JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 489-504 (2018)
DOI: 10.6688/JISE.201803_34(2).0011

RDF Keyword Search Using a Type-based Summary*

XIAO-QING LIN[1,3], ZONG-MIN MA[2,+] AND LI YAN[2]
[1]*School of Computer Science and Engineering*
Northeastern University
Shenyang, Liaoning, 110819 P.R. China
[2] *College of Computer Science and Technology*
Nanjing University of Aeronautics and Astronautics
Nanjing, Jiangsu, 211106 P.R. China
[3]*School of Information Engineering*
Eastern Liaoning University
Dandong, Liaoning, 118003 P.R. China

Keyword search enjoys great popularity due to succinctness and easy operability for exploring RDF data. SPARQL has been recommended as the standard query language. Thus, keyword search based on keywords-to-SPARQL attracts more and more attention. However, existing solutions have main limitations that the summary index used for translation is incomplete and thus results returned are wrong or short of some answers. To address the issues, we propose an original RDF keyword search paradigm based on the translation of keywords-to-SPARQL queries. We present a new type-based summary which summarizes all the inter-entity relationships from RDF data. We exploit an efficient search algorithm to quickly find the top-*k* subgraphs connecting all entering keyword elements. Then, a transforming algorithm is leveraged to translate top-*k* subgraphs into top-*k* SPARQL queries that are eventually executed by a SPARQL query engine. The experiments show that our approach takes shorter query response time and more accurate results are achieved than existing techniques.

Keywords: RDF keyword search, SPARQL, type-based summary, query translation, RDF data graph

1. INTRODUCTION

Since the resource description framework (RDF) has been the standard for expressing and exchanging semantic metadata, there are large amounts of continuously growing RDF data from different sources that include the DBLP dataset, the LUBM dataset, the DBpedia dataset, and *etc.* RDF data is a collection of statements, called triples, of the form (*s*, *p*, *o*), where *s* is called the subject, *p* is called the predicate connecting the subject and the object, and *o* is called object. SPARQL [15] is a standard query language for pattern matching against RDF graphs. The syntax resembles SQL, but SPARQL is far more powerful, enabling queries spanning multiple disparate data sources containing heterogeneous semi-structured data.

It is very difficult for non-expert users to issue SPARQL queries. First, users need to know SPARQL query syntax. Second, in order to construct SPARQL queries, the vocabulary of the underlying RDF data are required for users. Keyword search has been a

Received December 23, 2016; revised April 6, 2017; accepted April 30, 2017.
Communicated by Shyi-Ming Chen.
+ Corresponding author: zongminma@nuaa.edu.cn.
* This work was supported by the National Natural Science Foundation of China (61370075 and 61772269).

popular tool of exploring RDF data for non-expert users. Therefore, we present an approach to translate keyword queries into SPARQL queries in order to make it easier for non-expert users to compose SPARQL queries. Users only need to enter keywords and then top-k query results answered can be directly returned. We provide users with a friendly interface for querying RDF data on one hand and a better query performance can be obtained by using existing SPARQL search engines on the other hand.

General keyword search can be divided into two categories, according to different query processing ways. One directly constructs query results from keywords and discussed in the literature [2, 6-8, 11], where the subgraphs containing keywords are located on the RDF data through efficient indexes such as summary indexes, path indexes, and other auxiliary indexes. The other firstly constructs conjunctive queries and then executes the queries by an underlying search engine, studied in the literature [4, 10, 17, 19], where usually three steps, keywords mapping, constructing queries and ranking queries are included. [19] proposes a query construction method by providing users a series of incremental refinement steps to form queries in order to specify query intention of users explicitly. Moreover, it requires users to grasp a little of domain knowledge. There are other works related to keyword search studied in the literature [1, 12-14, 16]. In addition, a new approach provides a general framework [20] by Granular Computing. Just as the name itself stipulates, it deals with representing information in the form of some aggregates and their ensuing processing, which can be applied into the big data (*e.g.*, RDF data) for dynamic clustering.

There are several advantages for the keyword search based on translation [17]. As the summary performed by subgraph exploration is smaller than the real dataset, keyword search based on translation is faster. Second, the generated queries are presented to users in order to express users' intent more clearly. In particular, we can exploit the optimization capabilities of existing search engines to improve query performance. Further, in [17], the SCHEMA method, it has been proved that keyword search based on translation is faster than direct keyword query answering.

To our best knowledge, there exist few works on keyword search based on keywords-to-SPARQL translation including [4, 10, 17]. In [10], RDF graph structure information is extracted to construct structure indexes to produce conjunctive queries. However, structure index overhead is large and more query response time is taken. In [4], the definition for the type-based summary in [4] follows SCHEMA. There exists limitation in SCHEMA that it returns wrong results or no results for certain datasets because the summary of SCHEMA loses some structure information in RDF data as to how one type of the entity is connected to other types of the entity. In SCHEMA, all vertices with the same type are indistinguishably mapped to the same type vertex, only one relationship between the type vertices is kept, as shown in the Example 1. The same case is for the vertex "Lecturer2". The existing problems for SCHEMA are also elaborated in [11] so they present a summary type based by partitioning and summarizing, which produces more index overheads.

Example 1: In Fig. 1, "AssistantProfessor0" is mapped to the type vertex "AssistantProfessor" on the summary, "University389", "University942", and "University643" are all mapped to the same type vertex "University". However, the properties "mastersDegree-

From", "undergraduateDegreeFrom" and "doctoralDegreeFrom" can't be all kept by SCHEMA because SCHEMA can only keep one relationship.

Inspired by the problems, we present a new keyword search algorithm based on key-words-to-SPARQL translation. The property path alternativePath of SPARQL 1.1 is employed to incorporate properties with multi-edges when constructing the type-based summary. As far as we know, this is the first work that applies the property paths of SPARQL to the translation of keywords-to-SPARQL queries. To summarize, our principal contributions are shown below:

(1) Through identifying and addressing limitations in method [4, 17], we design a new type-based inter-entity relationship summary from the underlying RDF graph.
(2) We develop an efficient search algorithm to search for top-*k* subgraphs over the type-based summary not over the entire RDF data graph. Further, a transforming algorithm is leveraged to translate top-*k* subgraphs into top-*k* SPARQL queries.
(3) A detailed series of experiments are conducted on synthetic and real datasets to determine the effectiveness of our approach. Experimental results show that our approach is more efficient and more scalable than existing methods.

In what follows, Section 2 formulates the problem dealt with by our approach and provides an overview of our approach. Details on the type-based summary and the keyword search with type-based summary are tackled in Sections 3 and 4, respectively. Section 5 presents experimental evaluation. Section 6 concludes this paper.

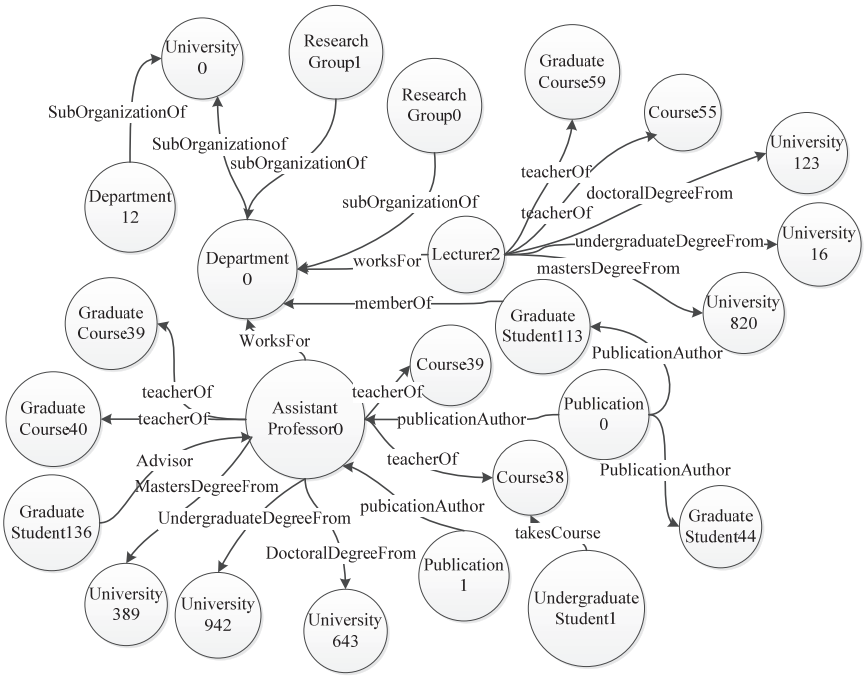


Fig. 1. Inter-entity relationship graph sample for LUBM.

2. PROBLEM DEFINTIONS AND OVERVIEW OF THE APPROACH

Our approach is to construct the SPARQL query by finding inter-entity relationships. A RDF data model is required. The definition for a RDF data graph is similar to that in [17], as shown in Definition 1.

Definition 1: The data graph G is a tuple (V, E) where V is the union of disjoint sets, V_E, V_T and V_K. Here V_E represents the set of entity vertices (*i.e.*, IRIs), V_T represents the set of type vertices, and V_K represents a set of keyword vertices. E is the union of disjoint sets E_R, E_A, and E_T, where E_R is the set of inter-entity edges that connects two entity vertices, E_A is the set of entity keyword edges and E_T is the set of entity-type edges. Each edge is the form $p(v_1, v_2)$ with $v_1, v_2 \in V$ and p is the property connecting v_1 and v_2.

In our scenario, the user query Q_u is a set of keywords (k_1, ..., k_m). The system queries Q_S are SPARQL queries. SPARQL is the standard pattern-matching language for querying RDF [15]. A simple SPARQL query form is shown below.

SELECT X WHERE P

The SPARQL query consists of two clauses, where the select clause identifies the variables to appear in the query and the where clause provides the basic pattern to match against the data graph. P is a graph pattern and X is a distinguished list of query variables denoted by "?" or "$". To construct the SPARQL query, it is a critical problem how to translate subgraph paths to SPARQL queries. Rather than mapping each vertex and each edge of produced top-k subgraphs to SPARQL queries, as SCHEMA does, we directly translate subgraphs to the SPARQL queries through property paths of SPARQL.

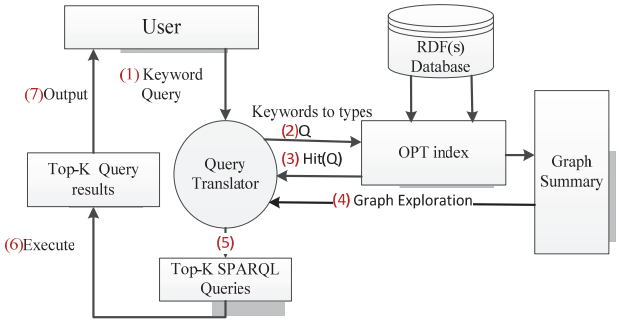


Fig. 2. Architecture and workflow.

To start with an overview of our approach, the following key steps are illustrated in Fig. 2.

- Retrieve each keyword types W_i the keyword k_i belongs to by using an OPT index for a given keyword query $Q = k_1, k_2, ..., k_m$. (Steps 1-3)
- Exploit a graph exploration algorithm to look up top-k matching subgraphs for all possible combinations $C = W_1 \times W_2 \times ... \times W_m = \{c = (w_1, ..., w_m) | w_i \in W_i, i = 1, ..., m\}$. (Step 4)

- Take the hit combination C and top-k matching subgraphs for the query translator to generate a set of top-k SPARQL translations of keyword query Q. (Step 5)
- Execute the top-k SPARQL queries (Step 6) by an underlying SPARQL search engine and return top-k SPARQL query results answered to users. (Step 7)

The edge weight is set to 1. The graph exploration starts from each of matched keyword elements. Then we need to find top-k subgraphs connecting all these keyword elements. The top-k subgraphs are a ranked set by top-k combined distances. We take the assumption that the answer roots are distinct as mentioned in [6, 17]. A subgraph $G'=\{V', E'\}$ is a qualified candidate when there is a root vertex $r \in V'$ that is reachable by $v_i \in V'$ for $i \in [1, m]$ and the minimal subgraph G' has the smallest combined distance $sg = \sum_{i=1}^{m} d(r, v_i)$.

3. TYPE-BASED SUMMARY

We deal with the type-based summary that is the key of the translation. During translating keyword queries to SPARQL queries, the relationships between the entities matched by keywords are needed. The vertex on the type-based summary matched by the keyword is called the keyword element.

The subject of each RDF triple connects the object by the property which denotes a relationship between the subject and the object. The triples with the same subject types and the same object types have same properties in most cases. As for different properties, we incorporate them. According to this observation, we construct a type-based summary by summarizing all the inter-entity type relationships from RDF triples that response to query patterns. Exploration over the summary will be much less than that over a large RDF data graph.

By the perspective of the objective query, we define the type-based summary that serves the translation of subgraphs to SPARQL queries. In [17], all the entities of the same type are mapped to one vertex of its summary. For example, in Fig. 1, for triples, "(AssistantProfessor0 undergraduateDegreeFrom University942)", "(AssistantProfessor0 mastersDegreeFrom University389)" and "(AssistantProfessor0 doctoralDegreeFrom University643)", their subjects are all mapped to "AssistantProfessor" and objects are mapped to "University'. However, only one property of properties "undergraduateDegreeFrom", "mastersDegreeFrom", "doctoralDegreeFrom" can be kept. Whatever property is selected, the other two properties will be lost. In the case, we add alternative relationships between two type vertices of the summary by "|", an alternative path operator of SPARQL 1.1 (see Section 5.2). Thus, the relationships between "AssistantProfessor" and "University" are incorporated into be "mastersDegreeFrom|undergraduateDegree From| doctoralDegreeFrom", as illustrated in Fig. 3.

Definition 2: A type-based triple is a triple $t\ (c_s, p, c_o) \in (I \cup B) \times I \times (I \cup B)$, where the subject and the object of the triple are represented by the type of the subject and the object of the triple, respectively. Given a type-based triple $p'\ (c'_s, c'_o)$, there exists at least a group of RDF triples. $[[p'(c'_s, c'_o)]] := \{p'(s, o) / p'(s, o) \in E, \text{type}(s, c'_s) \in E, \text{type}(o, c'_o) \in E$ and p' denotes the property connecting s with $o\}$.

Example 2: To explain Definition 2, "(AssistantProfessor worksFor Department)" is a type-based triple, where "AssistantProfessor" and "Department" are types and "worksFor" is the relationship between these two types. Each type-based triple has at least one mapping in the RDF database, for instance, the RDF triple "(AssistantProfessor0 worksFor Department0)" and *etc.*

Definition 3: A type-base summary G' (V', E') of a data graph G (V, E) is an aggregation of all the distinct type-based triples. The property path operator alternativePath "|" is employed to incorporate properties with multi-edges. Thus, all possibilities are tried. Every vertex, $v' \in V_C \subset V'$ denotes an aggregation of all the vertices $v \in V$ having the type v'. $[[v']] := \{v/ \text{ type } (v, v') \in E\}$. Each edge $p(v'_1, v'_2) = p_1| p_2|, ..., |p_n$, iff $p_1, p_2, ..., p_n$, and $\forall i \in [1, n]$, $p_i(s_i, o_i) \in E \cap \text{type } (s_i, v'_1) \in E \cap \text{type}(o_i, v'_2) \in E$, $v'_1, v'_2 \in V'$. Every edge $p(v_1, v_2) \in E'$ represents a distinct property conjunction. $V' = V_c \cup \{Others\}$. $[[Others]] = \{v/ \neg \exists c \in V_c$ with type $(v, c) \in E$ and *Others* is the set of all the vertices in V with no given type$\}$

A type-based triple in Definition 2 is employed to construct the type-based summary. In Definition 3, we define the inter-entity type relationship summary. To explain the definition, Example 3 is given.

Example 3: Fig. 3 illustrates a type-based summary sample that summarizes all the inter-entity type relationships from the partial LUBM dataset.

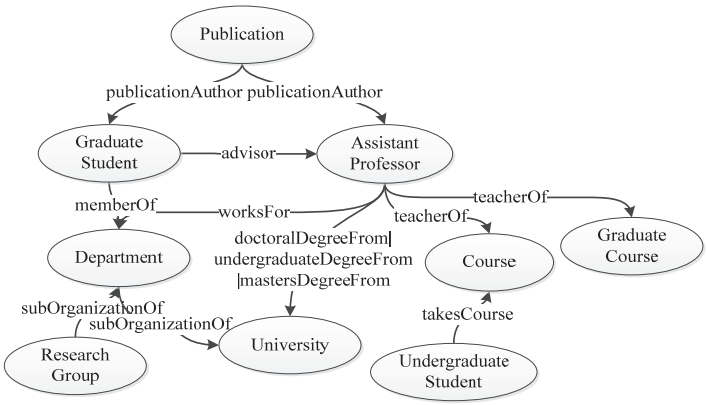


Fig. 3. Type-based summary sample.

Example 4: We present the limitation of the type-based summary for SCHEMA by an example and, meanwhile we prove the completeness of our type-based summary. Based on the data in Fig. 1, Table 1 shows the produced SPARQL queries by two different summaries, the type-based summary of SCHEMA and our type-based summary. Table 2 shows the SPARQL query results answered by these two approaches.

Notice that q_1, q_2 or q_3 in Table 1 is the possible SPARQL query produced by SCHEMA which means that the produced SPARQL query may be q_1, q_2 or q_3 because only one property is kept by SCHEMA. But our type-based summary incorporates the multi-edge properties into a combined property using alternativePath operator "|", like

"ub:doctoralDegreeFrom|ub:undergraduateDegreeFrom|ub:mastersDegreeFrom" shown in q_4 of Table 1. As a result, Table 2 shows whatever SPARQL queries SCHEMA may produce, the SPARQL query results are incomplete and lose a portion of data.

Table 1. SPARQL queries translated by different type-based summaries.

Query Id	SPARQL queries
q_1(SCHEMA)	Select *?ap ?u* where{*?ap* rdf:type ub:AssistantProfessor. *?u* rdf:type ub:University. *?ap* ub:doctoralDegreeFrom *?* }
q_2(SCHEMA)	Select *?ap ?u* where{*?ap* rdf:type ub:AssistantProfessor. *?u* df:type ub:University. ?*ap* ub:undergraduateDegreeFrom *?*}
q_3(SCHEMA)	Select *?ap ?u* where{*?ap* rdf:type ub:AssistantProfessor. *?u* df:type ub:University. *?ap* ub:mastersDegreeFrom *?u*}
q_4(OURS)	Select ?ap *?u* where{*?ap* rdf:type ub:AssistantProfessor. *?u* df:type ub:University. *?ap* ub : doctoralDegreeFrom|ub : undergraduateDegreeFrom|ub : mastersDegreeFrom *?u*}

Table 2. SPARQL query results answered for q_1, q_2, q_3 and q_4.

Query Id	SPARQL query results answered
q_1(SCHEMA)	<http://www.Department0.University0.edu/AssistantProfessor0> ub:doctoralDegreeFrom <http://www.University643.edu>
q(SCHEMA)$_2$	<http://www.Department0.University0.edu/AssistantProfessor0> ub:undergraduateDegreeFrom <http://www.University924.edu>
q_3(SCHEMA)	<http://www.Department0.University0.edu/AssistantProfessor0> ub:mastersDegreeFrom <http://www.University389.edu>
q_4(OURS)	<http://www.Department0.University0.edu/AssistantProfessor0> ub:doctoralDegreeFrom <http://www.University643.edu> <http://www.Department0.University0.edu/AssistantProfessor0> ub:undergraduateDegreeFrom <http://www.University924.edu> <http://www.Department0.University0.edu/AssistantProfessor0> ub:mastersDegreeFrom <http://www.University389.edu>

Algorithm 1: Summarize inter-entity type relationships

Input: RDF triples, D

Output: S_D, a summary graph with inter-entity type relationships

1 **Variables:** T_D, storing distinct type-based triples
2 **for** each RDF triple $t_i(s, p, o) \subset D$ and $p \in E_R$ and $i \in [1, n]$
3 Generate a type-based triple $tt(c_s, p, c_o)$ with $t_i(s, p, o)$, $t_{st}(s, \text{type}, c_s)$ and $t_{ot}(o, \text{type}, c_o)$
4 Insert distinct $tt(c_s, p, c_o)$ into T_D;
5 **for** $tt_i \in T_D$ and $i \in [1, n]$
6 **for** $tt'_j \in T_D$ and $j \in [1, n]$
7 **if** tt'_j.s is equal to $tt_i.s$ and $tt'_j.o$ is equal to $tt_i.o$ and $tt'_j.p$ is not equal to $tt_i.p$
8 Set $tt_i.p$ to be a combined property $tt'_j.p| tt_i.p$, Remove tt'_j;
9 Return T_D;

Next, we describe how to build a type-based summary which is shown in Algorithm 1. Firstly, for each RDF triple, we replace the subject of the triple with its type and replace the object of the triple with its type (line 2-4). Then, the merging multi-edge property operations lie in line 5-8. For a type-based triple $tt_i \in T_D$, if there is a type-based triple $tt'_j \in T_D$ whose subject and object are equal to the subject and the object of tt_i and whose property is not equal to the one of tt_i, the two properties of the type-based triples will be incorporated into a combined property and tt'_j is removed from T_D. The combined property is denoted as "$tt'_j.p|tt_i.p$". We repeat this process until the iteration ends (line 6-8). The time complexity of the algorithm is $O(n^2)$ in which n is the number of the type-based triples, namely the size of T_D.

4. KEYWORD SEARCH WITH TYPE-BASED SUMMARY

4.1 Search for Top-*k* Matching Subgraphs on Type-based Summary

The matched entities over the type-based summary are called keyword elements. RDF graph data is preprocessed and is stored in a new OPT index that is similar to the OPS index in [18]. However, our OPT index contains the object, the property and the corresponding type. An object key o_i is associated to a sorted vector of n_i property keys, $\{p_1^i, p_2^i, ..., p_n^i\}$. Every property key p_i is successively linked to an associated sorted list of $k_{i,j}$ type keys. All RDF triples with E_A and E_T relationships are kept to build the OPT index. In order to find top-k matching subgraph faster, we convert the type-based summary graph to an undirected graph. The vertices are substituted by integer IDs. The search algorithm for producing top-k subgraphs is shown in Algorithm 2. We utilize $R[u]$ to indicate the bookkeeping for the vertex u. Specially, in each element of R, we maintain two structures, namely "<r, $dist_1$, ..., $dist_i$>", where r is a connecting vertex, $dist_i$ is the distance from r to w_i and a collection of "<r, *propertyPath*, w_i>" structure stored in a vector, where *propertyPath* is a property path from r to w_i. The shortest distances are pre-computed from u to each vertex over the type-based summary by Floyd algorithm and this information is organized in a hash table denoted as M_{NK}.

Algorithm 2 contains three main steps: Step 1, push each keyword element w of each keyword element combination c into the queue S; Step 2, call function visit (u) to visit each element u of S when u hasn't been visited and push each neighbor b of u into S; these processes continue until S is empty; Step 3, execute Steps 1 and 2 until C is empty. For function visit (u), we look up u's combined distance to the other keyword elements (line 11-21) by which we can know if we have found an answer root. Specially, a generous depth value d_{max}=8 is used to reduce search space (line 14). A pruning threshold τ_{prune} is the current kth smallest combined distance (line 20-21). The function genPropertyPath (u, w_j) is to keep property paths from u to w_j used for query translation later (line 22-27). The time complexity of the algorithm is $O((m*|C|*(d_{max}+e))$, where $|C|$ is the number of keyword element combinations, m is the number of keywords, and e is the number of edges adjacent to each header vertices visited. The worst case complexity is $O((m*|C|*(|G|+e))$ in which d_{max} is equal to $|G|$ the size of the type-based summary.

Algorithm 2: Search for top-k Matching Subgraphs

Input: C, a set of $c = \{w_1, w_2, ..., w_m\}$; $G = \{V, E\}$; queue S; R, nodes visited ← ∅; pruning threshold, $\tau_{prune} \leftarrow \infty$
Output: A, answers found

```
1 for each combination c = {w1, w2, ..., wm} ∈ C
2     for each keyword element w ∈ c
3         Push w into S;
4         while the queue S is not empty
5             Pop the head of S, set u=it;
6             if u has not been visited
7                 Call visitNode(u) to visit u;
8             for each neighbor b of u
9                 Push b into queue S;
10 Return A (if found) or nil (if not)
11 visitNode(u) {
12     Insert a element, ⟨u, ⊥, ..., ⊥⟩ into R;
13     for each j ∈ [1, m]
14         if the distance from u to jth keyword element is less than dmax
15             Insert the shortest distance MNK(u, wj) from u to wj into R[u].distj;
16     if the combined distance ∑_{i=1}^{m} R[u].disti is not infinity
17         for j ∈ [1, m]
18             Call genPropertyPath (u, wj), set R[u].pathj = genPropertyPath (u, wj);
19     Insert u into A;
20     if the number of A is greater than k
21         Set τprune to be the kth smallest of {sumDist (u) |u ∈ A };     }
22 genPropertyPath(u, wj){
23     if the distance from u to wj is less than 1
24         Insert a property path pattern ⟨u, property_{u→wj}, wj⟩ into P;
25     else
26         Insert a property path pattern ⟨u, ppath_{u→vn}, vn⟩ into P;
27     Return P;                }
```

4.2 Property Paths in SPARQL

Property paths are a new feature introduced in SPARQL 1.1 [15] 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 nodes. A trivial case is a property path of length exactly 1, which is a triple pattern. Variables can't be used as part of the path itself, only the ends. Property paths allow for more concise expression for some SPARQL basic graph patterns and they also add the ability to match connectivity of two resources by an arbitrary length path. The SPARQL 1.1 specification is followed.

Definition 4: 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. ^e is the reverse path, e_1/e_2 is a sequence path of e_1 followed by e_2 and e_1|e_2 is an alternative path of e_1 or e_2. We do not consider e+, e∗,

e?, !{iri_1|...| iri_k} and !{^iri_1|...|^iri_k} because keyword search seldom encounters these cases above. To illustrate Definition 4, the following example is presented.

Example 5: "ub:worksFor/ub:subOrganizationOf" is a sequence path of "ub:worksFor" followed by "ub:subOrganizationOf" and "ub:headOf|ub:worksFor" is an alternative path of "ub:headOf" or "ub:worksFor".

Definition 5: A property path pattern is a triple in $(I \cup L \cup V) \times PP \times (I \cup L \cup V)$. Here I represents IRI, and L and V are same as that in Definition 1. In our approach, subgraph paths are stored in the form similar to property path patterns, namely the structures with "< >" in line (24, 26) of Algorithm 2, as shown in Example 5.

Example 6: "<*?x*, {worksFor, suborganizationOf}, *?y*>" is a property path pattern in which "*?x*" and "*?y*" represents the subject variable and the object variable, respectively and "{worksFor, suborganizationOf}" is a set of property path expressions.

4.3 Keyword Query Translation with Property Paths

Now we translate the top-k subgraphs into top-k SPARQL queries. Algorithm 3 contains three main steps: Step 1, for each element of A, define a variable s_{where} initialized with "where {" and translate the vertices and edges of the connecting vertex in A to m keywords; Step 2, define a variable s_{select} initialized with "select distinct *"; combine s_{where} and s_{select} and then insert s_{where+} s_{select} into Q; Step 3, execute Step 1 and Step 2 until A is empty or i equals k. The time complexity of the algorithm is O (k*m*|$Path'$|), where k is the number of SPARQL queries to be computed and m is the number of the keywords, and |$Path'$| is the average size of the property paths produced. A complete translation from a subgraph to a SPARQL query can be obtained as follows:

- **Processing of Vertices** All the vertices in the top-k subgraphs are taken as distinguished variables so the select clause is "select distinct *", where adding "distinct" is to avoid generating duplicated results (line 8). The function var (v) returns the SPARQL query variable denoted as "*?v*" (line 21).
- **Mapping of subgraph paths** The subgraph path in a vector *Path* is translated to property paths of SPARQL query by checking *Path*'s size (lines 11-20). The predicate path with length 1 is directly returned (lines 14-15). The property path with more than one property are transformed by adding "/" (lines 16-17). When path to tail is encountered, the property path is brought to an end without adding "/" (lines 18-19).
- **Mapping of edges with E_A relationships** Each E_A edge that connects the keyword to the entity is translated to two triple pattern expressions of SPARQL (var*(x)* $edgeLabel_{entity \to keyword}$ *keyword*) and (var(x) rdf:type $name_{type}$) (lines 6-7), where $edgeLabel_{entity \to keyword}$ and $name_{type}$ are derived from $Map_{property}$ and Map_{type}.

Algorithm 3: Top-k Query Generation

Input: A, a set of property path patterns; s_{select}, select clause, s_{where}, where clause, Q, SPARQL queries produced; $Map_{property}$, storing (*keyword*→*type*, *property*) pairs; Map_{type} keeping (*typeId*, *type*) pairs; *Path*, a property path vector;

```
Output: Q
1 for A is not empty and i=1, …, k
2     Set a string s_where = "where {";
3     for j=1, …, m
4         Set a property path pattern variable p = A.get(i).path_j;
5         s_where ← s_where + var (p.s) transPropertyPath(p.PP) var(p.o);
6         s_where ← s_where + var(p.o) Map_property.get(k_j → p.o) k_j;
7         s_where ← s_where + var(p.o) rdf:type Map_type.get(p.o);
8     Set a string s_select = "select distinct *";
9     s_where ← s_where + "}";
10    Insert (s_select+s_where) into Q;
11 transPropertyPath(Path) {
12     for i=0, …, Path.size()
13         Set a variable pt = Path.get(i);
14         if Path.size() = 1
15             Set a variable pp = pt;
16         if i ≤ Path.size() – 2
17             Set a variable pp = pp + pt/;
18         if i = Path.size() – 1
19             Set a variable pp = pp + pt;
20     Return pp;              }
21 var(v) { Return ?v;   // ?v is a SPARQL query variable denoted by adding "?" }
```

5. EXPERIMENTAL EVALUATION

LUBM [5] is the Lehigh University benchmark commonly used in the semantic web community. By its generator, we have a dataset of 5 million triples to 28 million triples. The real RDF datasets contain WordNet, Barton, DBpedia Infobox and BSBM. Two existing methods, SUMM [11] and SCHEMA [17] are utilized to do comparative tests. Our experiments are conducted on a SMP machine with 2.93 GHz Intel Core2 processors and 4GB memory. We use [3] to store and query RDF data.

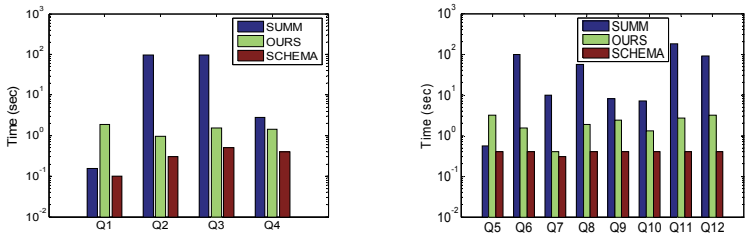
5.1 Search Performance

Table 4 presents 12 typical keyword queries [11]. Computing the response time is similar to that in [11, 17]. The response time starts from entering keywords until getting top-k query results answered that contains the translation time of keywords-to-SPARQL queries and the executing time of SPARQL queries. The query response time is in log scale. In Figs. 4 (a) and (b), the response time of SCHEMA, SUMM and our approach investigated on the same queries.

The summary graph by SCHEMA lacks some inter-entity relationships as mentioned earlier. Thus, in what follows, we compare the query response time of SUMM with our approach that has provable guarantees on the correctness of the query results answered. As illustrated in the Fig. 5, for both SUMM and ours approach, the cost of query evaluation generally becomes more expensive as the size of the data increases. Our approach outperforms SUMM on query Q_2–Q_4. As for k, we execute 30 SPARQL queries with length 2, 3 and 4 on LUBM at k=10, 15 and 20, respectively. The average time

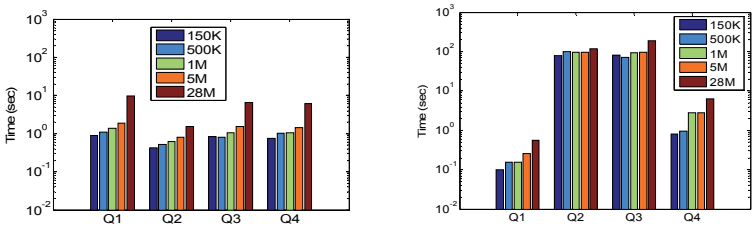
Table 4. Query examples.

Queries	Keywords
Q_1	Pulication19, Lecturer6
Q_2	Research5, Fullprofessor9, Publication17
Q_3	FullProfessor9, GraduateStudent0, Publication18, Lecturer6
Q_4	Department0, GraduateStudent1, Publication18, AssociateProfessor0
Q_5	Afghan, Afghanistan, al-Qaeda, al-Qa'ida
Q_6	3rdbase, 1stbase, baseball team, solo dance
Q_7	Knuth, Addison-Wesley, Number theory
Q_8	Data Mining, SIGMOD, Database Mgmt.
Q_9	Bloomberg, New York City, Manhattan
Q_{10}	Bush, Hussein, Iraq
Q_{11}	deflation, railroaders
Q_{12}	ignitor, microprocessor, lawmaker



(a) LUMB. (b) Other datasets.

Fig. 4. Query response time: (a) LUBM (b) other datasets.



(a) Our approach. (b) SUMM.

Fig. 5. Query response time: (a) our approach vs. (b) SUMM.

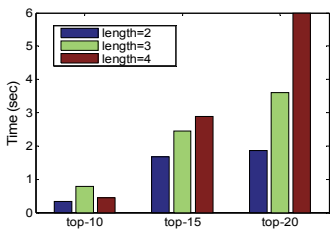


Fig. 6. Impact of k on search performance.

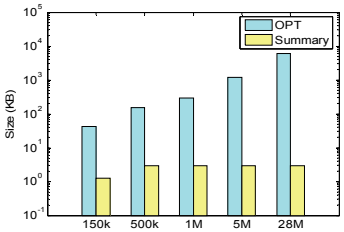


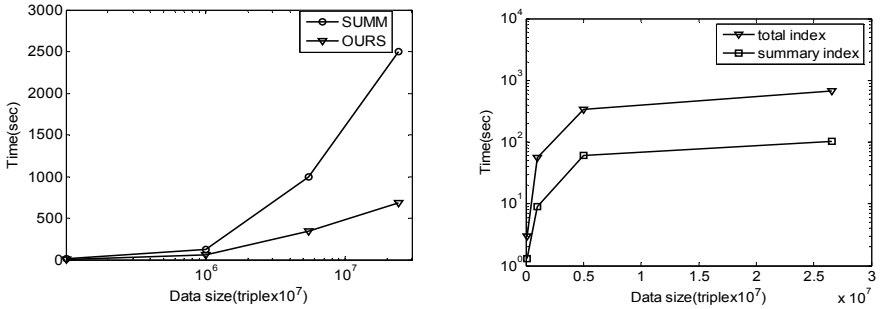
Fig. 7. Index size.

is illustrated in Fig. 6. The time increases linearly when k becomes larger. In addition, the impact of query length on the search performance is minimal when k is 10. The impact of query length is substantial when a higher k is used instead. Thus, we set k=10.

Though our query response is implemented by Algorithms 2 and 3, the response time complexity of our approach mainly depends on the time complexity of Algorithm 2. We compare the response time complexity with that of SUMM and SCHEMA, respectively. Our response time complexity of Algorithm 2 is $O((m*|C|*(d_{max}+e))$. The response time complexity of SUMM is $O(m*|K'|*m*|L|*|g|)$, where $|K'|$ is the average number of the keyword elements matching keywords, $|L|$ is the number of portals in the partition, and $|g|$ is the size of each subgraph to be visited. Note that our response time complexity is lower than that of SUMM, for $|C|$ is approximately equal to $|K'|*m$ and "d_{max}+ e" is less than $|L|*|g|$. Although SCHEMA canno guarantee the correctness of query results for LUBM, SCHEMA has a lower response time complexity $O(m*|K'|*|N'|)$ than that of our approach, where $|K'|$ is the average number of the keyword elements matching keywords, $|N'|$ is the average number of neighbors for the vertex to be visited, $|K'|$ is less than $|C|$, and $|N'|$ is less than e.

5.2 Index Performance

Computing the index time is similar to that in [11, 17]. We implement SUMM and OURS approach on the same datasets. For LUBM, the size of the data is varied from 100,000 triples to 28 million triples illustrated in Fig. 8 (b). The plotted total time includes the time spent to construct the summary index and the time spent to construct the OPT index. In Fig. 7, the size of the OPT index increases linearly when the size of the dataset gets larger. However, the size of the summary index changes little as the number of the RDF triples increases. Our approach performs an order of magnitude faster than SUMM because partitioning millions of subgraphs and summarizing these subgraphs are required for SUMM as shown in Fig. 8 (a).



(a) Index time: OURS vs. SUMM. (b) LUBM.

Fig. 8. Index time: (a) index time: OURS vs. SUMM (b) LUBM.

The index time complexity of SUMM is $O(|T|*|V|+|P|*|S|)$, where $|T|$ is the number of types, $|V|$ is the number of vertices of RDF graph, $|P|$ is the number of partitions and $|S|$ is the number of distinct summaries. Because the index construction or our approach

is implemented by Algorithm 1, our index time complexity is $O(n^2)$. n is less than $|V|$, $|P|$ and $|S|$. Therefore, our index time complexity is much lower than that of SUMM.

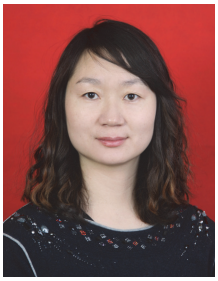
6. CONCLUSIONS

In this paper, we present a new approach by using a type-based summary to translate keyword queries to SPARQL queries which makes it easier for no-expert users because users are not required to master RDF query languages and the underlying RDF schema. In addition; (i) We distill a complete type-based summary from RDF dataset in order to find inter-entity relationships used for keywords-to-SPARQL translation; (ii) During translating keyword queries into SPARQL queries, we exploit the property path expressions of SPARQL 1.1 including predicatePath, SequencePath, ReversePath and AlternativePath; (iii) Efficient algorithms are leveraged to find top-k subgraphs and translate top-k sub-graphs into SPARQL queries. The experimental results on synthetical and real datasets show that our approach is scalable, efficient and portable. In the future, we plan optimize SPARQL queries using property paths to improve query performance.

REFERENCES

1. G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using banks," in *Proceedings of the 29th IEEE International Conference on Data Engineering*, 2002, pp. 431-440.
2. Y. Chen, W. Wang, Z Liu, and X. Lin, "Keyword search on structured and semi-structured data," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2009, pp. 1005-1010.
3. L. Garrison, R. Stevens, and A. Jocuns, "Efficient rdf storage and retrieval in jena2," *Exploiting Hyperlinks*, Vol. 51, 2004, pp. 35-43.
4. K. Gkirtzou, G. Papastefanatos, and T. Dalamagas, "RDF keyword search based on keywords-to-SPARQL translation," in *Proceedings of the 1st International Workshop on Novel Web Search Interface and Systems*, 2015, pp. 3-5.
5. Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Web Semantics Science Services and Agents on the World Wide Web*, Vol. 3, 2005, pp. 158-182.
6. H. He, H. Wang, J. Yang, and P. S. Yu, "Blinks: ranked keyword searches on graphs," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2007, pp. 305-316.
7. V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *Proceedings of International Conference on Very Large Data Bases*, 2005, pp. 505-516.
8. M. Kargar and A. An, "Keyword search in graphs: finding r-cliques," in *Proceedings of the VLDB Endowment*, 2011, pp. 681-692.
9. G. Klyne, J. J. Carroll, "Resource description framework (RDF): Concepts and abstract Syntax," W3C Recommendation, http://www.w3.org/TR/2004/REC-rdf-concepts-200 40210/.

10. G. Ladwig and T. Tran, "Combining query translation with query answering for efficient keyword search," in *Proceedings of International Conference on the Semantic Web: Research and Applications*, 2010, pp. 288-303.
W. Le, F. Li, A. Kementsietsidis, and S. Duan, "Scalable keyword search on large RDF data," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 26, 2014, pp. 2774-2788.
11. G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2008, pp. 903-914.
12. X. Lian, E. D. Hoyos, A. Chebotko, B. Fu, and C. Reilly, "K-nearest keyword search in rdf graphs," *Journal of Web Semantics*, Vol. 22, 2013, pp. 40-56.
13. F. Liu, C. Yu, W. Meng, and A. Chowdhury, "Effective keyword search in relational databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2006, pp. 563-574.
14. E. Prud'hommeaux and A. Seaborne, "W3C. SPARQL 1.1 overview," http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/, 21 March 2013.
15. L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold, "Column-store support for RDF data management: not all swans are white," in *Proceedings of International Conference on Very Large Data Bases*, 2008, pp. 1553-1563.
16. T. Tran, H. Wang, S. Rudolph, and P. Cimiano, "Top-*k* exploration of query candidates for efficient keyword search on graph-shaped (RDF) data," in *Proceedings of IEEE International Conference on Data Engineering*, 2009, pp. 405-416.
17. C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," in *Proceedings of the VLDB Endowment*, 2008, pp. 1008-1019.
18. G. Zenz, X. Zhou, E. Minack, W. Siberski, and W. Nejdl, "From keywords to semantic queries-incremental query construction on the semantic web," *Journal of Web Semantics*, Vol. 7, 2009, pp. 166-176.
19. G. Peters and R. Weber, "DCC: A framework for dynamic granular clustering," *Granular Computing*, Vol.1, 2016, pp. 1-11.



Xiao-Qing Lin received the M.S. degree in Computer Application Technology from Shenyang Aerospace University, China. She is currently pursuing the Ph.D. degree in the School of Computer Science and Engineering, Northeastern University, China. Her research interests include RDF keyword search and the Semantic Web.

Zong-Min Ma is currently a Full Professor at Nanjing University of Aeronautics and Astronautics, China. He received his Ph.D. degree from the City University of Hong Kong, 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.

Li Yan is a Full Professor at Nanjing University of Aeronautics and Astronautics, China. She received her Ph.D. degree from Northeastern University, China. Her research interests include databases, XML and the Semantic Web with a special focus on spatiotemporal information and uncertainty. She has published more than sixty papers on these topics. She is also the author of two monographs published by Springer.