang, W., Liu, Z. Y., & Lin, X. M. (2009). Keyword search on structured and semistructured data. In *ACM SIGMOD International Conference on Management of Data* (pp. 1005-1010). doi:10.1145/1559845.1559966

Dass, A., Aksoy, C., Dimitriou, A., Theodoratos, D., & Wu, X. Y. (2016). Diversifying the results of keyword queries on linked data. In *International Conference on Web Information System Engineering* (pp. 199-207). doi:10.1007/978-3-319-48740-3_14

Elbassuoni, S., Ramanath, M., Schenkel, R., & Weikum, G. (2010). Searching RDF graphs with SPARQL and keywords. *A Quarterly Bulletin of the Computer Society of the IEEE Technical Committee on Data Engineering*, *33*(1), 16–24.

Fakas, G. J. (2011). A novel keyword search paradigm in relational databases: Object summaries. *Data & Knowledge Engineering*, *70*(2), 208–229. doi:10.1016/j.datak.2010.11.003

Fu, H., Gao, S., & Anyanwu, K. (2011). Cosi: Context-sensitive keyword query interpretation on RDF databases. In *International Conference on World Wide Web* (pp. 209-212). doi:10.1145/1963192.1963291

García, G., Izquierdo, Y., Menendez, E., Dartayre, F., & Marco, A. (2017). Casanova: RDF Keyword-based Query Technology Meets a Real-World Dataset. In *Proceedings of the 20th International Conference on Extending Database Technology* (pp. 656-667).

Garrison, L., Stevens, R., & Jocuns, A. (2004). Efficient RDF storage and retrieval in jena2. *Exploiting Hyperlinks*, *51*(2), 35–43.

Gkirtzou, K., Papastefanatos, G., & Dalamagas, T. (2015). RDF keyword search based on keywords-to-SPARQL translation. In *International Workshop on Novel Web Search Interface and Systems* (pp. 3-5). doi:10.1145/2810355.2810357

Guo, Y., Pan, Z. X., & Heflin, J. (2005). LUBM: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, *3*(2-3), 158–182. doi:10.1016/j.websem.2005.06.005

Han, S., Zou, L., Yu, J. X., & Zhao, D. Y. (2017). Keyword Search on RDF Graphs - A Query Graph Assembly Approach. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (pp. 227-236). doi:10.1145/3132847.3132957

Harth, A., & Decker, S. (2005). Optimized index structures for querying RDF from the web. In *Latin American Web Congress* (pp. 71-80). doi:10.1109/LAWEB.2005.25

He, H., Wang, H. X., Yang, J., & Yu, S. P. (2007). Blinks: ranked keyword searches on graphs. In *ACM SIGMOD International Conference on Management of Data* (pp. 305-316).

Hulgeri, A., & Nakhe, C. (2002). Keyword searching and browsing in databases using banks. In *IEEE International Conference on Data Engineering* (pp. 431-440).

Izquierdo, Y. T., García, G. M., Menendez, E. S., Casanova, M. A., Dartayre, F., & Levy, C. H. (2018). QUIOW: A keyword-based query processing tool for RDF datasets and relational Databases. In *Proceedings of the 29th International Conference on Database and Expert Systems Applications* (pp. 259-269). doi:10.1007/978-3-319-98812-2_22

Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., & Karambelkar, H. (2005). Bidirectional expansion for keyword search on graph databases. In *International Conference on Very Large Data Bases* (pp. 505-516).

Kargar, M., & An, A. (2011). Keyword search in graphs: Finding r-cliques. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *4*(10), 681–692. doi:10.14778/2021017.2021025

Kasneci, G., Ramanath, M., Sozio, M., Suchanek, F. M., & Weikum, G. (2009). Star: steiner-tree approximation in relationship graphs. In *IEEE International Conference on Data Engineering* (pp. 868-879).

Klyne, G., Carroll, J. J., & McBride, B. (2004). *Resource description framework (RDF): Concepts and abstract syntax*. World Wide Web Consortium Recommendation.

Ladwig, G., & Tran, T. (2010). Combining Query Translation with Query Answering for Efficient Keyword Search. *In Extended Semantic Web Conference* (pp. 288-303). doi:10.1007/978-3-642-13489-0_20

Le, W., Li, F. F., Kementsietsidis, A., & Duan, S. (2014). Scalable keyword search on large RDF data. *IEEE Transactions on Knowledge and Data Engineering*, *26*(11), 2774–2788. doi:10.1109/TKDE.2014.2302294

Li, G. F., Yan, L., & Ma, Z. M. (2019). Pattern match query over fuzzy RDF graph. *Knowledge-Based Systems*, *165*, 460–473. doi:10.1016/j.knosys.2018.12.014

Li, G. L., Ooi, B. C., Feng, J. H., Wang, J. Y., & Zhou, L. Z. (2008). Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *ACM SIGMOD International Conference on Management of Data* (pp. 903-914). doi:10.1145/1376616.1376706

Lian, X., Hoyos, E. D., Chebotko, A., Fu, B., & Reilly, C. (2013). K-nearest keyword search in RDF graphs. *Journal of Web Semantics*, *22*(8), 40–56. doi:10.1016/j.websem.2013.08.001

Lin, X. Q., Ma, Z. M., & Yan, L. (2018). RDF Keyword Search Using a Type-based Summary. *Journal of Information Science and Engineering*, *34*(2), 489–504.

Liu, F., Yu, C., Meng, W. Y., & Chowdhury, A. (2006). Effective keyword search in relational databases. In *ACM SIGMOD International Conference* (pp. 563-574).

Ma, R., Jia, X. Y., Cheng, J. W., & Angryk, R. A. (2016). SPARQL queries on RDF with fuzzy constraints and preferences. *Journal of Intelligent & Fuzzy Systems*, *30*(1), 183–195. doi:10.3233/IFS-151745

Ma, Z., Capretz, M. A. M., & Yan, L. (2016). Storing massive Resource Description Framework (RDF) data: A survey. *The Knowledge Engineering Review*, *31*(4), 391–413. doi:10.1017/S0269888916000217

Neumann, T., & Weikum, G. (2008). RDF-3x: A risc-style engine for RDF. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *1*(1), 647–659. doi:10.14778/1453856.1453927

Neumann, T., & Weikum, G. (2010). xrdf3x: Fast querying, high update rates, and consistency for RDF databases. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *3*(1-2), 256–263. doi:10.14778/1920841.1920877

Peng, P., Lei, Z., & Zheng, Q. (2017). Answering top-k query combined keywords and structural queries on RDF graphs. *Information Systems*, *67*, 19–35. doi:10.1016/j.is.2017.03.002

Prud'hommeaux, E., & Seaborne, A. (2013). SPARQL 1.1 query language. *W3C*.

Rivero, C. R., Hernández, I., Ruiz, D., & Corchuelo, R. (2015). Discovering and Analysing Ontological Models From Big RDF Data. *Journal of Database Management*, *26*(2), 48–61. doi:10.4018/JDM.2015040104

Sinha, S. B., Lu, X. G., & Theodoratos, D. (2018). Personalized Keyword Search on Large RDF Graphs based on Pattern Graph Similarity. In *Proceedings of the 22nd International Database Engineering & Applications Symposium* (pp. 12-21). doi:10.1145/3216122.3216167

Sun, C., Chan, C. Y., & Goenka, A. K. (2007). Multiway slca-based keyword search in xml data. In *International Conference on World Wide Web* (pp. 1043-1052). doi:10.1145/1242572.1242713

Taha, K., & Elmasri, R. (2009). OOXKSearch: A Search Engine for Answering XML Keyword and Loosely Structured Queries Using OO Techniques. *Journal of Database Management*, *20*(3), 18–50. doi:10.4018/jdm.2009070102

Tran, T., Wang, H. F., Rudolph, S., & Cimiano, P. (2009). Top-k exploration of query candidates for efficient keyword search on graphshaped (RDF) data. In *IEEE International Conference on Data Engineering* (pp. 405-416).

Vasilyeva, E., Thiele, M., Bornhövd, C., & Lehner, W. (2015). Considering User Intention in Differential Graph Queries. *Journal of Database Management*, *26*(3), 21–40. doi:10.4018/JDM.2015070102

Virgilio, R. D., Cappellari, P., Maccioni, A., & Torlone, R. (2012). Path-Oriented Keyword Search Query over RDF. In *Semantic Search over the Web* (pp. 81–107). Springer.

Wang, D., Zou, L., Pan, W. Q., & Zhao, D. Y. (2012). Keyword Graph: Answering Keyword Search over Large Graphs. In *International Conference on Advanced Data Mining and Applications* (pp. 635-649). doi:10.1007/978-3-642-35527-1_53

Wang, H. F., Tran, T., Liu, C., & Fu, L. Y. (2011). Lightweight integration of ir and db for scalable hybrid search with integrated ranking support. *Journal of Web Semantics*, *9*(4), 490–503. doi:10.1016/j.websem.2011.08.002

Weiss, C., Karras, P., & Bernstein, A. (2008). Hexastore: Sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, *1*(1), 1008–1019. doi:10.14778/1453856.1453965

Wen, Y. L., Jin, Y. D., & Yuan, X. J. (2018). KAT: Keywords-to-SPARQL Translation Over RDF Graphs. In *Proceedings of the 23rd International Conference on Database Systems for Advanced Applications* (pp. 802-810).

Wylot, M., Hauswirth, M., Cudré-Mauroux, P. & Sakr, S. (2018). RDF Data Storage and Query Processing Schemes: A Survey. *ACM Computing Surveys, 51*(4), 84:1-84:36.

Yan, L., Ma, R., Li, D., & Cheng, J. (2017). RDF approximate queries based on semantic similarity. *Computing*, *99*(5), 481–491. doi:10.1007/s00607-017-0554-9

Zenz, G., Zhou, X., Minack, E., Siberski, W., & Nejdl, W. (2009). From keywords to semantic queries-incremental query construction on the semantic web. *Journal of Web Semantics*, *7*(3), 166–176. doi:10.1016/j.websem.2009.07.005

Zheng, W. G., Zou, L., Lian, X., Yu, J. X., Song, S. X., & Zhao, D. Y. (2015). How to build templates for RDF question/answering. In *ACM SIGMOD International Conference* (pp. 1809-1824). doi:10.1145/2723372.2747648

Zhou, Q., Wang, C., Xiong, M., Wang, H. F., & Yu, Y. (2007). Spark: Adapting keyword query to semantic search. In *International Semantic Web Conference and Asian Semantic Web Conference* (pp. 694-707). doi:10.1007/978-3-540-76298-0_50

Zou, L., Huang, R. Z., Wang, H. X., Yu, J. X., He, W. Q., & Zhao, D. Y. (2014). Natural language question answering over RDF: a graph data driven approach. In *ACM SIGMOD International Conference on Management of Data* (pp. 313-324). doi:10.1145/2588555.2610525

Zongmin Ma is currently a full professor at Nanjing University of Aeronautics and Astronautics, China. His research interests include databases, the semantic web, and knowledge representation and reasoning with a special focus on information uncertainty. He has published more than one hundred and seventy papers on these topics. He is also the author of four monographs published by Springer. He is a senior member of the IEEE.

Xiaoqing Lin received her M.S. degree and Ph.D. degree from Shenyang Aerospace University, China and Northeastern University, China. Her research interests include RDF keyword search and the semantic web.

Li Yan is currently a full professor at Nanjing University of Aeronautics and Astronautics, China. Her research interests include fuzzy data modeling and spatiotemporal information management. She has published more than fifty papers on these topics. She is the author of two monographs published by Springer.

Zhen Zhao respectively received his M.S. degree and Ph.D. degree from Dalian Jiaotong University, China and Northeastern University, China. His research interests include fuzzy data management and machine learning.