KeyLabel Algorithms for Keyword Search in Large Graphs

Yue Wang*, Ke Wang†, Ada Wai-Chee Fu‡, and Raymond Chi-Wing Wong§
*† *School of Computing Science, Simon Fraser University*
Email: {ywa138, wangk†}@cs.sfu.ca*
‡ *Department of Computer Science and Engineering, The Chinese University of Hong Kong*
Email: adafu@cse.cuhk.edu.hk
§ *Department of Computer Science and Engineering, The Hong Kong University of Science and Technology*
Email: raywong@cse.ust.hk

***Abstract*—Graph keyword search is the process of extracting small subgraphs that contain a set of query keywords from a graph. This problem is challenging because there are many constraints, including distance constraint, keyword constraint, search time constraint, index size constraint, and memory constraint, while the size of data is inflating at a very high speed nowadays. Existing greedy algorithms guarantee good performance by sacrificing the accuracy to generate approximate answers, and exact algorithms promise exact answers but require a high memory consumption for loading indices and advanced knowledge about the maximum distance constraint. For big data applications, existing techniques are inefficient and impractical due to huge memory consumption and varied distance constraint. We propose a new keyword search algorithm that finds exact answers with low memory consumption and without advanced knowledge of maximum distance constraint. This algorithm builds a compact index structure offline based on a recent labeling index for shortest path queries. At the query time, it finds the answer efficiently by examining a small portion of the index related to a query.**

***Keywords*-keyword search; large graphs; indexing; query**

I. Introduction

Graph keyword search finds a substructure from a graph, which could be modeled from relational databases [4], [11], XML documents [3], [5], [17], web pages [13], and social networks [7], [10], to cover a set of input keywords. Each node of the graph, annotated with some attributes or keywords, represents a point of interest, an XML document, a web page, or a participant of social networks. Edges in such graphs could represent foreign key relationships, IDREF/ID, hyper-links, and friendships or other routes. A top-k graph keyword search is about retrieving k sets of closely connected nodes, where each set collectively covers some specific keywords, i.e., keyword constraint, by nodes within a maximum distance, i.e., distance constraint, specified in the query. For example, an answer set could be a set of related papers in a citation network that cover a specified set of topics, a set of well-connected experts in a social network that cover a set of skills, and a set of nearby point-of-interests that cover several themes. Notice that this type of search identifies query keywords, not query nodes, thus, should not be confused with nearest neighbor searches where some query nodes are specified, such as [15].

As a concrete example, graph keyword search can be applied to construct queries for the RDF (Resource Description Framework) data [9]. The basic unit of RDF data is a tuple consisting of a subject, a predicate and an object. The predicate defines the relationship between the subject and the object. Constructing semantic queries like SPARQL queries for RDF data [14] requires sufficient knowledge about the underlying object-predicate-subject structure. For large scale applications, imposing such knowledge on the user might be unreasonable. More often, the user may know some attribute values in a query, but not the exact structure in which they are involved [16]. In this case, by modeling the RDF data as an attributed graph, keyword search can find subgraphs containing the user's interested entities and values. These subgraphs can then be used to derive candidate SPARQL queries.

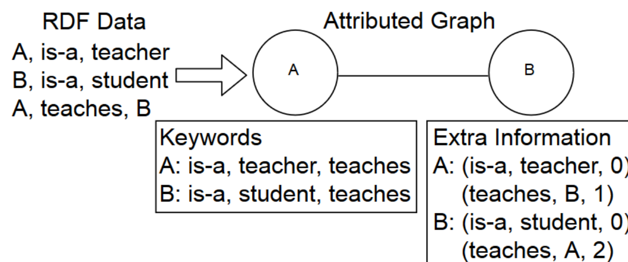

Figure 1. RDF data to attributed graph modeling

Figure 1 shows an example about the modeling. We need to convert the RDF data with three "subject, predicate, object" triples into an attributed graph for keyword search. All the subjects like A and B are entities, so we model them as nodes in the attributed graph. For entity-to-attribute tuples, such as "A, is-a, teacher" and "B, is-a, student", we view the predicate and the object as keywords of the subject node. For entity-to-entity tuples, such as "A, teaches, B", we view the predicate as a keyword of both the subject node and the object node and link them by an edge.

In order to convert the answers of keyword search to SPARQL queries, we create the following extra information to encode different types of tuples. Note that such

extra information is not part of the attributed graph and is used only to convert the answers of keyword search to the SLAROL queries represented by such answers. For each entity-to-attribute tuple, we assign the subject node a $(predicate, object, 0)$ triple, such as the (is-a, teacher, 0) triple of A and the (is-a, student, 0) triple of B, where 0 means it's an entity-to-attribute relationship. We can use this kind of triple to reconstruct the entity-to-attribute tuple given either the predicate or the object keyword. For each entity-to-entity tuple, we assign a $(predicate, object, indicator)$ triple to the subject, such as the (teaches, B, 1) triple of A and the (teaches, A, 2) of B, where 1 and 2 indicate whether the entity is a subject or an object. Similarly, we can use this kind of triple to reconstruct the entity-to-entity tuple given the predicate keyword.

Now, users can apply keyword search algorithms on the attributed graph to find candidate answer sets with nodes containing the query keywords. This step doesn't involve the extra information file. Then, the triples of related nodes in the extra information file allow us to retrieve the entity-to-entity or entity-to-attribute relationships and generate an optional SPARQL query for each candidate answer set so that users can select a query and apply it as the input to the SPARQL engine.

One of the early approaches to graph keyword search is finding Steiner trees to cover query keywords [1]. [2] provides a dynamic programming algorithm to find top-k minimum cost group Steiner trees, but its time complexity is exponential. In [8], the authors propose a polynomial delay algorithm, denoted by RClique, for searching r-cliques, where a r-clique is a set of nodes that cover collectively all query keywords and have a pairwise distance no larger than r. RClique finds 2-approximation answers where the distance between each pair of nodes in the answer produced could be twice of that allowed by the distance constraint.

RClique algorithm requires a pre-determined distance upper bound θ for all queries in order to index the distance information from each node to all other nodes within its θ-neighborhood. Requiring knowing a maximum distance constraint leads to either a loose upper bound, or not being able to process all queries. In a dense graph with a large average degree, the neighborhood of a node expands quickly as the distance increases. For a big graph, such full materialization of indexing, even only a limited neighborhood of each node, might take very long time and requires a huge storage. During query time, the indexed information of all nodes containing query keywords needs to be retrieved for processing. Random access of disk resident node-based index will result an expensive I/O cost. With 2-approximation, some of the returned top-k answers may violate the distance constraint and the best solutions may not be in the returned top-k answers.

Another approach is proximity based search in the greedy manner. [10] provides a greedy but NP-hard solution called RareFirst that will return 2-approximate top-1 answer. The GDensity framework introduced in [12] improves it by applying density indexing and likelihood ranking mechanisms to find the top-k exact node sets covering all the query keywords with smallest diameters. The idea is that if less hops are needed to reach more nodes from a node, there is higher chance for all query keywords to be covered within smaller neighborhood of the node. The search starts from nodes in the order of their likelihood ranks and increases the search range by 1 in each iteration. The program terminates once the priority queue of answers is full. Unlike RClique algorithm, GDensity provides exact answers instead of 2-approximation.

GDensity algorithm requires a distance upper bound θ for indexing too. The indexing is fast and results in small index size because it only records discrete distance distribution within each node's θ-neighborhood. However, GDensity requires the whole graph to be loaded into memory for computing shortest paths during query time, which is impractical for big data. The search space could expand quickly for a dense graph. In fact, the search is pruned mainly based on the least frequent query keyword and other query keywords are checked by the expensive shortest path computation. Also, GDensity might not be able to terminate early when the number of candidate answers is smaller than that of expected answers. If the graph is weighted or the distance constraint of the query is large, GDensity will waste time on re-examination of query keyword coverage with the range increased by only 1 at each iteration.

For large and dense graphs, especially when query keywords are frequent, node-based keyword search algorithms, such as RClique and GDensity, need to search a large amount of tight communities that could be anywhere in the graph. To address the weaknesseses of these existing algorithms, we design a keyword-based algorithm called KeyLabel algorithm based on the recent Hop Doubling Label Indexing algorithm defined in [6]. Our KeyLabel algorithm builds a new inverted index by associating the linkage information with each keyword, instead of each node, and ordering the indexed data in a distance sensitive manner. During query time, it loads and analyzes only a small amount of indexed information about query keywords. The novelty of KeyLabel is its indexing and querying strategies that push both keyword constraint and distance constraint to prune the search.

Our contributions are summarized as follows: (1) We introduce a new keyword-distance centralized indexing algorithm, KeyLabel-Indexing, to overcome the bottlenecks of node-centralized RClique-Indexing and GDensity-Indexing. (2) We design a fast querying algorithm, KeyLabel-Querying, to speed up query processing and reduce memory space cost at the same time as compared to RClique-Querying and GDensity-Querying. (3) Evaluation on real-world graphs supports that our KeyLabel

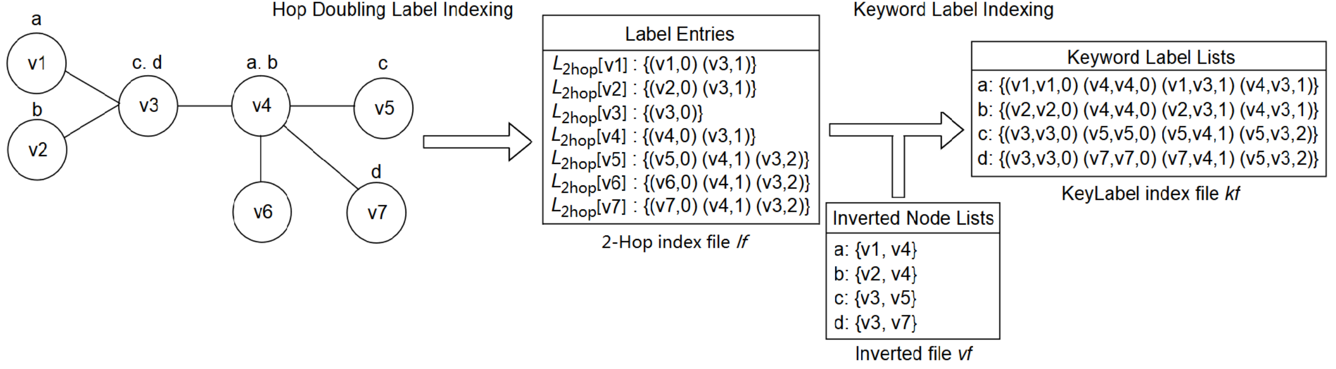

Figure 2. (Offline) KeyLabel-Indexing

algorithm outperforms the other two in most cases.(4) We evaluate the efficiency and effectiveness of KeyLabel in reconstructing SPARQL queries on RDF data.

The paper is organized into sections as follows. Section II specifies definitions and formal problem statements. Section III introduces the process of KeyLabel-Indexing algorithm. Section IV describes the procedure of KeyLabel-Querying algorithm. Section V evaluates the efficiency of KeyLabel algorithm and its effectiveness on constructing the queries on RDF data. Section VI concludes the paper. Section VII is the acknowledgement.

II. Problem Definition

Symbol	Description
$G = (V, E, W)$	G a node-attributed graph, where V is the node set, E is the edge set, W is the keyword set, and $W[v]$ is the set of keywords that node v contains
$Q = (QW, Dia, Tpk)$	a keyword query, where QW is the set of keywords, Dia is the diameter constraint, and Tpk is the top-k constraint
$dist(u, v)$	the shortest distance between node u and node v
$L_{2hop}[v]$	the label entry list of node v in $(p, dist)$ format
$vf[w]$	the list of nodes that contain keyword w, where vf is the keyword-to-nodes inverted file generated from G

Table I
Frequently used notations

For the convenience of presentation, we focus on undirected and unweighted graphs, but it's easy to modify current KeyLabel algorithm to deal with directed and/or weighted graphs. Table I shows the frequently used notations in our paper.

Definition 1. (Diameter). Given a graph $G = (V, E, W)$ and a node set $V_i \subseteq V$, the diameter of V_i is the maximum distance among the shortest distances of all node pairs in V_i.

Definition 2. (Minimal Cover). Given a graph $G = (V, E, W)$, a node set $V_i \subseteq V$ and a keyword set $W_i \subseteq W$, V_i is a cover of W_i if $W_i \subseteq \bigcup_{v \in V_i} W(v)$. If V_i is a cover of W_i, and no subset of V_i is a cover of W_i, then V_i is a minimal cover of W_i.

Problem (Keyword Search with Diameter Ranking). Given a graph $G = (V, E, W)$ and a query $Q = (QW, Dia, Tpk)$, the diameter ranking top-k keyword search finds Tpk node sets $\{V_1, V_2, ..., V_{Tpk}\}$ with smallest diameters such that each set V_i has a diameter no larger than Dia and is a minimal cover of QW.

The diameter ranking orders matching sets V_i by the largest distance between any two members in V_i. Other ranking methods such as pairwise total distance, which is the sum of shortest distances of all pairs of nodes in the answer set, or even the combination of diameter and pairwise total distance are also supported by our *KeyLabel* algorithm. Notice that if there are less than Tpk minimal covers with diameters no larger than Dia, all such minimal covers will be returned as the answer set. In the following two sections, we'll introduce the detailed mechanisms of indexing and querying of KeyLabel algorithm separately.

III. KeyLabel-Indexing

KeyLabel-Indexing is the offline step of KeyLabel and is performed only once. This step combines the label entry result of Hop Doubling Label Indexing with an inverted file vf to build a new index that attaches partial linkage information of each node to their keywords. Figure 2 shows the two components of KeyLabel-Indexing. The graph is first indexed into label entries of nodes as described in Section III-A, then together with inverted node lists, the labels are assigned to keywords in a new format as explained in Section III-B.

A. Hop Doubling Label Indexing

This step adopts the hop doubling label indexing in [6] for point-to-point distance querying on weighted and directed graphs. It constructs incoming and outgoing label lists for

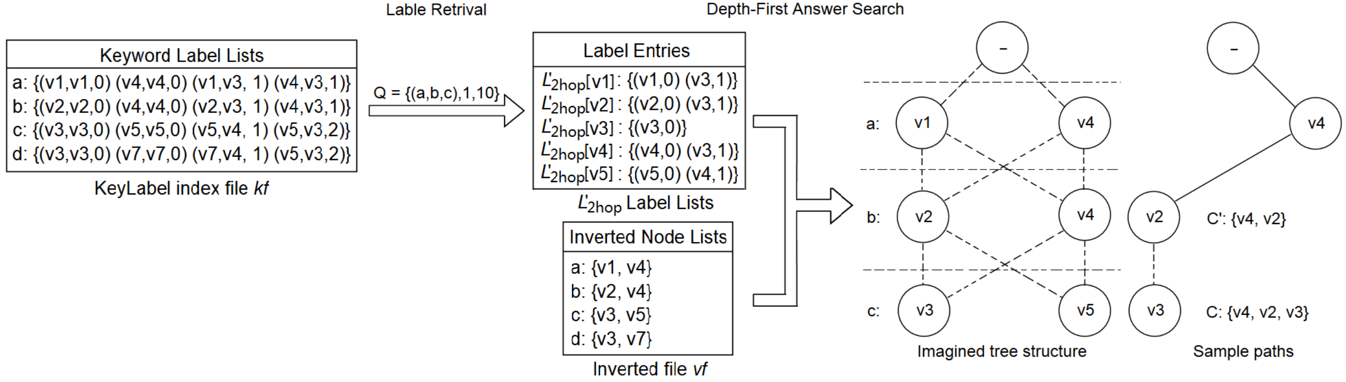

Figure 3. (Online) KeyLabel-Querying

each node so that the shortest distance between any pair of nodes can be found by simply looking up the two nodes' label lists. For the convenience of presentation, we focus on undirected graphs, so every node has only one label list. The label of a node v, denoted by $L_{2hop}[v]$, is a set of label entries in the format $(p, dist)$, where p is a pivot node that acts as an intermediate node for answering distance queries involving v in 2 hops, and $dist$ is the shortest distance between v and p.

The shortest distance between nodes u and v can be found by searching their label entry lists to see whether there are two label entries, one from each list, having the same pivot node. If $dist(u, v) \neq \infty$, we can find a pivot node p such that $(p, dist_1) \in L_{2hop}[u]$ and $(p, dist_2) \in L_{2hop}[v]$, and there doesn't exist another pivot node q such that $(q, dist_3) \in L_{2hop}[u]$, $(q, dist_4) \in L_{2hop}[v]$ and $dist_3 + dist_4 < dist_1 + dist_2$. In this case, $dist(u, v) = dist_1 + dist_2$. Thus, the shortest distance $dist(u, v)$ can be found by searching $L_{2hop}[u]$ and $L_{2hop}[v]$ for a pivot p with minimum $dist$ sum.

As shown in [6], for an unweighted graph, the computation cost of Hop Doubling Label Indexing algorithm is $O(|V|logM(|V|/M + log|V|))$, the I/O cost is $O(|V|log|V|/M \times |V|/B)$, and the index size is $O(h|V|)$, where h is a small constant, M is the memory size and B is the disk block size. Since the index is stored on disk and the index size is proportional to the number of nodes, not the number of edges, this method is scalable to large graphs.

B. Keyword Label Indexing

As the second component of the offline indexing step, the keyword label indexing combines the 2-hop index file lf generated by the Hop Doubling Label Indexing algorithm with the keyword-to-nodes inverted file vf. All the label entries of nodes containing each keyword will be bounded to the keyword and recorded in the output keylabel file kf. To associate the label entries of a node v to a keyword k that v contains, we change the label entries of v from the format $(p, dist)$ to the format $(v, p, dist)$ so that all $(p, dist)$ label entries of each node can be easily regrouped later. We then sort all the label entries associated with the keyword k in ascending order of $dist$ before they are written into the keyword label file kf. The sorting allows us to retrieve the label entries with $dist$ up to a certain diameter constraint without examining the whole label entry list.

Taking the keyword b in Figure 2 for example, we first read the nodes {v2, v4} from the inverted file vf. Then we read label entries {(v2, 0), (v3, 1)} of node v2 and label entries {(v4, 0), (v3, 1)} of node v4 from the 2-hop label index file lf. We reconstruct label entries from $(p, dist)$ to $(v, p, dist)$ format to indicate which node the label entry belongs to. After sorting the new label entries based on $dist$, we have {(v2, v2, 0), (v4, v4, 0), (v2, v3, 1), (v4, v3, 1)} as the label list of keyword b.

IV. KeyLabel-Querying

KeyLabel-Querying is the query processing step of KeyLabel at query time. Figure 3 gives an example of the procedure of KeyLabel-Querying. First, Label Retrieval loads the label entries of each query keyword from kf, reformats and reassigns them to nodes as described in Section IV-A. Then Depth-First Answer Search is performed on an imagined tree-like structure to search for candidate answers as explained in Section IV-B.

A. Label Retrieval

For a given query $Q = (QW, Dia, Tpk)$, Label Retrieval first loads the label entries of query keywords from the keylabel index file kf. Since label entries have already been sorted on $dist$, we can use a sequential scan to extract only the prefix of $kf[w]$ for every keyword w in QW cut off by Dia. This is because when we later try to find the shortest distance between two nodes by looking for the smallest sum of $dist$s of two label entries, one from each node's label entry list, the result must exceed Dia if the $dist$ of either label entry already exceeds Dia. Typically, the memory is large enough to hold all such prefixes for a query because the number of keywords in a query is small and each node

has a limited number of label entries with $dist$ no larger than Dia.

After obtaining the list of label entries in the $(v, p, dist)$ format from kf, we change them back to the original $(p, dist)$ form to construct a new 2-hop label entry list $L'_{2hop}[v]$ of node v, and sort the list in ascending order of pivot nodes p. The sorting of p allows us to calculate shortest distance between two nodes by checking the intersection of pivot nodes in a linear time of their total label size. This arrangement will save huge processing time because we need to frequently perform shortest distance calculation between two nodes in later depth-first answer search.

B. Depth-First Answer Search

Now, we can find the Tpk answers by a depth-first search with pre-order traversal. As shown in Figure 3, the node list of three query keywords a, b and c read from the inverted file vf will be loaded into memory and viewed as three levels in an imagined tree-like structure with an empty root. Each node list represents one level for DFS traversal. Note that the linkage information between nodes in the imagined structure is not maintained, so we just need a little memory space for keeping nodes in the most recently visited path from the root to current node.

In the imagined tree-like structure, all nodes in level t are the children of every node in level t-1. Suppose we have a current DFS visit on node v2 in level 2 and a previous DFS visit on node v4 in level 1, the path that contains most recently visited nodes in each level from level 1 to level 2 form a node set C' = (v4, v2). The diameter ranking score of C' is a lower bound on the score of any node set that is a super set of C'. The score is monotonic because adding a node to a node set does not decrease its score. Suppose we have a next DFS visit on node v3 in level 3, which leads to a node set C = (v4, v2, v3). Let $DB(C)$ denote the diameter score of C. The scores of C can be computed from those of C' with additional calculation of distance information between node v3 and any other node in C as following: $DB(C) = max\{DB(C'), dist(\text{v3, v4}), dist(\text{v3, v2})\}$. Note that we need to remove duplication from the node set before calculating the scores.

Due to the monotonicity of node set's scores, if $DB(C') > Dia$, all DFS visits that result in a super set of C' can be skipped for candidate answer examination. When a path reaches a leaf node in the imagined tree structure and the related node set passes the diameter check, we consider the node set to be a candidate answer that meets the keyword and diameter constraints. To check whether the node set is a minimal cover of query keywords, we remove one node each time from the set to see whether the remaining nodes cover all the query keywords. If the candidate answer is a minimal cover of QW, it would be inserted into the top-k priority answer queue based on the various ranking methods of different problems.

V. Experimental Results

In this section, we evaluate the claims we made about the proposed method KeyLabel in Section I. The experimental comparison of RClique, GDensity and KeyLabel algorithms is done under a C++ environment in Linux OS using a machine with an Intel(R) Core(TM) i5 2.67GHz CPU, 8GB RAM and 7200RPM SATA hard disk.

A. Tested Data Sets

Data set	Node	Edge	Avg Keyword/Node	Avg Degree
DBLP1	317K	664K	0.344	4.19
DBLP2	317K	1M	2.27	6.62
BTC	168M	181M	0.3	2.2
RDF	894K	963K	14.68	2.15

Table II
Information about data sets

The statistics about node number, edge number, average keyword per node and average degree of each tested data set are summarized in Table II. DBLP1 and DBLP2 represent graphs of varied degree density for comparison among three algorithms. BTC represents varied size graphs for evaluating scalability. RDF data set is used to evaluate the effectiveness and accuracy of constructing SPARQL queries from keyword search. All selected data sets are undirected and unweighted.

DBLP1 The DBLP1 graph is modeled from the DBLP XML data (*http://dblp.uni-trier.de/xml/*). It contains the information about a collection of papers and authors as nodes and related citations and authorship as edges. Words in paper title and author names are viewed as keywords. We extract the same number of nodes as DBLP2 data set from the original XML data in order to compare the influence of average degree and average keyword on performance.

DBLP2 Similar to DBLP1, the SNAP DBLP2 data set is a collection of research papers in computer science field (*http://snap.stanford.edu/data/com-DBLP.html*), in which nodes represent authors, and edges represent co-authorship of two authors if they published some paper together. Authors belong to the same community if they published to the same journal or conference, which is used as a keyword. DBLP2 has higher average degree and average keyword per node than DBLP1 data set, which increases the general difficulty of keyword search.

BTC The BTC semantic graph is converted from the Billion Triple Challenge 2009 data set. To distribute synthetically generated keywords with size equal to 0.1% of the node size, we sort nodes in increasing order of their degrees and evenly divide keywords into three categories marked as "easy", "medium" and "hard". For each "easy" keyword, we randomly draw 100 nodes from the first 1/3 sorted nodes and assign the keyword to them. Similarly, we randomly draw 300 and 500 nodes from the second 1/3 and last 1/3 of the

sorted nodes for each keyword in the “medium” and “hard” categories. Keywords with larger average degree of nodes and higher frequency would induce larger search space and thus increase the difficulty of keyword search for all the three algorithms.

RDF The RDF data provided by DBpedia organization (*http://benchmark.dbpedia.org/benchmark_10.nt.bz2*) defines a benchmark for data extracted from Wikipedia in RDF format. We'll apply the modeling mentioned in the introduction before doing keyword search.

B. Indexing Comparison on DBLP Data

Dataset-Algorithm	Time (s)	Size (MB)	Memory Usage (MB)
DBLP1-RClique	18626	12595	176
DBLP1-GDensity	346	30	710
DBLP1-KeyLabel	600	685	1373
DBLP2-RClique	76162	48742	351
DBLP2-GDensity	503	10	782
DBLP2-KeyLabel	771	3174	2322

Table III
INDEXING TIME, INDEX SIZE AND MAXIMUM INDEXING MEMORY USAGE COMPARISON

As shown in Table III, RClique-Indexing needs more than 10 times longer time and larger disk space than the other two algorithms because of its full materialization of indexing. RClique-Indexing requires minimum memory space because it only examines the θ-neighborhood of one node each time. GDensity-Indexing has minimum indexing time and index size because it only calculates and stores the distance distribution within every node's θ-neighborhood. However, When the graph is too large to be held in main memory, GDensity algorithm will become inapplicable.

Unlike the indexing of RClique and GDensity that requires a diameter upper bound θ, KeyLabel-Indexing provides full indexing that can be used to answer queries with any diameter constraint. The indexing time of KeyLabel algorithm is the total time used for Hop Doubling Label Indexing and keyword label list indexing. Although larger memory space is needed to load label entry lists and assign them to the related keywords, the procedure is done only once offline and later allows a very small portion of such label entries to be retrieved at querying time.

C. Querying Comparison on DBLP Data

The difficulty of queries is decided by four factors, including diameter constraint, top-k constraint, query size and keyword difficulty. The diameter constraint (*Dia*) is set according to the average diameter and largest diameter of each data set. Too large diameter will cause almost the whole graph to be searched, which is not practical. Too small diameter often leads to a small searching range and consequently results in very little search time, which is not good for comparison. The top-k constraint (*Tpk*) allows us to observe how the query search time varies when different numbers of answers are requested. Different query sizes (*Qsz*) are tested because more keywords would lead to more combinations of candidate answers to verify. The difficulty of keywords (*Dif*)is decided by the frequency of keywords, which affects the size of search space.

For the comparison of querying time of three algorithms, we draw three sets of curves showing their query speed with different settings of the four most important factors. For each data set, we test 3 *Dia*s, 3 *Tpk*s, 4 *Qsz*s and 3 *Dif*s, resulting in 108 combinations of parameter setting in total. We create 10 queries for each of the 108 combinations. The average search time of queries that have the same value of a particular parameter would be used as performance criteria.

Dataset-Algorithm	Memory Usage (MB)
DBLP1-RClique	3822
DBLP1-GDensity	447
DBLP1-KeyLabel	8
DBLP2-RClique	7054
DBLP2-GDensity	694
DBLP2-KeyLabel	56

Table IV
MAXIMUM QUERYING MEMORY USAGE COMPARISON

As shown in Table IV, RClique-Querying has very large memory space usage because the linkage information of nodes that contain query keywords will be loaded when query comes. GDensity-Querying has to keep the whole graph structure in memory when doing query, so the memory space used for query search is relatively large. KeyLabel-Querying only retrieve query keywords' label entries with distance no more than Dia for querying, so it requires much less memory space usage than the other two algorithms. Due to the complexity of queries, some query search might take very long time, so we setup a threshold of 100 seconds for each single query search. All query times exceed this threshold will be set to 100 seconds before averaging.

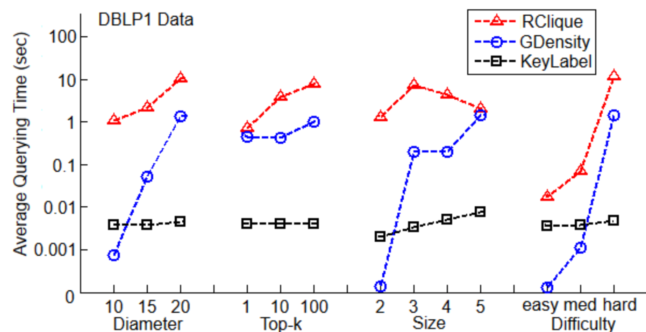

Figure 4. Querying performance on DBLP1

DBLP1 Data. KeyLabel-Querying can finish all of the 1080 tested queries within 0.1 second, while 40 queries exceed 100 seconds with RClique-Querying, and 5 queries

exceed 100 seconds with GDensity-Querying. In Figure 4, KeyLabel-Querying has almost same average query time for all settings because most of the time is spent on basic setup. Since the average degree and average keyword of DBLP1 data set are very low, GDensity-Querying needs less average querying time than KeyLabel-Querying for some easy settings. But as queries become harder, our KeyLabel-Querying obviously outperforms GDensity-Querying. RClique-Querying requires the largest average querying time in all cases.

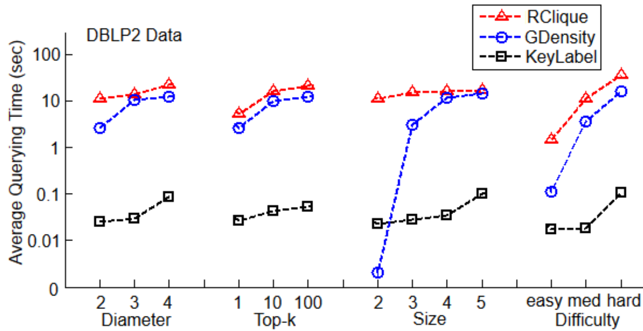

Figure 5. Querying performance on DBLP2

DBLP2 Data. The average querying time of testing 1080 queries on DBLP2 data is shown in Figure 5. No query exceeds the threshold for KeyLabel-Querying, and the longest search time is less than 4 seconds. 77 queries take more than 100 seconds to finish for GDensity-Querying. For RClique-Querying, the number increases to 204. Our KeyLabel-Querying outperforms the other two algorithms more clearly on the DBLP2 data set because it has higher average degree and average keyword per node than the DBLP1 data set. Again, RClique-Querying requires largest average querying time in all cases.

D. Scalability Test on Large BTC 2009 Data

When the data set is large, RClique and GDensity might become impractical because both algorithms would take too much time for indexing or require large memory space for querying. We tested the scalability of our KeyLabel algorithm on the large BTC 2009 data set, which has much larger size than the other three data sets and cannot be indexed or queried by RClique (over 100 hours for indexing) and GDensity (not enough memory space) in our machine. To verify the influence that the size of graphs has on our KeyLabel algorithm, we also test 50M-node and 100M-node data subsets extracted from the original 168M-node graph. The number of keywords for the three data sets is set to 0.1% of the number of their nodes.

Table V shows the indexing time, index size, maximum main memory usage for indexing and maximum main memory usage querying of KeyLabel algorithm on BTC data sets. Even for the largest 168M-node data set, our KeyLabel

Node (M)	50	100	168
Indexing Time (s)	1661	12704	25601
Index Size (MB)	2867	6861	12288
Indexing Memory Usage (MB)	3934	4748	5722
Querying Memory Usage (MB)	487	950	1851

Table V
INDEXING AND QUERYING STATISTICS ON BTC

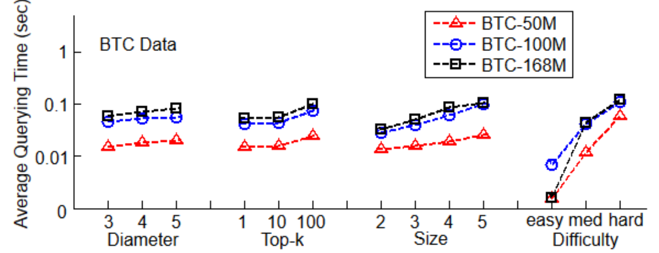

Figure 6. Scalability test of querying time on BTC

algorithm does not require too much memory space at querying time, and the indexing cost is acceptable. Figure 6 shows the search time performance of KeyLabel algorithm on BTC data sets. Although the index size of the BTC data set with 168M nodes reaches 12GB, all queries can finish within 1 second. The querying time increases almost linearly with the four factors of query setting for all three data sets, but the influence of keyword difficulty is obviously larger than that of other three factors.

E. Accuracy Test on DBpedia RDF Data

As a concrete application, we evaluate our KeyLabel algorithm for constructing optional SPARQL queries for the RDF data set introduced in Section V-A. We compare the accuracy of computed queries with the result of RClique. Note that GDensity produces exact answers as our KeyLabel algorithm instead of 2-approximation, so we skip comparing the accuracy with GDensity.

We randomly select 20 subgraphs with diameter 4 and construct the related SPARQL queries from the subgraphs as our "ground truth". For each ground truth SPARQL query Q_S, we create a keyword search query Q by picking 1 to 2 meaningful keywords from each node in the subgraph induced by Q_S to simulate users' partial knowledge about Q_S. Then we apply RClique and KeyLabel algorithms to produce top-10 answers for Q and rebuild SPARQL queries Q'_S by retrieving the linkage information for the nodes in each of top-10 answers. At last, we identify whether any rebuilt SPARQL query Q'_S matches the corresponding ground truth SPARQL query Q_S and compare the highest ranked match of both algorithms. The higher rank of a match, the more useful the graph keyword search for constructing SPARQL queries for RDF data.

Table VI shows the result of accuracy test on RDF data. Taking Query10 for example, 6 means that among

Queries	KeyLabel	RClique	Queries	KeyLabel	RClique
Query1	1	2	Query2	2	6
Query3	1	1	Query4	1	-
Query5	1	1	Query6	1	1
Query7	1	1	Query8	1	4
Query9	2	-	Query10	6	-
Query11	2	2	Query12	1	1
Query13	1	1	Query14	1	1
Query15	1	1	Query16	1	1
Query17	1	1	Query18	1	1
Query19	2	1	Query20	1	1

Table VI
THE FIRST HIT RANK OF RECONSTRUCTED SPARQL QUERIES

the 10 answers returned by the KeyLabel algorithm, the SPARQL query rebuilt from the 6th ranked answer is the first match to the corresponding ground truth SPARQL query. - , on the other hand, indicates that none of the SPARQL queries rebuilt from the 10 answers returned by the RClique algorithm is a match to the corresponding ground truth SPARQL query. Except for Query19, KeyLabel algorithm has the same or higher ranked hit than RClique. The lower ranked hit of RClique is caused by the 2-approximation of answers returned, which returns many loose answers having a large diameter.

VI. CONCLUSION

Many large graph search problems can be modeled as graph keyword search problems. We studied this problem for large graphs where previous approaches suffered from serious bottlenecks. The proposed KeyLabel algorithm takes the result of Hop Doubling Label Indexing algorithm, assigns reformatted label entries to keywords to generate an index based on both keywords and distance of nodes, and performs fast keyword search with small memory space usage using that index. KeyLabel algorithm shows good performance even for very large data sets. KeyLabel is proven to outperform RClique and GDensity in most cases. The experimental result also verifies the efficiency and accuracy of KeyLabel algorithm in mapping keyword based queries to SPARQL queries on RDF data sets.

VII. ACKNOWLEDGMENT

This work is partially supported by a Discovery Grant from Natural Sciences and Engineering Research Council of Canada, and partially done during a visit to SA Center for Big Data Research hosted in Renmin University of China. This Center is partially funded by a Chinese National 111 Project "Attracting International Talents in Data Engineering and Knowledge Engineering Research".

REFERENCES

[1] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440. IEEE, 2002.

[2] B. Ding, J. Xu Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE 2007*, pages 836–845. IEEE, 2007.

[3] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: ranked keyword search over xml documents. In *SIGMOD*, pages 16–27. ACM, 2003.

[4] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316. ACM, 2007.

[5] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on xml graphs. In *ICDE*, pages 367–378. IEEE, 2003.

[6] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *Proceedings of the VLDB Endowment*, 7(12):1203–1214, 2014.

[7] M. Kargar and A. An. Discovering top-k teams of experts with/without a leader in social networks. In *CIKM*, pages 985–994. ACM, 2011.

[8] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *Proceedings of the VLDB Endowment*, 4(10):681–692, 2011.

[9] G. Klyne and J. J. Carroll. Resource description framework (rdf): concepts and abstract syntax. 2006.

[10] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *SIGKDD*, pages 467–476. ACM, 2009.

[11] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914. ACM, 2008.

[12] N. Li, X. Yan, Z. Wen, and A. Khan. Density index and proximity search in large graphs. In *CIKM*, pages 235–244. ACM, 2012.

[13] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal. Retrieving and organizing web pages by information unit. In *World Wide Web*, pages 230–244. ACM, 2001.

[14] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

[15] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k nearest keyword search on large graphs. *PVLDB*, 6(10):901–912, 2013.

[16] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE*, pages 405–416. IEEE, 2009.

[17] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD*, pages 527–538. ACM, 2005.