Exact Top-k Nearest Keyword Search in Large Networks

Minhao Jiang[†], Ada Wai-Chee Fu[‡], Raymond Chi-Wing Wong[†]
[†]The Hong Kong University of Science and Technology [‡]The Chinese University of Hong Kong
{mjiangac, raywong}@cse.ust.hk adafu@cse.cuhk.edu.hk

ABSTRACT

Top-k nearest keyword search has been of interest because of applications ranging from road network location search by keyword to search of information on an RDF repository. We consider the evaluation of a query with a given vertex and a keyword, and the problem is to find a set of k nearest vertices that contain the keyword. The known algorithms for handling this problem only give approximate answers. In this paper, we propose algorithms for top-k nearest keyword search that provide exact solutions and which handle networks of very large sizes. We have also verified the performance of our solutions compared with the best-known approximation algorithms with experiments on real datasets.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

2-hop Labeling, Nearest Keyword Search, Keyword-lookup Tree

1. INTRODUCTION

Massive networks are emerging as a rich source of useful information. In addition to the structural characteristics of the network, users may aim to search for keyword-related data, similar to the popularity of keyword search on the internet repositories. Many real networks, especially the social networks, contain keywords in the vertices. For example, in a road network, a vertex may be labeled with a keyword of "hospital", "highschool", or "seven-eleven", etc. As pointed out in [4] and [26], some keyword queries in a network are generated from a vertex inside the network with an interest of looking for vertices in a near-vicinity of the network. An example would be the search for "kindergarten" from a home location in a road network, and the interest may be in the nearest 10 results. Another example is the search in a social network where the query source is from an individual in the network, and the query is looking for some closely related others with a certain interest or a certain skill.

We study the problem of top-k nearest keyword search in a network, where the query consists of a vertex in the network, a keyword, and an integer k for the desired number of top answers. As in [26], we refer to this problem as k**-NK**. Given such a query and a data network, an obvious solution is to apply Dijkstra's search from the query vertex to find the nearest vertices that contain the keyword. However, this will not be efficient for massive graphs.Recently, efficient query evaluation methods were proposed in [4, 26]. However, there are two issues with the existing algorithms in [4] and [26]: (1) The algorithms do not return exact answers to the queries. Given a graph $G = (V, E)$ with vertex set V, and edge set E, the algorithm in [4] incurs a $(2\log_2|V| - 1)$ approximation factor, which can be quite large given large values of $|V|$, and as shown in [26], the resulting error is significant in their empirical study in real graphs and good solutions can be missed. (2) Both methods [4, 26] assume that the index can reside in main memory. This may not be true for massive graphs and when the index size is many times bigger than the given graph. Therefore, it remains an open problem of handling graphs and indices resided on disk.

We take a very different approach to the problem. Instead of constructing oracles that return approximate answers, we use an indexing method that can return exact distance answers with fast response time. 2-hop labeling techniques, introduce in [12], can be found in the literature for distance querying and reachability querying in a given graph. Recent works in [14, 2, 19] have shown that a small index size is achievable for many real massive graphs. In particular, [19] proposed an I/O efficient algorithm with scalable complexity bounds for scale-free graphs. Hence, 2-hop labeling technique can serve as a tool for handling the distance requirement of the k-NK problem. We devise top-k keyword search algorithms with the concepts behind the labeling technique. We introduce a novel data structure called the KT index which facilitates efficient keyword search to handle the performance issue of frequent keywords. We propose efficient in-memory and disk-based algorithms for k-NK when the index resides on disk.

The contributions of this paper are the following. We propose algorithms for the top-k nearest keyword search problem in a massive graph. Our algorithms return the exact answers for such queries. For graphs that are very big so that the indices do not fit in the main memory, we propose algorithms for efficient disk-based query processing. We illustrate the efficiency of our proposed method by an empirical study on real and synthetic datasets. We show that both index construction and query evaluation for k-NK in large networks can be handled by a commodity machine.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2749447.

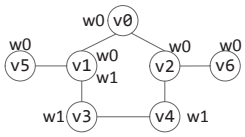


Figure 1: Example Graph G_1

This paper is organized as follows. We introduce the problem definition in Section 2, and describe the existing solutions in Section 3. In Section 4, the framework of our solution is introduced along with the concepts of label indexing. In Section 5, we introduce our memory-based solutions for k-NK. Section 6 contains extensions for disk residence indexing for k-NK, multiple keyword querying, and dynamic updates. An empirical study on our solutions is given in Section 7. Section 8 is about related works. Section 9 concludes the paper.

2. PROBLEM DEFINITION

In this section, we introduce more formally the problem of top-k nearest keyword search in a given network.

We are given a weighted undirected graph (network)[1] $G(V, E)$ where V is the set of vertices in G and E is the set of edges in G. We also use $n = |V|$ and $m = |E|$ to denote the number of vertices and edges in G. Each edge (u, v) in G is given a positive length, $d_G(u, v)$. A path $p = (v_1, ..., v_l)$ is a sequence of l vertices in V such that for each v_i $(1 < i < l)$, $(v_i, v_{i+1}) \in E$. We also denote a path $(x, ..., y)$ by $x \rightarrow y$. The length of a path is the sum of the lengths of the edges along the path. Given 2 vertices u, v in V, the distance between u and v, denoted by $dist(u, v)$, is the minimum length among all paths from u to v in G. Each vertex $v \in V$ contains a set of zero or more keywords which is denoted by $doc(v)$. The set of all keyword occurrences that appear in the vertices in G is denoted by $doc(V)$ (the same keyword at two vertices are considered two different occurrences). $|doc(V)| = \sum_{v \in V} |doc(v)|$. We denote the set of vertices containing keyword w by V_w. If vertex v contains keyword w, we say that $w \in v$. Some of our notations are listed in Table 1.

DEFINITION 1 (**k-NK querying**). *: Given a weighted undirected simple graph (network) $G(V, E)$, a top-k nearest keyword (k-NK) query is a triple (q, w, k), where $q \in V$ is a query vertex in G, w is a keyword, and k is a positive integer. Given a query $Q = (q, w, k)$, the result is a set of k vertices in V_w, $R = \{v_1, ..., v_k\}$ such that there does not exist any vertex $u \notin R, u \in V_w$ such that $dist(q, u) < \max_{v \in R} dist(q, v)$.*

EXAMPLE 1. *Consider the unweighted graph G_1 in Figure 1, where each vertex v_i is attached with a list of keywords that it contains. For example, v_0 contains keyword w_0, v_1 contains keywords w_0 and w_1, etc. If the query is $(v_2, w_0, 2)$ then $\{v_2, v_0\}$ or $\{v_2, v_6\}$ will be returned as the answer.*

3. EXISTING SOLUTIONS

Given the k-NK problem, with a query of (q, w, k), we can apply Dijkstra's search to compute the shortest paths from the query vertex to find the nearest vertices with keyword w. However, when the given graph is massive, Dijkstra's search will become too costly, and the problem becomes challenging. In this section, we summarize the current state-of-the-art solutions for this problem in [4] and [26], respectively. Both solutions are approximate.

[1]We shall use the terms *graph* and *network* interchangeably.

Notation	Meaning
$G(V, E)$	given graph with vertex set V and edge set E
$dist(u, v)$	distance between u and v in G
$v_1 \rightarrow v_2$	path $(v_1, ..., v_2)$
q	given query vertex point
w	given query keyword
k	top k results are needed
V_w	set of vertices in G containing keyword w
W	set of all keywords in V
$doc(V)$	the set of all keyword occurrences in V
L	2-hop label, $L = \{(u, d) \in L(v) : v \in V\}$
$L(v)$	set of label entries for vertex v in L
(u, d)	a label entry in $L(v)$
L_w	set of label entries in L with an end point in V_w
$LB(v)$	$((u_0, d_0), (u_1, d_1), (u_2, d_2), ...)$ where $(v, d_i) \in L(u_i)$ and $d_0 \geq d_1 \geq ...$
$[x..y]$	a fragment $(u_x, u_{x+1}, ...u_y)$, where $((u_x, d_x), ..., (u_y, d_y))$, $x \leq y$, is a subsequence of $LB(v)$

Table 1: Some Notations Used

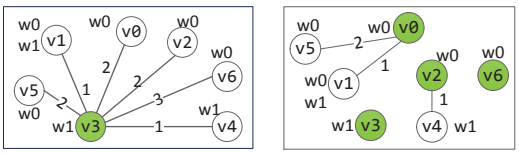


Figure 2: Two Distance Oracles $\mathcal{O}_a$ and $\mathcal{O}_b$ for G_1

3.1 Approximate k-NK in a graph by PMI

In the PMI (Partitioned Multi-Indexing) scheme [4], given an undirected graph $G = (V, E)$, a set of r distance oracles is used, where $r = p \log |V|$. Let the oracles be $\mathcal{O}_1, \mathcal{O}_2, ..., \mathcal{O}_r$. Oracle $\mathcal{O}_i$ is built from a partitioning of V formed by $2^{\lceil i/p \rceil - 1}$ randomly selected seed vertices, each of which collects vertices closer to it than to other seeds. Each partition is transformed into a star where the seed c is in the center and each other vertex is linked to the seed with a distance that preserves its original distance in the graph from the seed c. A set of $p \log |V|$ distance oracles are constructed. With each oracle $\mathcal{O}_i$, we can estimate the distance $dist(u, v)$ for any two given vertices u and v as follows: if u and v are in the same star with a center seed c, then the estimated distance $\overline{dist}_{\mathcal{O}_i}(u, v) = dist(u, c) + dist(v, c)$; if u and v are not in the same star, in $\mathcal{O}_i$, then $\overline{dist}_{\mathcal{O}_i}(u, v) = \infty$. The overall estimated distance after consulting all the oracles is given by $\overline{dist}(u, v) = \min_i \overline{dist}_{\mathcal{O}_i}(u, v)$. The query time is given by $O(k \cdot \log |V|)$. It is shown in [30] that when $p = \theta(|V|^{1/\log |V|})$, the estimated distance has an approximation factor of $2 \log_2 |V| - 1$ with a high probability.

EXAMPLE 2. *For the graph G_1 in Figure 1, two distance oracles $\mathcal{O}_a$ and $\mathcal{O}_b$ are shown in Figure 2. The shaded vertices are the chosen seeds. The estimated value for $dist(v_0, v_2)$ is 4 from $\mathcal{O}_a$ and ∞ from $\mathcal{O}_b$, so the overall estimation is 4, which is four times the true distance of 1 in graph G_1. Given a k-NK query of $(v_2, w_0, 2)$, $\{v_2, v_1\}$ will be returned as an answer while the true answer should be $\{v_2, v_0\}$ or $\{v_2, v_6\}$.*

3.2 Approximate k-NK by shortest path trees

The authors of [26] point out that the error introduced by the star summary in [4] can be large. In their method, multiple oracles are built as in [4], but in each oracle, shortest path trees replace the star trees. They have devised some interesting tech-

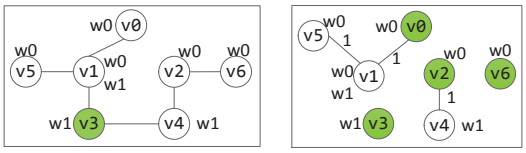


Figure 3: Shortest Path Forests T_a and T_b for G_1

$L(v_0)$	$\{(v_0, 0)\}$
$L(v_1)$	$\{(v_1, 0), (v_0, 1)\}$
$L(v_2)$	$\{(v_2, 0), (v_0, 1)\}$
$L(v_3)$	$\{(v_3, 0), (v_0, 2), (v_1, 1), (v_2, 2)\}$
$L(v_4)$	$\{(v_4, 0), (v_0, 2), (v_2, 1), (v_3, 1), (v_1, 2)\}$
$L(v_5)$	$\{(v_5, 0), (v_0, 2), (v_1, 1)\}$
$L(v_6)$	$\{(v_6, 0), (v_0, 2), (v_2, 1)\}$

Figure 4: A 2-hop Label Index for G_1

niques for handling k-NK on a tree $T = (V, E)$ based on the concepts of entry node and entry edge on the compact tree designed for 1-NK in a tree [32]. Two solutions are provided for k-NK for a graph: one is *graph-boundk*, assuming a bound on k, and another solution is *graph-pivot*. The query time complexities are $O((\log |V_w| + \overline{k}) \log |V|)$ and $O(k \log^2 |V|)$, respectively, where $\overline{k}$ is the upper bound on k. The index size without the k bound is given by $O(|doc(V)| \cdot \log^2 |V|)$. From their empirical studies, there is less error compared with PMI [4]. However, their solutions still miss 10% to 20% of the true answers.

EXAMPLE 3. *In Figure 3, we show the shortest path forests T_a and T_b for the graph G_1 in Figure 1, which are constructed based on $\mathcal{O}_a$ and $\mathcal{O}_b$. From these forests, the distance estimated for $dist(v_0, v_2)$ is 4, which is much greater than the true distance of 1 in G_1. Given a k-NK query of $(v_2, w_0, 3)$, $\{v_2, v_6, v_1\}$ will be returned as an answer while the true answer is $\{v_2, v_6, v_0\}$.*

4. PROPOSED SOLUTION FOR K-NK

We first give an outline of our main solution. We take a very different approach compared to previous studies. We make use of recent research results on shortest path querying for massive graphs where it is shown that indexing based on vertex labels can be very effective [12, 20, 14, 2, 19]. Our method makes use of the 2-hop labeling technique, which constructs a **label** for each vertex. With such labeling for all vertices, a distance query for s, t can be answered by looking up the labels of s and t only.

4.1 Components of the Proposed Solutions

In this subsection, we introduce the main components of our solution. The components handles keywords of different frequencies.

- Forward search (FS) component. The basic idea is to compute all possible shortest paths between the query vertex q and all vertices in V_w that contain the keyword w. This is highly effective if the keyword is not frequent and there are not many vertices in V_w. This is introduced in Section 5.1.
- Forward backward search (FBS) for frequent keywords. To deal with frequent keywords, a forward backward search is proposed. We describe this component in Section 5.2. In Section 5.3 we describe a hybrid approach that combines the strengths of FS and FBS.

In the next subsection, we introduce the basic ideas of label indexing for the querying of shortest paths, which are important for the components in our solution.

4.2 2-hop Labeling

Consider an undirected weighted graph $G = (V, E)$, where each edge (u, v) in E has a length of $d_G(u, v)$, which is a positive real number. 2-hop labeling supports distance queries between any two vertices s and t. A **label** $L(v)$ is created for each vertex $v \in V$. $L(v)$ is a set of **label entries** (u_i, d_i) where $u_i \in V$, and d_i is a positive real number serving as a distance value of a path between u_i and v. We say that u is a **pivot** in label entry (u, d). The set of $L(v)$ for all $v \in V$ forms the 2-hop label index, L. L is created in such a way that there exists a shortest path $s \rightsquigarrow t$ in G of length ℓ if and only if we can find a pivot vertex u such that $(u, d_1) \in L(s)$ and $(u, d_2) \in L(t)$ such that $d_1 + d_2 = \ell$, and there does not exist any other vertex u' such that $(u', d_1') \in L(s)$ and $(u', d_2') \in L(t)$ with $d_1' + d_2' < \ell$. We say that the pair (s, t) is **covered** by u. We refer to $(s \rightarrow u \rightarrow t, d_1 + d_2)$ as a **path** for an answer to the distance query for s, t. Hence, the distance query can be answered by looking up $L(s)$ and $L(t)$. Given a graph, we first construct a 2-hop label index by a state-of-the-art 2-hop construction algorithm [12, 20, 14, 2, 19], then we build our proposed indices on the 2-hop index.

EXAMPLE 4. *For the undirected graph G_1 in Figure 1, a corresponding 2-hop label index L is shown in Figure 4. Suppose we have a distance query asking for $dist(v_1, v_3)$, we look up $L(v_1)$ and $L(v_3)$, and find the entries $(v_0, 1)$ and $(v_1, 0)$ in $L(v_1)$, and $(v_0, 2)$ and $(v_1, 1)$ in $L(v_3)$, respectively. Thus, we return the value of $0 + 1 = 1$ as $dist(v_1, v_3) = (v_1 \rightarrow v_1 \rightarrow v_3)$, since it is the smallest distance sum for the matching pairs.*

5. EXACT K-NK QUERY EVALUATION

In this section, we propose our in-memory solutions for answering k-NK queries. When the problem size is within the capacity of the main memory, we can use the index methods described as follows for query evaluation. In Section 5.1, we introduce a method called FS to be used when the query keyword is not frequent in the graph. In Section 5.2, we propose another method called FBS to be used when the query keyword is frequent. Then, we study a hybrid method combining the strengths of the 2 methods proposed (Section 5.3).

5.1 Forward Search (FS)

Given a query (q, w, k), forward search (FS) is to find the answers by forward search from $L(q)$ and $L(y)$ where $y \in V_w$. For each v, the label entries (u, d) in $L(v)$ are sorted by u. We can examine the shortest distance between each vertex $y \in V_w$ and q by searching $L(y)$ and $L(q)$. The top-k nearest vertices are extracted among all vertices in V_w by a max heap of size k.

EXAMPLE 5. *For the undirected graph G_1 in Figure 1 and its 2-hop label index L in Figure 4, there are 7 vertices in the graph, most of which contain one keyword, w_0 or w_1, but v_1 contains two keywords w_0 and w_1. For graph G_1, and the index L, $V_{w_1} = \{v_1, v_3, v_4\}$. $L_{w_1} = L(v_1) \cup L(v_3) \cup L(v_4)$.*

Given query $(v_5, w_1, 2)$, since $V_{w_1} = \{v_1, v_3, v_4\}$, we check the label entries in $L(v_1), L(v_3)$ and $L(v_4)$ with 3 linear scans and match with the label entries in $L(v_5)$. The result is $\{v_1, v_3\}$, which is the exact solution.

The scanning of the label entries for a vertex y takes $O(|L(y)| + |L(q)|) = O(\frac{|L|}{|V|})$ expected time. When processing the labels for each vertex in V_w, we maintain the nearest k vertices by a max

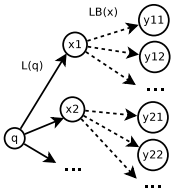


Figure 5: Forward Backward Search on L and LB

heap of size k in $O(|V_w| \log k)$ time. On average, we have $|V_w| = O(\frac{|doc(V)|}{|W|})$, so totally it takes $O(|V_w|\frac{|L|}{|V|}) + O(|V_w| \log k) = O(\frac{|doc(V)|L||}{|W||V|})$ time. In addition, we need $O(|L|)+O(|doc(V)|) = O(|L|)$ space in memory where $O(|L|)$ space is for 2-hop index L and $O(|doc(V)|)$ is for an inverted list to find V_w. We call this method forward search or **FS**. FS can be very effective for keywords with low frequencies.

5.2 Forward Backward Search (FBS)

The forward search algorithm becomes inefficient when the keyword w has high frequencies resulting in many candidates in V_w to be checked. Here, we propose an efficient mechanism to handle such cases by incorporating forward search and backward search on the label index. We call this method forward backward search or **FBS** algorithm. For FBS, in addition to the 2-hop index L, we also build the following two indices.

1. *2-hop Label Backward Index*, LB.
2. *Keyword-lookup Tree Index*, KT.

5.2.1 LB index

We first describe the 2-hop label backward index, i.e., LB index, which consists of an index element $LB(v)$ for each vertex $v \in V$. Similar to L, for each vertex v, $LB(v)$ consists of a list of label entries (u, d). It is constructed in such a way that $(u, d) \in LB(v)$ if and only if $(v, d) \in L(u)$. For each $LB(v)$, we assume that all the entries of the form (u, d) are sorted by non-increasing order of the distance value, d. For example, with the label index in Figure 4, $LB(v_1) = \langle(v_4, 2), (v_5, 1), (v_3, 1), (v_1, 0)\rangle$.

We denote the set of 2-hop label entries containing a vertex in V_w as an end point by L_w. That is, $L_w = \bigcup_{v \in V_w} L(v)$. The objective of FBS is to avoid scanning the whole L_w. To this end, we search from the query vertex q and from the vertex x in each label entry $(x, d) \in L(q)$. We search for label entries in the backward index $LB(x)$ with vertices that contain keyword w, i.e. $(y, d') \in LB(x)$ where y contains w. A simple illustration of FBS is shown in Figure 5. The solid lines correspond to label entries $(x, d) \in L(q)$, while the dotted lines correspond to label entries $(y, d') \in LB(x)$ with $(x, d) \in L(q)$. We can obtain one possible answer y from the path $(q \rightarrow x \rightarrow y, d+d')$. For example, given $q = v_5$ and $w = w_1$, from Figure 4, $(v_1, 1) \in L(v_5)$, since $(v_3, 1) \in LB(v_1)$ and $v_3 \in V_{w_1}$, one possible answer is $(v_5 \rightarrow v_1 \rightarrow v_3, 2)$.

5.2.2 KT Index

Since $LB(x)$ may contain vertices which do not contain keyword w, we need a mechanism to look up the entries of vertices that do contain w efficiently. Moreover, if we do this lookup according to an ordering of the distances in the entries, we can retrieve results in a distance order so that we can stop when k results are obtained. We introduce such a mechanism based on the tree index KT.

The keyword-lookup tree index KT is made up of an index element $KT(v)$ for each vertex $v \in V$. $KT(v)$ is a forest built by breaking the set of label entries of $LB(v)$ into fragments. Each tree node in $KT(v)$ contains some keyword information of a fragment, so that the entries containing the query keyword in a fragment can be retrieved efficiently. KT utilizes a hash function H.

DEFINITION 2. *The hash function H maps vertices and keywords to a binary number with h bits as follows. For each keyword w, the hash function $H(w)$ sets exactly one of the h bits to 1. For a vertex v, $H(v)$ is the superimposition (bitwise OR) of $H(w)$ for each keyword w in v. Similarly, for a vertex set X, $H(X)$ is the superimposition of $H(v)$ for each vertex v in X.*

In the remaining discussions we use $\wedge$ ($\vee$) to stand for bitwise AND (OR). Given a set X of vertices and a keyword w, we can determine that no vertex in X contains w if $H(X) \wedge H(w) = 0$. A non-zero result indicates that w may be contained in some vertex in X.

For each vertex v, the keyword-lookup tree index $KT(v)$ is a forest in which each tree node contains the compressed hash value for a certain subset of the keywords in u with $(u, d) \in LB(v)$, such that the tree nodes in the forest together cover all such keywords. $KT(v)$ enables efficient lookup of nodes containing a given keyword with non-decreasing distances d. Before we introduce the index, we first state some relevant notations and definitions.

We denote the label entries in $LB(v)$ in a way that is consistent with a sorted non-ascending order of the distances d as $LB(v) = [(u_0, d_0), (u_1, d_1), ..., (u_{|LB(v)|-1}, d_{|LB(v)|-1})]$, so that $d_0 \geq d_1 \geq d_2 \geq ... \geq d_{|LB(v)|-1}$.

DEFINITION 3 ($bit(x)$ AND $bit_i(x)$). *The value of $bit(x)$ is the number of 1's in the binary representation of x. $bit_i(x)$ is the value of the i-th significant 1-bit in the binary representation of x.*

For example, $13 = (1101)_2$, then $bit(13) = 3$. $bit_1(13) = (1000)_2 = 8, bit_2(13) = (0100)_2 = 4, bit_3(13) = (0001)_2 = 1$.

We partition the $|LB(v)|$ label entries in the sorted $LB(v)$ into $bit(|LB(v)|)$ groups according to the $bit_i(x)$ values. For examples, if $|LB(v)| = 13$, we form $bit(13) = bit((1101)_2) = 3$ groups. The first group consists of the first $bit_1(|LB(v)|)$ label entries in the sorted $LB(v)$, the second group consists of the next $bit_2(|LB(v)|)$ label entries, etc. Likewise, we can extract the vertices in the label entries and form an ordered vertex set of the form $(u_0, u_1, ..., u_{|LB(v)|-1})$. Sub-sequences in this ordered vertex set are called **fragments**.

Hence, each group in $LB(v)$ corresponds to a fragment S of 2^i vertices for some i, and forms a **tree** in the $KT(v)$ forest. Each **node** t in the tree for S corresponds to a subfragment, say, $F(t) = (u_x, u_{x+1}, ...u_y)$ in S, and the node contains the hash value of $H(\{u_x, u_{x+1}, ..., u_y\})$. We also denote this fragment by $[x..y]$. We say that t covers $[x..y]$. The root of this tree covers S. The **length** of $[x..y]$ is given by $y - x + 1$. For convenience, we may refer to a node t by its fragment $F(t)$.

In general, we partition a fragment S of size 2^i, for some $i > 0$, into 2 fragments of length 2^{i-1}, and set the node of fragment S_{2^i} as the parent of two nodes covering the 2 shorter fragments. The partitioning is continued recursively until a fragment covers only one vertex, forming a full binary tree with $2^{i+1} - 1$ nodes.

In Figure 6, we consider $LB(v)$ with $|LB(v)| = 13$, $LB(v)$ is partitioned into 3 groups forming 3 trees, T_1, T_2, and T_3, which are for fragments [0..7], [8..11], [12..12], respectively. The range **"[x..y]"** in a node indicates that the node contains the hash value $H(\{u_x, u_{x+1}, ..., u_y\})$ and covers the fragment $[x..y]$. For simplicity, we may also refer to a node by its fragment. Hence, the root of T_1 is node [0..7] and its children are [0..3] and [4..7]. The numbers in round brackets will be explained shortly.

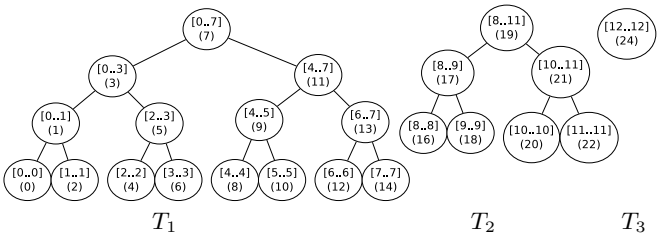


Figure 6: An example of $KT(v)$ for $|LB(v)|$=13

5.2.3 *Storing $KT(v)$ in an array*

Due to the structure of KT, it is possible to encode the trees as an array of hash values without the need of storing pointers between tree nodes. Here we propose a scheme to store $KT(v)$ for $LB(v)$ in an array of length $2|LB(v)|-1$. Let $KT(v,i)$ be the i-th element in the array for $0 \le i \le 2|LB(v)|-2$. We assign the node in $KT(v)$ with hash value $H(\{u_x, u_{x+1}, ..., u_y\})$ to $KT(v, x+y)$, i.e. $KT(v, x+y) = H(\{u_x, u_{x+1}, ..., u_y\})$. One example of encoding a $KT(v)$ for $|LB(v)| = 13$ in an array is shown in Figure 7. In the tree view of the index in Figure 6, the number i in a round bracket means that $KT(v,i)$ stores the hash value of the node. The KT structure allows efficient bitwise operations. A crucial property of this array encoding scheme is given in Lemma 1. For a tree or subtree T in $KT(v)$, let us call the leftmost (rightmost) leaf node the left (right) end of T.

LEMMA 1. *In a forest built for $LB(v) = \{(u_i, d_i)|0 \le i \le |LB(v)|-1\}$, the highest tree node z in a subtree having (u_x) at the right end must have a left end covering $(u_{(x+1)\ \wedge\ x})$, i.e., z covers fragment $[((x+1)\ \wedge\ x)..\ x]$.*

PROOF. We prove by induction. When $|LB(v)| = 2^i$ for some $i \ge 0$, the lemma trivially holds since $KT(v)$ is a single full binary tree. Assume the lemma holds for $1 \le |LB(v)| \le k-1$. We next prove that the lemma is true when $|LB(v)| = k \ne 2^i$ for any i. Let $low(k)$ be the value of the least significant 1 bit of k. For example, $low(12) = low((1100)_2) = (100)_2 = 4$. A forest F for a length k fragment consists of a forest F' for a length $k - low(k)$ fragment, built by the first $k - low(k)$ label entries, and a tree T' for the fragment of $low(k)$, built by the last $low(k)$ label entries. If $0 \le x \le k - low(k) - 1$, u_x is in F'. By the induction hypothesis, the lemma holds.

If $k - low(k) \le x \le k-1$, u_x is in T'. Consider another tree T'' built for some fragment $[s..t]$ where $s = 0$ and $t = low(k) - 1$. Obviously the structure of T'' is exactly the same as T' since their lengths are the same and an one-to-one matching exists between the node for $[s..t]$ in T'' and the node for $[(s + k - low(k))..(t + k - low(k))]$ in T'. Let the highest node in T' with u_x at the right end be $z = [x'..x]$, The node in T'' matching z is given by $[x' - k + low(k)..x - k + low(k)]$. By the induction hypothesis, the lemma holds for T'', thus,

$$x' - k + low(k) = (x - k + low(k) + 1)\ \wedge\ (x - k + low(k)) \quad (1)$$

Consider the binary representation of $x' - k + low(k)$ and $k - low(k)$. Since $low(k)$ is the value of the least significant 1 bit of k and $x' - k + low(k) < low(k)$, adding $k - low(k)$ to each term in Equation (1) only creates some new more significant 1-bits, and does not change the original one bits of the term. Therefore, $x' = ((x+1)\ \wedge\ x)$, and the highest tree node of a subtree with u_x at the right end is $[x'..x] = [((x+1)\ \wedge x)..\ x]$. Therefore, the lemma holds when $|LB(v)| = k$. □

For example, let $x = 11$, the highest tree node containing u_{11} at the right end is the node for fragment $[8..11]$, since $11 = (1011)_2$.

0	1	2	3	4	5	6	7	8
0..0	0..1	1..1	0..3	2..2	2..3	3..3	0..7	4..4

9	10	11	12	13	14	15	16	17
4..5	5..5	4..7	6..6	6..7	7..7	–	8..8	8..9

18	19	20	21	22	23	24
9..9	8..11	10..10	10..11	11..11	–	12..12

Figure 7: The array for keeping the forest of $KT(v)$ for $|LB(v)|$=13. $KT(v,0) = H(\{u_0\})$, $KT(v,1) = H(\{u_0, u_1\})$...

$x+1 = (1100)_2$, $x \wedge (x+1) = (1000)_2$. Note that $x \wedge (x+1)$ basically removes the least significant bit one from $x+1$.

Lemma 1 gives rise to an efficient tree traversal order that will be introduced next. The following lemmas show that the encoding scheme is correct and space-efficient.

LEMMA 2. *Each KT array entry stores at most one tree node.*

PROOF. We assign the tree node for fragment $[x..\ y]$ to the $(x+y)$-th element $KT(v, x+y)$ in the array. Firstly, nodes in different trees cannot be assigned to the same array element because each tree T_i corresponds to $[x_i..\ y_i]$ with $y_i < x_{i+1}$, and the array elements assigned to tree T_i have indices in $[2x_i, 2y_i]$. This implies that they do not overlap. Secondly, when T_i contains more than one node, the root covers fragment $[x..\ y]$ with $x \ne y$ and has two children, for fragments $[x..\ \lceil (x+y)/2 \rceil - 1]$ and $[\lceil (x+y)/2 \rceil ..\ y]$, respectively. Each node in the left subtree is assigned to $KT(v,i)$ for some $i \le \lceil (x+y)/2 \rceil - 1 + \lceil (x+y)/2 \rceil - 1 \le x + y - 1$. Similarly, each node in the right subtree is assigned to $KT(v,j)$ for some $j \ge \lceil (x+y)/2 \rceil + \lceil (x+y)/2 \rceil \ge x + y + 1$. Thus, only the root node is assigned to $KT(v, x+y)$. This argument holds recursively for the subtrees rooted at all descendants, which completes the proof. □

LEMMA 3. *The number of unused elements in the array for $KT(v)$ is given by $bit(|LB(v)|) - 1 = O(log|LB(v)|)$.*

PROOF. Given two trees for two successive fragments $[x..y]$ and $[y+1..z]$, the only unused element $KT(v,i)$ for $2x \le i \le 2z$ is $KT(v, 2y+1)$ since the maximum used element in $[x..\ y]$ is $y + y = 2y$ and the minimum used element in $[y+1..\ z]$ is $y + 1 + y + 1 = 2y + 2$. Totally we have $bit(|LB(v)|)$ trees, thus, the number of unused element is $bit(|LB(v)|) - 1$. □

From Lemmas 2 and 3, we use an array of size $2n-1$ to store $2n - bit(n)$ used elements, n of which correspond to the entries in $LB(v)$. Hence, the array size is $2|LB(v)| - 1$. The next lemma is important for the search process of $KT(v)$.

LEMMA 4. *If a tree node t on $KT(v)$ covers fragment $[x..\ x + \ell - 1]$ of length ℓ with $x \ge \ell$, then $KT(v)$ must contain another tree node t' covering fragment $[x - \ell..\ x - 1]$.*

PROOF. We prove by induction. The lemma trivially holds when $|LB(v)| = 1$. Assume it holds for $1 \le |LB(v)| \le k-1$. Consider $|LB| = k$. If $k = 2^i$ for some i, then $KT(v)$ is a full binary tree, the lemma clearly holds. Otherwise, $KT(v)$ consists of a list of full binary trees $\{T_1, T_2, ..., T_s\}$. Denote the number of leaf nodes in T_i by $|T_i|$. The first $s-1$ tree $\{T_1, T_2, ..., T_{s-1}\}$ is the $KT(v)$ when $|LB(v)| = k - |T_s|$, so if t is in $\{T_1, T_2, ..., T_{s-1}\}$, the lemma holds. If t is in T_{s_1}, since the last $s-1$ trees $\{T_2, T_3, ..., T_s\}$ form $KT(v)$ when $|LB(v)| = k - |T_1|$, the lemma also holds. □

Algorithm 1 KTsearchNext:*Search Next Shortest in* $LB(v)$

Input: keyword w, $KT(v)$ index, the previous shortest entry $(u_{w_{i-1}}, d_{w_{i-1}})$ containing w
Output: the next shortest (u_{w_i}, d_{w_i}) containing w

```
1: x ← w_{i-1} − 1; ℓ ← 1; x' ← ∞; ℓ' ← ∞
2: while true do
3:   // check the fragment F = [x − ℓ + 1.. x] of length ℓ
4:   if KT(v, 2x − ℓ + 1) ∧ H(w) = 0 or ℓ = 1 then
5:     if ℓ = 1 and KT(v, 2x − ℓ + 1) ∧ H(w) ≠ 0 and u_x contains
       keyword w then
6:       return (u_x, d_x) ∈ LB(v)
7:     if x − ℓ + 1 = 0 then
8:       return NULL
9:     x ← x − ℓ
10:    if x > x' then
11:      // still in subtree search mode
12:      ℓ ← x − (x ∧ (x + 1)) + 1
13:    else
14:      if x = x' then
15:        // u_x precedes F, go back to length-doubling search mode
16:        ℓ ← ℓ'
17:      if (x ∧ (x + 1)) ≤ x − 2ℓ + 1 then
18:        // Cases 1 and 2 in length-doubling search mode
19:        ℓ ← 2ℓ
20:  else
21:    // begin subtree search mode in the next iteration, and mark that
       we enter subtree search mode in x'
22:    if x' > x − ℓ then
23:      x' ← x − ℓ          // u_{x'} precedes F
24:      ℓ' ← ℓ
25:    ℓ ← ℓ/2
```

5.2.4 *Searching the next node in* $KT(v)$ *containing* w

The label entries (u, d) in $LB(v)$ are sorted in non-ascending order of d. In Algorithm FBS, we want to retrieve entries of the form (u, d) from $LB(v)$ where u contains the query keyword w in non-decreasing distance values of d. With $KT(v)$, we can efficiently extract the required entries without scanning the whole $LB(v)$.

Denote the i-th shortest label entry in $LB(v)$ that contains w by (u_{w_i}, d_{w_i}). Given $(u_{w_{i-1}}, d_{w_{i-1}})$, the next label entry we are interested in is (u_{w_i}, d_{w_i}). In the following, assume that the trees $T_1, T_2, ...$ in $KT(v)$ are listed from left to right as shown in Figure 6. The vertices with smallest distance values are on the right. We try to find the leaf node of u_{w_i} in $[0..\ w_{i-1} - 1]$ from right to left in the two following search modes: *length-doubling search mode* and *subtree search mode*.

The general idea is to locate an ancestor of the u_{w_i} leaf node by trying lengths 1, 2, 4, 8, etc. fragments in length-doubling search mode. When we find such a possible ancestor, we search its subtrees to find u_{w_i} in the subtree search mode. If the subtrees do not contain u_{w_i}, we resume the length-doubling search mode and continue with the search. The pseudocode is given in Algorithm 1.

Length-Doubling Search Mode. More specifically, given query for keyword w, each search in $LB(v)$ begins from the length 1 fragment containing $u_{w_{i-1}-1}$, i.e., the $u_{w_{i-1}-1}$ leaf node. In general, when we search from a node r covering fragment $F = [a..\ b]$ of length ℓ, if $H(w) \wedge H(F)$ is zero, no vertex in fragment F contains keyword w, then we turn to the next candidate r_1 covering a length ℓ or 2ℓ fragment on the left. We repeat this process until u_{w_i} is found or we reach [0..0]. There are several cases when doubling the length from ℓ to 2ℓ.

Case 1 : r is a tree root in $KT(v)$. A node r_1 covering fragment $[a - 2\ell..\ a - 1]$ must exist in $KT(v)$, since the tree on the left of r must cover a fragment of length at least 2ℓ. In Figure 6, if r covers [8..11], then r_1 is [0..7]. If r=[12..12], then r_1 = [10..11]. Therefore, r_1 is the next candidate to be visited.

Case 2 : r is a left child of its parent. From Lemma 4, the node r_1 covering fragment $[a-2\ell..\ a-1]$ also exists in $KT(v)$. Therefore, the next step is to search r_1. In Figure 6, if r covers [8..9] of length 2, r_1 will be [4..7].

Case 3 : r is a right child of its parent. From Lemma 4, the nodes for fragments $f_1 = [a - \ell..\ a - 1]$ and $f_2 = [a - 3\ell..\ a - \ell - 1]$ exist in $KT(v)$. In this case, we do not double the length ℓ immediately. Instead, we visit the length ℓ fragment, f_1, first, and if the bitwise AND value is zero, we then continue to check the length 2ℓ fragment, f_2. In Figure 6, if r = [6..7], we would visit [4..5] first and then [0..3].

The if condition at Line 17 distinguishes between Cases 1,2 and Case 3. From Lemma 1, $[x \wedge (x + 1)..\ x]$ is the highest node with u_x at the right end, if $(x \wedge (x+1)) > x-2l+1$, then $[x-2l+1..\ x]$ cannot exist in the tree, which means that the current node is under Case 3, so we do not double the length.

From the 3 cases above, we can double the length of the fragment in 1 or 2 steps.We will show that in this way an ancestor of u_{w_i} can be located after checking $O(\log(w_{i-1} - w_i))$ fragments.

Subtree Search Mode. Next we consider the subtree search mode. When the bitwise AND value in a node r covering $[y..\ x]$ of length ℓ (Line 4 in Algorithm 1) is non-zero, we switch to the subtree search mode. The first step is to mark the entry into the subtree search mode at node r by assignments $x' \leftarrow x - \ell$ and $\ell' \leftarrow \ell$ (Lines 23-24). We mark that we begin subtree search mode in node $r' = r = [(x' + 1)..\ (x' + \ell')]$. The second step is to search the right subtree under r, and if we fail to find u_{w_i} in the right subtree, we turn to the left subtree under r. By the recursive search, we may reach a length 1 fragment (u_x) with non-zero bitwise AND value, in which case we can determine whether u_x contains w by a binary search of the keywords contained in u_x (Line 5).

If the bitwise AND returns zero at some node, so that a fragment $[a..\ b]$ cannot contain w, we next try to search the highest tree node containing u_{a-1} at the leaf at the right end. We reset $x = a - 1$, and search the fragment $[(x + 1)\ \wedge\ x..\ x]$, based on Lemma 1. Line 12 of Algorithm 1 ensures that the next iteration will check $KT(v, ((x + 1)\ \wedge\ x) + x)$, for fragment $[((x + 1) \wedge x)..\ x]$.

One special case is when u_x falls outside the fragment of r', $[x' + 1..\ x' + \ell']$, recorded at the entry of the subtree search mode, i.e. when $x = x'$ (at Line 14). It means that this fragment does not contain w after all. In this case, we go back to length-doubling mode and search a length $2\ell'$ or ℓ' fragment on the left of r' (the length is set via Lines 16 and 19).

Analysis of Algorithm 1 Algorithm 1 is a key function in FBS. It traverses the KT tree and checks if the given keyword may exist in the subtree under the current node by means of the hash function H. A false positive alarm occurs if the bitwise AND operation at Line 4 of Algorithm 1 returns a non-zero value when keyword w is not contained in any vertex in $[x - \ell + 1..\ x]$. Such an alarm will affect the runtime. Let p be the probability that a correct result is returned by the bitwise AND operation.

THEOREM 1. *Assume in each node of* $KT(v)$*, the false positive rate is at most* $1 - p$*, and* $p > \frac{1}{2}$*. Given* $u_{w_{i-1}}$ *and* $KT(v)$*, assuming equal probability for* $u_j = u_{w_i}$ *for all* $0 \le j < w_{i-1}$*, the expected time for searching the next* (u_{w_i}, d_{w_i}) *is* $O(\log(w_{i-1} - w_i) + \log \frac{|doc(V)|}{|V|})$.

PROOF. Let the distance between w_{i-1} and w_i be $l = w_{i-1} - w_i$. We define two functions $f(h)$ and $g(h)$. $f(h)$ is the expected

cost of searching a tree T in KT index with height h when T actually contains the query keyword w. $g(h)$ is the expected cost of searching a tree T in KT index with height h where T does not contain w. In a $KT(v)$, the l label entries between $\{u_{w_i}, d_{w_i}\}$ and $\{u_{w_{i-1}}, d_{w_{i-1}}\}$ are partitioned into $h_1 = O(\log l)$ trees, so in Algorithm 1, we search the $O(\log l)$ trees to find the w_i. The cost of Algorithm 1 is bounded by $cost_1 = \sum_{i=0}^{h_1} g(i) + f(h_1)$. We analyse $f(h)$ and $g(h)$ next to show that $cost_1$ is $O(h_1 + \log \frac{|doc(V)|}{|V|})$.

When searching a tree node containing w of height h, it is possible that w is in its left subtree or in its right subtree with 0.5 probability each. When $h = 0$, it corresponds to a single vertex u_x, so we can check whether u_x contains w by a binary search. In brief, the cost $f(h)$ is as follows.

$$f(h) = \begin{cases} 1 + \frac{f(h-1)}{2} + \frac{g(h-1)+f(h-1)}{2} & for\ h > 0 \\ O\left(\log \frac{|doc(V)|}{|V|}\right) & for\ h = 0 \end{cases}$$

The analysis of $g(h)$ is similar to $f(h)$. On a tree node of height h that does not contain w, the probability that we further check its children is $1 - p$. When $h = 0$, a binary search in $O(\log \frac{|doc(V)|}{|V|})$ time can ensure that w is not in the tree. Thus, $g(h)$ is given by

$$g(h) = \begin{cases} 1 + (1-p) \times 2 \times g(h-1) & for\ h > 0 \\ O\left(\log \frac{|doc(V)|}{|V|}\right) & for\ h = 0 \end{cases}$$

Suppose $p > \frac{1}{2}$. Let a constant $c = 2(1-p)$.

For $h \geq 0$, $g(h) = O\left(c^h \log \frac{|doc(V)|}{|V|} + \frac{1-c^h}{1-c}\right)$, and $f(h) =$

$$O\left(h + \log \frac{|doc(V)|}{|V|} + \frac{h}{1-c} + \frac{1-c^h}{1-c}\left(\frac{1}{2}\log \frac{|doc(V)|}{|V|} - \frac{1}{1-c}\right)\right)$$

That is, $g(h) = O(c^h \log \frac{|doc(V)|}{|V|})$, $f(h) = O(h + \log \frac{|doc(V)|}{|V|})$. Consider the upper bound $cost_1$ of Algorithm 1, $cost_1 = O(h_1 + \log \frac{|doc(V)|}{|V|})$. Therefore, the time complexity of Algorithm 1 is $O(\log(w_{i-1} - w_i) + \log \frac{|doc(V)|}{|V|})$. □

The probability p can be made large by limiting the number of frequent keywords and using a universal hash function [10]. A false positive affects efficiency when searching u_{w_i} given $u_{w_{i-1}}$, but such a tree node must correspond to a fragment $[x..y]$ with $w_{i-1} < x \leq y < w_i$. In real datasets, $w_i - w_{i-1}$ is usually small and the fragment $[x..y]$ typically contains a very small set of keywords. We will introduce an enhanced KT construction to further reduce this keyword set. Therefore, probability p is high in practice and Algorithm 1 performs well in our experiments.

5.2.5 KT construction

The KT construction has not been introduced yet. Inspired by the tree traversal order in KTsearchNext, we can build the tree in a similar way. The generation of the nodes in $KT(v)$ follows a right-to-left preorder traversal of the trees in $KT(v)$. Suppose we have trees $T_1, ..., T_k$ in $KT(v)$, we visit the trees in reverse order from T_k to T_1, in each tree, we visit the root first, then recursively visit the right subtree, followed by the left subtree. E.g., the nodes in Figure 6 are constructed in the following order: 24, 19, 21, 22, 20, 17, 18, 16, 7, 11, 13, 14, 12, 9, 10, 8, 3, 5, 6, 4, 1, 2, 0.

Enhanced KT Construction. The KT Construction algorithm described in the above can be further improved by some careful observations. In Algorithm 1, two cases lead to visiting a node r of fragment $[x - \ell + 1..x]$.

Case 1 : we visit $[x - \ell + 1..x]$ in length-doubling search mode. Hence, before visiting r, we must have visited either $[x + 1..x + \ell]$ or $[x + 1..x + \ell/2]$. Thus, we can be sure that u_i with $x + 1 \leq i \leq x + \ell/2$ does not contain the query keyword w.

Case 2 : we visit r in subtree search mode. Suppose that r' of fragment $[x' - \ell' + 1..x']$ with length ℓ' is the node where we begin the subtree search. We must have visited r' in length-doubling search mode and according to Case 1, we can be sure that u_i with $x' + 1 \leq i \leq x' + \ell'/2$ does not contain w. Thus, u_i with $x + 1 \leq i \leq x + \ell/2$ cannot contain w since $x \leq x'$ and $\ell \leq \ell'$.

In either case, when visiting a node r of fragment $[x - \ell + 1.. x]$, we know that u_i with $x + 1 \leq i \leq x + \ell/2$ cannot contain the query keyword w. If we have known that fragment $[x+1.. x+\ell/2]$ does not contain w, we must have visited the node r_1 of fragment $[x + 1.. x + \ell/2]$ or some ancestors of r_1, which means we can at least make sure that u_j with $x + \ell/2 + 1 \leq j \leq x + \ell/4$ also cannot contain w according to Cases 1 and 2 above. Overall, when visiting a node r of fragment $[x - \ell + 1.. x]$, it always holds that u_i with $x + 1 \leq i \leq x + \ell/2 + \ell/4 + ... + 1 = x + \ell - 1$ does not contain the query keyword w.

It is equivalent to say that if we query a keyword w' which is contained in fragment $[x + 1.. x + \ell - 1]$, we cannot visit the node r of fragment $[x - \ell + 1.. x]$ in Algorithm 1. Therefore, we can improve the hash value of r by excluding a keyword w' if w' is contained in fragment $[x+1.. x+\ell-1]$, i.e. $KT(v, 2x-\ell+1) = H(\{w|w \in u_i$ for some $i, x-\ell+1 \leq i \leq x$, and $w \notin u_j$ for every $j, x+1 \leq j \leq x+\ell-1\})$ For instance, in Figure 6, the hash value of node 11 of fragment [4..7] does not consider any keyword that appears in u_8, u_9 or u_{10}. From our experiment, the above strategy typically improves the query time by 10 to 20 %.

Algorithm 2 *KT Construction*

Input: 2-hop backward index $LB(v)$
Output: keyword-lookup tree index $KT(v)$

```
1: sort all (u, d) ∈ LB(v) in non-ascending order of d
2: let the sorted sequence be (u_i, d_i) for 0 ≤ i ≤ |LB(v)| − 1
3: KT(v) ← an array from KT(v, 0) to KT(v, 2|LB(V)| − 2)
4: t ← |LB(v)| − 1
5: s ← t ∧ (t + 1)
6: while true do
7:    KT(v, s+t) ← H({w|w ∈ u_i for some i, s ≤ i ≤ t, and w ∉ u_j
      for every j, t + 1 ≤ j ≤ 2t − s}) where {(u_i, d_i), (u_j, d_j)} ⊆
      LB(v)
8:    if s = t then
9:       if s = 0 then
10:         return KT(v)
11:      t ← s − 1
12:      s ← t ∧ (t + 1)
13:   else
14:      s ← ⌈(s + t)/2⌉
```

5.2.6 Query Evaluation

Next, we consider how to answer a k-NK query given L, LB and KT by algorithm FBS. For a query (q, w, k), we denote the i-th label entry in $L(q)$ by (x_i, d_i). For each x_i, we denote the j-th shortest label entry in $LB(x_i)$ containing keyword w by $(y^i_{w_j}, d^i_{w_j})$. The main idea in the query evaluation process is to maintain a double-ended priority queue PQ of size k containing some candidate answers, i.e. some paths $(q \to x_i \to y^i_{w_j}, d_i + d^i_{w_j})$. The key value for the PQ is the value of $d_i + d^i_{w_j}$.

Priority Queue Initialization. In the beginning, we scan the entries in $L(q)$. For each $(x_i, d_i) \in L(q)$, we extract the shortest label entry $(y^i_{w_1}, d^i_{w_1})$ in $LB(x_i)$ containing w by searching $KT(v)$. Some paths $(q \to x_i \to y^i_{w_1}, d_i + d^i_{w_1})$ are generated as candidates and pushed into the priority queue PQ. The size of PQ is at most k. When we try to insert a new candidate

Algorithm 3 *FBS Query Evaluation for k-NK*

Input: vertex q, keyword w, k, 2-hop index L, 2-hop backward index LB and keyword-lookup tree KT
Output: the top-k nearest vertices from q with keyword w

double-ended priority queue $PQ \leftarrow \emptyset$ where the shortest distance (minimum key) is d^* and the longest distance (maximum key) is $d^{\#}$
$ans \leftarrow \emptyset$
for each $(x_i, d_i) \in L(q)$ **do**
  $(y^i_{w_1}, d^i_{w_1}) \leftarrow$ KTsearchNext(w, $KT(x_i)$, $(y^i_{|LB(x_i)|}, \infty)$)
  if $(y^i_{w_1}, d^i_{w_1}) \neq \emptyset$ **then**
    InsertPQ($(q \rightarrow x_i \rightarrow y^i_{w_1}, d_i + d^i_{w_1})$)
while $|ans| < k$ and $PQ \neq \emptyset$ **do**
  pop $(q \rightarrow x_{i^*} \rightarrow y^{i^*}_{w_{j^*}}, d^*)$ from PQ
  $ans = ans \cup \{y^{i^*}_{w_{j^*}}\}$
  for each $(q \rightarrow x_i \rightarrow y^i_{w_j}, d_i + d^i_{w_j})$ linked to $y^i_{w_j} = y^{i^*}_{w_{j^*}}$ **do**
    while $j < k$ **do**
      $j \leftarrow j + 1$
      $(y^i_{w_j}, d^i_{w_j}) \leftarrow$ KTsearchNext(w, $KT(x_i)$, $(y^i_{w_{j-1}}, d^i_{w_{j-1}})$)
      if $(y^i_{w_j}, d^i_{w_j}) \neq \emptyset$ and $y^i_{w_j} \notin ans$ **then**
        InsertPQ($(q \rightarrow x_i \rightarrow y^i_{w_1}, d_i + d^i_{w_1})$)
        break
return ans

Function $InsertPQ(q \rightarrow x \rightarrow y, d)$

if $\exists (q \rightarrow x' \rightarrow y, d') \in PQ$ **then**
  if $d' > d$ **then**
    replace $(q \rightarrow x' \rightarrow y, d')$ with $(q \rightarrow x \rightarrow y, d)$
  link $(q \rightarrow x \rightarrow y, d)$ with y
else
  if $|PQ| < k - |ans|$ **then**
    $PQ = PQ \cup (q \rightarrow x \rightarrow y, d)$
  else
    if $d^{\#} > d$ **then**
      replace $(q \rightarrow x^{\#} \rightarrow y^{\#}, d^{\#})$ with $(q \rightarrow x \rightarrow y, d)$

$(q \rightarrow x_i \rightarrow y^i_{w_1}, d_i + d^i_{w_1})$ into PQ, we firstly check whether another $(q \rightarrow x' \rightarrow y^i_{w_1}, d') \in PQ$ exists. If so, we link them together in a linked list for $y^i_{w_1}$ and set the shortest distance in the linked list as the shortest distance for $y^i_{w_1}$. If such an entry does not exists and PQ is not full, we insert $(q \rightarrow x_i \rightarrow y^i_{w_1}, d_i + d^i_{w_1})$ into PQ. If PQ is full, we compare the longest path (with maximum key) in PQ with the candidate path, if the candidate is shorter, it replaces the longest path with the maximum key for PQ.

Extracting Minimum Entries from PQ. After initialization, we pop the path $(q \rightarrow x_{i^*} \rightarrow y^{i^*}_{w_{j^*}}, d^*) \in PQ$ with the minimum distance d^* from PQ one by one. Each time such an entry is popped, we update PQ by an attempt to insert the path p to the next $y^{i^*}_{w_{j^*+1}}$. The next $y^{i^*}_{w_{j^*+1}}$ entry is looked up from the index tree of $KT(x_i)$. Note that since the popped $y^{i^*}_{w_{j^*}}$ may be linked to multiple x_i, we search the next $y^i_{w_{j+1}}$ for all of them. The insertion process is similar to the PQ initialization. We stop the above process when k minimal values (answers) are popped from the priority queue or the priority queue is empty.

The invariants during the query evaluation process after the initialization of PQ are that the next best answer is the minimum key value in the priority queue, and that for each x_i in $L(q)$, all the $LB(x_i)$ entries not examined so far have key values greater than the current entry from $LB(x_i)$ that is in PQ or have values greater than the maximum value in the queue. From these invariant properties, we can show that the algorithm is correct.

Complexity Analysis. In Algorithm 3, there are $O(|L(q)|)$ label entries in $L(q)$ and $O(k|L(q)|)$ label entries in LB are inserted into and removed from PQ. We can implement the double-ended priority queue PQ by an interval heap [33]. The PQ size is k, so it takes $O(k|L(q)| \log k)$ time to maintain the PQ. Besides, we need to search $KT(x)$ up to k times to search the next $y^{i^*}_{w_{j^*}}$ for each x. Denote the distance between w_{j-1} and w_j by l_j, $l_1 = w_1$, then $\sum_{j=1}^{k} l_j \leq |LB(x)|$. From Theorem 1, the total expected time complexity of the k searches of $KT(x)$ is $O(\sum_{i=1}^{k}(\log l_i + \log |doc(V)|/|V|))$.

Note that $\sum_{i=1}^{k} \log l_i = \log \prod_{i=1}^{k} l_i \leq \log(\sum_{j=1}^{k} l_j/k)^k \leq k \log \frac{|LB(x)|}{k}$, where $|LB(x)|$ is the average size of a backward label. Overall, a k-NK query can be answered in $O(k|L(q)|(\log k + \log \frac{|LB(x)|}{k} + \log \frac{|doc(V)|}{|V|})) = O\left(k \frac{|L|}{|V|} \log\left(\frac{|L|}{|V|} \frac{|doc(V)|}{|V|}\right)\right)$ expected time. Unlike the case with FS algorithm, the complexity does not depend on the keyword frequency, $|V_w|$. From studies in [19, 2], $|L|/|V|$ is small in many real graphs and the bound will become $O(k \log(|doc(V)|/|V|)$.

5.3 Combining FS and FBS

Since FS and FBS are efficient for keywords with low and high frequencies, respectively, we adopt a hybrid approach, called **FS-FBS**; if keyword frequency is high, use FBS search, otherwise, use FS search. A simple threshold is a median point so that half the keyword occurrences are handled by FS and half by FBS. Let us analyze this threshold assuming a Zipf's distribution [38, 25]. Consider the frequency $freq(w)$ of word w and its rank $r(w)$ in a document, which means that w is the $r(w)$-th most frequent word. Zipf's law states that $freq(w) \propto \frac{1}{r(w)^\alpha}$ where $\alpha \approx 1$, or $freq(w) = cr(w)^{-\alpha}$ where c is a constant.

For a graph with $|W|$ keywords and $|doc(V)|$ keyword occurrences, suppose we set $\sqrt{|W|}$ as a threshold. The $\sqrt{|W|}$-most frequent keywords are handled by FBS algorithm, while the others are by FS. Assume that $\alpha = 1$. The sum of frequencies of the top i highest ranked words is proportional to the i-th Harmonic number, $H_i = \sum_{r=1}^{i} \frac{1}{r} = \ln i + \gamma + \epsilon_i < \ln i + 1$. The total word frequency for the top $\sqrt{|W|}$ is proportional to $\sum_{r=1}^{|W|^{1/2}} \frac{1}{r} \approx \ln(|W|^{1/2})$. Therefore, the top $\sqrt{|W|}$ keywords are expected to contain about half of the $|doc(V)|$ keyword occurrences since $\frac{\ln(|W|^{1/2})}{\ln(|W|)} = 1/2$.

Let $freq_{max}$ be the frequency of the most frequent word. Assume that the frequency of the least frequent word is 1, which is typically true in real datasets, we have $freq_{max} = c \times 1^{-\alpha}$ and $1 = c \times |W|^{-\alpha}$. Therefore, the frequency of the $\sqrt{|W|}$-th most frequent word w' is $freq(w') = \sqrt{freq_{max}}$. For instance, In our experiment, for the DBLP dataset, FS only takes $freq(w') \approx \sqrt{freq_{max}} = 800$ 2-hop distance searches in the worst case. The average frequency of a keyword handled by FS is given by $\frac{|doc(V)|}{2(|W| - \sqrt{|W|})}$, which is typically very small. E.g., for DBLP, it is given by $\frac{12842501}{2(331301-576)} \approx 19$. Thus, query processing is highly efficient.

6. EXTENSIONS

Here, we extend our work in three ways: a disk-based approach for k-NK, handling of multiple keywords, and index maintenance for dynamic updates of keywords and the graph.

6.1 Disk-based Query Evaluation

We introduce an IO-efficient algorithm for disk-based querying. The idea of this method is consistent with Figure 5. We call this algorithm **High-index-Low-index-Querying**, or **HLQ** for short.

High and Low Indices. Given a 2-hop index L, we generate an LB index as the FBS algorithm. From LB, we derive index L_w for each keyword w. For each vertex v, $L_w(v)$ is a list of label entries (u, d). If $(u, d) \in LB(v)$ and u contains keyword w, then $(u, d) \in L_w(v)$, and thus, L_w is a w-related backward index. Querying the k-th nearest vertices containing keyword w from vertex q is based on a forward search on $L(q)$ and a backward search on L_w. After L_w for each keyword w is generated, it is partitioned into the w-related **high index** and the w-related **low index**, or HI_w and LI_w for short. The partitioning is based on two vertex sets V_{HI} and V_{LI}, where $V_{HI} = \{v \in V | \forall u \notin V_{HI}, |LB(v)| \geq |LB(u)|\}$ and $V_{LI} = V - V_{HI}$. For each keyword w, $L_w(v) \subseteq HI_w$ iff $v \in V_{HI}$, and we denote each $(u, d) \in L_w(v) \subseteq HI_w$ as $(v \rightarrow u, d) \in HI_w(v) \subseteq HI_w$. Similarly, $L_w(v) \subseteq LI_w$ iff $v \in V_{LI}$, and we denote each $(u, d) \in L_w(v) \subseteq LI_w$ as $(u, d) \in LI_w(v)$. We can set different $|V_{HI}|$ to control the size of high index and low index. In our empirical study, $|V_{HI}| = |V| \times 1\%$ is a good value for the index. Thus, we obtain a small vertex set V_{HI} that covers a large proportion of label entries, while HI_w is the keyword w-related backward index covered by the small vertex set.

For each $v \in V_{LI}$ and keyword w, all the $(u, d) \in LI_w(v)$ are stored in the non-descending order of dist d as a list of pairs (u, d) on a disk. We can locate the beginning of the list given v and w by a disk-based $B+$ tree. However, the order of label entries in HI stored on the disk is quite different. Consider the label entries in $HI_w(v)$, that is, $\{(v \rightarrow u_1, d_1), (v \rightarrow u_2, d_2), ..., (v \rightarrow u_s, d_s)\} \subseteq HI_w$. Denote the rank of entry $(v \rightarrow u_i, d_i)$ by $r_v(u_i, d_i)$, implying that distance d_i is the $r_v(u_i, d_i)$-th shortest distance among the s entries of $HI_w(v)$. For each HI_w, all $(v \rightarrow u, d) \in HI_w$ are stored in the non-descending order of $r_v(u, d)$ as a list of tuple (v, u, d) on the disk.

Query Evaluation. Given a query (q, w, k), we use a forward search on $L(q)$, and a backward search on HI_w and LI_w in the disk-based query evaluation. There are mainly 3 steps as follows.

Step 1: Read all the label entries in $L(q)$ into memory. Initialize a max-heap of size k for the answer set.

Step 2: Create an empty set H' in memory. For each $(x, d) \in L(q)$, if $x \in V_{HI}$, put (x, d) into H' for Step 3. Otherwise, if $x \in V_{LI}$, read k label entries $(y, d') \in LI_w(x)$ from disk. Meanwhile, try to update the answer set by the path $(q \rightarrow x, d) + (x \rightarrow y', d')$ with distance $d + d'$. When $d + d'$ is not shorter than the current k-th shortest result in the answer set, we stop reading $LI_w(x)$. Then, the next x in $(x, d) \in L(q)$ is checked iteratively.

Step 3: Read the high label entries $(x_h \rightarrow y_h, d_h) \in HI_w$ from disk. If there exists an entry $(x_h, d) \in H'$ from Step 2, try to update the answer set by the path $(q \rightarrow x_h, d) + (x_h \rightarrow y_h, d_h)$ with distance $d + d_h$. When an x_h is encountered such that the k highest ranked label entries in $HI_w(x_h)$, i.e. $\{(x_h \rightarrow u_1, d_1), (x_h \rightarrow u_2, d_2), ..., (x_h \rightarrow u_k, d_k)\}$, have been read, the algorithm stops.

Partitioning the second hop index L_w into the HI and LI indices is based on the high-coverage property of HI in the 2-hop labeling [19, 2, 1]. A large proportion of shortest paths $(q \rightarrow x \rightarrow y)$ go through some $x \in HI$. Finding the first hops $(q \rightarrow x)$ is easy by reading $L(q)$, and we can discover most answers by getting their second hops $(x \rightarrow y)$ with $x \in V_{HI}$ by a partial scan of HI_w. The remaining paths going through $x \in V_{LI}$ should be rare, and hence, their second hops in $LI_w(x)$ can be retrieved with a small I/O cost.

Complexity Analysis. If a vertex u contains r different keywords $w_1, w_2, ..., w_r$, each label entry (u, d) will appear r times in L_{w_1}, L_{w_2}, ..., L_{w_r}, and is partitioned into either HI or LI. Thus, on average, we require $O(|doc(V)|/|W|)$ times the storage size of the 2-hop label index, totally $O(\frac{|doc(V)||L|}{|W|})$ space. Let B be the disk block size. We can show that the overall expected I/O complexity is given by $O(\frac{|L|}{|V|}\lceil\frac{k}{B}\rceil + \lceil\frac{k|V_{HI}|}{B}\rceil)$.

6.2 Handling Multiple Keywords

For multiple keyword queries, we consider two possibilities in the query requirement: a conjunction of keywords $(w_1 \bigwedge w_2 ... \bigwedge w_p)$ and a disjunction of keywords $(w_1 \bigvee w_2 ... \bigvee w_p)$.

We discuss the extension for disjunctive multiple keyword querying in the following. Querying $(q, w_1 \bigvee w_2 ... \bigvee w_p, k)$ returns the nearest k vertices of q, each of which contains w_1, w_2, ..., or w_p. The index construction of disjunctive query is exactly the same as that of the index for single keyword queries. We begin with FBS to handle the keywords $\{w'_1, ..., w'_h\} \in \{w_1, ..., w_p\}$ with high frequencies that are indexed in KT and LB index. We initialize a priority queue as in the single-keyword case introduced in Section 3, and update the priority queue by searching the KT index and the LB index. When searching a node (x, y) on tree $KT(v)$, we continue the length-doubling mode when $H(\{w'_1, ..., w'_h\}) \wedge KT(v, x + y)$ is zero; otherwise, we switch to the subtree search mode. The recursive search ends when a leaf node containing one of the keywords is reported. Meanwhile, the same operations as in single keyword case are conducted to maintain the priority queue. The remaining keywords with low frequencies $\{w'_{h+1}, ..., w'_p\} \in \{w_1, ..., w_p\}$ can be handled by FS. The vertices in the candidate set $\bigcup_{h<j\leq p} V_{w'_j}$ are checked to update the answers from FBS.

For conjunctive querying, we follow a similar approach. We use FS when keyword with lowest frequencies is infrequent; otherwise, we run FBS similarly except that the subtree search mode only begins when $H(\{w_1, ..., w_p\}) \wedge KT(v, x + y) = H(\{w_1, ..., w_p\})$.

6.3 Handling Dynamic Updates

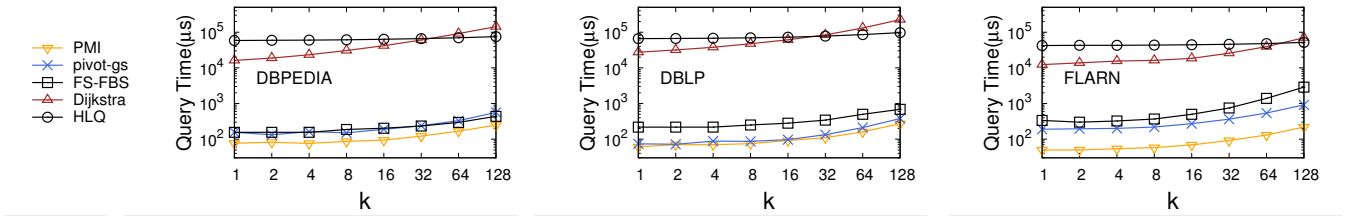
Our solutions support keyword updates naturally. Maintaining the set V_w is sufficient for updating in algorithm FS. When a keyword w is inserted into or removed from a vertex v, we simply update v in the set V_w, so that when running FS, the vertices containing w can be efficiently identified. In algorithm FBS, we maintain word updates by updating the KT index. When a keyword w is added to or deleted from a vertex v, we update $KT(u)$ for each $\{u, d\} \in L(v)$. The path in $KT(u)$ from the root to the leaf node containing v should be updated by recomputing the hash values.

When the graph structure is updated, we may adopt existing methods [37, 3, 6] to update the 2-hop structure. After label entries are inserted into or removed from the 2-hop index, the corresponding entries in the LB index and affected trees in the KT index are updated. If a new label entry (u, d) is inserted into $L(v)$, the entry (v, d) will be included in $LB(u)$. The KT index can adopt lazy updates by sharing a leaf node by multiple vertices for insertion in $LB(v)$. For lazy deletions in $LB(v)$, we may have leaf nodes with no vertex. More studies will be needed on the effect of updates.

7. EXPERIMENTS

We compare the proposed algorithms, i.e. memory-based **FS-FBS** and disk-based **HLQ**, with the baseline algorithm **Dijkstra**, and the best-known approximation algorithms, **PMI** algorithm [4] and **pivot-gs** algorithm [26]. Both PMI and pivot-gs were implemented by the authors of [26]. Algorithm Dijkstra runs a Dijkstra search in memory to find the vertices containing the query keyword. The experiments are run on a Linux machine with 3.4GHz CPU, 32GB memory, 1TB 7200 RPM hard-disk, GNU C++.

Datasets. We use three real datasets, DBPEDIA, DBLP, and Florida road network FLARN, as listed in Table 2. DBPEDIA is a RDF



	k	1	2	4	8	16	32	64	128
HR	PMI	0.47	0.52	0.54	0.56	0.56	0.57	0.61	0.65
	pivot-gs	0.87	0.89	0.89	0.88	0.85	0.83	0.83	0.84
S-ρ	PMI	1.00	0.72	0.49	0.37	0.29	0.32	0.32	0.33
	pivot-gs	1.00	0.97	0.86	0.75	0.67	0.60	0.55	0.53

(a) DBPEDA

	k	1	2	4	8	16	32	64	128
HR	PMI	0.30	0.31	0.32	0.33	0.34	0.38	0.41	0.46
	pivot-gs	0.68	0.68	0.70	0.70	0.70	0.72	0.74	0.78
S-ρ	PMI	1.00	0.54	0.43	0.36	0.28	0.24	0.20	0.20
	pivot-gs	1.00	1.00	0.92	0.88	0.83	0.79	0.74	0.71

(b) DBLP

	k	1	2	4	8	16	32	64	128
HR	PMI	0.72	0.75	0.75	0.75	0.75	0.76	0.76	0.77
	pivot-gs	0.95	0.92	0.91	0.89	0.88	0.88	0.88	0.87
S-ρ	PMI	1.00	0.66	0.76	0.78	0.81	0.81	0.82	0.84
	pivot-gs	1.00	0.92	0.92	0.93	0.93	0.92	0.93	0.93

(c) FLARN

Figure 8: Comparison of Average Query Time and Accuracy by Hit Rate(HR) and Spearman's rho(S-ρ) [HR(FS-FBS) = S-ρ(FS-FBS) $\equiv$ 1]

	$\|V\|$	$\|E\|$	$\|doc(V)\|$	$\|W\|$
DBPEDIA	1,121,413	4,529,420	8,312,816	541,376
DBLP	1,695,469	4,726,801	12,842,501	331,301
FLARN	1,070,376	1,356,399	6,966,665	2,730

Table 2: Real Dataset Statistics

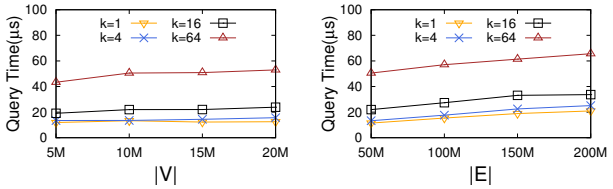


Figure 9: Scalability of FS-FBS in synthetic datasets

graph crawled from Wikipedia[2]. If there is a page link between the Wikipedia pages of two persons, their vertices are connected by an unweighted edge. The DBLP network[3] contains 1,060,763 publications, 631,589 authors, and 3,117 conferences or journals as vertices. As in [26], we assign the weight of an edge (u, v) by $\log_2 deg(u) + \log_2 deg(v)$ where $deg(u)$ is the degree of vertex u. In the FLARN network[4], the keywords of vertices are obtained from the OpenStreeMap project[5] with a bounding box. From this source, only 7127 vertices contain keywords. Following [26], we have assigned a random number (between 0 and 4) of keywords to the vertices with no keywords and the stop words are removed.

We generated 1000 queries for each dataset. In a query (q, w, k), the vertex q is randomly picked from the vertex set, and the keyword w is selected following the keyword distribution in the dataset. k is varied from 1 to 128.

Comparison with Approximation Algorithms. We compare FS-FBS, the approximation algorithms PMI [4] and pivot-gs [26], Dijkstra's algorithm and the disk-based algorithm HLQ in Figure 8 showing their query times and the accuracies of the approximation algorithms. The query times of the Dijkstra algorithm and the disk-based HLQ algorithm are also reported. Since the coding of PMI and pivot-gs provided by the authors of [26] can only run on Windows platform, and requires 128GB memory, we compare our exact algorithm FS-FBS with the approximation algorithms in a Windows workstation with 3GHz CPU and 128GB memory.

The accuracies of the approximation algorithms are evaluated by *hit rate* and *Spearman's rho* as in [26]. Hit rate gives the percentage of answers that are among the top-k. Spearman's rho [31] measures the correlation coefficient of the ranks of the reported result and the exact result. Note that the hit rate and Spearman's rho of our algorithm FS-FBS are always 1, which are $\approx 80\%$ better than those of PMI and improve over 20% of those of pivot-gs. Meanwhile, the query time of FS-FBS is always small without any loss of accuracy. Furthermore, in DBPEDIA, where the accuracies of the two approximation algorithms are low, FS-FBS is even faster than the state-of-the-art algorithm pivot-gs. FS-FBS not only achieves good performance in querying, but also has very a competitive indexing cost, as shown by the indexing times and index sizes in Figure 13.

Figure 8 also shows how the query time varies with k. With the growth of k, the query time of FS-FBS or HLQ only increases very slowly. Even when $k = 128$, FS-FBS can finish in milliseconds. This shows the efficiency of the FS-FBS algorithm. Although the disk-based algorithm HLQ is not as fast as the memory-based solutions, it does not require keeping the entire index in memory, instead, only loading a piece of the index is sufficient for querying.

Results on Scalability of FS-FBS. Two sets of experiments are conducted to test the scalability of our algorithm. We first randomly sample subgraphs with various sizes from DBPEDIA dataset to show the differences of querying performances of FS-FBS. The query time and the sizes of sampled subgraphs can be found in Figure 10, whose caption shows the average hit rate and Spearman's rho. PMI algorithm is inaccurate, while pivot-gs is almost dominated by FS-FBS with exact answers and faster query time.

The second set of experiments is based on large synthetic graphs generated by *Generalized Linear Preference(GLP)* model [7] following [14, 19]. We randomly assign keywords into the graphs following Zipf's law by setting $\alpha = 1.4$ according to linguistics studies on English language [25]. The default setting of the graph is $(|V| = 10M, |E| = 50M, |doc(V)| = 50M, |W| = 1M)$. The query times of varying $|V|$ and $|E|$ are reported in Figure 9. Due to the huge memory consumption, neither PMI nor pivot-gs can build the index for the large graphs in the 128GB RAM machine.

Effects of Keyword Frequencies. We verify the performance of our memory-based algorithm when querying keywords with different frequencies in Figure 11 (a). The cases only using FS or FBS are also compared to show the improvement of adopting the hybrid approach FS-FBS in Figures 11 (b) and (c). Each figure shows a trend of 1000 top-32 queries. When querying keywords with low frequencies on the left side of Figure 11(a), the query time steadily increases with the frequency, since we use FS to handle them. When the frequency is high on the right side of Figure 11(b), the query is processed by FBS, so the query time is quite sta-

[2]http://dbpedia.org
[3]http://www.informatik.uni-trier.de/~ley/db
[4]http://www.dis.uniroma1.it/challenge9/download.shtml
[5]http://wiki.openstreetmap.org/

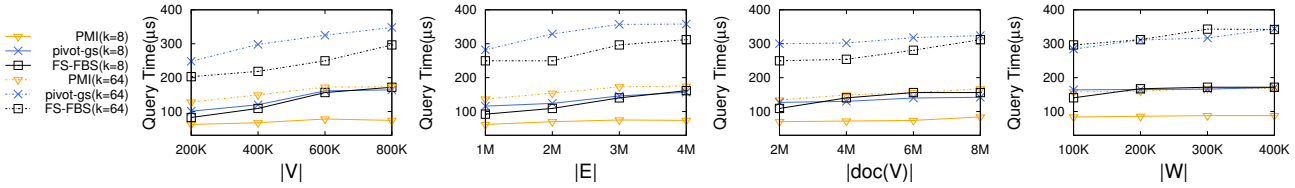
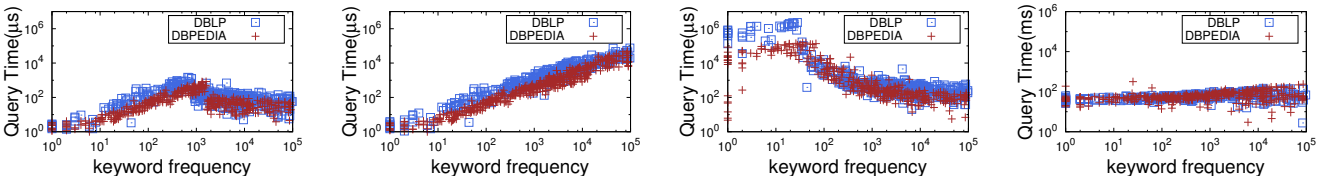


Figure 10: Scalability of Query Time (DBPEDIA) [HR(PMI) ≈ 0.57, HR(pivot-gs) ≈ 0.86, S-ρ(PMI) ≈ 0.49, S-ρ(pivot-gs) ≈ 0.77]



(a) FS-FBS (b) FS (c) FBS (d) HLQ

Figure 11: Query Time varying Keyword Frequencies

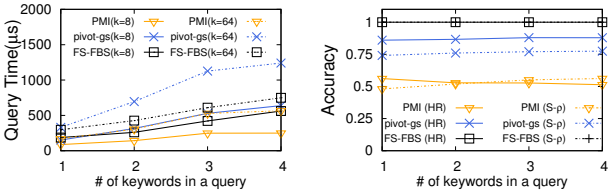


Figure 12: Multi-Keyword Querying in DBPEDIA

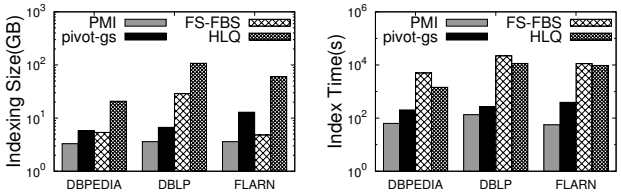


Figure 13: Indexing Statistics

ble when the frequency increases. FS or FBS alone fails to perform as fast as FS-FBS. With FS, querying keywords with high frequencies is slow. The FBS performance can be explained by the time complexity derived in Section 5.2. When the frequency is high, the keyword is more likely to be contained in a label entry, so the distance l_i between two consecutive label entries with the keyword is smaller. FBS cannot handle keywords with low frequencies as fast as FS. Furthermore, when the frequency is high, FBS is slower than FS-FBS since it maintains a larger keyword set in the index. The experimental results are consistent with our analysis in Section 5. We also show that the query time of HLQ is steady in Figure 11(d) since the query time is dominated by disk I/O for querying.

Results on Multiple Keyword Querying. Results of querying DBPEDIA with multiple disjunctive keywords are shown in Figure 12. The index for multiple keyword querying is exactly the same as the one in a single keyword case. We vary the number of keywords from 1 to 4. The selection of keywords follows the keyword distribution as in the single-keyword querying. FS-FBS dominates pivot-gs in the query time since FS-FBS can combine the keywords in one single search of the KT-index, while pivot-gs involves m searches for m keywords. When doubling the number of keywords, the query time of FS-FBS grows by only 60 % on average, compared with nearly 90% longer time for PMI and pivot-gs. The results for DBLP and FLARN are similar in the trends.

Results on Indexing. We adopt the **HopDB** algorithm [19] and **CH** algorithm [15] to construct the 2-hop index. Given a 2-hop index, we construct the memory-based index for FS-FBS and the disk-based index for HLQ. The results on the indexing time and the index size are shown in Figure 13. The 2-hop indexing time is included in the indexing time of memory-based algorithm FS-FBS and disk-based algorithm HLQ. The memory-based index size is small enough to fit in memory for efficiency querying. The disk-based index is a few times larger, but only loading a few pages rather than the entire index into memory is sufficient for querying. In two of the three datasets, the index size of FS-FBS is smaller than the best-known approximation algorithm pivot-gs, and only slightly larger than the approximation algorithm PMI which is not accurate, as shown in Figure 8. The results show that we can handle graphs with millions of vertices and edges on a commodity machine with acceptable indexing time and index size.

In the memory based algorithm FS-FBS, a frequency threshold is set when the FS algorithm is combined with the FBS algorithm. Other than the $\sqrt{|W|}$ threshold from Section 5.3, we also consider a few simple choices, including the median, lower quartile and upper quartile where FBS will handle half, a quarter, and 3 quarters of the keyword occurrences, respectively. For each threshold, we measure the average query time, and select the best threshold. The selected thresholds for DBPEDIA, DBLP, and FLARN are 618, 2221, and 1556, respectively. The time for threshold selection has been included in the indexing time in Figure 13. For HLQ, we select the threshold of $|V_{HI}| = |V| \times 1\%$ for the boundary between HI and LI. Varying the threshold by up to a factor of 10 has little impact on the querying time.

8. RELATED WORK

As discussed in Section 3, existing works that are exactly related to the k-NK problem are reported in [32, 4, 26]. There are other keyword search problems that are of some different characteristics. The general idea of keyword search is to find a subgraph in a given graph that contains the query keywords. The subgraph can be of the form of a tree in some cases. BANKS in [5] converts a relational database into a graph and answers to keyword queries are directed subtrees in the graph. Given a directed graph, the keyword search in [16] returns top ranked subtrees in the graph that cover the query keywords. Blinks [18] also considers directed graph and given a keyword query, an answer is a subtree in the graph that covers the keywords and the root of the subtree can reach all the keywords. Top-k results are top k subtrees with different roots. The graph type of r-clique is introduced in [21] as the form of expected answers. An r-clique is a set of vertices in the given graph which covers

the given query keywords and the distance between any pair of the vertices in this set is no longer than r. Both exact and approximate algorithms have been proposed in [21]. Querying the neighbors of a vertex in a compressed social network is considered in [24].

The methodology of 2-hop labeling is proposed in [12] and this technique has been found very useful for shortest path problems in road networks and in other networks. The problem of P2P (Point to Point) distance querying has been well studied for road networks. Some previous works include [1, 28, 15, 27, 29]. For other networks, P2P distance querying has been considered by numerous works such as [30, 17, 11, 12, 20, 34, 14, 2, 19]. Some of these works adopt 2-hop labeling techniques, including [12, 20, 14, 2, 19]. The technique of storing a tree in an array is adopted in other data structures such as the binary heap [35] and the Fenwick tree [13]. However, to the best of our knowledge, no existing structure supports a space-effective scheme with time-efficient retrieval operations for our application as the proposed KT tree.

Keyword search in spatial databases has been of great interest and a lot of interesting results have been produced, some recent works include [9, 36, 23, 8, 22]. However, since spatial data are based on spatial coordinates, the techniques in these work cannot be applied to general graphs as in our k-NK problem.

9. CONCLUSION

In this paper, we study the problem of top-k keyword search in large networks, which is useful for different applications. While existing solutions provide only approximate answers with large error bounds, we propose the first exact algorithms for this problem. We have designed algorithms for in-memory querying if the index can reside in memory. We also propose a disk-based algorithm for querying from an index on disk. We have conducted experiments in three large real networks and demonstrated that our methods are highly efficient. The in-memory querying can be finished in milliseconds, while the disk-based querying takes no more than a small fraction of a second.

ACKNOWLEDGEMENTS: We thank the authors of [26] for sharing their datasets, and kind help with their source code. We thank the anonymous reviewers for their very helpful comments and suggestions. We thank the CSE department of CUHK and Siu-Hang Or for the use of the workstation in the experiments. This research was supported by the RGC GRF research grant 412313 Proj_id 2150758 of Hong Kong.

10. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, pages 230–241, 2011.

[2] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, 2013.

[3] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*, pages 237–248, 2014.

[4] B. Bahmani and A. Goel. Bringing order to social search. In *WWW*, 2012.

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

[6] R. Bramandia, B. Choi, and W. K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *17th WWW*, pages 845–854, 2008.

[7] T. Bu and D. Towsley. On distinguishing between internet power law topology generators. In *INFOCOM*, pages 638–647. IEEE, 2002.

[8] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware optimal route search. *VLDB*, 5(11):1136–1147, 2012.

[9] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384. ACM, 2011.

[10] J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *9th STOC*, pages 106–112. ACM, 1977.

[11] L. Chang, J. Yu, L. Qin, H. Cheng, and M. Qiao. The exact distance to destination in undirected world. *The VLDB Journal*, 2012.

[12] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal of Computing*, 32(5):1338–1355, 2003.

[13] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327Ű336, 1994.

[14] A. Fu, H. Wu, J. Cheng, and R. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. In *PVLDB*, volume 6, April 2013.

[15] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[16] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, pages 927–940, 2008.

[17] A. Gubichev, S. J. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*, pages 499–508, 2010.

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

[19] 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. 2014.

[20] R. Jin, N. Ruan, Y. Xiang, and V. E. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD Conference*, pages 445–456, 2012.

[21] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. In *PVLDB*, pages 681–692, 2011.

[22] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu. Collective spatial keyword queries:a distance owner-driven approach. In *SIGMOD*, 2013.

[23] J. Lu, Y. Lu, and G. Cong. Reverse spatial and textual k nearest neighbor search. In *SIGMOD*, pages 349–360. ACM, 2011.

[24] H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *KDD*, pages 533–541, 2010.

[25] S. T. Piantadosi. Zipf's word frequency law in natural language: A critical review and future directions. *Psychonomic bulletin & review*, 21(5):1112–1130, 2014.

[26] M. Qiao, H. Cheng, J. Yu, and W. Tian. Top-k nearest keyword search on large graphs. In *VLDB*, 2013.

[27] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, 2008.

[28] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, pages 568–579, 2005.

[29] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.

[30] A. Sarma, S. Gollapudi, M. Najork, and R. Panigraph. A sketch-based distance oracle for web-scale graphs. In *WSDM*, pages 401–410, 2010.

[31] C. Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904.

[32] Y. Tao, S. Papadopoulos, C. Sheng, and K. Stefanidis. Nearest keyword search in xml documents. In *SIGMOD*, 2011.

[33] J. van Leeuwen and D. Wood. Interval heaps. *The Computer Journal*, 36(3):209–216, 1993.

[34] F. Wei. Tedi: efficient shortest path query answering on graphs. In *SIGMOD Conference*, pages 99–110, 2010.

[35] J. W. J. Williams. Algorithm-232-heapsort, 1964.

[36] D. Wu, M. Yiu, G. Cong, and C. Jensen. Joint top-k spatial keyword query processing. *TKDE*, 2011.

[37] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*, pages 1323–1334. ACM, 2014.

[38] G. K. Zipf. *Human behavior and the principle of least effort.* addison-wesley press, 1949.