Filomat 32:5 (2018), 1861–1873
https://doi.org/10.2298/FIL1805861L

Published by Faculty of Sciences and Mathematics,
University of Niš, Serbia
Available at: http://www.pmf.ni.ac.rs/filomat

RDF Keyword Search Using Multiple Indexes

Xiaoqing Lin[a,b], Fu Zhang[a], Danling Wang[b], Jingwei Cheng[a]

[a]*School of Computer Science and Engineering, Northeastern University, Shenyang, 110819, China*
[b]*School of Information Engineering, Eastern Liaoning University, Dandong, 118003, China*

Abstract. Since SPARQL has been the standard language for querying RDF data, keyword search based on keywords-to-SPARQL translation attracts more intention. However, existing keyword search based on keywords-to-SPARQL translation have limitations that the schema used for keyword-to-SPARQL translation is incomplete so that wrong or incomplete answers are returned and advantages of indexes are not fully taken. To address the issues, an inter-entity relationship summary (ER-summary) is constructed by distilling all the inter-entity relationships of RDF data graph. On ER-summary, we draw circles around each vertex with a given radius r and in the circles we build the shortest property path index (SP-index), the shortest distance index (SD-index) and the r-neighborhoods index by using dynamic programming algorithm. Rather than searching for top-k subgraphs connecting all the keywords centered directly as most existing methods do, we use these indexes to translate keyword queries into SPARQL queries to realize exchanging space for time. Extensive experiments show that our approach is efficient and effective.

1. Introduction

We are inundated with a significant amount of rapidly growing RDF data from disparate domains since the RDF (Resource Description Framework) [10] has been proposed for modeling Web objects. It has been widely used in various applications (e.g., DBPedia, Yago, Wordnet). RDF data can be represented as a collection of triples denoted as SPO (subject, property, object). SPARQL [1, 17] is the official standard for searching over RDF repositories. Although the SPARQL query has a similar format to the SQL query for relational engines, it is a graph pattern-matching query language and supports conjunctions of triple patterns.

If the database schema is unknown, it is a challenging task even for users who are familiar with the SPARQL query, not to mention non-expert users. First, the non-expert user must master the SPARQL query syntax. Second, the vocabulary of the underlying RDF data is required. Keyword search is a very succinct way by which non-expert users needn't to grasp professional knowledge about the schema or query languages. So, we present an approach of translating keyword queries into SPARQL queries. We provide non-expert users with a friendly interface as well as a good query performance.

Keyword search on RDF data can be summarized into two categories by query processing ways. One returns query results by directly finding candidate answers containing keywords on the RDF graph, discussed in [2, 4, 7–9, 12, 14] and the other constructs queries by the schema extracted from the RDF data

2010 *Mathematics Subject Classification*. 97R50.
Keywords. Data bases; Information system.
Received: 26 October 2017; Accepted: 20 November 2017
Communicated by Hari M. Srivastava
Email address: linda164@163.com (Xiaoqing Lin)

and then obtains query results by performing produced queries, discussed in [6, 11, 19, 20]. For the latter, three processes, keywords mapping, constructing queries and ranking queries are usually included. In [20], to clearly express the user intention, the queries are selected by the user interaction which requires some field knowledge. More work related to keyword search is discussed in [13–15, 18].

Due to several following advantages [19], keyword search based on translation is attracting more and more intention. First, the exploring on the summary that is much smaller than the real dataset accelerates the subgraph searching. Second, the user can select the best one by the user intention from the produced queries. In addition, we can exploit the optimization capabilities of existing search engines to improve query performance. It has been proved that keyword search based on translation is faster than direct keyword query answering in [19], called the SCHEMA method.

To our knowledge, few works on keyword search based on keywords-to-SPARQL translation are done so far (e.g., [6, 11, 19, 20]. In [11], structure indexes are extracted from RDF graph to help construct conjunctive queries but it costs much overhead on the structure indexes and takes more query time, and the authors of [6] define a type-based summary by following SCHEMA. The summary in SCHEMA looses too much information in RDF data as to how one type of the entity is connected to other types of the entity. For SCHEMA, all vertices of the same type are indistinguishably mapped to the same type vertex, only one property between the type vertices is kept. SCHEMA performs well on the datasets with regular topological structure (e.g., DBLP), but it returns wrong or incomplete answers for LUBM. The problem in SCHEMA is also detailed in [12] so they propose a new summary index based on types by partitioning and summarizing which leads to more overhead.

Inspired by these observations, we present a new approach to translate keyword queries to SPARQL queries using multiple indexes built in advance to exchange space for time. By the indexes we can quickly find top-k subgraphs and then translate them into SPARQL queries which are eventually executed by a SPARQL search engine. To summarize, we make the following contributions.

(1) We distill an ER-summary that summarizes relationships of all the inter-entities corresponding to relationships between query variables. The entity vertices and literal vertices are encoded to integer IDs for computation.

(2) We develop algorithms to find top-k subgraphs connecting all the keyword elements and translate the top-k subgraphs into SPARQL queries by multiple indexes (e.g., SD-index, SP-index and the r-neighborhoods index).

(3) We confirm the performance of our approach is superior to existing methods by a detailed series of experiments on real and benchmark RDF repositories.

In what follows, Section 2 provides an overview of our approach. Details on indexing graph data and keyword search with multiple indexes in Section 3 and Section 4 respectively. Section 5 presents experimental evaluation. Finally, the conclusion is arrived in Section 6.

2. Overview of our approach

We start with an overview of the following different steps. The architecture workflow is shown in Figure 2. Our work consists of pre-computing indexes and the top-k query translation denoted by two dashed boxes of Figure 2 respectively. I use Figure 1 to illustrate our approach.

Preprocessing. We process the RDF data to obtain the keywords-to-entities index used for the mapping of keywords to entities. An ER-summary is constructed that summarizes all the inter-entity relationships of the RDF data and over which SD-index, SP-index and the r-neighborhoods index are built.

Top-k Query Translation. Keywords are mapped to entities on ER-summary. Compute all the possible combinations of the entities w=$\{w_1, w_2, \ldots, w_m\}$. Using r-neighborhoods index and SD-index, Start from each of these matched keyword entities, top-k subgraphs connecting all keyword elements can be quickly found by the combined distance sumDist=$\Sigma_{i\in[1,m]}d(u, w_i)$ where u is the connecting node, w_i is matched entities and d(u, w_i) is the shortest distance from u to w_i derived from SD-index. For each subgraph, a SPARQL query is produced by the mapping of subgraph entities to query variables and subgraph property paths to relationships between query variables. Finally, translate top-k subgraphs into top-k SPARQL queries, execute the queries by a SPARQL search engine and return the query results to the user.

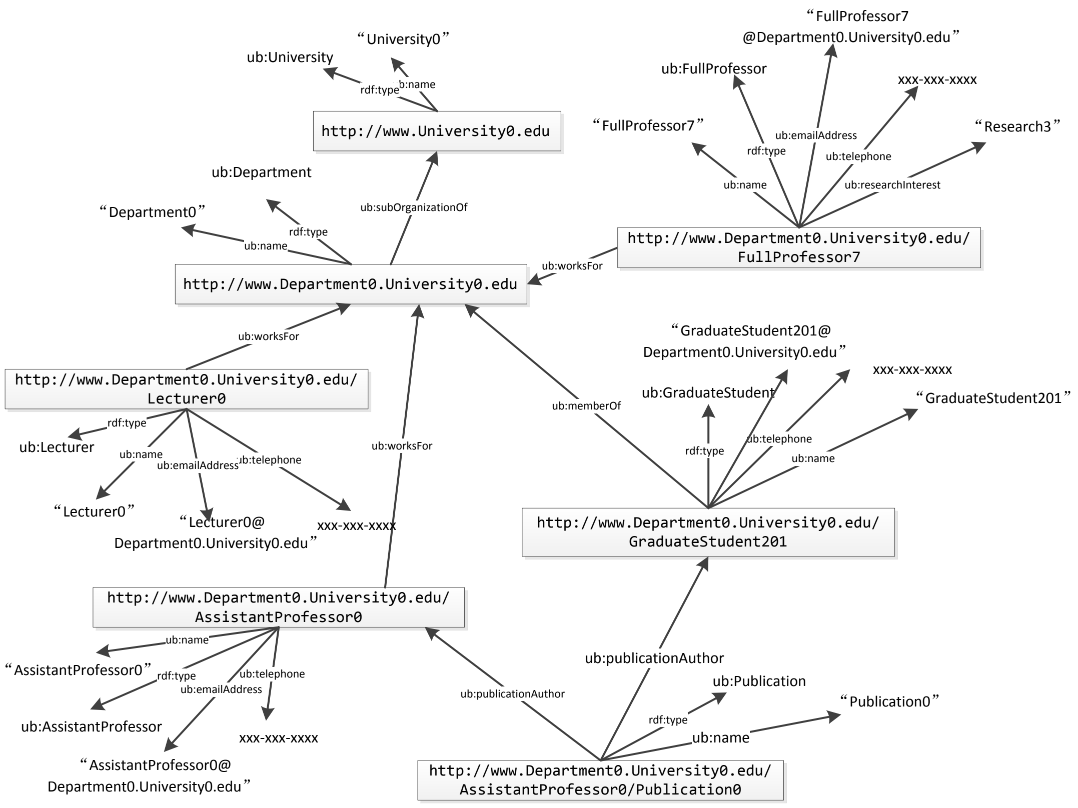


Figure 1: RDF data graph

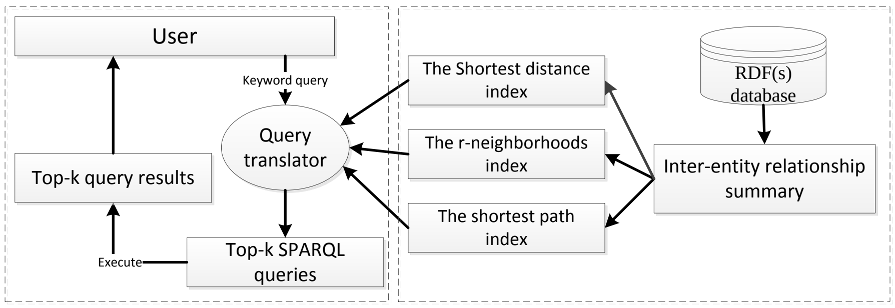


Figure 2: Architecture of our approach

3. Indexing graph data

We address the off-line indexing process where graph data is preprocessed and kept in specific data structures of a keyword and a graph index (ER-summary). We build multiple indexes over the graph index for the translation of keywords-to-SPARQL queries.

3.1. *The keyword index*

We employ B^+Tree due to less storage space for indexes than the size of original data by the compressed B^+Tree (see, e.g., [16]). A B^+Tree structure is built to support quickly mapping keywords to the entities with corresponding properties used for the translation. Figure 3 is our B^+Tree structure for the keyword index. The triples with object, predicate and subject shown in Table 1 are kept in leaf nodes of the B^+Tree. O_{id} (e.g., 1, 2,. . . ,7 in Figure 3) is the keyword integer ID converted by a string hash function and p_{ij} (e.g., p_{11},. . ., p_{61}) represents the property for s_{ij} (e.g.,s_{11},. . . ,s_{63}). Subjects and predicates of RDF triples are all converted into integer IDs by a mapping dictionary [3]. Because RDF triples can be converted into ID triples so as to compress the triple store, we compress the ID triples by [16]. Not only the subjects but also the predicates are required for the later translation of top-k subgraphs to SPARQL queries so we converted RDF triples into ID triples rather than simple IDs.

Unlike the method by [16], all six possible permutations of subject, predicate and object are constructed in clustered B$^+$Tree indexes, our keywords-to-entities index only maintains a (object, predicate, subject) permutation. Each lookup finding all triples containing keyword integer IDs must be a path from the root node to leaf node and all lookups have same path length, which leads to equivalent query performance. The time complexity of lookup is O (h) where h is the height of the B$^+$tree.

Object	**Predicate**	**Subject**
43549996	ub:name	5951 (http://www.Department5.University0.edu/FullProfessor9)
43549996	ub:name	1740 (http://www.Department1.University0.edu/FullProfessor9)
43549996	ub:name	523 (http://www.Department0.University0.edu/FullProfessor9)
43549996	ub:name	7046 (http://www.Department6.University0.edu/FullProfessor9)

Table 1: Data sample for leaf nodes of B$^+$-Tree

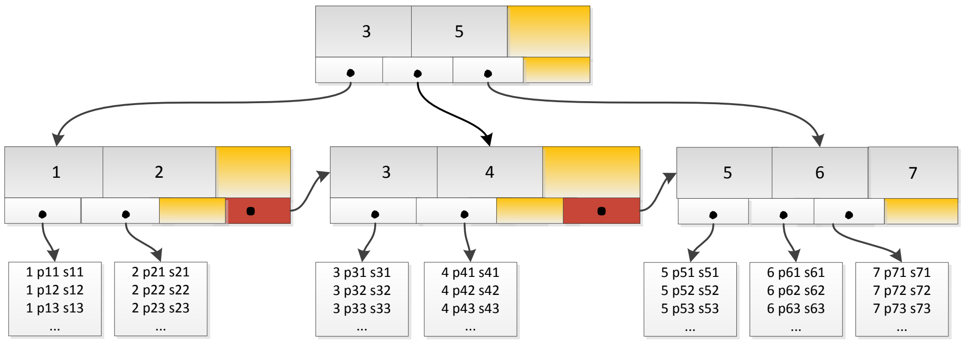


Figure 3: A B$^+$-tree structure for the keywords-to-entities index

3.2. *The inter-entity relationship summary (ER-summary)*

We build ER-summary including all the relationships between inter-entities of RDF data. First, we introduce the motivation for building the summary by an example. For a keyword query to find the department that FullProfessor0 works for whose corresponding SPARQL query is " select ?x ?y where {?x ub:worksFor ?y. ?x rdf:type FullProfessor.?y rdf:type Department. ?x ub:name 'FullProfessor0'}", we need to know the relationships between query variables "?x" and "?y" in order to translate the keyword query to the SPARQL query. Therefore ER-summary is distilled from RDF data so as to obtain the relationships between entities corresponding to the relationships between query variables. We define ER-summary as follows.

Definition 1. *: An inter-entity relationship summary G′ of a 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* $E' \subset E$*. E′ connects two entity vertices .* V_E *represents the set of entity vertices(i.e.,IRIs). The edge* $e(v_1, v_2)$ *exists in the index if and only if there is an edge* $e(v_1,v_2) \in E$ *and* $v_1, v_2 \in V_E$*.*

When calculating indexes, ER-summary is transformed into an undirected graph where all the vertices are represented by the integer IDs and the edge weight is all set to 1, as shown in Figure 4. All RDF data are from Figure 1. The edge together with edge direction is kept in a table used for late translation.

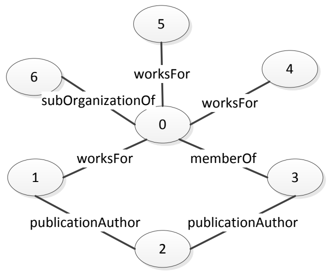


Figure 4: Integer inter-entity relation summary

3.3. Build multiple indexes by performing dynamic programming algorithm (DP)

We describe the process of building indexes on ER-summary including the r-neighborhoods index, SD-index and SP-index by DP algorithm. Similar to the r-cliques in [9], we define r-neighborhoods as shown in Definition 2. Each vertex on ER-summary is treated as the center and around which circles with radius *r* are drawn, by which the summary graph is partitioned into multiple blocks that are similar to our *r* radius neighborhoods. It takes less time to search on blocks than on the whole summary graph. In Figure 5, note that there are intersections between circles so that we needn't to build between-block indexes as [7] does. Those answers where keyword elements are not very close to each other are less useful and informative [9]. We suppose the answer connecting all keyword elements must be found in some circle as long as the radius *r* is reasonably given.

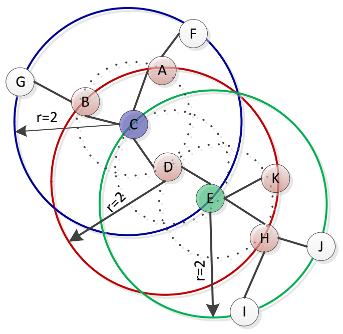


Figure 5: r=2 radius area

Definition 2. *(r-neighborhood): Given a graph G, a r-neighborhood of G is a set of nodes in which the shortest distance between any two nodes is no larger than r. The shortest distance between two nodes is the sum of the weights of the edges in G on the shortest path between the two nodes.*

We precompute the shortest distances from any two nodes as SD-index used for search for top-k subgraphs by calculating the combined distance. Through SP-index we can produce property paths of any two vertices, e.g., the property path of "$2 \rightarrow 1 \rightarrow 0$" is "$ub:publicationAuthor/ub:worksFor$" where "/" is the property path operator of SPARQL 1.1.

We build the r-neighborhoods index on ER-summary which can greatly improve search performance on the fly. *r* controls the size of the neighborhood. The pseudo codes are shown in Algorithm 1.

Data structure. The *AdjacentList* is an adjacent list for ER-summary graph shown in Figure 4. We introduce *Q* to achieve recursive calculations. Each node $< nodeId, d, p >$ contains three attributes, node integer ID *nodeId*, the shortest distance from the starting node *d*, and the shortest path from the starting node *p*. When visiting a new node, we can get the states of the node just visited by the three attributes. *MapA* is a list of form $< center, neighbors >$ where *center* is the starting node and *neighbors* is an ordered list by the distance from starting nodes within radius *r*. *MapD* and *MapP* keep SD-index and SP-index respectively.

The algorithm. We build the r-neighborhoods index, SD-index and SP-index on G (line 2-23). At each 'for' iteration, d is set to be 0 which shows a node has not been reached yet (line 4). We push an entry $< nodeId, d, p >$ from $AdjacentList$ into q (line 6). When q is not empty, the algorithm proceeds in the iterations of 'while' (line 8-22). In each iteration, we pop the top entry of q to a node of form $< tId, td, tp >$. Next, we visit the node if the node has not been visited. The node ID *tId* is kept in an array *neighbors* so as to build the r-neighborhoods index (line 23). SD-index and SP-index are stored in *MapD* and *MapP* respectively (line 16-17). Then add *td* and *neighbors* of the node will be expanded where *tp* is the path from starting that grows with the expansion (line 19-22). We repeat the loop until Q is empty or td >R (line 10-11). The time complexity is $O(|V| * (|E| + |V|))$ where $|E|$ is the number of edges, and $|V|$ is the number of vertices.

Algorithm 1: Building indexes by dynamic programming algorithm

```
Input: integer inter-entity relationship summary, G={V,E}, the designated radius, $r$
Output: $map_D$: the shortest distance index, $map_P$: path index, $map_A$: r-neighborhoods index
1  Variables: AdjacentList: an adjacent list for G; Q: q queue, each element form of (< nodeId, d, p >); visited:
     an array indicating whether the node is visited, $map_A$: a list of form of < center, neighbors > where
     neighbors is an ordered list;
2  for $i \leftarrow 0$ to AdjacentList.size()-1 do
3      $nodeId \leftarrow AdjacentList.get(i)$;
4      $d \leftarrow 0$;
5      $p \leftarrow nodeId$;
6      $Q.enqueue(< nodeId, d, p >)$;
7      $visited \leftarrow \emptyset$;
8      while $!Q.empty()$ do
9          $< tId, td, tp > \leftarrow Q.deqeue()$;
10         if $td > R$ then
11             break;
12         if $visited[tId] == 0$ then
13             $neighbors \leftarrow tId$;
14             if $map_D.contains(tId \rightarrow nodeId)$ then
15                 continue;
16             $map_D \leftarrow (nodeId \rightarrow tId, td)$;
17             $map_P \leftarrow (nodeId \rightarrow tId, tp)$;
18             $visited[tId] \leftarrow 1$;
19             $td \leftarrow td + 1$;
20             for each node eId that is adjacent to tId do
21                 $object \leftarrow (eId, td, tp \leftarrow tp + eId)$;
22                 Q.enqueue(object);
23     $map_A \leftarrow (< nodeId, neighbors >)$;
```

4. Keyword search with multiple indexes

We present the process of searching for top-k subgraphs using multiple indexes, SD-index, SP-index and the r-neighborhoods index built in Section 3.3 and the translation of keywords-to-SPARQL queries.

4.1. Searching for top-k subgraphs connecting all keyword elements

We look for top-k subgraphs using multiple indexes. We call entities containing keywords keyword elements. The algorithm is shown in Algorithm 2.

Data structure. W is a possible combination of keyword elements $w = (w_1, w_2, \ldots, w_n)$. A keeps answers found and each element in A is form $< u, sumDist, w >$ where u is the connecting node, *sumDist* is the distance sum from u to the keyword elements. A threshold T_{prune} is the current k-th shortest combined distance. For a new answer to be in the top k, its combined distance is no greater than T_{prune}.

The algorithm. We use a cursor to traverse the search list for each keyword element searchList (w_i) from the r-neighborhoods index. Cursor c_i advances on the list searchList (w_i) by calling next (line 5), which returns the next node in the list. As soon as we visit a node of the search list searchList (w_i), we check its combined distance *sumDist* from the other keyword elements by SD-index (line 11-12). Using the *sumDist*, we can immediately determine if we have found the connecting node of an answer. If the *sumDist* is less than the pruning threshold the top-k subgraph is kept otherwise it is abandoned (line 18-22).

Algorithm 2: Searching for top-k subgraphs using indexes

```
Input: $map_D$, $map_A$
Output: A: $top - k$ subgraphs
1  foreach $i \in [1, m]$ do
2      $c_i \leftarrow$ new Cursor(searchList($w_i$),0);
3  while $\exists j \in [1, m] : c_j.peekDist() \neq \infty$ do
4      $i \leftarrow$ pick from $[1, m]$ in a round-robin fashion;
5      $u \leftarrow c_i.next()$;
6      for each keyword element combination $w \in W$ do
7          if $u \neq \perp$ then
8              visit (u,w);
9  visit $(u, w)$ begin
10     $sumDist_u \leftarrow 0$;
11     foreach $j \in [1, m]$ do
12         $sumDist_u \leftarrow sumDist_u + map_D.get(u \rightarrow w_j)$;
13     if $sumDist_u < \infty$ then
14         if $|A| < k$ then
15             A.add($< u, sumDist_u, w >$);
16             $T_{prune} \leftarrow$ the k−th largest of $\{sumDist(v)|v \in A\}$;
17         else
18             if $sumDist_u < T_{prune}$ then
19                 insert it to A by the combined distance;
20                 $T_{prune} \leftarrow$ the k−th largest of $\{sumDist(v)|v \in A\}$;
21             else
22                 return;
```

4.2. *Translating for top-k subgraphs to SPARQL queries*

We translate top-k subgraphs derived by last section into SPARQL queries. We can obtain the inter-entity relationships by searching for top-k subgraphs on ER-summary. The keywords-to-entities relationships can be obtained from the keywords-to-entities index in Section 3.1.

4.2.1. *SPARQL and the property path of SPARQL*

We present the select query form as shown in Definition 3 by which it is easier to transform the subgraph into the SPARQL query graph.

Definition 3. *(SPARQL query) A simple SPARQL query is defined as sparql→ SELECT varlist WHERE(gp) where varlist=$(v_1, v_2, \ldots, v_n)$ is an ordered list of variables and varlist ⊆var(gp) and gp is the basic graph pattern that is a set of triple patterns.*

The ends of a property path for SPARQL 1.1 may be variables or RDF terms and a property path can't contain variables. To start with the definition of property path expressions, the SPARQL 1.1 specification [17] is followed.

Definition 4. $e := (iri)|(^\wedge e)|(e_1/e_2)|(e_1|e_2)|(e^+)|(e^*)|(e?)|(!\{iri_1|\ldots|iri_k\})|(!\{^\wedge iri_1|\ldots|^\wedge iri_k\})$

iri_1,..., iri_k are IRIs. Negated property sets are the expressions starting with "!". $^\wedge e$ denotes the reverse path construction, e_1/e_2 represents 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. $e_1|e_2$, $e+$, $e*$, $e?$, $!\{iri_1|\ldots|iri_k\}$ and $!\{^\wedge iri_1|\ldots|^\wedge iri_k\}$ are not considered.

4.2.2. *Translating top-k subgraphs into SPARQL queries*

A SPARQL query can be a query graph, therefore each one of top-k subgraphs can be mapped to be a SPARQL query graph, such as Figure 6 (b). The subgraph form of $< u, sumDist, w >$ derived by last section can construct a graph by SP-index where the paths from $u \to w_1$, $u \to w_2$, ..., $u \to w_m$ can be connected together to be a graph as shown in Figure 6 (a). u is a connecting node, w is the possible keyword element combination and p_1, p_2, p_3, p_m are all property paths. The algorithm is shown in Algorithm 3.

The algorithm. In each element of A, there are three attributes, the connecting node u and keyword element combination w are used to translate top-k subgraphs into SPARQL queries. Nevertheless, *sumDist* is not used to translate subgraphs but to find top-k subgraphs mentioned in the last section. Each time we produce the SPARQL queries by combing the triple patterns with types, keyword predicates and property paths (line 4-10). The results are kept in Q (line 12). "select distinct * where {" in Line 3 and "}" in Line 11 are necessary for each produced SPARQL query. The algorithm time complexity is O ($k * m$) where k is the number of produced SPARQL queries and m is the number of the keywords.

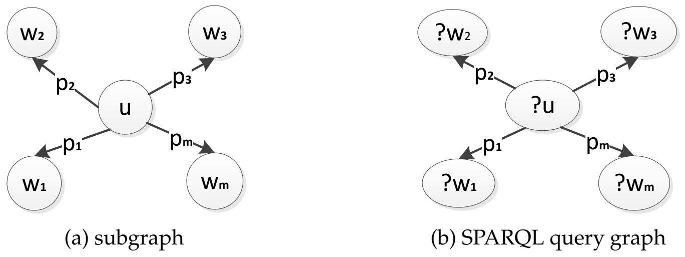


Figure 6: Mapping between the subgraph a and the SPARQL query graph b

5. Experiments

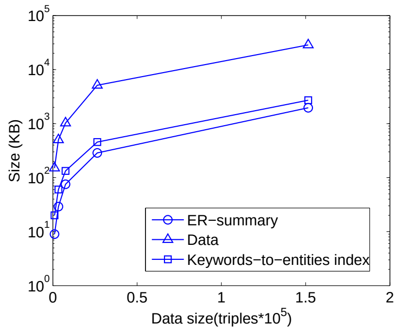
We test the performance of indexing and querying on both synthetical and real datasets. LUBM, one synthetic dataset and two real datasets SwetoDblp and WordNet are utilized where LUBM (Lehigh University Benchmark) is a popular benchmark, SwetoDblp is a large-size ontology on bibliography data of computer science publications from DBLP, and WordNet is a lexical database for the English language that organizes English words into synonym sets according to part of speech and enumerates linguistic relations between these sets. Two existing methods in [19] and [12] denoted as SCHEMA and SUMM are utilized to do comparative tests. The SPARQL queries produced are executed by the search engine [5].

Index performance We test the performance of indexes including ER-summary, the keywords-to-entities index, SD-index, SP-index and r-neighborhoods index, in terms of indexing time and index size. Figure 7 (a) shows the indexing time and index size of ER-summary and the keywords-to-entities index over LUBM

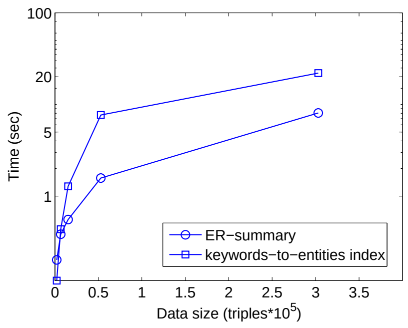
Algorithm 3: Translating top-k subgraphs to SPARQL queries

```
Input: A: top-k subgraphs, $map_D$
Output: Q: top-k SPARQL queries
1  for $a \in A$ do
2      (u,sumDist,w)← $a$;
3      $q$ ← select distinct * where { ;
4      for $i \in [1, m]$ do
5          $p_i$ ← property path of $u \rightarrow w_i$;
6          $q \leftarrow q$+?$u$   $p_i$   ?$w_i$.   ;
7          $predicate_i$ ← the predicate of $k_i \rightarrow w_i$;
8          $q \leftarrow q$+$k_i$   $predicate_i$ $w_i$.;
9          $type_i$ ← the type of $w_i$;
10         $q \leftarrow q$+$w_i$   $rdf : type$   $type_i$.;
11     $q \leftarrow q$+};
12     Q.add($q$);
```

while varying the size of the data and we can notice the size of ER-summary and the keywords-to-entities index is much less than that of the data. Figure 7 (b) shows the time of building ER-summary and the keywords-to-entities index.



(a) Size



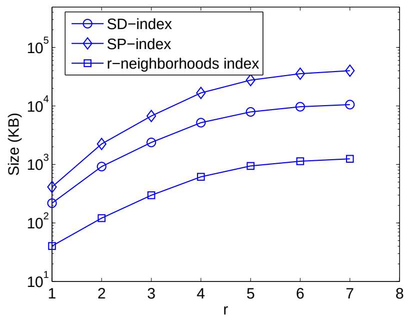
(b) Time

Figure 7: Index size a and time b over LUBM

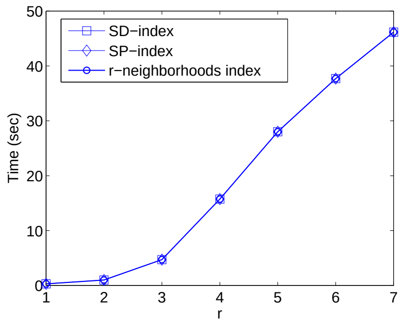
In Figure 8, we examine the impact of r to the size and the time of indexes that contain SD-index, SP-index and r-neighborhoods index which are necessary for constructing top-k subgraphs. The size and the time of the three indexes increase linearly when the radius r varies from 1 to 7 on LUBM. This is reasonable that larger r produces larger size indexes and requires more time. The size remains small (0.21-39.2 M) and the time remains low (0.295-46.16 s). DP is used to produce these three indexes at the same time so the time for the three indexes is same shown in Figure 8 (b). We set a generous default value r=7.

By Figure 9, note that the number of distinct entities on ER-summary has great impact on the size of SD-index, SP-index and the r-neighborhoods index. The size of the index increases as the number of distinct entities increases. Even so, the size of indexes is still small because the number of distinct entities is much less than the data shown in Figure 10. The number of entities is much smaller than the number of triples whatever the dataset, which proves it is correct for us to distill ER-summary from RDF data to reduce search space. WordNet and SwetoDblp contain more entities than LUBM.

In Figure 11, we present the size and the time of constructing ER-summary and the keywords-to-entities index with the datasets of the size 24M. Since the different graph topology from the datasets, the size



(a) Impact of r to the index size



(b) Impact of r to the indexing time

Figure 8: Impact of r to the index size a impact of r to the indexing time b for LUBM

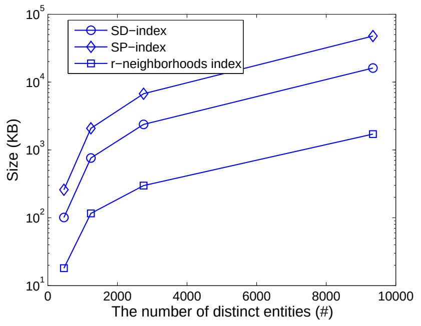


Figure 9: Impact of the number of distinct entities to size of indexes for LUBM

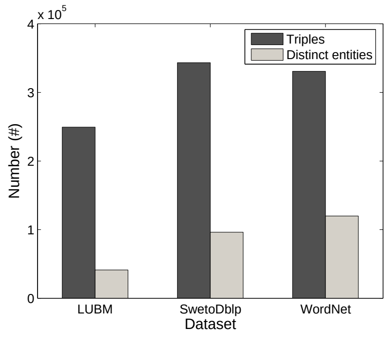
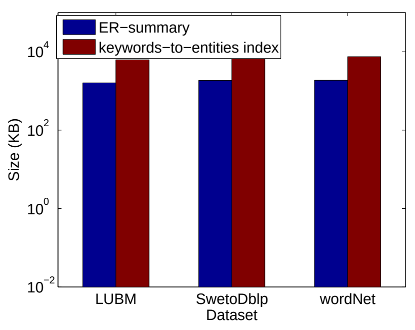
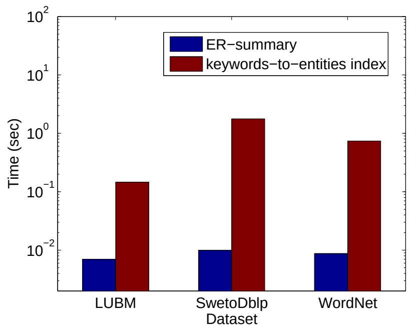


Figure 10: The number of triples and entities



(a) Size



(b) Time

Figure 11: Size a and time b for inter-entity relationship summary and keywords-to-entities

and the time for LUBM is all smaller than other two datasets. LUBM has the least number of entities on the same size dataset. So it has the least size and the least time for constructing ER-summary and the keywords-to-entities index.

Query performance The sample queries are described in Table 2. Figure 12 illustrates our query time over both synthetical and real datasets. The query time consists of the time of producing SPARQL queries and the time of executing the SPARQL queries. In Figure 12, the producing time is less than the executing time because the indexes for ER-summary are built in advance without traversing the index graph on the

	Queries	# Keyword Nodes	Datasets
Q_1	Publication19, Lecturer6	(53, 16)	LUBM
Q_2	Research5, FullProfessor9, Publication17	(32, 11, 196)	LUBM
Q_3	FullProfessor9, GraduateStudent0, Publication18, Lecturer6	(11, 37, 102, 16)	LUBM
Q_4	Department0, GraduateStudent1, Publication18, AssociateProfessor0	(3, 37, 102, 37)	LUBM
Q_5	The Compiler Design Handbook, Dynamic Compilation.	(23, 1)	SwetoDblp
Q_6	2002-01-03, Traditional Software Design.	(6091, 1)	SwetoDblp
Q_7	2002, Dynamic Compilation.,	(1426, 1)	SwetoDblp
Q_8	1990, Graph Algorithms.	(585, 1)	SwetoDblp
Q_9	breach of contract	(2)	WordNet
Q_{10}	nothing, nonentity	(2, 2)	WordNet
Q_{11}	first base, third base, baseball team, solo homer	(4, 3, 17, 4)	WordNet
Q_{12}	amphibious landing	(2)	WordNet

Table 2: Sample query workload

fly, which realizes our goal of trading more space to use less time. Nonetheless, the index size remains low (0.018-46.5 MB). The executing time is mainly affected by the property path length of SPARQL queries produced and the size of the dataset. The executing time gets longer as there are longer property paths in the SPARQL queries. Our query time is not related directly to the number of keywords. The query time is in log scale.

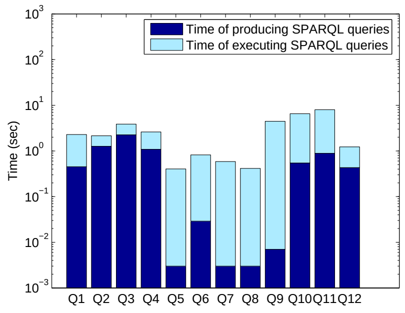


Figure 12: Our query performance

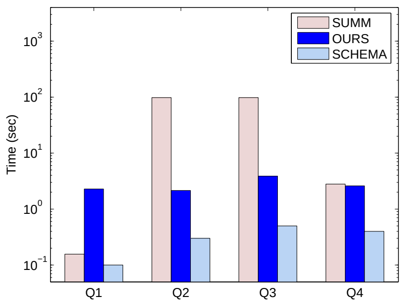
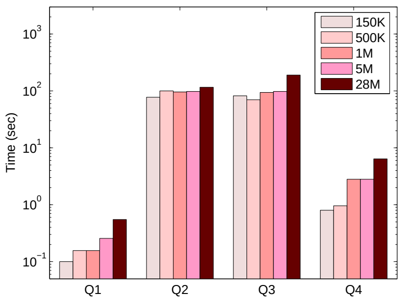
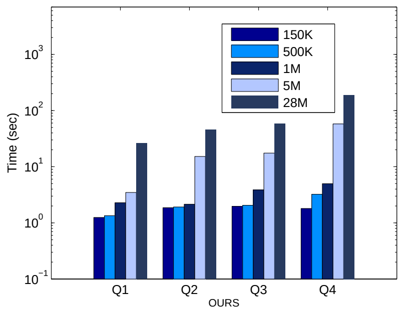


Figure 13: Performance comparison on LUBM



(a) SUMM



(b) OURS

Figure 14: SUMM a and our approach b over LUBM

We use four same queries as that in [12] to compare query time shown in Figure 13. Although SCHEMA has the best query performance, its schema summary is incomplete which leads to wrong or incomplete answers. In what follows, we discuss the query time for SUMM and our approach in guaranteeing the correctness of the query results. Our approach outperforms SUMM except for Q_1 as shown in Figure 13.

We investigate the scalability of our approaches, with respect to the data size of LUBM. In Figure 14 (a) and (b), the query time for SUMM and ours is shown while varying the size of LUBM. We can see the query time becomes longer as the size of the data increases for both SUMM and our approach. However, the query time remains low (1.25-100 s). As the data size increases, there may be more keyword elements matching keywords, which leads to more possible keyword element combinations on one hand and the time of executing SPARQL queries produced increases on the other hand.

6. Conclusion

We focus on the translation of keywords queries to SPARQL queries by using multiple indexes. It costs too much to build SD-index, SP-index for all pairs on ER-summary, so we build these indexes in *r* radius fields through drawing circles, which lowers the index cost significantly. While translating, the property path of SPARQL 1.1 is exploited that makes the translation more simple efficient. Experimental results on both synthetical and real datasets show that our solution improves the query performance under guaranteeing the correctness. In the future, we hope to focus on the optimization of the produced SPARQL queries and the maintenance of these indexes so as to improve SPARQL query efficiency.

7. Acknowledgments

This work was supported by the National Natural Science Foundation of China (61672139) and Natural Science Foundation of Liaoning Province of China (2015020048).

References

[1] W3c. sparql 1.1 overview. <http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/>, 21 March 2013. W3C Recommendation.
[2] Yi Chen, Wei Wang, Ziyang Liu, and Xuemin Lin. Keyword search on structured and semi-structured data. In *ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, Usa, June 29 - July*, pages 1005–1010, 2009.
[3] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient sql-based rdf querying scheme. In *International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September*, pages 1216–1227, 2005.
[4] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, and Gerhard Weikum. Searching rdf graphs with sparql and keywords. *Bulletin of the Technical Committee on Data Engineering*, (1):16–24, 2010.
[5] Lari Garrison, Reed Stevens, and Andrew Jocuns. Efficient rdf storage and retrieval in jena2. *Exploiting Hyperlinks*, 51(2):35–43, 2004.
[6] Katerina Gkirtzou, George Papastefanatos, and Theodore Dalamagas. Rdf keyword search based on keywords-to-sparql translation. pages 3–5, 2015.
[7] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: ranked keyword searches on graphs. In *ACM SIGMOD International Conference on Management of Data, Beijing, China, June*, pages 305–316, 2007.
[8] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *International Conference on Very Large Data Bases*, pages 505–516, 2005.
[9] Mehdi Kargar and Aijun An. *Keyword search in graphs: finding r-cliques*. VLDB Endowment, 2011.
[10] Graham Klyne, Jeremy J.Carroll, and Brian MeBride. W3c, resource description framework(rdf): concepts and abstract syntax. <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>, 10 February 2004. W3C Recommendation.
[11] Gnter Ladwig and Thanh Tran. *Combining Query Translation with Query Answering for Efficient Keyword Search*. Springer Berlin Heidelberg, 2010.
[12] Wangchao Le, Feifei Li, , Anastasios Kementsietsidis, and Songyun Duan. Scalable keyword search on large rdf data. *Knowledge and Data Engineering IEEE Transactions on*, 26(11):2774–2788, 2014.
[13] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 903–914, 2008.
[14] Xiang Lian, Eugenio De Hoyos, Artem Chebotko, Bin Fu, and Christine Reilly. K-nearest keyword search in rdf graphs. *Journal of Web Semantics*, 22(8):40–56, 2013.
[15] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. *In SIGMOD*, pages 563–574, 2006.

[16] Thomas Neumann and Gerhard Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[17] Eric Prud'hommeaux and Andy Seaborne. W3c, sparql query language for rdf. <`http://www.w3.org/TR/2008/REC-rdf-query-20080115/`>, 15 January 2008. W3C Recommendation.

[18] Chong Sun, Chee Yong Chan, and Amit K. Goenka. Multiway slca-based keyword search in xml data. In *Proceedings of the 16th international conference on World Wide Web*, pages 1043–1052, 2007.

[19] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *IEEE International Conference on Data Engineering*, pages 405–416, 2009.

[20] Gideon Zenz, Xuan Zhou, Enrico Minack, Wolf Siberski, and Wolfgang Nejdl. From keywords to semantic queriesincremental query construction on the semantic web. *Web Semantics Science Services and Agents on the World Wide Web*, 7(3):166–176, 2009.