Scalable Keyword Search on Large RDF Data

Wangchao Le, Feifei Li, Anastasios Kementsietsidis, and Songyun Duan

Abstract—Keyword search is a useful tool for exploring large RDF data sets. Existing techniques either rely on constructing a distance matrix for pruning the search space or building summaries from the RDF graphs for query processing. In this work, we show that existing techniques have serious limitations in dealing with realistic, large RDF data with tens of millions of triples. Furthermore, the existing summarization techniques may lead to incorrect/incomplete results. To address these issues, we propose an effective summarization algorithm to summarize the RDF data. Given a keyword query, the summaries lend significant pruning powers to exploratory keyword search and result in much better efficiency compared to previous works. Unlike existing techniques, our search algorithms always return correct results. Besides, the summaries we built can be updated incrementally and efficiently. Experiments on both benchmark and large real RDF data sets show that our techniques are scalable and efficient.

Index Terms—Keywords search, RDF graph, RDF data

1 Introduction

The resource description framework (RDF) is the *de-facto* standard for data representation on the web. So, it is no surprise that we are inundated with large amounts of rapidly growing RDF data from disparate domains. For instance, the linked open data (LOD) initiative integrates billions of entities from hundreds of sources. Just one of these sources, the DBpedia data set, describes more than 3.64 million *things* using more than 1 billion RDF triples; and it contains numerous keywords, as shown in Fig. 1.

Keyword search is an important tool for exploring and searching large data corpuses whose structure is either unknown, or constantly changing. So, keyword search has already been studied in the context of relational databases [3], [7], [15], [20], XML documents [8], [23], and more recently over graphs [14], [17] and RDF data [11], [24]. However, existing solutions for RDF data have limitations. Most notably, these solutions suffer from: (i) returning incorrect answers, i.e., the keyword search returns answers that do not correspond to real subgraphs or misses valid matches from the underlying RDF data; (ii) inability to scale to handle typical RDF data sets with tens of millions of triples. Consider the results from two representative solutions [14], [23], as shown in Figs. 2 and 3. Fig. 2 shows the query results on three different data sets using the solution specifically designed for RDF in [24], the *Schema* method. While this solution may perform well on data sets that have regular topological structure (e.g., DBLP), it returns incorrect answers for others (e.g., Lehigh University Benchmark (LUBM) [13] etc.) when compared to a naive, but *Exact* method. On the other hand, classical techniques [14] proposed for general graphs can be used for RDF data, but they assume a distance matrix built on the data, which makes it prohibitively expensive to apply to large RDF data set as shown in Fig. 3.

Motivated by these observations, we present a comprehensive study to address the keyword search problem over big RDF data. Our goal is to design a scalable and exact solution that handles realistic RDF data sets with tens of millions of triples. To address the scalability issues, our solution builds a new, succinct and effective summary from the underlying RDF graph based on its *type*s. Given a keyword search query, we use the summary to prune the search space, leading to much better efficiency compared to a baseline solution. To summarize, our contributions are:

- We identify and address limitations in the existing, state-of-the-art methods for keyword search in RDF data [24]. We show that these limitations could lead to incomplete and incorrect answers in real RDF data sets. We propose a new, correct baseline solution based on the backward search idea.
- We develop efficient algorithms to summarize the structure of RDF data, based on the *types* in RDF graphs, and use it to speed up the search. Compared to previous works that also build summary [11], [24], our technique uses different intuitions, which is more scalable and lends significant pruning power without sacrificing the soundness of the result. Further, our summary is light-weight and updatable.
- Our experiments on both benchmark and large real RDF data sets show that our techniques are much more scalable and efficient in correctly answering keyword search queries for realistic RDF workloads than the existing methods.

In what follows, we formulate the keyword search problem on RDF data in Section 2, survey related work in Section 3, present our solutions in Sections 4, Section 5, Section 6, and 7, show experimental results in Section 8, and conclude in Section 9. Table 1 lists the frequently used symbols.

- *W. Le and F. Li are with the School of Computing, University of Utah, Salt Lake City, UT 84112. E-mail: {lew, lifeifei}@cs.utah.edu.*
- *A. Kementsietsidis and S. Duan are with the IBM Thomas J. Watson Research Center, 19 Skyline Dr., Hawthorne, NY 10532. E-mail: tasosk@ca.ibm.com, sduan@us.ibm.com.*

Manuscript received 23 Dec. 2012; revised 16 Jan. 2014; accepted 17 Jan. 2014. Date of publication 22 Jan. 2014; date of current version 26 Sept. 2014. Recommended for acceptance by J. Pei.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2014.2302294

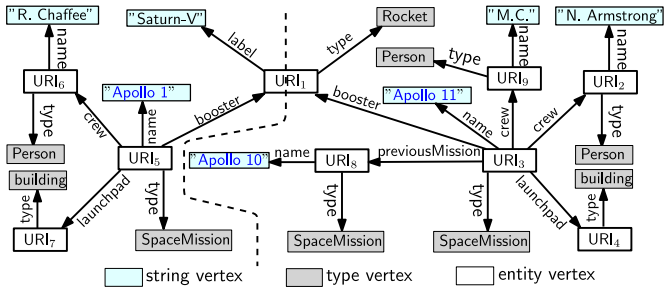


Fig. 1. Keywords in a sample from DBpedia.

2 Preliminaries

An RDF data set is a graph (RDF graph) composed by triples, where a triple is formed by subject, predicate and object in that order. When such ordering is important semantically, a triple is regarded as a directed edge (the predicate) connecting two vertices (from subject to object). Thus, an RDF data set can be alternatively viewed as a directed graph, as shown by the arrows in Fig. 1. W3C has provided a set of unified vocabularies (as part of the RDF standard) to encode the rich semantics. From these, the rdfs:type predicate (or type for short) is particularly useful to our problem (see Section 5), since it provides a classification of vertices of an RDF graph into different groups. For instance in Fig. 1, the entity URI_3 has type SpaceMission. Formally, we view an RDF data set as an RDF graph $G = (V, E)$ where

- V is the union of disjoint sets, V_E, V_T and V_W, where V_E is the set of entity vertices (i.e., URIs), V_T is the set of type vertices, and V_W is 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 entity-entity edges (i.e., connecting two vertices in V_E), E_A is the set of entity-keyword edges (i.e., connecting an entity to a keyword), and E_T is the set entity-type edges (i.e., connecting an entity to a type).

For example in Fig. 1, all gray vertices are type vertices while entity vertices are in white. Each entity vertex also has associated keyword vertices (in cyan). The division on vertices results in a corresponding division on the RDF predicates, which leads to the classification of the edge set E discussed earlier. Clearly, the main structure of an RDF graph is captured by the entity-entity edges represented by the set E_R. As such, an alternative view is to treat an entity vertex and its associated type and keyword vertices as *one vertex*. For example, the entity vertices URI_5, URI_1 and URI_3 from Fig. 1, with their types and keywords, can be viewed as the structure in Fig. 4.

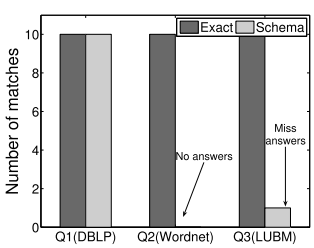


Fig. 2. *Schema* method [24].

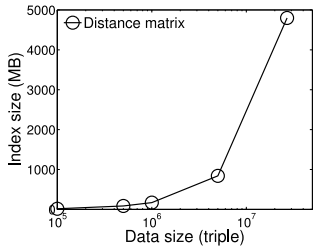


Fig. 3. Distance matrix [14].

In general, for an RDF graph $G = \{V, E\}$, we will refer this as the *condensed view* of G, denoted as $G_c = \{V'_E, E_R\}$. While $|V'_E| \equiv |V_E|$, every vertex $v' \in V'_E$ contains not only the entity value of a corresponding vertex $v \in V_E$, but also the associated keyword(s) and type(s) of v. For the ease of presentation, hereafter we associate a *single* keyword and a *single* type to each entity. Our techniques can be efficiently extended to handle the general cases. Also for simplicity, hereafter, we use $G = \{V, E\}$ to *represent the condensed view* of an RDF graph.

We assume that readers are familiar with the SPARQL query langauge; a brief review of SPARQL is also provided in the online appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2014.2302294, in Section 10.

2.1 Problem Statement

Intuitively, a keyword query against an RDF graph looks for (smallest) subgraphs that contain all the keywords. Given an RDF graph $G = \{V, E\}$, for any vertex $v \in V$, denote the keyword stored in v as $w(v)$. For the ease of presentation, we assume each vertex contains a single keyword. However the solutions we have developed can be seamlessly applied to general cases where a vertex has multiple keywords or no keywords. Formally, a keyword search query q against an RDF data set $G = \{V, E\}$ is defined by m unique keywords $\{w_1, w_2, \ldots, w_m\}$. A set of vertices $\{r, v_1, \ldots, v_m\}$ from V is a *qualified candidate* when:

TABLE 1
Frequently Used Notations

Symbol	Description
$G\{V, E\}$	the condensed view of an RDF graph.
$\mathcal{A}(q)$	top-k answers for a query.
r	answer root.
w_i	the i-th keyword.
$d(x, y)$	graph distance between node x and node y.
$C(q)$	a set of candidate answers.
α	used to denote the α-hop neighborhoods.
g	an answer subgraph of G.
S	the summaries of $\mathcal{P}$.
W_i	the set of nodes containing keyword w_i.
$\mathcal{P}$	partitions.
M	for bookkeeping the candidate answers.
$h(v, \alpha)$, h	the α-hop neighborhoods of v, a partition.
$h_t(v, \alpha)$, h_t	the covering tree of $h(v, \alpha)$, a covering tree.
$\mathcal{S}$	a path represented by a sequence of partitions.
d_l, d_u	the lower and upper bounds (for a path).
σ	a one-to-many mapping in converting h to h_t.
Σ	a set of σ's from a partition h.

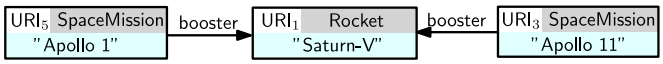


Fig. 4. Condensed view: combining vertices.

- $r \in V$ is called a *root answer node* which is reachable by $v_i \in V$ for $i \in [1, m]$
- $w(v_i) = w_i$.

If we define the answer for q as $\mathcal{A}(q)$ and the set of all qualified candidates in G with respect to q as $C(q)$, then

$$A(q) = \underset{g \in C(q)}{\operatorname{argmin}}\, s(g), \text{and } s(g) = \sum_{r, v_i \in g, i=1..m} d(r, v_i), \quad (1)$$

where $d(r, v_i)$ is the *graph distance* between vertices r and v_i (when treating G as *an undirected graph*). Intuitively, this definition looks for a subgraph in an RDF graph that has minimum length to connect all query keywords from a root node r. In prior works concerning keyword search in RDF data, the graph distance of $d(v_1, v_2)$ is simply the shortest path between v_1 and v_2 in G, where each edge is assigned a weight of 1 (in the case of general graph [14], the weight of each edge could be different). Note that if v_1 and v_2 belong to disconnected parts of G, then $d(v_1, v_2) = +\infty$. Also note that this metric (i.e., eq. (1)) is proposed by [14] and has been used by prior work on keyword search in RDF data [11], [24].

This definition has a top-k version, where the query asks for the top k qualified candidates from $C(q)$. Let the score of a qualified candidate $g \in C(q)$ defined as $s(g)$ in (1), then we can rank all qualified candidates in $C(q)$ in an ascending order of their scores, and refer to the ith ranked qualified candidate as $A(q, i)$. The answer to a top-k keyword search query q is an *ordered set* $\mathcal{A}(\text{II}, \|) = \{\mathcal{A}(\text{II}, \infty), \ldots, \mathcal{A}(\text{II}, \|)\}$. $\mathcal{A}(\text{II})$ is a special case when $k = 1$, and $\mathcal{A}(\text{II}) = \mathcal{A}(\text{II}, \infty)$. Lastly, we adopt the same assumption as in the prior works [14], [23] that the answer roots in $\mathcal{A}$ are *distinct*.

Unless otherwise specified, all proofs in this paper appear in the online appendix, in Section 11.

3 Related Work

For keyword search on generic graphs, many techniques [14], [17] assume that graphs fit in memory, an assumption that breaks for big RDF graphs. For instance, the approaches in [14], [17] maintain a distance matrix for all vertex pairs, and clearly do not scale for graphs with millions of vertices. Furthermore, these works do not consider how to handle updates. A typical approach used here for keyword-search is backward search. Backward search when used to find a Steiner tree in the data graph is NP-hard. He et al. [14] proposed a tractable problem that does not aim to find a Steiner tree and can be answered by using backward search. In this work we extend this problem to large RDF graphs with rigorous analysis, and without depending on the distance matrix.

Techniques for summarizing large graph data to support keyword search were also studied [9]. The graph data are first partitioned into small subgraphs by heuristics. In this version of the problem, the authors assumed edges across the boundaries of the partitions are weighted. A partition is treated as a *supernode* and edges crossing partitions are *superedge*s. The supernodes and superedges form a new graph, which is considered as a summary the underlying graph data. By recursively performing partitioning and building summaries, a large graph can be eventually summarized with a small summary and fit into memory for query processing. During query evaluation, the correspondent supernodes containing the keywords being queried are unfolded and the respective portion of graph are fetched from external memory for query processing. This approach is proposed for generic graphs, and cannot be extended for RDF data, as edges in RDF data are predicates and are not weighed. Furthermore, the portion of the graph that does not contain any keyword being queried is still useful in query evaluation, therefore, this approach cannot be applied to address our problem. A summary built in this manner is not updatable.

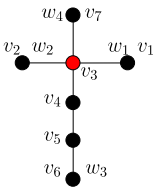
Keyword search for RDF data has been recently studied in [24], which adopted the popular problem definition from [14] as we do in this paper. In this approach, a schema to represent the relations among entities of distinct types is summarized from the RDF data set. Backward search is first applied on the schema/summary of the data to identify promising relations which could have all the keywords being queried. Then, by translating these relations into search patterns in SPARQL queries and executing them against the RDF data, the actual subgraphs are retrieved.

The proposed summarization process in [24] has a limitation: it bundles all the entities of the same type into one node in its summary, which loses too much information in data as to how one type of entities are connected to other types of entities. As a result, this approach could *generates erroneous results (both false positives and false negatives)*. We have given one example in Fig. 2. As another example, consider Fig. 1. In this approach, all vertices of the type *SpaceMission* are represented by one node named *SpaceMission* in the summary. Then, summarizing the predicate *previousMission* connecting URI_3 and URI_8 results in a self-loop over the node *SpaceMission* in the summary, which is incorrect as such a loop does not exist in the data. To be more concrete, when a user asks for all the space missions together with their previous missions, the search pattern in SPARQL would be {`?x PreviousMission ?x. ?x type SpaceMission.`}, which is resultless in DBpedia. Furthermore, such a summary does not support updates. While we also built our summarization using *type information*, our summarization process uses different intuitions, which *guarantees* (a) the soundness of the results; and (b) the support of efficient updates.

There are other works related to keyword search on graphs. In [19], a 3-in-1 method is proposed to answer keyword search on structured, semi-structured and unstructured data. The idea is to encode the heterogeneous relations as a graph. Similar to [14], [17], it also needs to maintain a distance matrix. An orthogonal problem to keyword search on graph is the study of different ranking functions. This problem is studied in [11], [12]. In this work, we adopt the standard scoring function in previous work in RDF [24] and generic graphs [14].

4 The Baseline Method

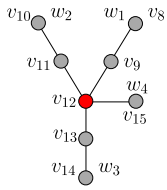
A baseline solution is based on the *"backward search"* heuristic. Intuitively, the "backward search" starts simultaneously



(a)

	expansion step			
	1	2	3	4
v_1	v_3	v_2 v_4 v_7	v_5	v_6
v_2	v_3	v_1 v_4 v_7	v_5	v_6
v_6	v_5	v_4	v_3	v_1 v_2 v_7
v_7	v_3	v_1 v_2 v_4	v_5	v_6

(b)



(c)

Fig. 5. Backward search.

from each vertex in the graph G that corresponds to a query keyword, and expands to its neighboring nodes recursively until a candidate answer is generated. A *termination condition* is used to determine whether the search is complete.

The state-of-the-art keyword search method on RDF graphs [24] has applied the backward search idea. Their termination condition is to stop the search whenever the expansions originating from m vertices $\{v_1, \ldots, v_m\}$ (each corresponding to a distinct query keyword) meet at a node r *for the first time*, where $\{r, v_1, \ldots, v_m\}$ is returned as the answer. Unfortunately, this termination condition is *incorrect*.

Counter example. Consider the graph in Fig. 5a and a top-1 query $q = \{w_1, w_2, w_3, w_4\}$. The steps for the four backward expansions performed on Fig. 5a are shown in Fig. 5b. Using the above termination condition, the backward expansions from the four vertices $\{v_1, v_2, v_6, v_7\}$ covering the query keywords $\{w_1, w_2, w_3, w_4\}$ meet for the first time in the second iteration, so the candidate answer $g = \{r = v_4, v_1, v_2, v_6, v_7\}$ is returned and $s(g) = 8$. However, if we continue to the next iteration, the four expansions will meet again at v_3, with $g' = \{r = v_3, v_1, v_2, v_6, v_7\}$ and $s(g') = 6$, which is the best answer. One may argue that the graph covering the query keywords is still correctly identified. However, it will be problematic if we also consider the graph in Fig. 5c as input for the search. There, the best possible answer would be $g'' = \{r = v_{12}, v_8, v_{10}, v_{14}, v_{15}\}$ and $s(g'') = 7 < s(g)$. Hence, g'' will be declared as the top-1 answer for q instead of g', which is clearly incorrect. Furthermore, later we will explain that even if we fix this error in the terminating condition, their method [24] may still return incorrect results due to the limitations in the summary it builds, as shown by the results in Fig. 2.

The correct termination. Next, we show the correct termination condition for the backward search on RDF data. The complete algorithm appears in Algorithm 1.

Data structures. Given $q = \{w_1, \ldots, w_m\}$ and a (condensed) RDF graph $G = \{V, E\}$, we use W_i to denote the set of vertices in V containing the keyword w_i (line 1). We initialize m empty priority queues (e.g., min-heaps) $\{a_1, ..a_m\}$, one for each query keyword (line 1). We also maintain a set M of elements (line 2), one for each distinct node we have explored so far in the backward expansion to track the state of the node, i.e., what keywords are reachable to the node and their best known distances. In what follows, we use $M[v]$ to indicate the bookkeeping for the node v. Specifically, in each element of M, we store a list of m (`vertex`, `distance`) pairs. A (`vertex`, `distance`) pair in the j-th entry of $M[v]$ indicates a (shortest) path from `vertex` that reaches v in `distance` hops and it is the shortest possible path starting from any instance of w_j (recall that there could have multiple copies of w_j in G). Next, we also use $M[v][j]$ to indicate the j-th pair in $M[v]$. For instance in Fig. 5a, consider an element $M[v_3] = \{(v_1, 1), (v_2, 1), \text{nil}, (v_7, 1)\}$ in M. The entry indicates that v_3 has been reached by three expansions from vertices v_1, v_2 and v_7, containing keywords w_1, w_2 and w_4 respectively—each can reach v_3 in one hop. However, v_3 has not been reached by any expansion from any vertex containing w_3 yet.

Algorithm 1: BACKWARD

Input: $q = \{w_1, w_2, \ldots, w_m\}$, $G = \{V, E\}$
Output: top-k answer $\mathcal{A}(q)$

1 Initialize $\{W_1, ..W_m\}$ and m min-heaps $\{a_1, ..a_m\}$;
2 $M \leftarrow \emptyset$; `// for tracking potential` $C(q)$
3 **for** $v \in W_i$ **and** $i = 1..m$ **do**
4 **for** $\forall u \in V$ **and** $d(v, u) \leq 1$ **do**
5 $a_i \Leftarrow (v, p \leftarrow \{v, u\}, d(p) \leftarrow 1)$; `// enqueue`
6 **if** $u \notin M$ **then** $M[u] \leftarrow \{\text{nil}, ..(v, 1).., \text{nil}\}$;
7 **else** $M[u][i] \leftarrow (v, 1)$; `↑the i-th entry`
8 **while not** *terminated* **and** $\mathcal{A}$ **not** *found* **do**
9 $(v, p, d(p)) \leftarrow \texttt{pop}(arg\min_{i=1}^{m}\{\texttt{top}(a_i)\})$;
10 **for** $\forall u \in V$ **and** $d(v, u) = 1$ **and** $u \notin p$ **do**
11 $a_i \Leftarrow (u, p \cup \{u\}, d(p) + 1)$;
12 update M the same way as in lines 6 and 7;
13 **return** $\mathcal{A}$ (if found) **or nil** (if not);

The algorithm. With the structures in place, the algorithm proceeds in iterations. In the first iteration (lines 3-7), for each vertex v from W_i and every neighbor u of v (including v itself), we add an entry $(v, p \leftarrow \{v, u\}, d(p))$ to the priority queue a_i (entries are sorted in the ascending order of $d(p)$ where p stands for a path and $d(p)$ represents its length). Next, we look for the newly expanded node u in M. If $u \in M$, we simply replace $M[u][i]$ with $(v, d(\mathsf{p}))$ (line 7). Otherwise, we initialize an empty element for $M[u]$ and set $M[u][i] = (v, d(\mathsf{p}))$ (line 6). We repeat this process for all W_i's for $i = 1..m$.

In the j-th ($j > 1$) iteration of our algorithm (lines 8-12), we pop the smallest top entry of $\{a_1..a_m\}$ (line 9), say an entry $(v, p = \{v, \ldots, u\}, d(p))$ from the queue a_i. For each neighboring node u' of u in G such that u' is not in p yet (i.e., not generating a cycle), we push an entry $(v, p \cup \{u'\}, d(p)+1)$ back to the queue a_i (line 11). We also update M with u' similarly as above (line 12). This concludes the j-th iteration.

In any step, if an entry $M[u]$ for a node u has no nil pairs in its list of m (`vertex`, `distance`) pairs, this entry identifies a candidate answer and u is a candidate root. Due to the property of the priority queue and the fact that all edges have a unit weight, the paths in $M[u]$ are the shortest paths to u from m distinct query keywords.Denote the graph concatenated by the list of shortest paths in $M[u]$ as g. We have:

Lemma 1. *$g = \{r = u, v_{\ell_1}, \ldots, v_{\ell_m}\}$ is a candidate answer with $s(g) = \sum_{i=1}^{m} d(u, v_{\ell_i})$.*

A node v is not fully explored if it has not been reached by at least one of the query keywords. Denote the set of vertices that are not fully explored as V_t, and the top entries from the m expansion queues (i.e., min-heaps) $a_1..a_m$ as $(v_1, p_1, d(p_1)), \ldots, (v_m, p_m, d(p_m))$. Consider two cases: (i) an

unseen vertex, i.e., $v \notin M$, will become the answer root; (ii) a seen but not fully expanded vertex $v \in M$ will become the answer root. The next two lemmas bound the optimal costs for these two cases respectively. For the first case, Lemma 2 provides a lower bound for the best potential cost.

Lemma 2. *Denote the best possible candidate answer as g_1, and a vertex $v \notin M$ as the answer root of g_1. Then it must have $s(g_1) > \sum_{i=1}^{m} d(p_i)$.*

For the second case, it is clearly that $v \in V_t$. Assume the list stored in $M[v]$ is $(v_{b_1}, d_1), \ldots, (v_{b_m}, d_m)$. Lemma 3 shows a lower bound for this case.

Lemma 3. *Suppose the best possible candidate answer using such an v ($v \in M$ and $v \in V_t$) as the answer root is g_2, then*

$$s(g_2) > \sum_{i=1}^{m} f(v_{b_i}) d_i + (1 - f(v_{b_i})) d(p_i), \quad (2)$$

where $f(v_{b_i}) = 1$ if $M[v][b_i] \neq$nil, and $f(v_{b_i}) = 0$ otherwise.

Notice that in Lemma 3, if $M[v][b_i] \neq \text{nil}$, then $d(p_i) \geq d_i$ due to the fact that a_i is a min-heap. It follows $s(g_2) \leq s(g_1)$.

The termination condition. These v's represent all nodes that have not been fully explored. For case (i), we simply let $s(g_1) = \sum_{i=1}^{m} d(p_i)$; for case (ii), we find a vertex with the smallest possible $s(g_2)$ value w.r.t. the RHS of (2), and simply denote its best possible score as $s(g_2)$.

Denote the kth smallest candidate answer identified in the algorithm as g, our search can safely terminate when $s(g) \leq \min(s(g_1), s(g_2)) = s(g_2)$. We denote this algorithm as BACKWARD. By Lemmas 1, 2, 3, we have Theorem 1:

Theorem 1. *The BACKWARD method finds the top-k answers $\mathcal{A}(q, k)$ for any top-k keyword query q on an RDF graph.*

5 TYPE-BASED SUMMARIZATION

The BACKWARD method is clearly not scalable on large RDF graphs. For instance, the keyword "Armstrong" appears 269 times in our experimental DBpedia data set, but only one is close to the keyword "Apollo 11", as in Fig. 1. If we are interested in the smallest subgraphs that connect these two keywords, the BACKWARD method will initiate many random accesses to the data on disk, and has to construct numerous search paths in order to complete the search. However, the majority of them will not lead to any answers. Intuitively, we would like to reduce the input size to BACKWARD and apply BACKWARD only on the most promising subgraphs. We approach this problem by proposing a type-based summarization approach on the RDF data. The idea is that, by operating our keyword search initially on the summary (which is typically much smaller than the data), we can navigate and prune large portions of the graph that are irrelevant to the query, and only apply BACKWARD method on the smaller subgraphs that guarantee to find the optimal answers.

The intuition. The idea is to first induce partitions over the RDF graph G. Keywords being queried will be first concatenated by partitions. The challenge lies on how to safely prune connections (of partitions) that will not result in any top-k answer. To this end, we need to calibrate the length of a path in the backward expansion that crosses a partition. However, maintaining the exact distance for every possible path is expensive, especially when the data is constantly changing. Therefore, we aim to distill an updatable summary from the distinct structures in the partitions such that any path length in backward expansion can be effectively estimated. The key observation is that neighborhoods in close proximity surrounding vertices of the same type often share *similar structures* in how they connect to vertices of other types.

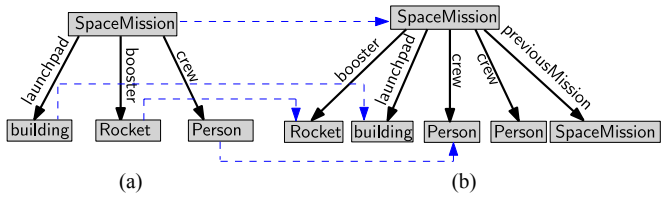


Fig. 6. Graph homomorphism across summaries.

Example 1. Consider the condensed view of Fig. 1. The graph in Fig. 6a is common for the 1-hop neighborhoods of URI_3 and URI_5 with the type SpaceMission.

This observation motivates us to study how to build a type-based summary for RDF graphs. A similar effort can be seen in [23], where a *single* schema is built for all the types of entities in the data. However, this is too restrictive as RDF data is known to be schemaless [10], e.g., entities of the same type do not have a unified property conformance.

5.1 Outline and Preliminaries

Our approach starts by splitting the RDF graph into multiple, smaller partitions. Then, it defines a minimal set of common type-based structures that summarizes the partitions. Intuitively, the summary bookkeeps the distinct structures from all the partitions. In general, the keyword search can benefit from the summary in two perspectives. With the summary,

- we can obtain the upper and lower bounds for the distance traversed in any backward expansion without constructing the actual path (Section 6); and
- we can efficiently retrieve *every* partition from the data by collaboratively using SPARQL query and any RDF store without explicitly storing the partition (Section 14).

We first introduce two notions from graph theory: *graph homomorphism* and *core*.

Homomorphism across partitions. As in Fig. 6a, type vertices at close proximity are a good source to generate induced partitions of the data graph. However, if we were to look for such induced partitions that are exactly the same across the whole graph, we would get a large number of them. Consider another type-based structure in Fig. 6b, which is extracted from 1-hop neighbors around the vertex URI_3 in Fig. 1. Notice the two graphs are different, however Fig. 6a is a subgraph of Fig. 6b. We consider discovering such embeddings between the induced partitions, so that one template can be reused to bookkeep multiple structures.

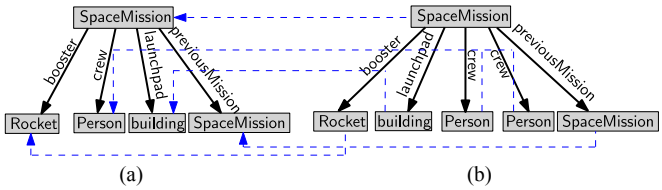


Fig. 7. Build a core (a) from (b).

Definition 1. *A graph homomorphism f from a graph $G = \{V, E\}$ to a graph $G' = \{V', E'\}$, written as $f : G \to G'$, is a mapping function $f : V \to V'$ such that (i) $f(x) = x$ indicates that x and $f(x)$ have the same type; and (ii) $(u, v) \in E$ implies $(f(u), f(v)) \in E'$ and they have the same label. When such an f exists, we say G is homomorphic to G'.*

Intuitively, embedding G to G' not only reduces the number of structures we need to keep but also preserve any path from G in G', as shown by Fig. 6 (more expositions in Section 6). Finally, notice that homomorphism is transitive, i.e., $G \to G'$ and $G' \to G''$ imply that $G \to G''$.

Cores for individual partitions. A *core* is a graph that is only homomorphic to itself, but not to any one of its proper subgraphs (i.e., there is no homomorphism from a core to any of its proper subgraphs).

Definition 2. *A core c of a graph G is a graph with the following properties: there exists a homomorphism from c to G; there exists a homomorphism from G to c; and c is minimal (in the number of vertices) with these properties.*

Intuitively, a core succinctly captures how different types of entities are connected, e.g., the partition in Fig. 7b is converted to its core in Fig. 7a by eliminating one branch.

5.2 Partition

The summarization process starts with splitting the data graph into smaller but semantically similar and *edge-disjoint* subgraphs. Given our observation that nodes with the same type often share similar type-neighborhoods, we induce a distinct set of partitions for G based on the types in G, using small subgraphs surrounding vertices of the same type. Our partitioning algorithm treats an input RDF data set as a *directed graph* G concerning only the type information, i.e., we use the condensed view of an RDF graph. For any vertex that does not have a type specified by the underlying data set, we assign an universal type NA to them. Notice that graph partitioning is a well studied problem in the literature, here we do not propose any new technique in that respect but rather focus on how to build semantically similar partitions for our purpose. The partitioning algorithm is shown in Algorithm 2

In Algorithm 2, suppose G has n distinct number of types $\{\mathsf{T}_1, \ldots, \mathsf{T}_n\}$, and we use the set V_i to represent the vertices from V that have a type T_i (line 4). We define the *α-neighborhood surrounding a vertex*, where α is a parameter used to produce a set of edge disjoint partitions $\mathcal{P}$ over G. Formally, for any vertex $v \in V$ and a constant α, the α-neighborhood of v is the subgraph from G obtained by expanding v with α hops in a breadth-first manner, denoted as $h(v, \alpha)$ (line 5), but subject to the constraint that the expansion only uses edges which have not been included by any partition in $\mathcal{P}$ yet. We define the i-hop neighboring nodes of v as the set of vertices in G that can be connected to v through a *directed path* with exactly i directed edges. Note that since we are using directed edges, it is possible the i-hop neighboring nodes of v is an empty set. Clearly the nodes in $h(v, \alpha)$ are a subset of the α-hop neighboring nodes of v (since some may have already been included in another partition).

To produce a partition $\mathcal{P}$, we initialize $\mathcal{P}$ to be an empty set (line 2) and then iterate all distinct types (line 3). For a type T_i and for each vertex $v \in V_i$, we find its α-neighborhood $h(v, \alpha)$ and add $h(v, \alpha)$ as a new partition into $\mathcal{P}$. The following lemma summarizes the properties of our construction:

Algorithm 2: Partition

Input: $G = \{V, E\}$, α
Output: A set of partitions in $\mathcal{P}$
1 Let $\mathcal{T} = \{\mathsf{T}_1, \ldots, \mathsf{T}_n\}$ be the distinct types in V;
2 $\mathcal{P} \leftarrow \emptyset$;
3 **for** $\mathsf{T}_i \in \mathcal{T}$ **do**
4 **for** $v \in V_i$ **do**
5 identify $h(v, \alpha)$ – the α neighborhood of v;
6 $E \leftarrow E - \{\text{triples in } h(v, \alpha)\}$ and
 $\mathcal{P} \leftarrow \mathcal{P} \cup h(v, \alpha)$;
7 **return** $\mathcal{P}$;

Lemma 4. *Partitions in $\mathcal{P}$ are edge disjoint and the union of all partitions in $\mathcal{P}$ cover the entire graph G.*

It is worth pointing out that Algorithm 2 takes an ad hoc order to partition the RDF data, i.e., visiting the set of entities from type 1 to type n in order. A different order to partition the data could lead to different performance in evaluating a keyword query. However, finding an optimal order to partition the RDF data set is beyond the scope the paper, therefore we decide not to expand the discussion on this issue.

Note that the order in which we iterate through different types may affect the final partitions $\mathcal{P}$ we build. But no matter which order we choose, vertices in the same type always induce a set of partitions based on their α-neighborhoods. For example, the partitions $\mathcal{P}$ of Fig. 1 (as condensed in Fig. 4) are always the ones shown in Fig. 8, using $\alpha = 1$.

5.3 Summarization

The intuition of summarization technique is as follows. The algorithm identifies a set of templates from the set of partitions $\mathcal{P}$. Such templates serve as a summary for the partitions. In addition, the summarization algorithm guarantees that every partition in $\mathcal{P}$ is homomorphic to one of the templates in the summary. As we will show in Section 6, this property allows the query optimizer to (i) efficiently estimate any path length in the backward expansion without frequently accessing the RDF data being queried; and (ii) efficiently reconstruct the partitions of interest by querying the RDF data without explicitly storing and indexing the partitions.

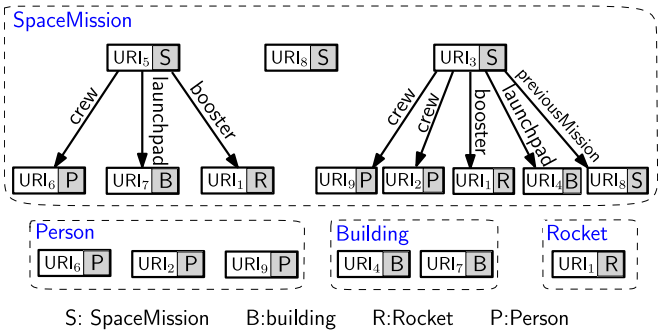


Fig. 8. Partitions $\mathcal{P}$ of the RDF data in Fig. 1, $\alpha = 1$.

We first outline our approach to summarize the distinct structures in a partition $\mathcal{P}$. Then, we discuss how to make it more practical by proposing our optimizations. Finally, we discuss the related indices in Section 5.4. The general framework of our approach is shown in Algorithm 3.

Given a partition $\mathcal{P}$, Algorithm 3 retrieves all the distinct structures and stores them in a set S. Algorithm 3 begins with processing partitions in $\mathcal{P}$ in a loop (line 2). For a partition h_i, we use its core c to succinctly represent the connections between different types in h_i (line 3). Once a core c is constructed for a partition, we scan the existing summary structures in S to check (a) if c is homomorphic to any existing structure s_i in S; or (b) if any existing structure s_i in S is homomorphic to c. In the former case, we terminate the scan and S remains intact (without adding c), as in lines 5 and 6; in the latter case, we remove s_i from S and continue the scan, as in lines 7 and 8. When S is empty or c is not homomorphic to any of the structures in S after a complete scan on S, we add c into S. We repeat the procedure for all partitions in $\mathcal{P}$.

Algorithm 3: Summarize structures in $\mathcal{P}$

Input: $\mathcal{P} = \{h(v_1, \alpha), h(v_2, \alpha), \ldots\}$
Output: A set of summaries in S

1 $S \leftarrow \emptyset$;
2 **for** $h_i \in \mathcal{P}$, $i = 1, \ldots, |\mathcal{P}|$ **do**
3 $\quad c \leftarrow core(h_i)$; //see discussion on optimization
4 $\quad$ **for** $s_j \in S$, $j = 1, \ldots, |S|$ **do**
5 $\quad\quad$ **if** $f : c \rightarrow s_j$ **then**
6 $\quad\quad\quad$ **goto** line 2; // also bookkeep $f : c \rightarrow s_j$
7 $\quad\quad$ **else if** $f : s_j \rightarrow c$ **then**
8 $\quad\quad\quad$ $S \leftarrow S - \{s_j\}$; //also bookkeep $f : s_j \rightarrow c$
9 $\quad S \leftarrow S \cup \{c\}$;
10 **return** S;

Improving efficiency and reducing $|S|$. There are two practical problems in Algorithm 3. First, the algorithm requires testing subgraph isomorphism for two graphs in lines 3, 5 and 7, which is NP-hard. Second, we want to reduce $|S|$ as much as possible so that it can be cached in memory for query processing. The latter point is particularly important for RDF data sets that are known to be irregular, e.g., DBpedia.

The optimization is as follows. Before line 3 of Algorithm 3, consider each partition $h(v, \alpha)$ in $\mathcal{P}$, which visits

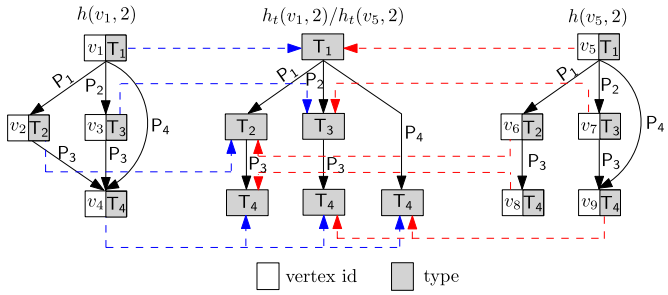


Fig. 9. A tree structure for two partitions.

the α-neighborhood of v in a breadth-first manner. We redo this traversal on $h(v, \alpha)$ and construct a *covering tree* for the edges in $h(v, \alpha)$, denoted as $h_t(v, \alpha)$. In more detail, for each visited vertex in $h(v, \alpha)$, we extract its type and create a *new* node in $h_t(v, \alpha)$ (even if a node for this type already exists). By doing so, we build a tree $h_t(v, \alpha)$ which represents all the distinct type-paths in $h(v, \alpha)$. In the rest of the algorithm (lines 3-10), we simply replace $h(v, \alpha)$ with $h_t(v, \alpha)$.

Example 2. As in Fig. 9, a tree $h_t(v_1, 2)$ is built for the partition $h(v_1, 2)$. Notice that the vertex v_4 is visited three times in the traversal (across three different paths), leading to three distinct nodes with type T_4 created in $h_t(v_1, 2)$. In the same figure, a tree $h_t(v_5, 2)$ is built from the partition $h(v_5, 2)$ and isomorphic to $h_t(v_1, 2)$.

There are two motivations behind this move. First, using the covering tree instead of the exact partition potentially reduces the size of the summary S. As seen in Fig. 9, two partitions with distinct structures at the data level (e.g., $h(v_1, 2)$ and $h(v_5, 2)$) could share an identical structure at the type level. Taking advantage of such overlaps is the easiest way to reduce the number of distinct structures in S. The second reason is efficiency. Whereas testing subgraph isomorphism is computationally hard for generic graphs, there are polynomial time solutions if we can restrict the testing on trees [22]—leading to better efficiency. For instance, to find the core of a covering tree h_t, it simply amounts to a bottom-up and recursive procedure to merge the homomorphic branches under the same parent node in the tree.

5.4 Auxiliary Indexing Structures

To facilitate the keyword search, along with the summary S, we maintain three auxiliary (inverted) indexes.

A *portal* node ℓ is a data node that is included in more than one partitions (remember that partitions are edge-disjoint, not node disjoint). Intuitively, a portal node joins different partitions. A partition may have multiple portals but usually much less than the total number of nodes in the partition. Portal nodes allow us to concatenate different partitions. In the first index, dabbed *portal index*, for each partition $h(v, \alpha)$, we assign it a unique id, and associate it with the list of portals in the partition. In practice, since the partition root node v is unique in each partition, we can simply use it to denote the partition $h(v, \alpha)$ when the context is clear.

Recall that we use $h_t(v, \alpha)$ to represent $h(v, \alpha)$, where a vertex in $h(v, \alpha)$ could correspond to more than one vertex

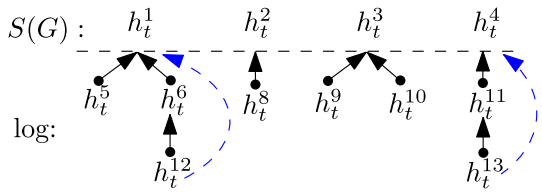


Fig. 10. All the homomorphism in building S.

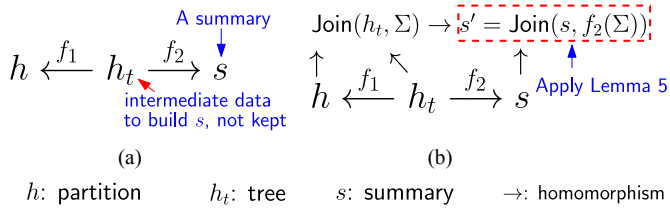


Fig. 11. Homomorphic mappings.

in $h_t(v,\alpha)$. $\sigma(v_i)$ registers the fact that there are more than one v_i in $h_t(v,\alpha)$ and $\sigma(v_i)$ also denotes the set of all v_is in $h_t(v,\alpha)$. Without loss of generality, let $\Sigma = \{\sigma(v_1), \sigma(v_2), \ldots\}$ denote all the one-to-many mappings in a partition. For instance, consider $h(v_1,2)$ and $h_t(v_1,2)$ in Fig. 9, $\Sigma \leftarrow \{\sigma(v_4) = \{T_4\}\}$. The second index, dabbed *partition index*, is to map the partition root v of $h(v,\alpha)$ to its Σ. Intuitively, this index helps rebuild from $h_t(v,\alpha)$ a graph structure that is similar to $h(v,\alpha)$ (more rigorous discussion in Section 6).

The third index, dabbed *summary index*, maps data nodes in partitions to summary nodes in S. In particular, we assign a unique id sid to each summary in S and denote each node in S with a unique id nid. For any node u in a partition $h(v,\alpha)$, this index maps the node u to an entry that stores the partition root v, the id sid of the summary and the id nid of the summary node that u corresponds to. Notice that since $h_t(v,\alpha)$ is built in a breadth-first traversal, we can easily compute the shortest path from v to any node in $h_t(v,\alpha)$ using this index.

To obtain the homomorphic mappings from each $h_t(v,\alpha)$ to a summary in S, one needs to maintain a log for all the homomorphisms found during the construction of S, as in lines 6 and 8. Once S is finalized, we trace the mappings in this log to find the mappings from data to summaries. As each partition (represented by its core) is either in the final S or is homomorphic to *one* other partition, the size of the log is linear to the size of G. An example for the log is in Fig. 10 (h_t^i is the covering tree for the i-th partition). It shows sets of trees (and their homomorphic mappings); each set is associated with a summary in S, that all trees in that set are homomorphic to. To find the final mappings, we scan each set of trees in the log and map the homomorphisms of each entry in a set to the corresponding entry in S, i.e., the blue arrows in Fig. 10. We remove the log once all the mappings to S are found.

6 Keyword Search with Summary

Next, we present a scalable and exact search algorithm. It performs a *two-level* backward search: one backward search at the summary-level, and one at the data-level. Only for identified connected partitions that are found to contain all the distinct keywords at the summary-level and whose score could enter the top-k answers, do we initiate a backward search at the data-level on the selected partitions. Remember that path-length computation is at the heart of backward search and pruning. While working at the summary-level, exact path lengths are not available. Therefore, we first show how to estimate the path length of the actual data represented by our summary. Then, we proceed to describe the algorithm in detail.

6.1 Bound the Shortest Path Length

At the summary-level, any shortest path in the underlying RDF graph must go through a number of partitions, and for each intermediate partition the path connects two of its portals, i.e., an entrance and an exit node. By construction, the shortest distance from the partition root v of a partition to any vertex u in the same partition can be computed with the *summary index*. By triangle inequality, the shortest distance $d(v_1,v_2)$ for any two vertices v_1 and v_2 in a partition with a partition root v can be upper bounded by $d(v_1,v_2) \leq d(v, v_1) + d(v,v_2)$, and lower bounded by $d(v_1,v_2) \geq |d(v,v_1) - d(v,v_2)|$. Yet, a possibly tighter lower bound can be found by using the correspondent summary of the partition that is rooted at v and Lemma 5.

Lemma 5. *Given two graphs g and h, if $f: g \rightarrow h$, then $\forall v_1, v_2 \in g$ and their homomorphic mappings $f(v_1)$, $f(v_2) \in h$, $d(v_1,v_2) \geq d(f(v_1), f(v_2))$.*

The homomorphic mappings between a partition h, its covering tree h_t, and its summary s in S are shown in Fig. 11a. Notice that due to the optimization we employ in Section 5.3, there is no homomorphism from h to s, so that we cannot apply Lemma 5 directly. To obtain a lower bound for the distance of any two vertices in h, we need to rebuild a homomorphic structure for h from its summary s and h_t.

To do so, we first define a mapping function Join, which takes as input a graph g, a list of disjoint sets of vertexes $\{V'_{t_1}, V'_{t_2}, \ldots\}$ and outputs a new graph g', written as $g' = \text{Join}(g(V,E), \{V'_{t_1}, V'_{t_2}, \ldots\})$. In particular, $V'_{t_i} \subseteq V$ and vertexes in V'_{t_i} are all of type t_i. The function Join constructs g' as follows:

1. Initialize g' as g.
2. For the vertexes in V'_{t_i} of g', merge them into a single node v'_i of type t_i, and all edges incident to the vertexes in V'_{t_i} are now incident to v'_i.
3. Repeat step 2 for all is.

Notice that the function Join itself is a homomorphic mapping, which constructs a homomorphism from g to g'. Also recall $\sigma(x)$ of a partition h registers the fact that a type node x in h has more than one replicas in h_t and hence $\sigma(x)$ is a one-to-many mapping from x in h to the set of *all* vertexes of type x in h_t and Σ of h_t is the set of all such mappings.

Example 3. Use the examples in Fig. 9 for illustration. $\text{Join}(h_t(v_1,2), \{\sigma(\mathsf{T}_4)\})$ rebuilds $h(v_1,2)$ in Fig. 9 and hence there is a homomorphism from $h(v_1,2)$ to $\text{Join}(h_t(v_1,2), \{\sigma(\mathsf{T}_4)\})$, i.e., isomorphism. On the other hand, Join $(h_t(v_5,2), \{\sigma(\mathsf{T}_4)\})$ equals $h(v_1,2)$. Although it does not reconstruct $h(v_5,2)$ this time, the Join function in this case still produces a homomorphic mapping from

w_1	w_2
t_1=(v_a, $\{(\ell_2, v_0)\}$, 5, 7)	t_2=(v_b, $\{(\ell_1, v_4), (\ell_0, v_5)\}$, 3, 5)
	t_3=(v_c, $\{(\ell_3, v_2)\}$, 5, 6)

Fig. 12. An entry in M for the partition rooted at v.

$h(v_5, 2)$ to $\mathsf{Join}(h_t(v_5, 2), \{\sigma(\mathsf{T}_4)\})$ since it is not hard to see that $h(v_5, 2)$ is homomorphic to $h(v_1, 2)$ in Fig. 9.

More formally, we have:

Lemma 6. *For a partition h and its covering tree h_t, there is a homomorphism from h to* $\mathsf{Join}(h_t, \Sigma)$.

In what follows, we show how to build a homomorphism from $\mathsf{Join}(h_t, \Sigma)$ to a graph s' derived from the summary s of h_t. With this and by the transitivity of homomorphism and Lemma 6, it follows h is homomorphic to s' and hence Lemma 5 can be applied (as shown in Fig. 11b).

By Algorithm 2, every h_t from a partition h is homomorphic to a summary s in S (see the relations in Fig. 11a). Assume the homomorphism is $f_2 : h_t \rightarrow s$. Given the Σ of a partition h, define $f_2(\Sigma) = \{f_2(\sigma(v)) \,|\, \sigma(v) \in \Sigma\}$ where $f_2(\sigma(v)) = \{f_2(u) \,|\, u \in \sigma(v) \wedge u \in h_t \wedge f_2(u) \in s\}$, i.e., the set of the mapped vertexes of $\sigma(v)$ in s by the homomorphism f_2. Further, we have the following result:

Lemma 7. *For a partition h, its covering tree h_t and its summary s that has $f_2 : h_t \rightarrow s$, there is a homomorphism from* $\mathsf{Join}(h_t, \Sigma)$ *to* $\mathsf{Join}(s, f_2(\Sigma))$.

By Lemmas 6, 7 and the transitivity of homomorphism, a partition h is homomorphic to $\mathsf{Join}(s, f_2(\Sigma))$, as shown in Fig. 11b. Notice f_2 is a part of our *summary index*, which maps a vertex in data to a vertex in summary. Finally, given any two vertices in a partition h, their shortest distance can be (lower) bounded by combining Lemmas 5, 6, 7 and using any shortest path algorithm, e.g., Dijkstra's algorithm, to find the shortest distance between the correspondent mappings on $\mathsf{Join}(s, f_2(\Sigma))$. In practice, we use the larger lower bound from either the summary or the triangle inequality.

6.2 The Algorithm

The algorithm is in Algorithm 4, dabbed the SUMM method.

Data structures. Like the BACKWARD method in Section 4, we define $\{W_1, W_2, \ldots, W_m\}$, where W_i is the set of vertexes containing the keyword w_i. We also initialize m priority queues $\{a_1, \ldots, a_m\}$ and maintain a set M of entries, one for each visited partition. Each entry in M stores a unique partition root followed by m lists. The i-th list records all the reachable vertexes containing keyword w_i and through which other partitions they connect to the current partition in the search. An entry of M is in the form of quadruples – each can be represented as (u, $\mathcal{S}$, d_l, d_u). Here, the node u is the first vertex in the backward expansion and contains the keyword w_i. The expansion reaches the current partition by routing through a sequence of the portals from some partitions, stored in $\mathcal{S}$ as a sequence of (*portal*, *partition root*) pairs. A sequence $\mathcal{S}$ defines a path (of partitions) that starts at u. Denote the lower and upper bounds for the path in $\mathcal{S}$ as d_l and d_u.

Example 4. A sub-sequence $\{(\ell, v_a), (\ell', v_b)\}$ of $\mathcal{S}$ indicates that the path first enters the partition rooted at v_a and exits the partition from one of its portals ℓ. From the portal ℓ, this path then enters a partition rooted at v_b and leaves the partition from the portal ℓ'. We are interested in (lower and upper) bounding the shortest distance that connects two adjacent portals in $\mathcal{S}$, e.g., $d(\ell, \ell')$ in the partition rooted at v_b.

Algorithm 4: SUMM

Input: $q = \{w_1, w_2, \ldots, w_m\}$, $G = \{V, E\}$
Output: top-k answer $\mathcal{A}$

1 initialize $\{W_1, ..W_m\}$ and m min-heaps $\{a_1, ..a_m\}$;
2 $M \leftarrow \emptyset$; // for tracking partitions
3 **for** $u \in W_i$ **and** $i = 1..m$ **do**
4 **if** $u \in h(v, \alpha)$ **then**
5 $t \leftarrow (u, \{\emptyset\}, 0, 0)$;
6 $a_i \Leftarrow (v, t)$; // enqueue
7 **if** $v \notin M$ **then** $M[v] \leftarrow \{\text{nil}, ..., \underline{t}, ..., \text{nil}\}$;
8 **else** $M[v][i] \leftarrow t$; ↑the i-th entry
9 **while not** *terminated* **and** $\mathcal{A}$ **not** *found* **do**
10 $(v, (\mathsf{u}, \mathcal{S}, d_l, d_u)) \leftarrow \mathtt{pop}(arg\min_{i=1}^{m}\{\mathtt{top}(a_i)\})$;
11 Denote the last entry in $\mathcal{S}$ as (ℓ, v_ℓ) and $\mathcal{L} = \{\ell'_1, \ell'_2, \ldots\}$ be the portals in the partition rooted at v;
12 **for** $\forall \ell' \in \mathcal{L}$ **do**
13 compute d'_l and d'_u for $d(\ell, \ell')$ or $d(\mathsf{u}, \ell')$;
14 let $t \leftarrow (\mathsf{u}, \mathcal{S} \cup (\ell', v_r), d_l + d'_l, d_u + d'_u)$;
15 update $M[v_r]$ with t; // see discussions
16 **if** $M[v_r]$ *is updated* **and** *nil* $\notin M[v_r]$ **then**
17 $a_i \Leftarrow (v_r, t)$; // enqueue
18 **for** *each new subgraph g incurred by t* **do**
19 retrieve g from data;
20 apply BACKWARD on g **and** update $\mathcal{A}$;
21 **return** $\mathcal{A}$ (if found) **or nil** (if not);

Example 5. In Fig. 12, assume $m = 2$ (i.e., the query has two keywords) and an entry in M for a partition rooted at v is shown as below. The entry records that there is a path (of partitions) from w_1 that reaches the current partition rooted at v. This path starts at v_a, enters the concerned partition from portal ℓ_2 and has a length of at least 5 hops and at most 7 hops. To reach the partition rooted at v, the path has already passed through a partition rooted at v_0. Same for w_2, the concerned partition is reachable from two paths starting at v_b and v_c respectively, both contain the keyword w_2.

The algorithm. With the data structures in place, the algorithm proceeds in iterations.

- *In the first iteration.* For each vertex u from W_i, we retrieve the partition root v that u corresponds to, from the *summary index*. Next, if there is an entry for v in M, we append a quadruple $t = (\mathsf{u}, \{\emptyset\}, 0, 0)$ to the i-th list of the entry; otherwise we initialize a new entry for v in M (with m empty lists) and update the i-th list with t, as in lines 7 and 8. We also add an entry (v, t) to the priority queue a_i (entries in the priority queue are sorted in ascending order by their

```
SELECT * WHERE{URI5 name "A1". URI5 type S.
OPTIONAL{URI5 launchPad ?x. ?x type B.}
OPTIONAL{URI5 booster ?y. ?y type R}
OPTIONAL{URI5 crew ?z. ?z type C} .
OPTIONAL{URI5 previousmission ?m. ?m type S} . }
```

Fig. 13. A query to retrieve the targeted partition.

lower bound distances in t's). We repeat this process for all W_i's for $i = 1, \ldots, m$, which completes the first iteration (lines 3-8).

- *In the j-th iteration.* We pop the smallest entry from all a_i's, say $(v, (\mathsf{u}, \mathcal{S}, d_l, d_u))$ (line 10). We denote the partition rooted at v as the current partition. Denote the last pair in $\mathcal{S}$ as (ℓ, v_ℓ), which indicates that the path leaves the partition rooted at v_ℓ and enters the current partition using portal ℓ. Next, for the current partition, we find its portals $\mathcal{L} = \{\ell'_\infty, \ell'_\in, \ldots\}$ from the portal index. For each ℓ' in $\mathcal{L}$, we compute the lower and upper bounds for $d(\ell, \ell')$ (or $d(\mathsf{u}, \ell')$ if $\ell = nil$) in the current partition using the approach discussed in Section 6.1, denoted as d'_l and d'_u (line 13). A portal ℓ' can connect the current partition to a set P' of neighboring partitions. For each partition in P', denoted by its partition root v_r, we construct a quadruple $t = (\mathsf{u}, \mathcal{S} \cup (\ell', v_r), d_l + d'_l, d_u + d'_u)$ as in line 14. We also search for the entry of v_r in M and update its i-th list with t in the same way as in the first iteration. However, if either of the following cases is satisfied, we stop updating the entry for v_r in M: (i) adding ℓ' to $\mathcal{S}$ generates a cycle; and (ii) $d_l + d'_l$ is greater than the k-th largest upper bound in the i-th list. Otherwise, we also push (v_r, t) to the queue a_i.

At any iteration, if a new quadruple t has been appended to the i-th list of an entry indexed by v in M, and all of its other $m-1$ lists are non-empty, then the partition rooted at v contains potential answer roots for the keyword query. To connect the partitions containing all the keywords being queried, we find all the possible combinations of the quadruples from the $(m-1)$ lists, and combine them with t. Each combination of the m quadruples denotes a connected subgraph having all the keywords being queried.

Example 6. In Fig. 12, denote the new quadruple just inserted to the first list of an entry in M as t_1. Since both of its lists are now non-empty, two combinations can be found, i.e., (t_1, t_2) and (t_1, t_3), which leads to two conjunctive subgraphs. Using the partition information in the quadruples, we can easily locate the correspondent partitions.

We study how to access the instance data for a partition in Section 7. Once the instance data from the selected partitions are ready, we proceed to the second-level backward search by applying the BACKWARD method to find the top-k answers on the subgraph concatenated by these partitions (line 20). In any phase of the algorithm, we track the top-k answers discovered in a priority queue.

• *Termination condition.* The following Lemmas provide a correct termination condition for the SUMM method.

Lemma 8. *Denote an entry in the priority queue as $(v, (u, \mathcal{S}, d_l, d_u))$, then for any v' in the partition rooted at v and the length of any path starting from u and using the portals in $\mathcal{S}$ is $d(u, v') \geq d_l$.*

Lemma 9. *Denote the top entry in the priority queue a_i as $(v, (u, \mathcal{S}, d_l, d_u))$, then for any explored path p from w_i in the queue a_i, the length of p, written as $d(p)$, has $d(p) \geq d_l$.*

We denote the set of all unexplored partitions in $\mathcal{P}$ as $\mathcal{P}_t$. For a partition h rooted at v that has not been included in M, clearly, $h \in \mathcal{P}_t$. The best possible score for an answer root in h is to sum the d_l's from all the top entries of the m expansion queues, i.e., $a_1, \ldots, a_m$. Denote these m top entries as $(v_1, (u_1, \mathcal{S}^1, d_l^1, d_u^1)), \ldots, (v_m, (u_m, \mathcal{S}^m, d_l^m, d_u^m))$, respectively. Then,

Lemma 10. *Let g_1 be a possible unexplored candidate answer rooted at a vertex in a partition h, with $h \in \mathcal{P}_t$,*

$$s(g_1) > \sum_{i=1}^{m} d_l^i. \tag{3}$$

Next, consider the set of partitions that have been included in M, i.e., the set $\mathcal{P} - \mathcal{P}_\sqcup$. For a partition $h \in \mathcal{P} - \mathcal{P}_t$, let the first quadruple from each of the m lists for its entry in M be: $t_1 = (\hat{u}_1, \hat{\mathcal{S}}_1, \hat{d}_l^1, \hat{d}_u^1), \ldots, t_m = (\hat{u}_m, \hat{\mathcal{S}}_m, \hat{d}_l^m, \hat{d}_u^m)$ (note that due to the order of insertion, each list has been implicitly sorted by the lower bound distance $\hat{d}_l$ in ascending order), where $t_j = nil$ if the j-th list is empty. Then, we have:

Lemma 11. *Denote the best possible unexplored candidate answer as g_2, which is rooted at a vertex in the partition h, where $h \in \mathcal{P} - \mathcal{P}_\sqcup$, then*

$$s(g_2) > \sum_{i=1}^{m} f(t_i)\hat{d}_l^i + (1 - f(t_i))d_l^i, \tag{4}$$

where $f(t_i) = 1$ if $t_i \neq nil$ otherwise $f(t_i) = 0$.

Finally, we can derive the termination condition for the search.

The termination condition. We denote the score of the best possible answer in an unexplored partition as $s(g_1)$, as defined by the RHS of (3); and the score of the best possible answer in all explored partitions as $s(g_2)$, as defined by the RHS of (4). Denote the candidate answer with the k-th smallest score during any phase of the algorithm as g. Then, the backward expansion on the summary level can safely terminate when $s(g) \leq \min(s(g_1), s(g_2))$. By Lemmas 10 and 11, we have:

Theorem 2. *SUMM finds the top-k answers $\mathcal{A}(q, k)$ for any top-k keyword search query q on an RDF graph.*

Sections 12 and 13 in the online appendix discuss the complexity of SUMM and further elaborate its correctness.

7 Accessing Data and Update

The SUMM algorithm uses the summary of the RDF data to reduce the amount of data accessed in the BACKWARD method. For the algorithm to be effective, we should be able to efficiently identify and retrieve the instance data from

LUBM	Wordnet	Barton	BSBM	DBpedia Infobox
14	15	30	1520	5199

Fig. 14. Number of distinct types in the data sets.

LUBM	Wordnet	Barton	BSBM	DBpedia Infobox
5	2	40	70	30

Fig. 15. Number of triples in the data sets ($\times 10^6$).

selected partitions. One option is to store the triples by partitions and index on their partition ids, i.e., adding another index to the algorithm. But then whenever an update on the partition happens, we need to update the index. Furthermore, the approach enforces a storage organization that is particular to our methods (i.e., not general). In what follows, we propose an alternative efficient approach that has no update overhead and requires no special storage organization. Our approach stores the RDF data in an RDF store and works by dynamically identifying the data of a partition using appropriately constructed SPARQL queries that retrieve only the data for that partition.

Since graph homomorphism is a special case of homomorphism on relational structure (i.e., binary relations) [16] and the fact that relational algebra [21] is the foundation of SPARQL, we can use the Homomorphism Theorem [1] to characterize the results of two conjunctive SPARQL queries.

Theorem 3. *Homomorphism Theorem [1]. Let q and q' be relational queries over the same data D. Then $q'(D) \subseteq q(D)$ iff there exists a homomorphism mapping $f : q \rightarrow q'$.*

Recall that $f_1 : h_t \rightarrow h$ (see Fig. 11a) and for each h_t, we extract a core c from h_t. By definition, c is homomorphic to h_t, thus c is homomorphic to h (transitivity). Using c as a SPARQL query pattern can extract h due to Theorem 3.

There are two practical issues the need our attention. First, there is usually a many-to-one mapping from a set of h_t's to the same core c—leading to a low selectivity by using c as the query pattern. To address this issue, we can bind constants from the targeted partition to the respective variables in query pattern. These constants include the root and the portals of the targeted partition which are retrievable from the indexes. The second issue is that in our construction of S, we do not explicitly keep every c. Instead, a core c is embedded (by homomorphism) to a summary $s \in S$, where c is a subtree of s. To construct a SPARQL query from s, we first need to find a mapping for the partition root in s, then the triple patterns corresponding to the subtree in s are expressed in (nested) `OPTIONAL`s from the root to the leaves. For example, the SPARQL query for the partition rooted at URI_5 in Fig. 8 can be constructed by using the summary in Fig. 7a. Notice that URI_5 is bound to the root to increase selectivity.

Our approach also supports efficient updates, which is addressed in Section 14 in the online appendix.

8 Experiments

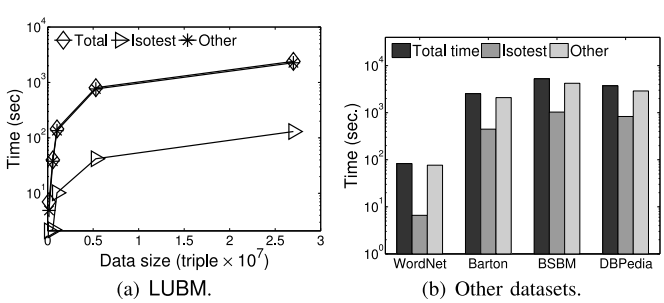
We implemented the Backward and Summ methods in C++. We also implemented two existing approaches proposed in [23] and [14]. We denote them as Schema and Blinks respectively. All experiments were conducted on a 64-bit Linux machine with 6 GB of memory.

Data sets: We used large synthetic and real RDF data sets for experiments. Synthetic data sets are generated by popular RDF benchmarks. The first is the Lehigh University Benchmark [13], which models universities with students, departments, etc. With its generator, we created a *default* data set of 5 million triples and varied its size up to 28 million triples. The second is the Berlin SPARQL Benchmark (BSBM) [5], which models relationships of products and their reviews. Unlike LUBM where data are generated from a fixed set of templates, BSBM provides interfaces to scale the data on both size and complexity of its structure. In particular, it provides means for adding new types in the data. The rest of the data sets are popular real RDF data sets, i.e., Wordnet, Barton and DBpedia Infobox. The number of distinct types for the data sets are shown in Fig. 14 and their sizes are reported in Fig. 15. Notice DBpedia and BSBM are very irregular in structure, i.e., both have more than 1,000 distinct types. BSBM is also large in size (70 million triples by default).

Implementation and setup: We used the disk-based B$^+$-tree implementation from the TPIE library to build a Hexstore-like [25] index on the RDF data sets. For subgraph isomorphism test, we used the VFLib. We assume each entity in the data has one type. For an entity that has multiple types, we bind the entity to its most popular type. To store and query RDF data with SPARQL, we use Sesame [6]. In all experiments, if not otherwise noted, we built the summary for 3-hop neighbors, i.e., $\alpha = 3$, and set $k = 5$ for top-k queries.

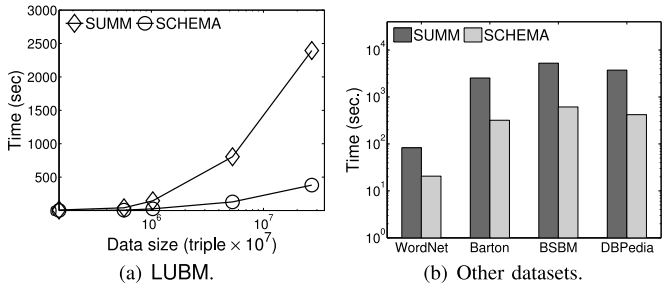
8.1 Evaluating Summarization Algorithms

Time on the summarization process. We start with a set of experiments to report the time (in log scale) in building a summary. For LUBM, we vary the size of the data from 100,000 triples to 28 million triples.In Fig. 16a, we plot the total time for building the summary, which includes: the time spent to find homomorphic mappings (i.e., performing subgraph isomorphism tests) and the time spent for the rest of operations, e.g., partitioning the graph, and constructing the inverted indexes. The latter cost dominates the summarization process for all the cases in LUBM data sets. The same trend can also be observed in all the other data sets, as shown in Fig. 16b. Notice that the summary used by Summ is built once and thereafter incrementally updatable whenever the data get updated. For comparison, we study the summarization performance for the Schema method. The



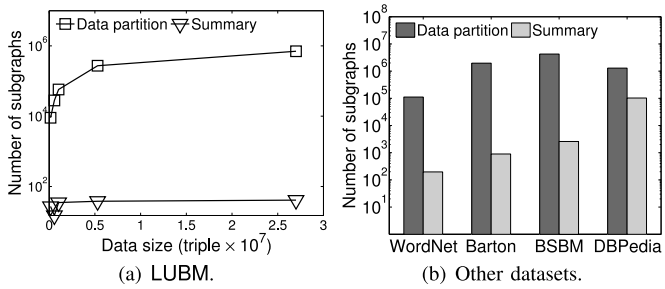
(a) LUBM. (b) Other datasets.

Fig. 16. Time for the summary construction.



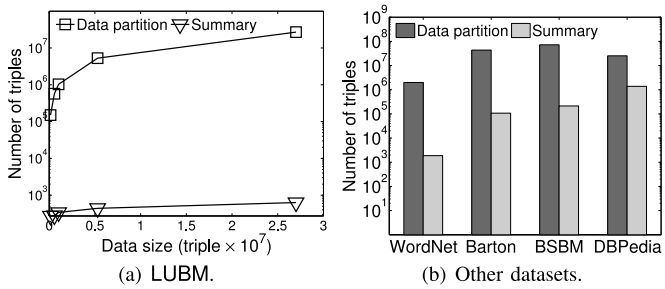
(a) LUBM. (b) Other datasets.

Fig. 17. Time for the summarization: SUMM versus SCHEMA.



(a) LUBM. (b) Other datasets.

Fig. 18. # subgraphs: partitions versus summary.



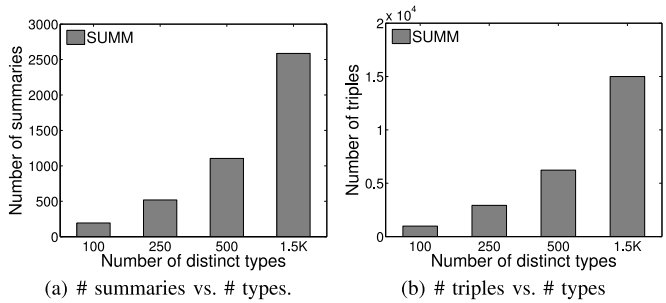
(a) LUBM. (b) Other datasets.

Fig. 19. # triples: partitions versus summary.

comparisons are shown in Fig. 17. Regardless of the quality of the summary, the SCHEMA method in general performs an order of magnitude faster than our SUMM method across all the data sets we experimented. However, as it will become more clearly shortly, while the summary built by SCHEMA might be useful in some settings, it does not yield correct results in all our experimental data sets.

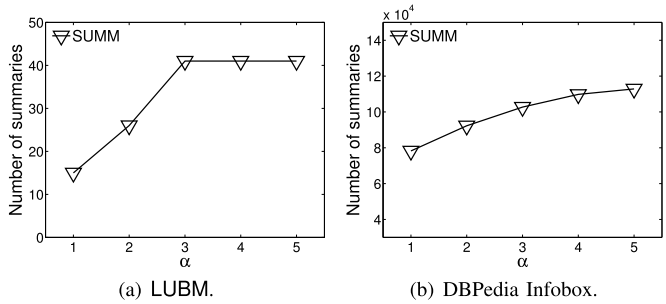
Size of the summary. As the SCHEMA method generates one (type) node in the summary for all the nodes in the data that have the same type, the size of summary (in terms of number of nodes) is equal to the number of distinct types from the data, as shown in Fig. 14. For our summarization technique, we plot the number of partitions and the number of summaries in Figs. 18a and 18b. In Fig. 18c for LUBM, the summarization technique results in at least two orders less distinct structures comparing to the number of partitions. Even in the extreme case where the data set is partitioned into about a million subgraphs, the number of distinct summaries is still less than 100. In fact, it remains almost a constant after we increase the size of the data set to 1 million triples. This is because LUBM data is highly structured [10].

Not all RDF data sets are as structured as LUBM. Some RDF data sets like BSBM and DBpedia are known to have a high variance in their *structuredness* [10]. In Fig. 18b, we plot



(a) # summaries vs. # types. (b) # triples vs. # types

Fig. 20. Vary the number of types in BSBM.



(a) LUBM. (b) DBPedia Infobox.

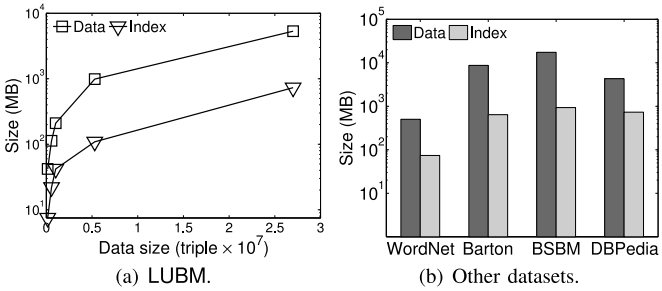
Fig. 21. Impact of α to the size of summary.

the number of distinct summaries for other data sets after applying the SUMM method. For Wordnet and Barton, SUMM distills a set of summaries that has at least three orders less distinct structures than the respective set of partitions. Even in the case of DBpedia and BSBM, our technique still achieves at least one order less distinct structures than the number of the partitions, as shown in Fig. 18b.

In Figs. 19a and 19b, we compare the number of triples stored in the partitions and in the summary. Clearly, the results show that the distinct structures in the data partitions can be compressed with orders-of-magnitude less triples in the summary, e.g., at least one order less for DBpedia Infobox, and at least three orders less for LUBM, Wordnet, Barton and BSBM. Since the summaries are all small, this suggests that we can keep the summaries in main memory to process keyword query. Therefore, the first level of backward search for SUMM method can be mostly computed in memory.

Impact from the distinct number of types. Adding new types of entities in an RDF data set implicitly adds more variances to the data and hence makes the summarization process more challenging. To see the impact on SUMM, we leverage the BSBM data generator. In particular, we generate four BSBM data sets with the number of distinct types ranged from 100 to 1,500. The results from SUMM on these data sets are shown in Figs. 20a and 20b. As we are injecting more randomness into the data by adding more types, the size of the summary increases moderately. Consistent to our observations for DBpedia in Fig. 18b and Fig. 19b, SUMM efficiently summarizes the BSBM data sets with orders less triples, even when the data set has more than a thousand distinct types.

Impact of α. Since trends are similar, we only report the impact of α on two representative data sets, i.e., LUBM and DBpedia. In Figs. 21a and 21b, we report the impact of α (a parameter on the max number of hops in each partition, see



(a) LUBM. (b) Other datasets.

Fig. 22. Size of the auxiliary indexes.

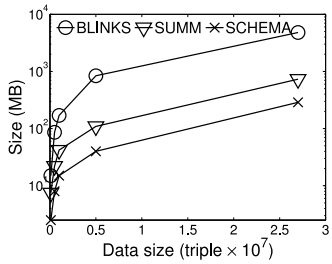


Fig. 24. Indexes on LUBM.

Section 5.2) on the size of summary. Intuitively, the smaller α is, the more similar the α-neighborhoods are, leading to a smaller size of summary after performing summarization. This is indeed the case when we vary α for all the data sets. The smallest summary is achieved when $\alpha = 1$ in both Figs. 21a and 21b. Notice that there is a tradeoff between the size of the summary and the size of the auxiliary indexes. A smaller partition implies that more nodes become portals, which increases the size of the auxiliary indexes. On the other hand, increasing α leads to larger partitions in general, which adds more variances in the structure of the partitions and inevitably leads to a bigger summary. However in practice, since the partitions are constructed by directed traversals on the data, we observed that most of the directed traversals terminate after a few hops. For instance, in LUBM, most partitions stop growing when $\alpha > 3$. A similar trend is visible in Fig. 21b. When we increase α, the number of distinct structures increases moderately.

Overheads of auxiliary indexes. In Figs. 22a and 22b, we study the size of the auxiliary indexes. Fig. 22a shows that for LUBM, the size of the auxiliary indexes is one order less than the respective data when we vary its size up to 28 million triples. Similar trends can be observed in Fig. 22b for the other data sets. This is because that for all indexes, we do not explicitly store the edges of the RDF data that usually dominate the cost in storing large graphs. In Fig. 23, we report the breakdown of the inverted indexes for all the data sets. The most costly part is to store the mappings in the *summary index* (i.e., SUMMIDX in the figure) and the other indexes are all comparably small in size. Thus, to efficiently process query, we can keep all but the *summary index* in main memory.

We also compare in Fig. 24 the indexing overhead of different methods as we vary the data size for LUBM. Notice that BLINKS is the most costly method in terms of storage (i.e., it demands at least one order of magnitude more space), as its distance matrix leads to a quadratic blowup in indexing size. BLINKS is no doubt the faster method for small data, but it clearly does not scale with large RDF data sets. Therefore, we do not report BLINKS in the evaluation of query performance.

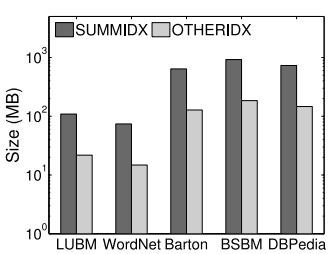


Fig. 23. Breakdown.

8.2 Query Performance

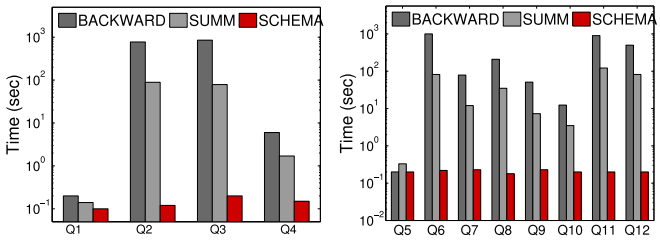
In this section, we study the performance of top-k keyword search using SUMM and BACKWARD. In particular, we compare the proposed methods with SCHEMA, which is the only existing method that can apply to large RDF data sets. To this end, we design a query workload that has various characteristics. Fig. 25 lists 12 typical queries, together with the number of keyword occurrences in the data sets. For LUBM, all the keywords are selected from the first university in the data set, except for keywords with respect to publications 17 and 18. For the two indicated keywords, we select one copy of each publication from the first university and pick the rest of them randomly from other universities. This is to simulate the cases in real data sets where not all the keywords in a query are close to each other. Notice that in LUBM, keywords such as publications 17 or professor 9 are associated with multiple entities. For the other data sets, we pick two representative queries for each of them. The queries are shown in Fig. 25. In particular, the second column # *nodes* shows the number of occurrences for the keywords being queried in the respective data set. For long running queries, we terminate the executions after 1,000 seconds. The response times are in log scale.

We first use SCHEMA to answer the queries in Fig. 25. SCHEMA generates a set of k SPARQL queries for each keyword query. Evaluating these queries are supposed to return k answer roots ranked by the scores. However, even if we have fixed the incorrect termination condition in

	Query	# nodes	Dataset
Q_1	[Pub_{19}, Lec_6]	(20,13)	L
Q_2	[$Research_5$, $FullProf_9$, Pub_{17}]	(9,4,83)	L
Q_3	[$FullProf_9$, $Grad_0$, Pub_{18}, Lec_6]	(4,15,40,5)	L
Q_4	[Dep_0, $Grad_1$, Pub_{18}, $AssocProf_0$]	(1,15,40,15)	L
Q_5	[Afghan, Afghanistan, al-Qaeda, al-Qa'ida]	(6,3,3,2)	W
Q_6	[3rd base, 1st base, baseball team, solo dance]	(14,13,17,4)	W
Q_7	[Knuth, Addison-Wesley, Number theory]	(1,1,35)	B
Q_8	[Data Mining, SIGMOD, Database Mgmt.]	(166,1,4)	B
Q_9	[Bloomberg, New York City, Manhattan]	(1,7,108)	D
Q_{10}	[Bush, Hussein, Iraq]	(1,1,48)	D
Q_{11}	[deflation, railroaders]	(32,70)	S
Q_{12}	[ignitor, microprocessor, lawmaker]	(3,110,43)	S

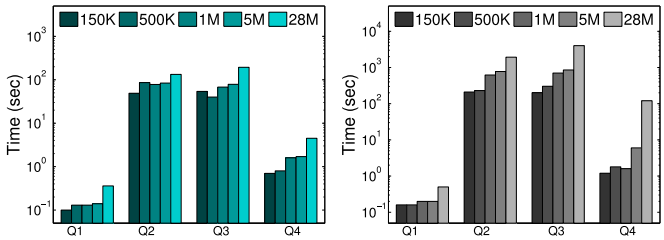
L:LUBM W:Wordnet B:Barton D:DBpedia Infobox S:BSBM

Fig. 25. Sample query workload.



(a) LUBM. (b) Other datasets.

Fig. 26. Query performance.



(a) SUMM. (b) BACKWARD.

Fig. 27. Query performance versus data size.

SCHEMA (as discussed in Section 4), our observation is that SCHEMA still returns incorrect results for *all* of the queries, as we have indicated in Fig. 2. This can be explained by the way it summarizes the data, as all nodes of the same type are indistinguishably mapped to the same type node in the summary. For instance, every FullProfessor has a publication in LUBM, whereas this does not mean that FullProfessor9 has a Publication17 for Q_2. Nevertheless, we report the response time of the SCHEMA method together with other methods for comparison. In what follows, we focus on discussing the query performance for the BACKWARD and SUMM methods, both of which have provable guarantees on the correctness of the query results. Their performance on the sampled query workload are plotted on Figs. 26a and 26b respectively.

The efficiency of a keyword search is determined by a collection of factors [14], with no single factor being the most deterministic one. In particular, we observe that for selective keywords (i.e., keywords that have fewer occurrences in the data) and especially those that are close to each other, e.g., Q_1, Q_5, both BACKWARD and SUMM answer the queries efficiently. In some cases, e.g., Q_5, BACKWARD outperforms SUMM since SUMM uses a lower bound to decide when to terminate. This inevitably requires SUMM to access more data than what is necessary to correctly answer the query.

However, being selective alone does not necessarily lead to better query performance, especially if the keyword being queried corresponds to a hub node that has a large degree, e.g., the Department0 in Q_4. On the other hand, as the keywords being queried become non-selective, e.g., Q_3 or Q_{11}, or the keywords are far away from one another, e.g., the solo dance and the baseball team in Q_6, the SUMM approach generally performs much better than the BACKWARD method. This is because only when the connected partitions are found to contain all the keywords, the SUMM needs to access the whole subgraph on disk. This leads to savings in dealing with keywords that are resultless, e.g., most of the occurrences for publications 17 and 18 in Q_2-Q_4 fall into this category. In addition, at the partition level, the backward expansion in the SUMM approach can be done almost completely in memory as the major indexes for expansion are lightweight (as shown in Section 8.1) and therefore can be cached in memory for query evaluation. In such cases, i.e., Q_2-Q_4 and Q_6-Q_{12}, we observe that the SUMM approach performs much better.

In Figs. 27a and 27b, we investigate the query performance while varying the size of the LUBM data set. As shown in the figure, for both SUMM and BACKWARD, the cost of query evaluation generally becomes more expensive as the size of the data increases. In contrast to BACKWARD, the difference in time for the SUMM method is relatively moderate as the size of the data changes. This is because for SUMM, keywords that are far apart can be pruned in the first level of the backward search. Unlike SUMM, BACKWARD has to pay equal effort in dealing with each copy of every keyword being queried, leading to a much slower convergence.

9 Conclusion

We studied the problem of scalable keyword search on big RDF data and proposed a new summary-based solution: (i) we construct a concise summary at the type level from RDF data; (ii) during query evaluation, we leverage the summary to prune away a significant portion of RDF data from the search space, and formulate SPARQL queries for efficiently accessing data. Furthermore, the proposed summary can be incrementally updated as the data get updated. Experiments on both RDF benchmark and real RDF data sets showed that our solution is efficient, scalable, and portable across RDF engines. An interesting future direction is to leverage the summary for optimizing generic SPARQL queries on large RDF data sets.

Acknowledgments

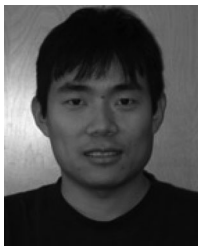
Feifei Li was supported in part by US National Science Foundation (NSF) grant IIS-1251019.

References

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.
[2] A. Aggarwal and J.S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Comm. ACM*, vol. 31, no. 9, pp. 1116-1127, 1988.
[3] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: Enabling Keyword Search over Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2002.
[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword Searching and Browsing in Databases Using Banks," *Proc. 18th Int'l Conf. Data Eng. (ICDE)*, 2002.
[5] C. Bizer and A. Schultz, "The Berlin SPARQL Benchmark," *Int'l J. Semantic Web and Information Systems*, vol. 5, pp. 1-24, 2009.
[6] J. Broekstra et al., "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," *Proc. First Int'l Semantic Web Conf. The Semantic Web (ISWC)*, 2002.
[7] Y. Chen, W. Wang, and Z. Liu, "Keyword-Based Search and Exploration on Databases," *Proc. 27th Int'l Conf. Data Eng. (ICDE)*, 2011.
[8] Y. Chen, W. Wang, Z. Liu, and X. Lin, "Keyword Search on Structured and Semi-Structured Data," *Proc. ACM Int'l Conf. Management of Data (SIGMOD)*, 2009.

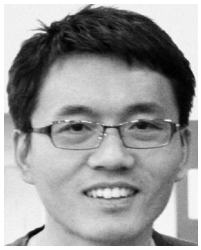
[9] B.B. Dalvi, M. Kshirsagar, and S. Sudarshan, "Keyword Search on External Memory Data Graphs," *Proc. VLDB Endowment*, vol. 1, pp. 1189-1204, 2008.
[10] S. Duan, A. Kementsietsidis, and K.Srinivas, O. Udrea, "Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 2011.
[11] H. Fu and K. Anyanwu, "Effectively Interpreting Keyword Queries on RDF Databases with a Rear View," *Proc. 10th Int'l Conf. Semantic Web (ISWC)*, 2011.
[12] K. Golenberg, B. Kimelfeld, and Y. Sagiv, "Keyword Proximity Search in Complex Data Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2008.
[13] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL Knowledge Base Systems," *J. Web Semantics*, vol. 3, pp. 158-182, 2005.
[14] H. He, H. Wang, J. Yang, and P.S. Yu, "Blinks: Ranked Keyword Searches on Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07)*, 2007.
[15] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient IR-Style Keyword Search Over Relational Databases," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB)*, 2003.
[16] P. Jeavons, "On the Algebraic Structure of Combinatorial Problems," *Theoretical Computer Science*, vol. 200, pp. 185-204, 1998.
[17] M. Kargar and A. An, "Keyword Search in Graphs: Finding R-cliques," *Proc. VLDB Endowment*, vol. 4, pp. 681-692, 2011.
[18] G. Li, B.C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: Efficient and Adaptive Keyword Search on Unstructured, Semi-structured and Structured Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 2008.
[19] Y. Luo, W. Wang, and X. Lin, "SPARK: A Keyword Search Engine on Relational Databases," *Proc. IEEE 24th Int'l Conf. Data Eng. (ICDE)*, 2008.
[20] A. Polleres, "From SPARQL to Rules (and Back)," *Proc. 16th Int'l Conf. World Wide Web (WWW)*, 2007.
[21] R. Shamir and D. Tsur, "Faster Subtree Isomorphism," *J. Algorithms*, vol. 33, pp. 267-280, 1999.
[22] C. Sun, C.Y. Chan, and A.K. Goenka, "Multiway SLCA-based Keyword Search in XML Data," *Proc. 16th Int'l Conf. World Wide Web (WWW)*, pp. 1043-1052, 2007.
[23] T. Tran, H. Wang, S. Rudolph, and P. Cimiano, "Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data," *Proc. IEEE Int'l Conf. Data Eng. (ICDE)*, 2009.
[24] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," *Proc. VLDB Endowment*, vol. 1, pp. 1008-1019, 2008.

Wangchao Le received the PhD degree from the University of Utah, specialized in data management and analysis. His research interests include database systems, in particular, temporal and multi-version data, query processing and optimization for large graph data, and semantic web. His dissertation is about supporting scalable data analytics on large linked data. He is currently working at Microsoft for its big data and search infrastructure.



Feifei Li received the BS degree in computer engineering from Nanyang Technological University in 2002 and the PhD degree in computer science from Boston University in 2007. He is currently an associate professor in the School of Computing, University of Utah. His research interests include database and data management systems and big data analytics.

Anastasios Kementsietsidis received the PhD degree from the University of Toronto. He is a research staff member at the IBM T.J. Watson Research Center. His research interests include various aspects of RDF data management (including, querying, storing, and benchmarking RDF data). He research interests also include data integration, cleaning, provenance and annotation, as well as (distributed) query evaluation and optimization on relational or semi-structured data.



Songyun Duan received the PhD degree from Duke University, with a dissertation on simplifying system management through automated forecasting, diagnosis, and configuration tuning. He is a researcher at the IBM T.J. Watson Research Center. His research interests include data management, machine learning, big-data analysis, and semantic web.