The VLDB Journal
DOI 10.1007/s00778-017-0464-7

REGULAR PAPER

I/O-efficient algorithms for top-k nearest keyword search in massive graphs

Qiankun Zhu[1] · **Hong Cheng**[1] · **Xin Huang**[2]

Received: 20 July 2016 / Revised: 30 March 2017 / Accepted: 11 May 2017

Abstract Networks emerging nowadays usually have labels or textual content on the nodes. We model such commonly seen network as an undirected graph G, in which each node is attached with zero or more keywords, and each edge is assigned with a length. On such networks, a novel and useful query is called *top-k nearest keyword* (k-NK) search. Given a query node q in G and a keyword λ, a k-NK query searches k nodes which contain λ and are nearest to q. The k-NK problem has been studied recently in the literature. But most existing solutions assume that the graph as well as the constructed index can fit entirely in memory. As a result, they cannot be applied directly to very large-scale networks which are commonly found in practice, but cannot fit in memory. In this work, we design an I/O-efficient solution, which uses a compact disk index to answer a k-NK query with constant I/Os. The key to an accurate k-NK result is a precise shortest distance estimation in a graph. In our solution, we follow our previous work Qiao et al. (PVLDB 6:901–912, 2013) which uses the shortest path tree as an approximate representation of a graph and uses the tree distance between two nodes as an accurate estimation of the shortest distance between them on a graph. With such representation, the original k-NK query on a graph can be reduced to answering the query on a set of trees and then assembling the results obtained from the trees. We exploit a compact tree-based index and study how to lay out the index to disk. We design a novel technique which decomposes the index tree into paths and subtrees and stores them in disk. Our theoretical analysis shows that the disk-based index is small in size and supports constant query I/Os. Extensive experimental study on massive trees and graphs with billions of edges and keywords verifies our theoretical findings and demonstrates the superiority of our method over the state-of-the-art methods in the literature.

Keywords I/O-efficient algorithms · Nearest keywords search · Top-k · Massive graphs

✉ Xin Huang
xhuang@ieee.org

Qiankun Zhu
qkzhu@se.cuhk.edu.hk

Hong Cheng
hcheng@se.cuhk.edu.hk

[1] Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, Sha Tin, Hong Kong

[2] School of Data Science, Fudan University, Shanghai, China

1 Introduction

Many real-world networks nowadays have keywords associated with nodes. Such keywords can represent properties of a node, e.g., profile of a user in a social network, keywords of a paper in a bibliographic network, and name or category of a location in a road network. We model such networks as an undirected weighted graph, in which each node is attached with zero or more keywords, and each edge is assigned with a length. We study the problem of top-k nearest keyword (k-NK) search on such a graph. A k-NK query is in the form of $Q = (q, \lambda, k)$, where q is the query node, λ is a keyword and k is a positive integer. It searches k nodes that carry λ and are nearest to q. Different from a large body of research on k-nearest neighbor (k-NN) search on spatial networks [1–3], we define G as a general graph without coordinates. Thus, our solution can apply to a wide range of networks.

Published online: 29 May 2017

1.1 Motivation

k-NK is an important and useful query in graphs, as personalized search based on graph structure and textual content has become increasingly popular. k-NK can be used as a standalone query, as well as a building block for tackling graph pattern matching problems with both structural and textual constraints. Below we describe two application scenarios of the k-NK query.

In a social network, a person looks for 10 people who have a certain skill, e.g., *Java*, to complete a task. Intuitively, if the 10 people are close to the person in their social relationship, they can work together more effectively. Thus, the problem is to find 10 people who know *Java* and are nearest to the person who serves as the coordinator. It can be answered by a k-NK query.

As another example, in a road network where locations are associated with keywords, people can use a k-NK query to search for certain targets, e.g., *hotels, restaurants*, that are nearest to their current location as a list of candidates to choose from.

Real-world applications that are similar to the k-NK query include Facebook Graph Search[1] and keyword search on Google Maps.[2] Facebook Graph Search is a semantic search engine that gives answers to user natural language queries from a user's friends and second degree connections. With the search scope limited within 2 hops of a user's neighborhood, the indexing and search mechanisms of Facebook Graph Search are greatly simplified. In contrast, we do not impose this restriction on the search scope for the k-NK query, and we return top-k nearest answers from the whole graph with the support of our novel index. It is a non-trivial task for us to design a compact index to index all matching keywords in the whole graph to support efficient query processing. Furthermore, when there is no matching answer from a user's friends and second degree connections, Facebook Graph Search may fail to return any result, while our solution may still find answers beyond the 2-hop vicinity of the query user. Keyword search on Google Maps is formulated as a search problem on spatial network with coordinates (i.e., latitude and longitude), where the location coordinates are used for distance calculation, and spatial indices (such as network Voronoi polygons [1], shortest path quadtree [2] or path-distance oracle [3]) can be leveraged in this setting. In contrast, we assume the input is a general graph, which can be weighted or unweighted, but has no coordinates, for processing a k-NK query. Without coordinates in our setting, Google Maps cannot be applied to solve the k-NK query.

The k-NK query has been studied recently in the literature [4–6]. Bahmani and Goel [4] designed a partitioned multi-indexing (PMI) scheme, which uses an inverted index of keywords to answer k-NK queries approximately. Qiao et al. [5] proposed a shortest path tree-based index to answer k-NK queries approximately. Jiang et al. [6] designed a 2-hop labeling index to find exact answers to the k-NK query. These methods focus on the memory setting, that is, they assume that the graph as well as the constructed index can fit entirely in memory. Although [6] provides a disk-based solution when the index does not fit in the main memory, our experiments show that their disk-based solution cannot really scale to a real-world Twitter network with 41 million nodes and 1.4 billion edges.

In this paper, we study how to process k-NK queries on such massive graphs. We use a semi-external memory model in which the graph nodes can fit in memory, but the edges and keywords cannot. In this computational model, we design a disk resident index which can support I/O-efficient query processing and is compact in size.

Shortest path computation is a key operation in answering a k-NK query, but also a costly operation in a massive graph. To speed up the calculation, we follow our previous work [5] which uses a set of spanning trees as an approximate representation of a graph and uses the shortest distance in trees as an approximation of the shortest distance in a graph. With such representation, the original k-NK query on a graph can be reduced to answering the query on a set of trees and then assembling the results obtained from the trees. Thus, we focus on building the index for k-NK query on a tree and study how to organize the index to disk that is both query I/O efficient and space efficient. Specifically, we design a novel technique which decomposes the index tree into paths and subtrees and stores them in disk. The disk-based index is small in size and answers a query in optimal I/O cost.

1.2 Contributions

This work is an extension of [5] to a disk-based index, and our main contributions are summarized as follows.

1. We study the k-NK search problem on massive graphs and propose an I/O-efficient solution in a semi-external memory model.
2. We propose a novel blocking technique which decomposes the index tree into paths and subtrees and stores them in disk. Our disk index is compact and supports constant query I/O costs under a reasonable assumption. For the case beyond our assumption, we extend our blocking technique and formally prove that our proposed technique can still answer a query in optimal I/O cost. We also compare our paths + subtrees blocking technique with other alternatives analytically in terms of their space complexity and query I/O complexity and show that our technique is superior.

[1] https://en.wikipedia.org/wiki/Facebook_Graph_Search.

[2] https://maps.google.com/.

3. We discuss how to incrementally update the index given frequent keyword insertions. We adopt a batch update mode and construct a separate incremental index to index the inserted keywords. To process a k-NK query, we use both the original index and the incremental index and consolidate the answers retrieved from both indices. The effectiveness of the incremental update mechanism is evaluated and confirmed in the experiment.
4. Experimental results show that our approach is both I/O efficient and space efficient in processing k-NK query on trees and graphs. Our method consistently outperforms a disk implementation of our previous k-NK solution [5] in terms of query time and I/O, result quality, index size and index construction time. In addition, our method is around 10 times faster than the disk-based solution HLQ [6] in query processing and 13.26 times faster in index construction, and our index size is 54.90 times smaller than that of HLQ on a small-scale Twitter network (denoted as SubTwitter). Furthermore when tested on the large-scale Twitter network with 1.4 billion edges, HLQ fails to complete the index construction in 100 h with explosion in index size.

1.3 Roadmap

The rest of the paper is organized as follows. Section 2 defines the k-NK problem and describes the semi-external memory model. We introduce the compact tree index structure borrowed from [5] in Sect. 3. Then, we propose how to lay out the index to disk in Sect. 4. Section 5 discusses how to process k-NK queries on a graph based on the solution on tree. We study how to maintain the index to support frequent keyword insertions in Sect. 6. Section 7 presents the experimental results on massive trees and graphs. Section 8 discusses related work, and Sect. 9 concludes the paper.

2 Problem definition

In this section, we formally define the problem of *top-k nearest keyword* (k-NK) search in a massive graph. Then, we describe the computation model and give an overview of our solution.

2.1 Preliminary concepts

We model a weighted undirected graph as $G(V, E)$, where $V(G)$ and $E(G)$ represent the vertex set and edge set of G, respectively. We use V and E to denote $V(G)$ and $E(G)$ if the context is obvious. Each edge $(u, v) \in E$ has a positive length, denoted as $\mathsf{length}(u, v)$. A path $p = (v_1, v_2, \ldots, v_l)$ is a sequence of nodes in V such that for each v_i $(1 \leq i < l)$, $(v_i, v_{i+1}) \in E$. The length of a path is the total length of all edges on the path. For any two nodes $u \in V$ and $v \in V$, the distance of u and v on G, $\mathsf{dist}(u, v)$, is the minimum length of all paths from u to v in G. Each node $v \in V$ contains a set of zero or more keywords which is denoted as $\mathsf{doc}(v)$. The union of keywords for all nodes in G is denoted as $\mathsf{doc}(V)$. Note that $\mathsf{doc}(V)$ is a multi-set and $|\mathsf{doc}(V)| = \sum_{v \in V} |\mathsf{doc}(v)|$. We use $V_\lambda \subseteq V$ to denote the set of nodes carrying keyword λ in V. We follow [5] to define the k-NK query in the same representation.

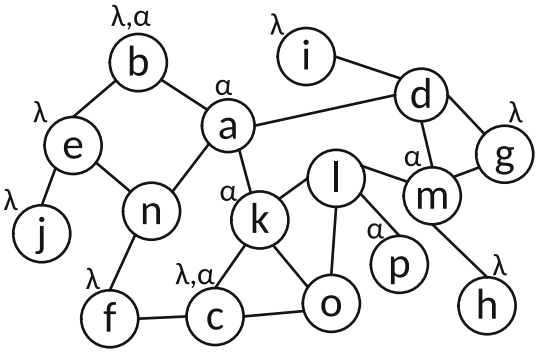

Fig. 1 A graph G with keywords λ and α

Definition 1 Given a graph $G(V, E)$, a top-k nearest keyword (k-NK) query is a triple $Q = (q, \lambda, k)$, where $q \in V$ is a query node in G, λ is a keyword and k is a positive integer. Given a query Q, a node $v \in V$ is a keyword node w.r.t. Q if v contains keyword λ, i.e., $v \in V_\lambda$. The result is a set of k keyword nodes, denoted as $R = \{v_1, v_2, \ldots, v_k\} \subseteq V_\lambda$, and there does not exist a node $u \in V_\lambda \setminus R$ such that $\mathsf{dist}(q, u) < \max_{v \in R} \mathsf{dist}(q, v)$. To further report the distance in the top-k result, we can use the form $R = \{v_1 \colon \mathsf{dist}(q, v_1), v_2 \colon \mathsf{dist}(q, v_2), \ldots, v_k \colon \mathsf{dist}(q, v_k)\}$.

Example 1 Figure 1 shows a graph G with keyword λ and keyword α. Suppose that the length of each edge is 1. Given a k-NK query $Q = (l, \lambda, 3)$, we need to find 3 nodes which carry keyword λ and are nearest to l. The result is $R = \{c \colon 2, g \colon 2, h \colon 2\}$.

In this paper, we perform exact match of keywords without considering the semantic similarity between them, which is the same as done by previous studies [4–6]. For the ease of presentation, we focus on the k-NK query which contains only one keyword. Please note that a query containing multiple keywords with AND and OR semantics can be easily handled according to [5]. Without loss of generality, we assume that the length of each edge is 1 in the rest of the paper for the ease of presentation, but our method can be applied to both weighted and unweighted graphs.

2.2 Computation model

We study the problem in the *semi-external memory* model [7,8]. M is the main memory size, and B is the disk block size, where $1 \ll B \leq \frac{M}{2}$. An I/O either reads a block of data from disk to memory or, conversely, writes a piece of data of size B in memory to a disk block. In particular, we define $scan(N) = \Theta(\frac{N}{B})$, where N is the amount of data being read from or written to disk in sequential order. However, accessing N data items at random costs $O(N)$ I/Os in the worst case. For a massive graph $G(V, E)$, the semi-external memory model [7] assumes that the internal memory can hold $c \cdot |V|$ data, for a small constant c. However, the internal memory cannot hold the whole graph as $M \ll \min\{|E|, |\mathsf{doc}(V)|\}$.

2.3 Solution overview

Shortest path computation is a key operation in answering a k-NK query, but also a costly operation in a massive graph. To speed up the calculation, we follow our previous work [5] which uses a set of spanning trees as an approximate representation of a graph and uses the shortest distance in trees as an approximation of the shortest distance in a graph. Thus, in the following, we first study how to answer a k-NK query in a spanning tree and then consolidate the answers from a set of trees as the approximate answers in a graph.

According to the semi-external memory model, we assume $M \ll \min\{|E|, |\mathsf{doc}(V)|\}$ and $M \geq c \cdot |V|$, where $c \geq 1$ is a small constant. For a spanning tree T of graph G, the tree nodes and edges can fit in main memory. For any keyword λ, $V_\lambda \subseteq V$ holds. Thus, tree T carrying keyword λ can entirely fit in main memory. As different keywords are independent, in the following, we tackle with a tree T carrying one keyword for index construction and query processing at a time. Note that this separate handling strategy on keywords will not increase the time or space complexity in index construction and query processing. We will study how to design a compact disk-based index which supports I/O-efficient query processing to a k-NK query.

Remark Disk-based index and I/O-efficient query processing are a promising approach to handle massive graphs. Several systems such as GraphChi [9], Grid Graph [10] and Chaos [11] have demonstrated that it is possible to process graphs with edges in the order of billions on a single machine relying on secondary storage. Meanwhile, many I/O-efficient graph algorithms [7,8,12–15] have been proposed in the literature.

Another possibility is to leverage a distributed computing platform with multiple machines and distributed indices. This can be naturally realized as the keywords are indexed independently. In the distributed computing platform, our disk-based index can still be deployed in multiple machines for two reasons. First, if the index for a single keyword exceeds the memory limit of a machine, it (or part of it) needs to be stored on disk. Second, even if the index can fit in the memory, a disk-based index ensures durability which is a desirable property.

In the remainder of the paper, we focus on designing a disk-based index and I/O-efficient query techniques. How to deploy our index in a distributed computing platform is not the focus of this paper.

3 An existing in-memory solution

In this section, we introduce an existing in-memory solution for processing k-NK queries proposed in [5]. Given a tree $T(V, E)$ carrying a keyword λ, we describe a compact index to process a k-NK query. We borrow the definitions and techniques described in this section as the basis for our solution.

3.1 Compact tree

We first create a compact representation of T for keyword λ, by adopting the compact tree data structure [16].

Definition 2 (*Compact tree* [16]) For a tree T and a keyword λ, a compact tree $\mathsf{CT}(\lambda)$ is a tree that keeps only two types of nodes in T: the keyword nodes that contain λ, and the nodes that have at least two direct subtrees containing nodes carrying λ.

A compact tree $\mathsf{CT}(\lambda)$ has at least $|V_\lambda|$ nodes and at most $2|V_\lambda| - 1$ nodes [16]. A compact tree $\mathsf{CT}(\lambda)$ can be much smaller than tree T, especially if $|V_\lambda| \ll |V|$. Building $\mathsf{CT}(\lambda)$ from T takes $O(|V_\lambda| \cdot \log |V_\lambda|)$ time [16].

Example 2 Figure 2a shows a tree T. The nodes that contain λ are marked with bold circles. Figure 2b illustrates the compact tree $\mathsf{CT}(\lambda)$. The keyword nodes containing λ are b, c, e, f, g, h, i and j, which are marked with bold circles. Node d is in $\mathsf{CT}(\lambda)$ because d has two direct subtrees with nodes carrying λ.

For every node v in $\mathsf{CT}(\lambda)$, a candidate list $\mathsf{cand}_\lambda(v)$ storing its descendant nodes carrying λ and their distance to v in T is created. Specifically, $\mathsf{cand}_\lambda(v)$ takes the form of:

$$\mathsf{cand}_\lambda(v) = \{v_1 : \mathsf{dist}_T(v, v_1), \ldots, v_j : \mathsf{dist}_T(v, v_j)\}$$

where each of v's descendants $v_i, i = 1, \ldots, j$, carries λ, and $\mathsf{dist}_T(v, v_i)$ denotes the distance between v and v_i in T. The list $\mathsf{cand}_\lambda(v)$ is sorted in nondecreasing order of the distance, i.e., $\mathsf{dist}_T(v, v_1) \leq \cdots \leq \mathsf{dist}_T(v, v_j)$.

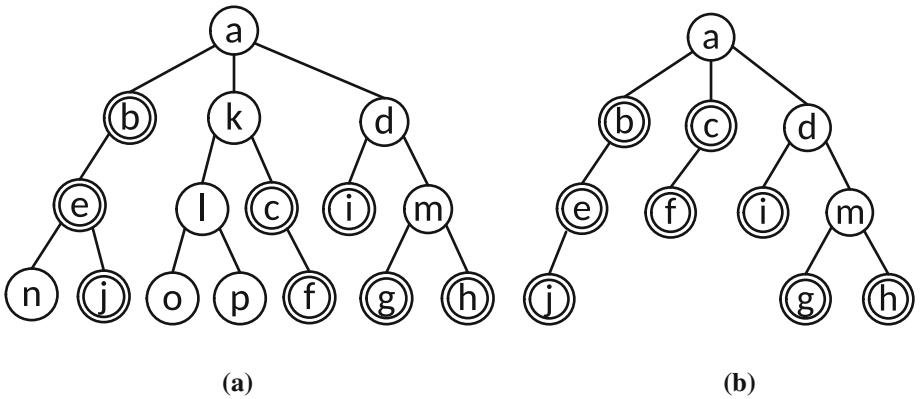

Fig. 2 Tree T and its compact tree $\mathsf{CT}(\lambda)$ with keyword λ, **a** tree T, **b** compact tree $\mathsf{CT}(\lambda)$

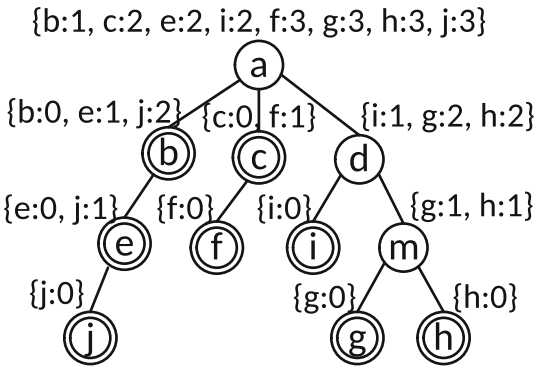

Fig. 3 Compact tree $\mathsf{CT}(\lambda)$ with candidate lists

Example 3 Figure 3 shows the compact tree with candidate lists. Node d carries a candidate list $\{i\colon 1, g\colon 2, h\colon 2\}$, because i, g, h are the descendants of d carrying λ and their distances to d are 1, 2, 2, respectively.

3.2 Use compact tree for query processing

Equipped with the compact tree $\mathsf{CT}(\lambda)$ with candidate lists, we discuss how to process a k-NK query $Q = (q, \lambda, k)$ on a tree T. We consider the following two cases: **(a)** $q \in \mathsf{CT}(\lambda)$; **(b)** $q \notin \mathsf{CT}(\lambda)$.

Case (a). For an ancestor node v of q, an entry $v_i\colon \mathsf{dist}_T(v, v_i)$ in $\mathsf{cand}_\lambda(v)$ records the keyword node v_i under v and the distance $\mathsf{dist}_T(v, v_i)$. Consider the path from q to keyword node v_i via their common ancestor v, the corresponding path length is $\mathsf{dist}_T(q, v) + \mathsf{dist}_T(v, v_i)$. Based on this observation, for an ancestor v of q, we add $\mathsf{dist}_T(q, v)$ to every entry in $\mathsf{cand}_\lambda(v)$, and we do this operation for every ancestor of q. Finally, we merge the candidate lists along the path from q to the root and return the top-k results. This procedure is shown in Algorithm 1 merge-list merges the candidate lists of q's ancestors in $\mathsf{CT}(\lambda)$ using two operators $\oplus$ and $\otimes_k$. The $\oplus$ operator adds a distance $\mathsf{dist}_T(q, v)$ to every distance entry $\mathsf{dist}_T(v, v_i)$, $1 \le i \le j$, in $\mathsf{cand}_\lambda(v)$, that is, $\mathsf{cand}_\lambda(v) \oplus \mathsf{dist}_T(q, v) = \{v_1\colon \mathsf{dist}_T(v, v_1) + \mathsf{dist}_T(q, v), \ldots, v_j\colon \mathsf{dist}_T(v, v_j) + \mathsf{dist}_T(q, v)\}$. The $\otimes_k$ operator merges two lists R_1 and R_2, and returns the top-k elements from the merged list.

Algorithm 1: merge-list $(Q, \mathsf{CT}(\lambda))$

Input: A k-NK query $Q = (q, \lambda, k)$, and a compact tree $\mathsf{CT}(\lambda)$.
Output: Answer for Q on $\mathsf{CT}(\lambda)$.

```
1 R ← ∅;
2 foreach ancestor v of q in CT(λ) do
3     R ← R ⊗_k (cand_λ(v) ⊕ dist_T(q, v));
4 return R;
```

Example 4 To answer a query $Q = (c, \lambda, 3)$ using $\mathsf{CT}(\lambda)$ in Fig. 3, for node c itself we have $\mathsf{cand}_\lambda(c) = \{c\colon 0, f\colon 1\}$. For the ancestor node a, we perform $\mathsf{cand}_\lambda(a) \oplus \mathsf{dist}_T(c, a)$ and get $\{b\colon 3, c\colon 4, e\colon 4, i\colon 4, f\colon 5, g\colon 5, h\colon 5, j\colon 5\}$. We merge these two candidate lists by $\otimes_k$ and get top-3 results as $R = \{c\colon 0, f\colon 1, b\colon 3\}$.

Case (b). When $q \notin \mathsf{CT}(\lambda)$, we need to find out how q is connected to nodes in $\mathsf{CT}(\lambda)$. For this purpose, we define the *entry node* of q. Intuitively, an entry node is a node that is closest to q in T and belongs to $\mathsf{CT}(\lambda)$ as well. A node $q \notin \mathsf{CT}(\lambda)$ has two entry nodes in $\mathsf{CT}(\lambda)$ as defined below.

Definition 3 (*Entry nodes pair* [5]) Given a tree $T(V, E)$, keyword λ and its compact tree $\mathsf{CT}(\lambda)$, for a node $q \in V(T)$, the entry nodes pair of q is a pair of nodes (u, u') in $\mathsf{CT}(\lambda)$, denoted as $\mathsf{ENP}_\lambda(q) = (u, u')$, where the path from q to any keyword nodes must pass through u or u', and there does not exist another node in $\mathsf{CT}(\lambda)$ that is closer to q in T. For a keyword node v in $\mathsf{CT}(\lambda)$, $\mathsf{dist}_T(q, v)$ is defined as $\min\{\mathsf{dist}_T(q, u) + \mathsf{dist}_T(u, v), \mathsf{dist}_T(q, u') + \mathsf{dist}_T(u', v)\}$. If u and u' are different, there is an edge between u and u' in $\mathsf{CT}(\lambda)$.

In the following, we give a remark that the entry nodes pair is sufficient to compute the shortest distance for any query node q to all keyword nodes in $\mathsf{CT}(\lambda)$.

Remark 1 Given a tree $T(V, E)$ with a keyword λ, for any vertex $q \in V$ and the entry nodes pair of q as (u, u'), two entry nodes u and u' are sufficient to cover all the shortest paths between q and other vertices carrying keyword λ.

Proof Consider a tree $T(V, E)$ and its compact tree $\mathsf{CT}(\lambda)$ for keyword λ. If $q \in \mathsf{CT}(\lambda)$, then one of the two entry nodes must be q, i.e., $u = q$ or $u' = q$. Then the entry nodes obviously cover all the shortest paths between q and other nodes carrying λ. If $q \notin \mathsf{CT}(\lambda)$, then by definition q does not carry keyword λ and has no more than one direct subtree containing nodes carrying λ. Let u be the nearest ancestor of q in tree T such that u appears in $\mathsf{CT}(\lambda)$. Then, we consider the following two cases: **(a)** q has no direct subtree containing nodes carrying λ. Obviously, for any vertex v carrying keyword λ, the shortest path between v and q must pass node u; **(b)** q has one direct subtree containing nodes carrying λ. Let u' be the nearest descendant of q in tree T such that u' appears in $\mathsf{CT}(\lambda)$. Thus, for any vertex v carrying keyword λ, it is easy to know that the shortest path between v and q must pass nodes u or u'. Thus, this remark holds.

Example 5 Consider tree T in Fig. 2a and compact tree $\mathsf{CT}(\lambda)$ in Fig. 2b. For node $l \notin \mathsf{CT}(\lambda)$, $\mathsf{ENP}_\lambda(l) = (a, c)$.

3.2.1 ENP construction

Given a tree $T(V, E)$ and its compact tree $\mathsf{CT}(\lambda)$ for keyword λ, we construct the entry nodes pair index, $\mathsf{ENP}(\lambda)$, which divides $[1, |V|]$ into several disjoint intervals, such that nodes in V with preorder in the same interval share the same entry nodes pair.

For every node v in T, we assign an interval $[s_v, t_v]$ where s_v is the preorder label of v on T and t_v is the maximum preorder label for all nodes in the subtree rooted at v. Algorithm 2 shows how to build $\mathsf{ENP}(\lambda)$ based on a recursive procedure partition. Consider an entry nodes pair (u, u') and the corresponding interval $[s, t]$. For each child node u'' of u' in $\mathsf{CT}(\lambda)$, nodes in the subtree of u'' are under the entry nodes pair (u', u''), rather than (u, u'). Thus, we remove the intervals of such subtrees from the interval $[s, t]$ which corresponds to the entry nodes pair (u, u'). Nodes with preorder in the remaining intervals have (u, u') as the entry nodes pair (line 10, 13–14). For each subtree rooted at u'', we recursively invoke partition using (u', u'') as the entry nodes pair and $[s_v, t_v]$ as the corresponding interval for partitioning. For example, in $\mathsf{CT}(\lambda)$ shown in Fig. 2b, the entry nodes pair (a, c) corresponds to an interval [6, 11]. Node c has a child node f in $\mathsf{CT}(\lambda)$ whose interval is [11, 11]. By excluding [11, 11] from [6, 11], [6, 10] remains. Nodes with preorder in [6, 10] have (a, c) as the entry nodes pair, and nodes with preorder in [11, 11] have (c, f) as the entry nodes pair. Table 1 shows the preorder label of nodes of tree T in Fig. 2a, and Table 2 shows the entry nodes pair index for $\mathsf{CT}(\lambda)$ in Fig. 2b.

Table 1 Preorder label of nodes in T

Node	a	b	e	n	j	k	l	o
Preorder	1	2	3	4	5	6	7	8
Node	p	c	f	d	i	m	g	h
Preorder	9	10	11	12	13	14	15	16

Algorithm 2: ENP-construct $(T, \mathsf{CT}(\lambda))$

Input: Tree $T(V, E)$ and its compact tree $\mathsf{CT}(\lambda)$.
Output: Entry nodes pair index $\mathsf{ENP}(\lambda)$.

```
1  ENP(λ) ← ∅;
2  r ← the root of CT(λ);
3  partition(ENP(λ), [1, |V|], (r, r));
4  return ENP(λ);
5  Procedure partition(ENP(λ), interval [s, t], (u, u'))
6  foreach child node u'' of u' in CT(λ) in increasing preorder do
7      v ← the child node of u' in the path from u' to u'' in T;
8      [s_v, t_v] ← the interval of v in T;
9      if s < s_v then
10         insert ([s, s_v − 1], (u, u')) into ENP(λ);
11     partition(ENP(λ), [s_v, t_v], (u', u''));
12     s ← t_v + 1;
13 if s ≤ t then
14     insert ([s, t], (u, u')) into ENP(λ);
```

Since there is a one-to-one mapping from $\mathsf{ENP}(\lambda)$ to edges of $\mathsf{CT}(\lambda)$, the size of $\mathsf{ENP}(\lambda)$ is $O(|V_\lambda|)$ [16]. Constructing $\mathsf{ENP}(\lambda)$ takes $O(|V_\lambda|)$ time. We will discuss how to organize the entry nodes pair index for all keywords on disk in Sect. 4.

3.2.2 Query processing

Algorithm 3 shows how to process a query $Q = (q, \lambda, k)$ on a compact tree $\mathsf{CT}(\lambda)$ via the entry nodes pair (u, u') of q. We get the top-k answers of $Q(u, \lambda, k)$ and $Q(u', \lambda, k)$ on $\mathsf{CT}(\lambda)$ by invoking merge-list in Algorithm 1. When we get the top-k results for the entry nodes u and u', we add the distance $\mathsf{dist}_T(q, u)$ and $\mathsf{dist}_T(q, u')$ to them, respectively, and merge the results (line 4–5).

3.2.3 Computing tree distance [5]

For processing a k-NK query on tree T as discussed above, we need to compute the tree distance $\mathsf{dist}_T(u, v)$ between two nodes u and v. The computation is done as follows. Given tree $T(V, E)$ with root r, we precompute and store the distance from r to every node in $V(T)$ using $O(|V|)$ space. For nodes u and v, we denote $\mathsf{LCA}(u, v)$ as their lowest common ancestor. The distance of u and v can be computed as $\mathsf{dist}_T(u, v) = \mathsf{dist}_T(r, u) + \mathsf{dist}_T(r, v) - 2\mathsf{dist}_T(r, \mathsf{LCA}(u, v))$. Using the techniques in [17], $\mathsf{LCA}(u, v)$ can be found in $O(1)$ time using an index of size $O(|V|)$. Thus, $\mathsf{dist}_T(u, v)$ can be com-

Table 2 Entry nodes pair index for $\mathsf{CT}(\lambda)$

Range	[1, 1]	[2, 2]	[3, 4]	[5, 5]	[6, 10]	[11, 11]
ENP	(a, a)	(a, b)	(b, e)	(e, j)	(a, c)	(c, f)
Range	[12, 12]	[13, 13]	[14, 14]	[15, 15]	[16, 16]	
ENP	(a, d)	(d, i)	(d, m)	(m, g)	(m, h)	

Algorithm 3: tree-knk $(Q, \mathsf{CT}(\lambda))$

Input: A k-NK query $Q = (q, \lambda, k)$, and a compact tree $\mathsf{CT}(\lambda)$.
Output: Answer for Q on $\mathsf{CT}(\lambda)$.

```
1 l_q ← the preorder label of q;
2 (u, u') ← binary-search(ENP(λ), l_q);
3 R ← ∅;
4 R ← R ⊗_k (merge-list((u, λ, k), CT(λ)) ⊕ dist_T(q, u));
5 R ← R ⊗_k (merge-list((u', λ, k), CT(λ)) ⊕ dist_T(q, u'));
6 return R;
```

puted in $O(1)$ time using $O(|V|)$ index space. Note that for tree $T(V, E)$ carrying multiple keywords, one LCA structure is sufficient, as it is independent of keywords. Since the memory $M \geq c|V|$, the LCA index can fit in memory; thus, we do not consider the I/O cost of accessing LCA in query processing.

Example 6 Given a query $Q = (l, \lambda, 3)$ and $\mathsf{CT}(\lambda)$ in Fig. 3, we find $\mathsf{ENP}_\lambda(l) = (a, c)$. Thus, we process two queries from l's entry nodes a and c, i.e., $Q(a, \lambda, 3)$ and $Q(c, \lambda, 3)$. For query $Q(a, \lambda, 3)$, the top-3 answers are $R_a = \{b{:}\,1, c{:}\,2, e{:}\,2\}$. By adding $\mathsf{dist}_T(a, l) = 2$ to R_a, we get $R_a \oplus 2 = \{b{:}\,3, c{:}\,4, e{:}\,4\}$. Similarly, for query $Q(c, \lambda, 3)$, the top-3 answers are $R_c = \{c{:}\,0, f{:}\,1, b{:}\,3\}$. By adding $\mathsf{dist}_T(c, l) = 2$ to R_c, we get $R_c \oplus 2 = \{c{:}\,2, f{:}\,3, b{:}\,5\}$. We merge these two lists and get the final result $R = \{c{:}\,2, b{:}\,3, f{:}\,3\}$.

3.3 Compact tree balancing

The problem is not perfectly solved using the compact tree described above, for the following two reasons. First, the index size for keyword λ is $\sum_{v \in V_\lambda} \mathsf{depth}(v, \mathsf{CT}(\lambda))$, which can be quite large if $\mathsf{depth}(v, \mathsf{CT}(\lambda))$ is large. Second, when processing a k-NK query $Q = (q, \lambda, k)$, we need to traverse the tree path from the query node q to the root of $\mathsf{CT}(\lambda)$. This process may incur high I/O cost if $\mathsf{depth}(v, \mathsf{CT}(\lambda))$ is large. Hence, reducing the depth of the compact tree is the key to reduce both index space and query time.

Qiao et al. [5] designed a distance preserving balanced tree to bound the tree depth, in their memory-based solution to the k-NK query. Different from applying balancing technique on the original shortest distance tree $T(V, E)$ in [5], we apply a similar balancing technique to the compact tree $\mathsf{CT}(\lambda)$ to reduce the tree depth. The compact tree has a smaller number of nodes than the original distance tree.

Definition 4 (*Balanced compact tree*) Given a tree $T(V, E)$ with a length on each edge, and a compact tree $\mathsf{CT}(\lambda)$ for keyword λ, the balanced compact tree, denoted as $\mathsf{BCT}(\lambda)$, is an unweighted tree with the following three properties.

P_1: $V(\mathsf{BCT}(\lambda)) = V(\mathsf{CT}(\lambda))$.
P_2: $\mathsf{depth}(\mathsf{BCT}(\lambda)) \leq \log_2 |V_\lambda|$.
P_3: For any two nodes u and v, let the lowest common ancestor of u and v on $\mathsf{BCT}(\lambda)$ be o. The following equation always holds: $\mathsf{dist}_T(u, v) = \mathsf{dist}_T(u, o) + \mathsf{dist}_T(v, o)$.

We will transform $\mathsf{CT}(\lambda)$ into a balanced compact tree $\mathsf{BCT}(\lambda)$. It preserves all distance information for any node pair on $\mathsf{CT}(\lambda)$ and $\mathsf{depth}(\mathsf{BCT}(\lambda)) \leq \log_2 |V_\lambda|$.

Constructing $\mathsf{BCT}(\lambda)$: $\mathsf{BCT}(\lambda)$ is constructed by recursively selecting a node in a subtree to rotate the subtree. We select a *median node* in a subtree to be the subtree root for rotation, which is defined as follows.

Definition 5 (*Median node* [5]) Given a tree T, the median node of T is a node r in T such that when using r as the root of T, for each child node u, T_u is a direct subtree of r and $|V(T_u)| \leq \frac{|V(T)|}{2}$ holds.

According to [5], a median node, denoted as r, uniquely exists in a tree T such that the subtree rooted at r contains more than $\frac{|V(T)|}{2}$ nodes and $\mathsf{depth}(r, T)$ is the maximum. The median node can be found by a traversal on the tree. For a compact tree $\mathsf{CT}(\lambda)$, the median node r is used to balance the size of each direct subtree of $\mathsf{CT}(\lambda)$ when using r as the root, as each direct subtree of r in $\mathsf{CT}(\lambda)$ contains at most half of the nodes in $\mathsf{CT}(\lambda)$. We recursively do this for each direct subtree of the root. In this way, we can balance the compact tree with $\mathsf{depth}(\mathsf{BCT}(\lambda)) \leq \log_2 |V_\lambda|$.

Algorithm 4 shows how to construct $\mathsf{BCT}(\lambda)$ for $\mathsf{CT}(\lambda)$. First, the median node r of $\mathsf{CT}(\lambda)$ is used as the new root and $\mathsf{CT}(\lambda)$ is rotated accordingly (line 1–2). For each direct subtree T_i of r, we recursively create $\mathsf{BCT}_i(\lambda)$ and insert $\mathsf{BCT}_i(\lambda)$ as a subtree of $\mathsf{BCT}(\lambda)$ (line 4–6). In this way, we get the balanced tree $\mathsf{BCT}(\lambda)$.

The second part of Algorithm 4 (line 7–13) computes candidate lists for nodes in $\mathsf{BCT}(\lambda)$. For each keyword node v, we propagate it to all its ancestors in $\mathsf{BCT}(\lambda)$, that is, add an entry $v{:}\,\mathsf{dist}_T(p, v)$ to the candidate list $\mathsf{cand}_\lambda(p)$ for every ancestor p of v in $\mathsf{BCT}(\lambda)$. Finally, we sort each candidate list in nondecreasing order of distances. As $\mathsf{depth}(\mathsf{BCT}(\lambda)) \leq$

Algorithm 4: BCT-construct (CT(λ))

Input: A tree CT(λ).
Output: A balanced compact tree BCT(λ) with candidate lists.

```
1  r ← the median node of CT(λ) ;
2  rotate CT(λ) with r as the root;
3  BCT(λ) ← a tree with a single node r;
4  foreach direct subtree T_i of r in CT(λ) do
5      BCT_i(λ) ← BCT-construct(T_i);
6      insert BCT_i(λ) as a subtree of r in BCT(λ);
7  foreach node v ∈ V(BCT(λ)) do
8      cand_λ(v) ← ∅;
9  foreach v ∈ V(BCT(λ)) and v carrying λ do
10     foreach ancestor p of v do
11         cand_λ(p) ← cand_λ(p) ⋃{v: dist_T(p, v)};
12 foreach v ∈ V(BCT(λ)) do
13     sort elements in cand_λ(v) in nondecreasing order of
       distances;
14 return BCT(λ);
```

$\log_2 |V_\lambda|$ holds, the time complexity to construct $\mathsf{BCT}(\lambda)$ and compute all candidate lists is $O(|V_\lambda| \cdot \log |V_\lambda|)$. For sorting candidate lists, $O(|V_\lambda| \cdot \log^2 |V_\lambda|)$ time is needed.

Query processing on $\mathsf{BCT}(\lambda)$ is similar to that on $\mathsf{CT}(\lambda)$, as discussed in Sect. 3.2. That is, we merge the candidate lists along the path from the query node (for case (a)) or its entry nodes pair [for case (b)] to the root of $\mathsf{BCT}(\lambda)$. But we only need to access at most $\log_2 |V_\lambda|$ nodes in $\mathsf{BCT}(\lambda)$, as opposed to $O(|V_\lambda|)$ nodes on $\mathsf{CT}(\lambda)$. The reduction in node access can lead to reduced I/O costs in our disk-based solution. We use the following example to illustrate processing a query $Q = (q, \lambda, k)$ on $\mathsf{BCT}(\lambda)$.

Example 7 Figure 4 shows the balanced compact tree $\mathsf{BCT}(\lambda)$ with candidate lists for $\mathsf{CT}(\lambda)$ in Fig. 3. For a query $Q = (b, \lambda, 4)$, we first find $\mathsf{ENP}_\lambda(b) = (a, b)$. Thus, we need to process queries $Q = (a, \lambda, 4)$ and $Q = (b, \lambda, 4)$. For query $Q = (a, \lambda, 4)$, the top-4 answers are $R_a = \{b{:}\,1, c{:}\,2, e{:}\,2, i{:}\,2\}$. By adding $\mathsf{dist}_T(a, b)$, we get $R_a \oplus 1 = \{b{:}\,2, c{:}\,3, e{:}\,3, i{:}\,3\}$. For query $Q = (b, \lambda, 4)$, we first get $\mathsf{cand}_\lambda(b) = \{b{:}\,0\}$. Then, we visit b's father node e and get $\mathsf{cand}_\lambda(e) \oplus \mathsf{dist}_T(b, e) = \{e{:}\,1, b{:}\,2, j{:}\,2\}$. Then,

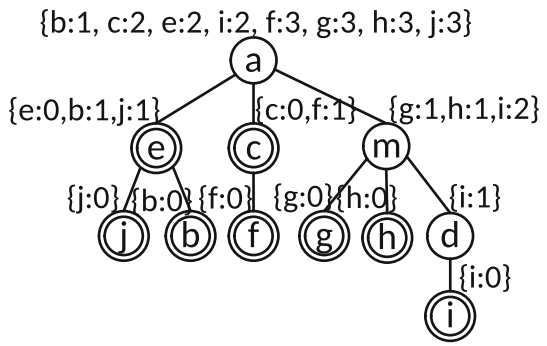

Fig. 4 BCT(λ) with candidate lists

we visit b's ancestor node a and get top-4 answers from $\mathsf{cand}_\lambda(a) \oplus \mathsf{dist}_T(b, a)$ as $\{b{:}\,2, c{:}\,3, e{:}\,3, i{:}\,3\}$. We merge these three lists and get $R_b = \{b{:}\,0, e{:}\,1, j{:}\,2, c{:}\,3\}$. Finally, we merge $R_a \oplus 1$ and R_b and get $R = \{b{:}\,0, e{:}\,1, j{:}\,2, c{:}\,3\}$.

We have the following theorem to ensure the correctness of query results computed from the balanced compact tree.

Theorem 1 *Given a* k-NK *query* $Q = (q, \lambda, k)$*, the answers computed from the balanced compact tree* $\mathsf{BCT}(\lambda)$ *are the same as the answers computed from the compact tree* $\mathsf{CT}(\lambda)$.

Proof First, we prove that a query node q in $\mathsf{BCT}(\lambda)$ can still reach all keyword nodes in $\mathsf{BCT}(\lambda)$ after tree balancing. This is because every keyword node is included in the candidate lists of all its ancestors (line 10, 11 of Algorithm 4). For a query node q, by accessing the candidate lists of all its ancestors, we can retrieve all keyword nodes for query processing.

Second, we prove that the tree distance between a query node q and any keyword node v is correctly calculated by our query processing technique. Let o be the lowest common ancestor of q and v in $\mathsf{BCT}(\lambda)$, then o must lie on the path from q to v in the original tree T. In $\mathsf{cand}_\lambda(o)$, we have an entry $v{:}\,\mathsf{dist}_T(o, v)$. In our query processing, we add $\mathsf{dist}_T(q, o)$ and $\mathsf{dist}_T(o, v)$ by the $\oplus$ operation, which leads to the tree distance $\mathsf{dist}_T(q, v)$, as $\mathsf{dist}_T(q, v) = \mathsf{dist}_T(q, o) + \mathsf{dist}_T(o, v)$ holds.

Based on the above two points, we prove that the answers computed from $\mathsf{BCT}(\lambda)$ are the same as the answers computed from $\mathsf{CT}(\lambda)$.

Theorem 2 *Given a tree* T *and keyword* λ*, constructing* $\mathsf{BCT}(\lambda)$ *takes* $O(|V_\lambda| \cdot \log^2 |V_\lambda|)$ *time, and* $\mathsf{BCT}(\lambda)$ *takes* $O(|V_\lambda| \cdot \log |V_\lambda|)$ *space. For all keywords, the index construction takes* $O(|\mathsf{doc}(V)| \cdot \log^2 |V|)$ *time, and the whole index takes* $O(|\mathsf{doc}(V)| \cdot \log |V|)$ *space. Answering a* k-NK *query* $Q = (q, \lambda, k)$ *takes* $O(k \cdot \log |V_\lambda|)$ *time.*

4 Paths + subtrees blocking index

In this section, we study how to lay out the balanced compact trees on disk, so that we only need a small number of block accesses to answer a k-NK query, and the index size on disk is small. We first give a brief overview of commonly used tree blocking techniques. Then, we propose a novel and highly efficient algorithm for blocking trees under a reasonable assumption. We split a balanced compact tree $\mathsf{BCT}(\lambda)$ into two levels. The top level is stored as paths and the bottom level is stored as subtrees. In this way, we can answer a query in constant I/Os and do not increase the index space complexity. We also discuss the case that violates the assumption and prove that our approach is still optimal in this case. Finally, we propose a practical technique to reduce the query I/O cost.

4.1 Commonly used tree blocking techniques

Our task is to block the balanced compact trees to disk. We discuss the following commonly used tree blocking techniques.

4.1.1 Blocking nodes

The first method is blocking tree nodes with the associated candidate lists one by one on disk. To answer a k-NK query, we have to perform random access to read the blocks containing the nodes that lie on the path from the query node to the root of the balanced compact tree. It costs $O(\log |V_\lambda| + \frac{|V_\lambda|}{B})$ I/Os in the worst case, where $\frac{|V_\lambda|}{B}$ is the number of blocks for reading the longest candidate list. Obviously, this node blocking method causes high I/O cost in query processing.

4.1.2 Blocking paths

The second method is blocking tree paths, that is, for each leaf node v in $\mathsf{BCT}(\lambda)$, storing the nodes and the associated candidate lists on the path from v to the root node in sequence on disk. To answer a k-NK query, we need to find out the path that the query node lies on and read the sequential blocks that contain the path. But this method increases the space cost for storing the index, as the internal nodes and their candidate lists lie on multiple paths and thus are stored multiple times. For instance, the root node r and its candidate list is stored in the paths starting from every leaf node. The size of $\mathsf{cand}_\lambda(r)$ is $\Theta(|V_\lambda|)$. The number of leaf nodes can reach $\Theta(|V_\lambda|)$ in the worst case, e.g., a balanced binary tree. As a result, the total space complexity of this blocking solution is $\Omega(|V_\lambda|^2)$ in the worst case.

4.1.3 Blocking layered trees

The third method is blocking subtrees with divided layer [12,13]. The main idea of this approach is described as follows. Given a height d and a tree T, we partition T into layers of height d, and the ith layer contains all nodes at the levels between id and $(i+1)d-1$, so that each layer can be stored in a block on disk. To retrieve the path from a node to the root node, we need to read the blocks containing the tree layers that this path belongs to, which costs $O(\frac{L}{d})$ I/Os where L is the length of this path. This technique can reduce I/O cost compared to blocking nodes and does not increase the space complexity. However, this blocking technique is not an ideal solution to lay out a balanced compact tree $\mathsf{BCT}(\lambda)$. The reasons are twofold. First, the size of a subtree can be bounded if and only if the maximum degree of $\mathsf{BCT}(\lambda)$ is bounded. If the maximum degree of $\mathsf{BCT}(\lambda)$ is not known in advance, we cannot guarantee that every layer can be blocked into a page [12]. Second, the nodes in $\mathsf{BCT}(\lambda)$ carry candidate lists of varying lengths. This makes it even more difficult to set the height d for tree partitioning so that each layer can fit into a page.

4.2 Paths + subtrees blocking

As discussed above, the three commonly used tree blocking techniques are not suitable for blocking the balanced compact trees. To solve this problem, we design a novel strategy by combining path and subtree blocking, denoted by PathST. We split $\mathsf{BCT}(\lambda)$ into two levels: the upper level is stored as paths, and the lower level is stored as subtrees. In this way, for any node v in $\mathsf{BCT}(\lambda)$, the path from v to the root can be assembled by merging a path and a subtree. To reduce the I/O cost, we want to store each of such paths and subtrees into a block. For this purpose, we first make a reasonable assumption:

$$\mathbf{A_1}: \bar{k} \cdot \log_2 |V_\lambda| \leq B,$$

where $\bar{k}$ is a real large constant and $k \leq \bar{k}$ holds for any k-NK query $Q = (q, \lambda, k)$. The assumption is reasonable due to the following reason. If we use a typical block size of $B = 64$ KB, and $\bar{k} = 1000$ which is large enough for any k-NK query in practice, $\bar{k} \cdot \log_2 |V_\lambda| \leq B$ holds even for $|V_\lambda| = 2^{64}$, which is an extremely large number for the tree nodes carrying λ.

According to the assumption, we modify the structure of $\mathsf{BCT}(\lambda)$ as follows. For each node $v \in \mathsf{BCT}(\lambda)$, if $|\mathsf{cand}_\lambda(v)| > \bar{k}$, we keep the first $\bar{k}$ entries in $\mathsf{cand}_\lambda(v)$ and denote the new list by $\mathsf{cand}^{\bar{k}}_\lambda(v)$. The new balanced compact tree is denoted as $\mathsf{BCT}_{\bar{k}}(\lambda)$.

Example 8 Figure 5a shows a $\mathsf{BCT}_{\bar{k}}(\lambda)$ for the balanced compact tree $\mathsf{BCT}(\lambda)$ with the parameter $\bar{k} = 2$. For node m, we keep the candidate list $\mathsf{cand}^{\bar{k}}_\lambda(m) = \{g{:}\,1, h{:}\,1\}$, instead of $\mathsf{cand}_\lambda(m) = \{g{:}\,1, h{:}\,1, i{:}\,2\}$ in Fig. 4. For the following examples in this section, we set the parameters $\bar{k} = 2$ and $B = 5$.

Now we start to lay out $\mathsf{BCT}_{\bar{k}}(\lambda)$ by paths and subtrees blocking into disks in the bottom-up manner.

4.2.1 Subtree blocking in bottom level

In the bottom level, we store a subtree into a block. For a node $v \in \mathsf{BCT}_{\bar{k}}(\lambda)$, the subtree rooted by v, denoted as $\mathsf{ST}(v)$, contains all v's descendants and itself. We use $|\mathsf{ST}(v)|$ to denote the total size of all candidate lists in $\mathsf{ST}(v)$. If $|\mathsf{ST}(v)| \leq B$, we say that $\mathsf{ST}(v)$ can fit into a block. To make the best use of the block, we define a maximal subtree bounded by a block as follows.

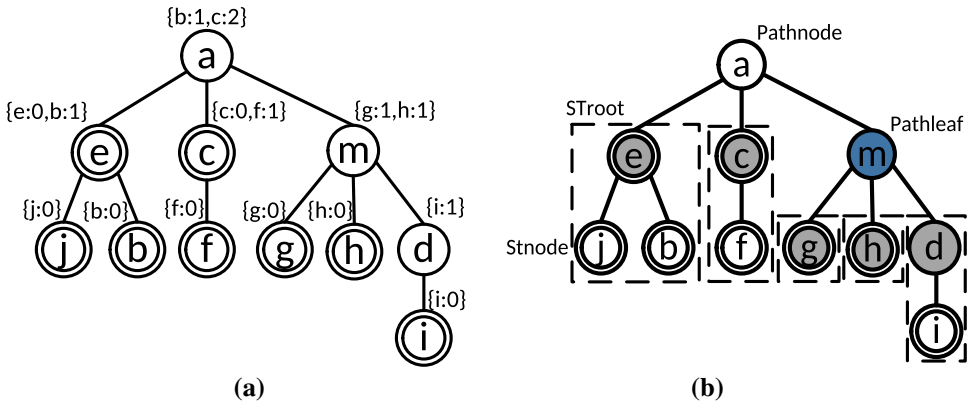

Fig. 5 Paths + subtrees blocking, **a** $\mathsf{BCT}_{\bar{k}}(\lambda)$ with $\bar{k} = 2$, **b** blocking $\mathsf{BCT}_{\bar{k}}(\lambda)$ with $B = 5$

Definition 6 For $v \in \mathsf{BCT}_{\bar{k}}(\lambda)$, $\mathsf{ST}(v)$ is a maximal subtree, if there does not exist a node u as the parent of v, s.t. $|\mathsf{ST}(u)| \leq B$.

Example 9 Consider Fig. 5b with $B = 5$. For node d, the subtree of d as $\mathsf{ST}(d)$ contains two nodes d and i with their candidate lists in $\mathsf{BCT}_{\bar{k}}(\lambda)$. The size of $\mathsf{ST}(d)$, $|\mathsf{ST}(d)| = |\mathsf{cand}^{\bar{k}}_{\lambda}(d)| + |\mathsf{cand}^{\bar{k}}_{\lambda}(i)| = 2 \leq B$. Thus, $\mathsf{ST}(d)$ can fit into a block. Moreover, for node m as a parent of d, $|\mathsf{ST}(m)| = 6 > B$, thus $\mathsf{ST}(m)$ cannot be held into a block. By Definition 6, $\mathsf{ST}(d)$ is a maximal subtree.

So in the bottom level of a balanced compact tree, PathST method stores a set of maximal subtrees, denoted as $\mathcal{S} = \{\mathsf{ST}(v) | \ v \in \mathsf{BCT}_{\bar{k}}(\lambda)$, and $\mathsf{ST}(v)$ is a maximal subtree$\}$, each of which fits into a block. For the tree in Fig. 5b, the maximal subtrees are $\mathcal{S} = \{\mathsf{ST}(e), \mathsf{ST}(c), \mathsf{ST}(g), \mathsf{ST}(h), \mathsf{ST}(d)\}$.

4.2.2 Path blocking in top level

Now we consider the nodes that are not blocked into maximal subtrees. We will store such nodes in the form of paths into blocks. According to the subtree and path concepts, we first categorize all nodes into four different types.

Definition 7 For node $v \in \mathsf{BCT}_{\bar{k}}(\lambda)$, u and $\{w_1, ..., w_l\}$ are the parent and children of v, respectively. v can be classified into one of the following four types:

- STnode: $|\mathsf{ST}(v)| \leq B$, $|\mathsf{ST}(u)| \leq B$;
- STroot: $|\mathsf{ST}(v)| \leq B$, $|\mathsf{ST}(u)| > B$;
- Pathleaf: $|\mathsf{ST}(v)| > B$, $|\mathsf{ST}(w_i)| \leq B$ for $1 \leq i \leq l$;
- Pathnode: $|\mathsf{ST}(v)| > B$, $\exists i$, $1 \leq i \leq l$, s.t. $|\mathsf{ST}(w_i)| > B$.

Example 10 In Fig. 5b, node e is a STroot, and node j is a STnode in $\mathsf{ST}(e)$, since $|\mathsf{ST}(e)| < B$, $|\mathsf{ST}(j)| < B$ and $|\mathsf{ST}(a)| > B$. Node m is a Pathleaf, since $|\mathsf{ST}(m)| > B$ and all its children $\{g, h, d\}$ have subtree of size less than B. Clearly, node a is a Pathnode.

The idea of path blocking is, for each Pathleaf node v in $\mathsf{BCT}_{\bar{k}}(\lambda)$, we store the path from v to the root and the associated candidate lists into a block. The path from v to the root is denoted as $\mathsf{rpath}(v)$. All these blocking paths are denoted by $\mathcal{P} = \{\mathsf{rpath}(v) \mid v$ is a Pathleaf $\}$. As we store the paths from every Pathleaf node to the root, it is clear that all Pathnode nodes are covered by such paths.

The following lemma shows that such a path in $\mathcal{P}$ can fit into a block.

Lemma 1 *Under assumption* $\mathbf{A_1}$*, for a* Pathleaf $v \in \mathsf{BCT}_{\bar{k}}(\lambda)$*, the path* $\mathsf{rpath}(v)$ *with the associated candidate lists can be held into a block.*

Proof Since $\mathsf{BCT}_{\bar{k}}(\lambda)$ is a balanced tree with the maximum height $\log_2 |V_\lambda|$, the length of $\mathsf{rpath}(v)$ is no greater than $\log_2 |V_\lambda|$. Moreover, for any node $u \in \mathsf{rpath}(v)$, $|\mathsf{cand}^{\bar{k}}_{\lambda}(u)| \leq \bar{k}$. As a result, the total size of $\mathsf{rpath}(v)$ with candidate lists is at most $\bar{k} \cdot \log_2 |V_\lambda| \leq B$ by assumption $\mathbf{A_1}$. □

It is worth noting that even under assumption $\mathbf{A_1}$, for $\mathsf{BCT}_{\bar{k}}(\lambda)$, the blocking path technique still incurs very high space cost $O(|V_\lambda| \bar{k} \log |V_\lambda|)$. This is because, for each leaf node $v \in \mathsf{BCT}_{\bar{k}}(\lambda)$, storing the nodes and the associated candidate lists on the path from v to the root node takes $O(\bar{k} \log |V_\lambda|)$ space, and there are at most $|V_\lambda|$ leaf nodes in $\mathsf{BCT}_{\bar{k}}(\lambda)$. However, after blocking the maximal subtrees in the bottom level, our PathST technique only needs to store the paths which cover those nodes not in maximal subtrees.

Algorithm 5 shows the procedure of blocking the balanced compact tree $\mathsf{BCT}_{\bar{k}}(\lambda)$ by PathST method. We first find the maximal subtrees and store them into blocks (line 2–4). If the data size of the current tree is greater than B, each of its direct subtrees will be processed by the block function recursively (line 6–9). If r is a Pathleaf node, the path $\mathsf{rpath}(r)$ and the associated candidate lists will be stored into a block (line 10, 11).

Algorithm 5: block($\mathsf{BCT}_{\bar{k}}(\lambda)$, B)

Input: A balanced compact tree $\mathsf{BCT}_{\bar{k}}(\lambda)$ and page size B
Output: A boolean value indicating whether $\mathsf{BCT}_{\bar{k}}(\lambda)$ is blocked

```
r ← the root of BCT_k̄(λ);
if the data size of BCT_k̄(λ) ≤ B then
    Store all data on BCT_k̄(λ) into a block sequentially;
    Return True ;
allblock ← True ;
foreach direct subtree T_i of r in BCT_k̄(λ) do
    isblock ← block(T_i, B) ;
    if isblock =False then
        allblock ← False ;
if allblock =True then
    Store data on the path from r to its ancestors into a block
    sequentially ;
Return False ;
```

4.2.3 Complexity analysis

In the following, we first analyze the space of $\mathsf{BCT}_{\bar{k}}(\lambda)$. Then, we analyze the space complexity of the PathST blocking technique and show our blocking technique does not increase the space complexity of $\mathsf{BCT}_{\bar{k}}(\lambda)$.

Lemma 2 $\mathsf{BCT}_{\bar{k}}(\lambda)$ *takes* $O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$ *space.*

Proof As there are at most $2|V_\lambda| - 1$ nodes in $\mathsf{BCT}_{\bar{k}}(\lambda)$ and each candidate list is bounded by $\bar{k}$ in length, the size of $\mathsf{BCT}_{\bar{k}}(\lambda)$ is bounded $O(|V_\lambda|\bar{k})$. On the other hand, the size of $\mathsf{BCT}_{\bar{k}}(\lambda)$ is no larger than that of $\mathsf{BCT}(\lambda)$ which is $O(|V_\lambda| \log |V_\lambda|)$. Thus, we prove the size of $\mathsf{BCT}_{\bar{k}}(\lambda)$ is bounded by $O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$.

Theorem 3 *For* $\mathsf{BCT}_{\bar{k}}(\lambda)$*, the space complexity of* PathST *index is* $O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$*. For all keywords, the whole index takes* $O(|\mathsf{doc}(V)| \cdot \min(\log |V|, \bar{k}))$ *space on disk.*

Proof The PathST index includes maximal subtrees $\mathcal{S}$ and paths $\mathcal{P}$. Firstly, we prove the size of $\mathcal{S}$, $|\mathcal{S}| = O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$. The set of maximal subtrees in $\mathcal{S}$ can be regarded as a non-overlapping forest, which is a subset of $\mathsf{BCT}_{\bar{k}}(\lambda)$. According to Lemma 2, we have $|\mathcal{S}| = O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$.

Secondly, we prove the size of all paths and candidate lists in $\mathcal{P}$, $|\mathcal{P}| = O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$. The number of paths in $\mathcal{P}$ is the same as the number of Pathleaf nodes, denoted as $|PF|$. According to Lemma 1, the size of each path is no greater than B. Thus, we have $|\mathcal{P}| \leq |PF| \cdot B$. For each Pathleaf v, the size of the subtree $\mathsf{ST}(v)$ satisfies $|\mathsf{ST}(v)| > B$. Thus, we have

$$|\mathcal{P}| \leq |PF| \cdot B \leq \sum_{v \text{ is a Pathleaf}} |\mathsf{ST}(v)|.$$

As the subtrees for all Pathleaf nodes are not overlapping and form a forest, the size of all these subtrees is bounded by $O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$. Thus, we can prove that $|\mathcal{P}| = O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$.

We prove that the total size of $\mathcal{S}$ and $\mathcal{P}$ is $O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$. For all keywords, the space complexity is $O(|\mathsf{doc}(V)| \cdot \min(\log |V|, \bar{k}))$. □

4.3 Auxiliary index

Besides the balanced compact trees stored on disks in the form of paths and subtrees, we need some auxiliary index for processing a k-NK query. We discuss how to organize such auxiliary index.

Given a k-NK query $Q = (q, \lambda, k)$, we need to get the entry nodes pair $\mathsf{ENP}_\lambda(q)$ from the index $\mathsf{ENP}(\lambda)$. The size of the entry nodes pair index for keyword λ is $O(|V_\lambda|)$ and that for all keywords is $O(|\mathsf{doc}(V)|)$. According to our semi-external memory model that $M \ll |\mathsf{doc}(V)|$, the ENP index cannot fit in the memory and needs to be stored on disk. As $\frac{|\mathsf{doc}(V)|}{B} < M$ is reasonable in practice, we use a secondary index of ENP, called MSI, to store the block addresses of ENP. Since ENP is sorted by node preorder, we can find the block address containing $\mathsf{ENP}_\lambda(q)$ by binary search. After that, we will read the corresponding block into memory to process the k-NK query.

To facilitate query processing, the detailed block layout for the ENP index is designed as follows. For any node v, we keep the following 6-tuple:

$$\langle u, u', addr_s(u), addr_p(u), addr_s(u'), addr_p(u') \rangle,$$

where u, u' are the entry nodes pair of node v, $addr_s(u)$ is the block address that stores the subtree containing u. It is null if u is a Pathleaf or Pathnode (i.e., u is not contained in a maximal subtree). $addr_p(u)$ is the block address that stores the path that u must go through to reach the root of a balanced compact tree. $addr_s(u')$ and $addr_p(u')$ are the two block addresses of u' as defined above.

Given the disk-based index and the secondary index MSI, we process a k-NK query $Q = (q, \lambda, k)$ as follows. We perform binary search on MSI to find the block address containing $\mathsf{ENP}_\lambda(q)$ in $O(\log \frac{|\mathsf{doc}(V)|}{B})$ time. Then, we read the block containing the 6-tuple of q with one I/O. Once we get the entry nodes pair (u, u') and their block addresses for the paths and subtrees, we use at most four I/Os to read the corresponding blocks, from which we obtain the path from q to the root node and the associated candidate lists to answer the k-NK query.

Table 3 Complexity analysis of different blocking techniques under assumption $\mathbf{A_1}$

Blocking methods	Space complexity	Disk size (in blocks)	Query I/O cost
PathST	$O(\|V_\lambda\| \cdot \min(\log \|V_\lambda\|, \bar{k}))$	$O\left(\frac{\|V_\lambda\| \cdot \min(\log \|V_\lambda\|, \bar{k})}{B}\right)$	$O(1)$
Blocking paths	$O(\|V_\lambda\| \bar{k} \log \|V_\lambda\|)$	$O\left(\frac{\|V_\lambda\| \bar{k} \log \|V_\lambda\|}{B}\right)$	$O(1)$
Blocking nodes	$O(\|V_\lambda\| \cdot \min(\log \|V_\lambda\|, \bar{k}))$	$O\left(\frac{\|V_\lambda\| \cdot \min(\log \|V_\lambda\|, \bar{k})}{B}\right)$	$O(\log \|V_\lambda\|)$
Blocking layered trees	$O(\|V_\lambda\| \cdot \min(\log \|V_\lambda\|, \bar{k}))$	$O\left(\frac{\|V_\lambda\| \cdot \min(\log \|V_\lambda\|, \bar{k})}{B}\right)$	$O\left(\frac{\log \|V_\lambda\|}{d}\right)$

Theorem 4 *The answer to any* k-NK *query* $Q = (q, \lambda, k)$ *can be computed by at most five I/Os under the assumption* $\mathbf{A_1}$.

4.4 Complexity comparison of different tree blocking techniques

For comparison, we list the complexity of different tree blocking techniques in Table 3 under the same assumption $\mathbf{A_1}$: $\bar{k} \cdot \log |V_\lambda| \leq B$. The complexity of PathST is provided by Theorems 3 and 4.

4.4.1 Blocking paths

For each leaf node $v \in \mathsf{BCT}_{\bar{k}}(\lambda)$, storing the nodes and the associated candidate lists on the path from v to the root node takes $O(\bar{k} \log |V_\lambda|)$ space. There are at most $|V_\lambda|$ leaf nodes in $\mathsf{BCT}_{\bar{k}}(\lambda)$. Thus, the blocking path technique takes $O(|V_\lambda| \bar{k} \log |V_\lambda|)$ space and occupies $O(\frac{|V_\lambda| \bar{k} \log |V_\lambda|}{B})$ blocks. It costs constant query I/O under assumption $\mathbf{A_1}$.

4.4.2 Blocking nodes

Blocking nodes has the same space complexity as the balanced compact tree $\mathsf{BCT}_{\bar{k}}(\lambda)$, as it stores $\mathsf{BCT}_{\bar{k}}(\lambda)$ to disk without any duplicate. So the space complexity is $O(|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k}))$ and it occupies $O(\frac{|V_\lambda| \cdot \min(\log |V_\lambda|, \bar{k})}{B})$ blocks. It costs $O(\log |V_\lambda|)$ query I/O in the worst case if the path is of length $O(\log |V_\lambda|)$ and the nodes on the path are stored in non-consecutive blocks.

4.4.3 Blocking layered trees

Blocking layered trees has the same space complexity as blocking nodes. The query I/O is $O(\frac{\log |V_\lambda|}{d})$ where d is the height of a layer.

In summary, PathST takes less space than blocking paths and uses lower query I/O cost than the methods of blocking nodes and blocking layered trees. Overall, our approach PathST achieves a good balance of space cost and query efficiency among all competitive methods.

4.5 Extend paths + subtrees blocking index

Our PathST blocking technique is under the assumption $\mathbf{A_1}$ with the condition $k \leq \bar{k}$. Now we discuss how to extend the blocking technique to support k-NK queries with an arbitrary k by removing the assumption $\mathbf{A_1}$. In the extension, we block $\mathsf{BCT}(\lambda)$ in two steps.

> Step 1: For any node v with $|\mathsf{cand}_\lambda(v)| \leq \frac{B}{2}$ and its parent w with $|\mathsf{cand}_\lambda(w)| > \frac{B}{2}$, we apply our PathST method (Algorithm 5) to block the subtree $\mathsf{ST}(v)$ to disk. Specifically, the upper level of $\mathsf{ST}(v)$ is stored as paths, and the lower level of $\mathsf{ST}(v)$ is stored as subtrees;
> Step 2: For any node v with $|\mathsf{cand}_\lambda(v)| > \frac{B}{2}$, we store node v with $\mathsf{cand}_\lambda(v)$ into blocks sequentially. This blocking technique is called node blocking.

As we can see, our new blocking strategy combines PathST method and node blocking. Particularly, we split $\mathsf{BCT}(\lambda)$ into three levels: the upper level is stored as nodes, the middle level is stored as paths, and the lower level is stored as subtrees.

Lemma 3 shows that the PathST algorithm can be applied in Step 1 for blocking the subtree $\mathsf{ST}(v)$.

Lemma 3 *For a node* v *with* $|\mathsf{cand}_\lambda(v)| \leq \frac{B}{2}$ *in* $\mathsf{BCT}(\lambda)$, *for any node* $u \in \mathsf{ST}(v)$, *the candidate lists on the path from* u *to* v *can fit into a block.*

Proof According to the median node definition, we know that for a node v' and its parent v in $\mathsf{BCT}(\lambda)$, $|\mathsf{cand}_\lambda(v')| \leq \frac{|\mathsf{cand}_\lambda(v)|}{2}$ holds. Then the total size of the candidate lists on the path from u to v is bounded by $|\mathsf{cand}_\lambda(v)| + |\frac{\mathsf{cand}_\lambda(v)}{2}| + |\frac{\mathsf{cand}_\lambda(v)}{2^2}| + \cdots + 1 \leq 2|\mathsf{cand}_\lambda(v)| \leq B$. Thus, the candidate lists on the path from u to v can fit into a block.

Thus, we can always find this type of node $u^* \in \mathsf{ST}(v)$ such that, (1) the path from u^* to v can fit into one block by Lemma 3, and (2) the subtree $\mathsf{ST}(u^*)$ rooted by u^* can fit into one block. This means the subtree $\mathsf{ST}(v)$ can be blocked by PathST into paths and subtrees in Step 1.

Step 1 blocks the subtree $\mathsf{ST}(v)$ if $|\mathsf{cand}_\lambda(v)| \leq \frac{B}{2}$ and $|\mathsf{cand}_\lambda(u)| > \frac{B}{2}$ hold where u is v's parent. Step 2 stores the

remaining nodes to disk. Both steps do not increase the space complexity. Thus, the extension blocking technique creates index of $O(|V_\lambda| \cdot \log |V_\lambda|)$ space which is the same as the space of $\mathsf{BCT}(\lambda)$ according to Theorem 2. The whole index takes $O(|\mathsf{doc}(V)| \cdot \log |V|)$ space for storing all balanced compact trees for all keywords.

Based on the index, we can answer a k-NK query $Q = (q, \lambda, k)$ for an arbitrary k as follows. We need to retrieve the candidate lists of the nodes in $\mathsf{rpath}(q)$. $\mathsf{rpath}(q)$ can be obtained by merging a path and a subtree blocked in Step 1, and a series of nodes blocked in Step 2. Since each path or subtree in Step 1 can fit into one block, we only need two I/Os to read them. In addition, we need to read a series of nodes with candidate lists of size no less than $\frac{B}{2}$. The I/O cost of reading these nodes with candidate lists is denoted by $\mathcal{C}$. Since the longest candidate list occupies at most $\lceil \frac{|V_\lambda|}{B} \rceil$ blocks, and the candidate list size of a child node is at most half of the candidate list size of its parent, we have the following bound: $\mathcal{C} \leq \lceil \frac{|V_\lambda|}{B} \rceil + \lceil \frac{|V_\lambda|}{2B} \rceil + \cdots + \lceil \frac{|V_\lambda|}{2^i B} \rceil \leq 2 \cdot (\frac{|V_\lambda|}{B} + \frac{|V_\lambda|}{2B} + \cdots + \frac{|V_\lambda|}{2^i B}) \leq \frac{4|V_\lambda|}{B}$, where $\frac{|V_\lambda|}{2^i} > \frac{B}{2}$ holds according to Step 2. As a result, we can answer a k-NK query for arbitrary k in $O(\frac{|V_\lambda|}{B})$ I/Os, which is optimal in the worst case, because outputting the results for a query $Q = (q, \lambda, k)$ with $k = |V_\lambda|$ takes $scan(|V_\lambda|) = \Theta(\frac{|V_\lambda|}{B})$ I/Os.

Theorem 5 *The answer to any* k-NK *query* $Q = (q, \lambda, k)$ *for an arbitrary* k *can be computed by* $O(\frac{|V_\lambda|}{B})$ *I/Os, which is optimal in the worst case.*

4.6 Further optimization heuristic

Since different keywords are independent of each other, we use one compact tree as a compact representation of a keyword. In real data, we observe that the size of compact trees for different keywords may vary greatly. Thus, we propose an optimization heuristic to handle those compact trees which are small enough to fit into memory.

For a keyword λ, if its compact tree $\mathsf{CT}(\lambda)$ fits into memory, our heuristic just stores $\mathsf{CT}(\lambda)$ in memory instead of disk and uses the following simple method to process a k-NK query $Q = (q, \lambda, k)$ on $\mathsf{CT}(\lambda)$. We compute the distance from q to all the other nodes in $\mathsf{CT}(\lambda)$ in $O(|V_\lambda|)$ time. We then do a partial sort and return top-k nodes as the answer. This can be done in $O(k \log k + |V_\lambda|)$ time. If $|V_\lambda|$ is small, we can store $\mathsf{CT}(\lambda)$ in memory and answer a query efficiently. This heuristic can avoid I/O costs for small compact trees. To implement this heuristic, we can set a threshold $\overline{V_\lambda}$. For a keyword λ, if $|V_\lambda| < \overline{V_\lambda}$, queries on λ will be processed in memory; otherwise, queries will be handled by disk-based index.

Theorem 6 *Given a* k-NK *query* $Q = (q, \lambda, k)$ *and* $\mathsf{CT}(\lambda)$ *in memory, the optimization heuristic computes the answer in* $O(k \log k + |V_\lambda|)$ *time and takes* $O(|V_\lambda|)$ *space in memory.*

5 k-NK query on a graph

In this section, we discuss how to answer a k-NK query $Q = (q, \lambda, k)$ on a massive graph $G(V, E)$ based on our previous solution on trees. The main idea is that we use a set of spanning trees as an approximate representation of a graph. We process query Q on the set of trees and then consolidate the exact answers on trees as the approximate answer on graph G.

5.1 Index construction

We select L nodes $v_1, v_2, \ldots, v_L$ from V randomly as landmarks, from each of which we build a shortest path tree of G. Other landmark selection strategies [18–20] can be applied as well, but it is not the focus of this paper. A shortest path tree of G is a spanning tree where the path from the root to any node is a shortest path between the two nodes in G. We use Dijkstra's algorithm [21] to build a shortest path tree from a node v_i, $1 \leq i \leq L$. Dijkstra's algorithm takes $O(|E| + |V| \log |V|)$ time and $O(|V|)$ space. In the semi-external model, as $M \geq c \cdot |V|$, we can read the adjacency list of a node in memory in $O(scan(|V|))$ I/Os and traverse $G(V, E)$ in $O(|V| + scan(|E|))$ I/Os. Given L tree roots, we need $O(L \cdot (|E| + |V| \log |V|))$ time and $O(L \cdot |V|)$ memory to construct shortest path trees. In addition, according to Theorem 2, constructing the balanced compact trees of all keywords on one shortest path tree takes $O(|\mathsf{doc}(V)| \cdot \log^2 |V|)$ time. Thus, the total index construction time for a graph is $O(L \cdot (|E| + |\mathsf{doc}(V)| \cdot \log^2 |V|))$. Compared with the CPU cost, the I/O cost for traversing the graph is not significant and not considered as the bottleneck of index construction.

Theorem 7 *For a graph* $G(V, E)$*, our method constructs the index using* $O(L \cdot |\mathsf{doc}(V)| \cdot \log^2 |V|)$ *time,* $O(L \cdot |V|)$ *memory space and* $O(L \cdot |\mathsf{doc}(V)| \cdot \log |V|)$ *disk space.*

5.2 Query processing

Given a k-NK query Q, we process Q on the L shortest path trees using the tree-based solution. We merge the results from the trees by the $\otimes_k$ operator and return the top-k results as the approximate answer. The efficiency and accuracy of our method depend on the parameter L. Increasing L will improve the accuracy of results, but also increases

the query processing I/Os linearly, as the query I/O cost is $O(L)$.

To further reduce the query I/O cost, we propose a heuristic, called *nearest landmarks*. The idea is, given a query node q, we select k_l nearest landmarks to q out of L landmarks according to their distance and process the k-NK query on the corresponding k_l trees, instead of L trees. The rationale is, if q is closer to a landmark v_i, the approximate graph distance provided by the shortest path tree rooted at v_i may be more accurate. This heuristic can effectively reduce the query I/O cost from $O(L)$ to $O(k_l)$, while still achieving good precision on distance estimation.

Algorithm 6 shows how to process a k-NK query on graph G. The algorithm first selects k_l nearest landmarks to q (line 2). Since a tree distance can be computed by the LCA index in $O(1)$ time, the selection step can be implemented by top-k partial sort in $O(k_l \cdot \log k_l + L)$ time. For a selected landmark v_i, Q is processed on the corresponding tree- based index T_i. The answers from k_l trees are merged to get the final result R (line 3–5).

Algorithm 6: graph-knk (G, Q)

Input: A graph $G(V, E)$ and a k-NK query $Q = (q, \lambda, k)$.
Output: The answer for Q on G.

```
1 R ← ∅;
2 S_k ← k_l nearest landmarks to q ;
3 foreach landmark v_i in S_k do
4 |   T_i ← shortest path tree rooted by v_i;
5 |   R ← R ⊗_k tree-knk(T_i, Q);
6 return R;
```

5.3 Complexity summary

Table 4 summarizes the indexing and query processing complexities on tree and graph, respectively. The index time and space costs on graph are L times those on tree, while the query time and I/O costs on graph are k_l times those on tree.

6 k-NK query on dynamic graphs

In practice, keywords may be inserted to the graph vertices at a high frequency, e.g., new tweets in Twitter. In this section, we study how to incrementally maintain the index given frequent keyword insertions. We adopt a batch update mode, i.e., we process keyword insertions accumulated over a period of time in a batch. Batch update (instead of real-time update) is acceptable in principle, as our query algorithm returns approximate (instead of exact) k-NK answers anyway. We validate this claim in our experiments.

We denote a keyword insertion as $\langle v, \lambda \rangle$ indicating that keyword λ is inserted to vertex v. We use $\Delta\mathsf{doc}(V)$ to denote a batch of inserted keywords over a period of time. When the number of insertions reaches a certain level, i.e., $|\Delta\mathsf{doc}(V)| \geq \delta \cdot |\mathsf{doc}(V)|$ where $\delta > 0$ is a batch size parameter, we start to process the insertions in $\Delta\mathsf{doc}(V)$. As the keywords are processed separately, we focus on a single keyword at a time. Denote the insertions of keyword λ as $\Delta V_\lambda = \{\langle v, \lambda \rangle : v \in V\}$ and $\Delta V_\lambda \subseteq \Delta\mathsf{doc}(V)$.

Given ΔV_λ, updating the existing index structure is non-trivial. The challenge is that the insertions can trigger a series of changes in the compact tree structure $\mathsf{CT}(\lambda)$, the entry node pair index $\mathsf{ENP}(\lambda)$, the balanced compact tree $\mathsf{BCT}(\lambda)$ and the block layout on disk. Updating all the intermediate data structures and final index can be very complicated and expensive. Instead of updating the existing index for λ, we propose to construct a separate index, denoted as $\mathsf{BCT}_{\Delta V_\lambda}(\lambda)$ for the affected vertices in ΔV_λ. Given a query (q, λ, k), we will retrieve top-k results from both indices $\mathsf{BCT}(\lambda)$ and $\mathsf{BCT}_{\Delta V_\lambda}(\lambda)$ and then merge the two answer lists and return the top-k answers from the merged list. For a small amount of batch update, the newly constructed index $\mathsf{BCT}_{\Delta V_\lambda}(\lambda)$ is often small enough to reside in memory. In case it is large, $\mathsf{BCT}_{\Delta V_\lambda}(\lambda)$ can be stored on disk. It is not hard to verify that the query result will be identical, be it obtained by a single updated index, or the combination of the old index and the incremental one. The reason is that the distance for each keyword node is estimated to be the same value no matter the keyword nodes carrying λ are organized in one or two indices.

The time complexity of the incremental index construction on a graph is $O(L \cdot |\Delta\mathsf{doc}(V)| \cdot \log^2 |V|)$, and the incremental index size is $O(L \cdot |\Delta\mathsf{doc}(V)| \cdot \log |V|)$. To process a query, the time complexity is $O(k_l \cdot k \cdot (\log |V_\lambda| + \log |\Delta V_\lambda|))$ as we will access both the original and incremental indices.

7 Experiments

In this section, we evaluate the performance of our I/O-efficient method for processing k-NK queries on tree and graph.

7.1 Experimental setting

We test our method on a large graph extracted from social network Twitter and several synthetic trees. On the graph, we compare our method with three baseline methods.

– pivot-gs [5]. As pivot-gs is an in-memory algorithm, we adapt it to a disk-based solution by simply laying out the tree nodes one by one to disk. It takes $O(\log |V|)$ I/Os to access the disk index of pivot-gs to process a

Table 4 Algorithm complexities on tree (T) and graph (G)

	Tree	Graph
Index time	$O(\|\mathsf{doc}(V)\| \cdot \log^2 \|V\|)$	$O(L \cdot \|\mathsf{doc}(V)\| \cdot \log^2 \|V\|)$
Index size	$O(\|\mathsf{doc}(V)\| \cdot \log \|V\|)$	$O(L \cdot \|\mathsf{doc}(V)\| \cdot \log \|V\|)$
Query time	$O(k \cdot \log \|V_\lambda\|)$	$O(k_l \cdot k \cdot \log \|V_\lambda\|)$
Query I/Os	$O(1)$	$O(k_l)$

Table 5 Data set statistics

Data set	$\|V\|$	$\|E\|$	$\|\mathsf{doc}(V)\|$	# Distinct keywords	Data size	Avg. # nodes associated with query keyword
T_1	104,466	104,465	2,084,375	334,522	12.6 MB	11,141
T_2	314,983	314,982	6,276,967	758,771	39.4 MB	34,605
T_3	3,148,149	3,148,148	62,796,424	3,663,316	416.7 MB	367,171
T_4	10,494,505	10,494,504	209,360,819	5,636,193	1.4 GB	1,404,811
SubTwitter	100,059	624,056	3,210,413	385,778	26.2 MB	17,378
Twitter	41,652,230	1,468,365,182	1,886,241,342	42,191,496	69.8 GB	13,621,114

k-NK query on a tree $T(V, E)$, and $O(\log^2 |V|)$ I/Os to process a query on a graph $G(V, E)$, as the graph distance is estimated from a distance oracle [22] of size $\log_2 |V|$ according to [5].

- HLQ [6]. We use the disk-based solution provided by the authors of [6] to process k-NK queries on graph.
- scan. scan processes k-NK queries on graph simply by Dijkstra's algorithm without any index.

On the synthetic trees, we compare our method with pivot-gs and the blocking layered tree technique, denoted as layer, mentioned in Sect. 4.1. In layered tree blocking, we block consecutive levels of the tree that maximally fit into one block to disk.

Our method is denoted as knk-io-mp, which stands for I/O-efficient k-NK solution with memory optimization heuristic. All methods are implemented in GNU C++ and tested on a Windows machine with an Intel Xeon 2.7 GHz CPU. Main indices are stored in disk. The disk block size is set to 64 KB. A 32 GB memory limit is set for index construction.

7.1.1 Data sets and queries

We use a large Twitter graph[3] with 41 million nodes, 1.4 billion edges and 1.8 billion keywords on nodes. In the Twitter graph, each node represents a Twitter user. For each node v, each discriminative word in v's tweets is regarded as a keyword and added into $\mathsf{doc}(v)$. An edge (u, v) between nodes u and v means that u follows v or vice versa. A weight $(\log_2 \mathsf{deg}(u) + \log_2 \mathsf{deg}(v))$ is assigned to edge (u, v) as a conceptual length, where $\mathsf{deg}(u)$ denotes the degree of node u. Compared with the unit edge weight setting, the numerical edge weight can effectively differentiate the length of all edges in a graph. Thus for any k-NK query, this helps produce a ranking of top-k answer nodes with less ties in their distances as the ground truth, which is important for fair and unambiguous ranking quality evaluation. We also sampled a subnetwork of the Twitter graph, called SubTwitter, with 100 K nodes, 624K edges and 3.2 million keywords. For both graphs, we set the number of landmarks to be $L = 15$ as a default setting. We have tested L in $[5, 40]$ and finally set $L = 15$ as it achieves a good balance between answer quality and efficiency.

In addition, we generate four synthetic unweighted trees, denoted T_1, T_2, T_3 and T_4, to test the performance on trees. The number of tree nodes ranges from 100K to 10M, and the number of keywords $|\mathsf{doc}(V)|$ ranges from 2 to 200M. The synthetic trees are generated randomly with a maximum degree of 20. Keywords assigned to the tree nodes are extracted from DBLP.[4] We view the XML document of DBLP as a tree T_D with keywords. For each node v_i in a synthetic tree, we randomly choose a node of T_D and copy its keywords to v_i. The statistics of the network and trees are shown in Table 5.

For each data set, we generated 10,000 k-NK queries in the form of $Q = (q, \lambda, k)$, where $q \in V$ is a randomly selected query node, and λ is a keyword randomly selected by following the keyword frequency distribution in the document collection. The reason is that frequent keywords are more

[3] https://snap.stanford.edu/data/twitter7.html.

[4] http://www.informatik.uni-trier.de/~ley/db.

likely to be asked in real-life queries. The average number of nodes associated with query keyword for each data set is listed in the last column of Table 5. We test $k = 1, 2, \ldots, 128$.

7.1.2 Evaluation metrics.

We evaluate the algorithm performance in terms of efficiency and effectiveness. The efficiency performance is measured by *query time* and *query I/Os*. We also evaluate *index construction time* and *index size*. The effectiveness performance evaluates the answer quality and is measured by three metrics: *hit rate*, *Spearman's rho* [23] and *error*. *Spearman's rho* measures the rank correlation between an approximate rank result and the ground truth. *Hit rate* and *error*, defined below, measure the quality of an approximate result. For a query $Q = (q, \lambda, k)$, denote the exact result as $R = \{u_1{:}d_1, \ldots, u_k{:}d_k\}$ in nondecreasing order of their distances, and $\overline{d} = d_k$ as the upper bound distance of the result R. Denote an approximate result set as $R' = \{u'_1{:}d'_1, \ldots, u'_k{:}d'_k\}$ in nondecreasing order of their distances. The hit rate is defined as:

$$\mathsf{hit}(R') = \left|\left\{i \in [1,k] | \mathsf{dist}\left(u'_i, q\right) \leq \overline{d}\right\}\right| / k$$

and the error is the average relative error of the estimated distances w.r.t. the ground truth:

$$\mathsf{err}(\mathsf{R}') = \sum_{1 \leq i \leq k} |d'_i/d_i - 1|/k$$

Note that the k-NK answers on a tree are exact. So we do not report or compare the effectiveness performance on synthetic trees.

7.2 Experimental results

7.2.1 Query time and I/Os on trees

We compare pivot-gs, layer and knk-io-mp on synthetic trees by varying the parameter k and report the query processing time (in microseconds) in Fig. 6a–d and the average query I/O cost in Table 6. Fig. 6a–d shows that knk-io-mp uses the least query time, in the range of a few hundred to one thousand microseconds to process a k-NK query. This can be explained by the constant query I/O complexity of knk-io-mp. The query time of pivot-gs is one order of magnitude longer than that of knk-io-mp, and the query time of layer is 1.5–2.6 times longer than that of knk-io-mp. The average query I/O costs by these three methods in Table 6 follow the same trend, and our method knk-io-mp uses the least I/Os for query processing. The average query I/O costs of pivot-gs are 4.5–7.8 times that of knk-io-mp. This result also explains the difference of the query time by the three methods. Note that pivot-gs cannot finish index construction on the largest tree T_4 due to the memory limit of 32GB.

Table 6 Average query I/Os on synthetic trees and Twitter graphs

Data set	pivot-gs	layer	knk-io-mp
T_1	14.086	2.782	1.818
T_2	15.102	3.614	2.031
T_3	18.176	5.873	2.993
T_4	–	7.106	2.774
SubTwitter	127.856	–	28.187
Twitter	–	–	36.413

7.2.2 Query time and I/Os on graphs

We compare pivot-gs, HLQ, scan and knk-io-mp on graphs by varying the parameter k. Figure 6e reports the query processing time (in microseconds) of the four methods on SubTwitter. knk-io-mp uses the least query time, around 1,000 microseconds to process a k-NK query. The query time of pivot-gs and HLQ is 5–10 times that of knk-io-mp, while the query time of scan is the longest, and it shows an increasing trend with k. On the large Twitter graph, pivot-gs cannot finish index construction due to the memory limit, and HLQ cannot finish it in a given 100 h time limit. So we only report the query time of knk-io-mp and scan. While the query time of knk-io-mp remains stable, that of scan increases with k. scan is two to three orders of magnitude slower than knk-io-mp. Table 6 shows the average query I/Os by pivot-gs and knk-io-mp on SubTwitter, and the query I/Os by knk-io-mp on Twitter. The HLQ program provided by their authors does not report the query I/O costs.

7.2.3 Index time and index size on trees

In this experiment, we compare pivot-gs, layer and knk-io-mp on synthetic trees and report the index size and index construction time in Table 7. The index time of layer and knk-io-mp is close, which means different tree blocking techniques do not affect the index construction time much. The index construction time of pivot-gs is slightly longer than that of layer and knk-io-mp. The index size of layer is the smallest as it causes no duplicate of the tree nodes in blocking. The index size of knk-io-mp is slightly larger than that of layer, and the index size of pivot-gs is three to four times larger than that of knk-io-mp.

7.2.4 Index time and index size on graphs

In this experiment, we compare pivot-gs, HLQ and knk-io-mp on graphs and report the index size and index construction

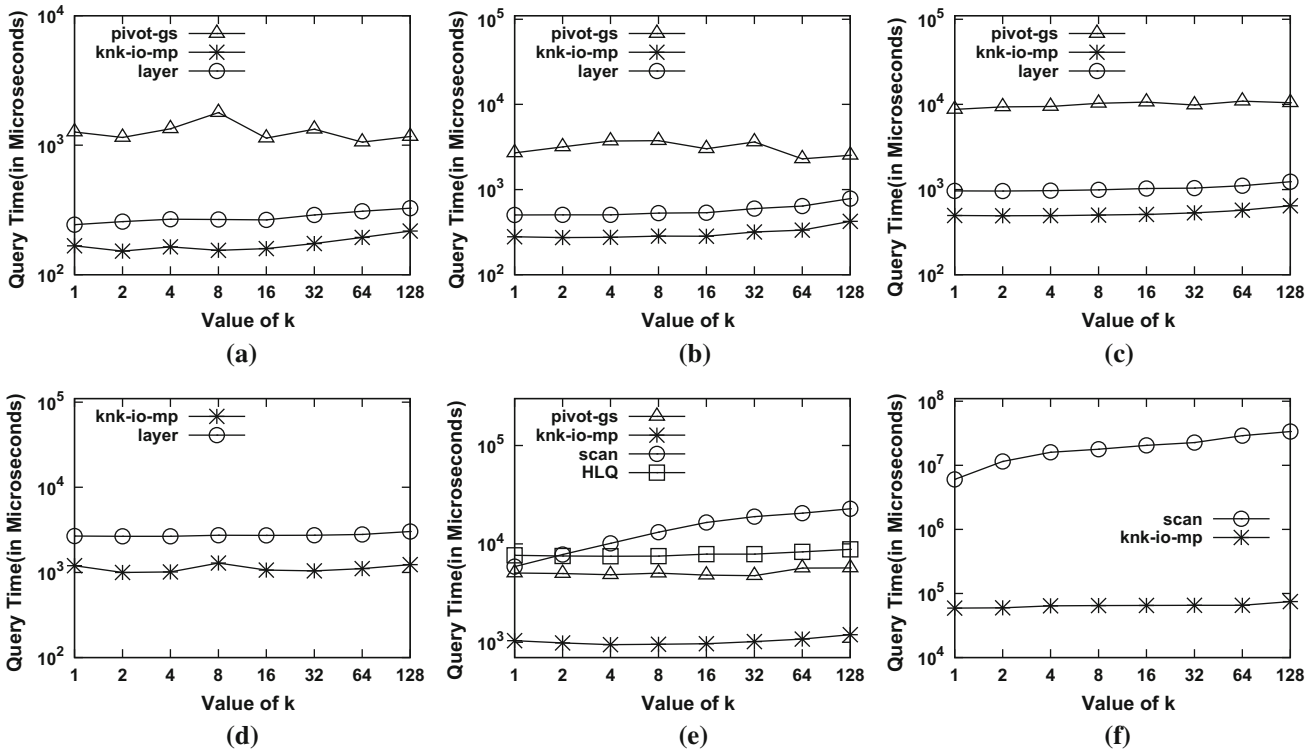

Fig. 6 Query time in microseconds by varying k, **a** T_1, **b** T_2, **c** T_3, **d** T_4, **e** SubTwitter, **f** Twitter

Table 7 Index time and index size on synthetic trees

Data set	Index time			Index size		
	pivot-gs (s)	layer (s)	knk-io-mp (s)	pivot-gs (MB)	layer	knk-io-mp
T_1	9.392	7.416	7.169	416.9	92.4 MB	116.7 MB
T_2	31.596	20.860	23.712	1318.2	335.9 MB	374.1 MB
T_3	539.783	362.259	348.76	15223.3	5098.1 MB	5671.2 MB
T_4	–	1539.372	1427.332	–	15.3 GB	16.4 GB

Table 8 Index time and index size on graphs

Data set	Index time			Index size		
	pivot-gs	HLQ	knk-io-mp	pivot-gs	HLQ	knk-io-mp
SubTwitter	334.629 s	1605.871 s	121.239 s	3342.7 MB	66.1 GB	1204.4 MB
Twitter	–	–	57.993 h	–	–	2.307 TB

time in Table 8. On SubTwitter graph, the index time of HLQ is 13.26 times longer than that of knk-io-mp, and the index time of pivot-gs is 2.76 times longer than that of knk-io-mp. The index size of HLQ is 54.90 times larger than that of knk-io-mp, and the index size of pivot-gs is 2.78 times larger than that of knk-io-mp. HLQ and pivot-gs cannot finish index construction on Twitter. Another observation we make is the index size we report is much larger than the original graph size in Table 5. This shows that disk-based index is very essential even in some cases the graph itself can be stored in memory.

7.2.5 Answer quality on graphs

We evaluate the answer quality of pivot-gs and knk-io-mp in terms of hit rate, Spearman's rho and error. As HLQ returns exact answers to a k-NK query, we do not need to evaluate its answer quality. We set the parameters of pivot-gs according to [5] and set the landmark number $k_l = K = 15$ for knk-io-mp. Figure 7 shows the results on SubTwitter by varying k. We observe that knk-io-mp outperforms pivot-gs in terms of all three metrics and for all k values. The answer quality difference is due to the different graph embedding

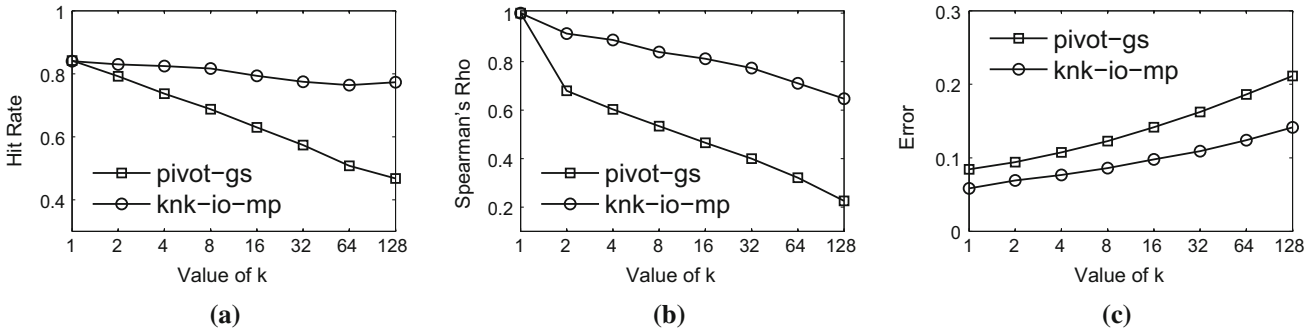

Fig. 7 Answer quality evaluation on SubTwitter by varying k, **a** hit rate, **b** Spearman's rho, **c** error

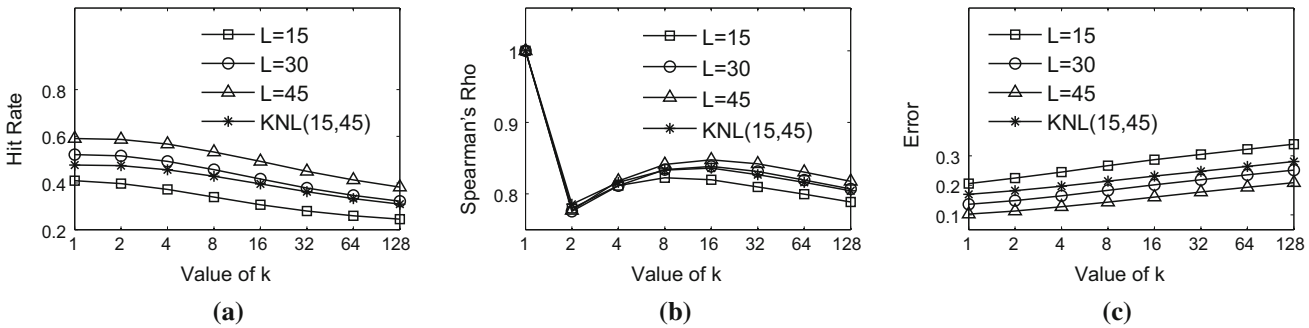

Fig. 8 Answer quality evaluation on Twitter network by varying k, **a** hit rate, **b** Spearman's rho, **c** error

techniques adopted: pivot-gs uses distance oracle [22] for shortest distance approximation, while knk-io-mp uses landmark embedding for shortest distance approximation. We believe the estimated shortest distance by distance oracle in pivot-gs is less precise, which explains the worse answer quality of pivot-gs.

For the large Twitter graph, pivot-gs cannot handle it due to the memory limit for index construction. So we only tested knk-io-mp on Twitter. In this experiment, we evaluate the performance by varying the number of landmarks L and testing our *nearest landmarks* heuristic (denoted as KNL(k_l, L)) and report the answer quality results in Fig. 8. We can see that when L increases, the answer quality improves as it gives more accurate distance estimation. When we use the nearest landmarks heuristic, we find that the answer quality by choosing $k_l = 15$ out of 45 landmarks is close to that by using 30 landmarks without the heuristic. But using 15 nearest landmarks can effectively reduce the query I/O cost by half compared with using 30 landmarks without the heuristic. This result proves the effectiveness of the nearest landmarks technique.

7.2.6 Memory optimization technique

We evaluate the memory optimization technique proposed in Sect. 4.6 on T_2 with $k = 128$. The results are reported in Table 9 by varying the threshold $\overline{V_\lambda}$. Given a k-NK query $Q = (q, \lambda, k)$, if $|V_\lambda| < \overline{V_\lambda}$, Q will be processed by a memory index rather than disk index. In the first row of the table, $\overline{V_\lambda} = 0$ shows the performance without the memory optimization technique. As $\overline{V_\lambda}$ grows, query I/Os, index time and size are greatly reduced. But the query time first decreases till $\overline{V_\lambda} = 3000$ and then increases again. This is because the optimization technique reduces query time by reducing the I/O cost when $\overline{V_\lambda}$ is small. When $\overline{V_\lambda}$ is large, the query time is dominated by CPU cost and increases as query time complexity is $O(k \cdot \log k + |V_\lambda|)$ instead of $O(k \cdot \log |V_\lambda|)$ in memory. These results demonstrate the effectiveness of the memory optimization technique. If we want to minimize the query time, we can set $\overline{V_\lambda} = 3000$. If our goal is to reduce the query I/O or disk index size, we can set $\overline{V_\lambda} = 10{,}000$.

7.2.7 Performance evaluation on keyword insertions

In this experiment, we evaluate the performance of our proposed approach in Sect. 6 for answering k-NK queries on dynamic graphs with keyword insertions. When the number of insertions $|\Delta\mathsf{doc}(V)| \geq \delta \cdot |\mathsf{doc}(V)|$ where $\delta > 0$ is a batch size parameter, we start to process the insertions in $\Delta\mathsf{doc}(V)$. The keyword frequency distribution in $\Delta\mathsf{doc}(V)$ is the same as that in the original set $\mathsf{doc}(V)$. We denote our approach by incremental and use a baseline method denoted by scratch which recomputes the index from scratch for

Table 9 Evaluation of memory optimization heuristic on tree T_2

$\overline{V_\lambda}$	Query time (μs)	Query I/Os	Index time (s)	Index size (MB)
0	535	4.116	72.3	865.6
1000	447	2.329	39.7	431.2
3000	415	1.957	33.1	352.0
10,000	470	1.457	31.2	264.7
30,000	568	1.085	25.8	202.6
100,000	658	0.677	14.9	127.7

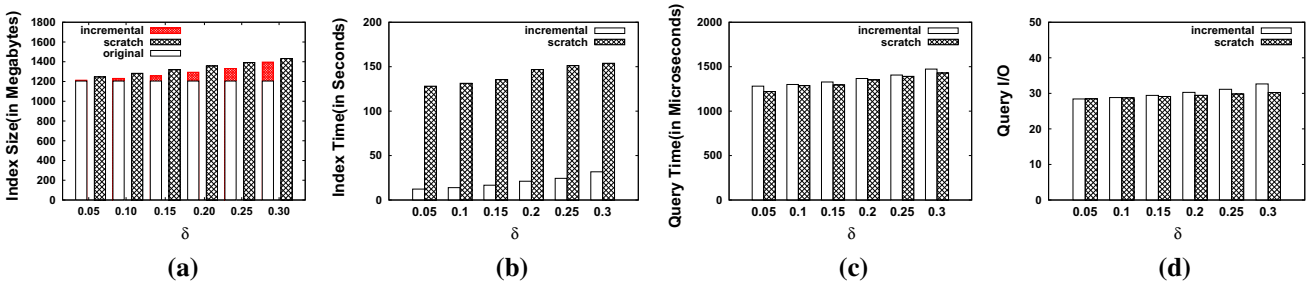

Fig. 9 Indexing and query performance on SubTwitter network with keyword insertions by varying δ, **a** index size, **b** index time, **c** query time, **d** query I/O

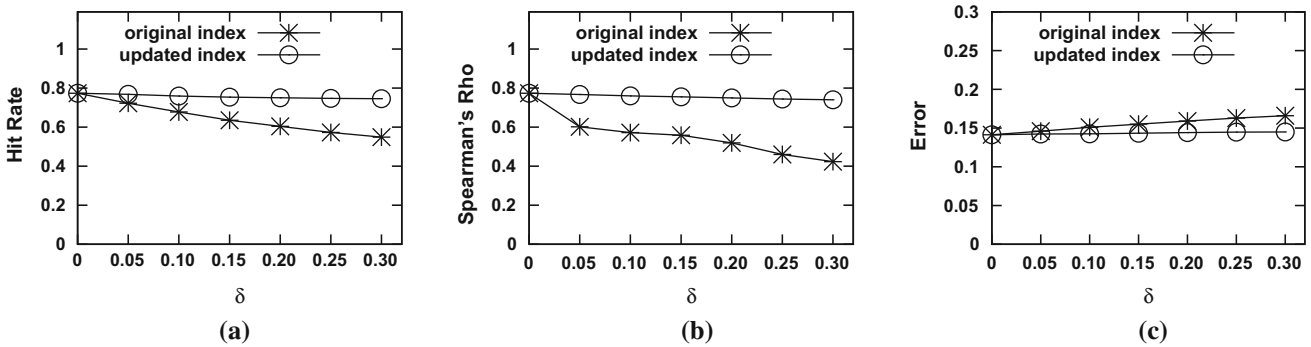

Fig. 10 Answer quality evaluation on SubTwitter network with keyword insertions by varying δ, **a** hit rate, **b** Spearman's rho, **c** error

comparison. We test 10,000 queries on SubTwitter network with k fixed as 128.

Index size and construction time We vary the number of keyword insertions by increasing δ and report the index size (in megabytes) and construction time (in seconds) by incremental and scratch. The results are shown in Fig. 9a, b. We observe the size of the incremental index is very small, e.g., 1/6 of the original index size when there are 30% keyword insertions ($\delta = 0.3$). The total size of the original index and the incremental one is slightly smaller than the size of the recomputed index from scratch. A possible reason is that, if we organize all keyword nodes carrying a keyword in a single compact tree (as done by scratch), some extra auxiliary nodes are needed to connect them according to the definition of compact tree. But if the newly inserted keyword nodes are organized in a separate tree (as done by incremental), they do not need to be connected to the original keyword nodes, thus reducing the number of such auxiliary nodes. In terms of the construction time in Fig. 9b, the incremental update is 5–10 times faster than recomputing from scratch.

Query time and I/O cost We report the query time (in microseconds) and I/O cost by incremental and scratch in Fig. 9c, d when we vary the parameter δ. The query time of incremental is at most 3% higher than that of scratch and the query I/O cost of incremental is at most 8% higher. This is because incremental needs to check both the original index and the incremental index to consolidate the results.

Answer quality We evaluate the answer quality when we vary the number of keyword insertions. As incremental and scratch return identical results, we just report the answer quality measures by incremental. For comparison, we report the answer quality results by using the original index (which is outdated given the keyword insertions). Figure 10a–c show the hit rate, Spearman's rho and error, respectively. We observe that all three answer quality measures remain stable when the incremental index is maintained and used for

query processing. In contrast, the answer quality deteriorates as more keywords are inserted, but the index is not updated.

7.2.8 Summary

In this section, we evaluate the performance of knk-io-mp and other baseline methods. The main conclusions are as follows.

1. On synthetic trees, knk-io-mp is 1.5–2.6 times faster than layer and one order of magnitude faster than pivot-gs in query processing time. Its index size is slightly larger than that of layer, but is around 1/3 of the index size of pivot-gs.
2. On SubTwitter graph, knk-io-mp again has the least query processing time, which is about one order of magnitude faster than HLQ, pivot-gs, and one to two orders of magnitude faster than scan. The index size of HLQ is 54.90 times larger than that of knk-io-mp, and the index construction time of HLQ is 13.26 times longer than that of knk-io-mp.
3. On the large Twitter graph with 41 million nodes and 1.4 billion edges, pivot-gs and HLQ fail to finish index construction due to the memory limit and 100 h time limit, respectively. But our method knk-io-mp can process a k-NK query in 36.4 I/Os on average (or 100 ms) on the Twitter graph.
4. In terms of the answer quality, knk-io-mp finds more accurate answers than pivot-gs, due to the different graph embedding techniques adopted, while HLQ returns exact answers.
5. We demonstrate the effectiveness of our incremental index update mechanism. The incremental index is very compact in size and can be constructed 5–10 times faster than recomputing from scratch. The incurred overhead in query time or I/O cost is very minor, and the query answer quality is the same as the answer quality by using a recomputed index.

8 Related work

The related work to our study includes keyword search, k nearest neighbors and I/O-efficient algorithms.

8.1 Keyword search

The problem of keyword search in a graph is to find a substructure of the graph containing the query keywords. The answer substructure can be a tree [24–26], a subgraph [27,28] or a r-clique [29]. Yu et al. [30] gives a survey on keyword search in databases and graphs. Keyword search has substantial differences from the k-NK query in our paper. A k-NK query looks for k nearest answer nodes, each one of which contains all the query keywords, but does not concern about the joint structure of these nodes as keyword search does. Yao et al. [31] and Cao et al. [32] study the problem of *keyword routing* on a road network. Recently, Qiao et al. [5] proposed memory algorithms for k-NK queries on a graph. But a straightforward adaptation of their algorithm to a disk-based solution yields poor performance, as shown in our experiments. Jiang et al. [6] proposed both memory and disk-based approaches, called HLQ, to find exact answers of top-k nearest keyword search in massive graphs. They use the 2-hop labeling index for exact distance computation, which is different from our balanced compact trees for approximate distance computation, and is more expensive in nature in terms of both query cost and indexing cost. We empirically compared our method with HLQ and observed the substantial differences in performance.

8.2 K nearest neighbors

(k-NN) search has been extensively studied in spatial networks [1–3]. Kolahdouzan and Shahabi [1] proposes to use network Voronoi polygons to divide a graph into disjointed subsets for k-NN search. Samet et al. [2] uses a shortest path quadtree to answer k-NN queries. Sankaranarayanan and Samet [3] proposes a path-distance oracle index to estimate ε-approximated distance for k-NN query answers. Nevertheless, all above approaches are designed for spatial networks with coordinates, which cannot be applied to graphs.

8.3 I/O-efficient algorithms

Recently, there have been studies on different fundamental algorithmic problems on databases and graphs with I/O-efficient solutions, such as triangle listing [14,15], strong connected components computing [8], depth first search [7] and top-k range query [33]. A survey of several elementary techniques used for designing I/O-efficient algorithms can be found in [12]. It is noted that [13] describes a blocking index for rooted trees to efficiently report the path from a node to the root in $O(L/B)$ I/Os, where L is the length of the traversed path. However, in this paper, we face a different problem of bottom-up path traversal in a weighted rooted tree, in which each node carries a candidate list of different size.

9 Conclusions

In this work, we investigate top-k nearest keyword (k-NK) search on massive networks and design an I/O-efficient solution. We borrow the key concepts including compact tree and its balancing technique from our previous work [5] to form

the basis of our solution. Then, we design novel tree blocking techniques to lay out the compact tree on disk in the form of paths and subtrees, which is compact and supports constant query I/Os. We also propose some heuristics to further optimize our solution. We experimentally test our proposed approach on a large-scale real-world graph and demonstrate its superior performance over the state-of-the-art approaches in terms of query I/O costs, index size and index construction time. Specifically, our method uses 36 I/Os on average (or 100 ms) to process one k-NK query on a Twitter graph with 41.6 million nodes and 1.4 billion edges. This shows that our solution is very promising to be practically applied on real-world networks, with the strong theoretical guarantee of optimal query I/O complexity.

Acknowledgements The work was supported by The Chinese University of Hong Kong Direct Grant No. 4055048.

References

1. Kolahdouzan, M.R., Shahabi, C.: Voronoi-based k nearest neighbor search for spatial network databases. In: VLDB, pp. 840–851 (2004)
2. Samet, H., Sankaranarayanan, J., Alborzi, H.: Scalable network distance browsing in spatial databases. In: SIGMOD, pp. 43–54 (2008)
3. Sankaranarayanan, J., Samet, H.: Query processing using distance oracles for spatial networks. IEEE Trans. Knowl. Data Eng. **22**(8), 1158–1175 (2010)
4. Bahmani, B., Goel, A.: Partitioned multi-indexing: bringing order to social search. In: WWW, pp. 399–408 (2012)
5. Qiao, M., Qin, L., Cheng, H., Yu, J.X., Tian, W.: Top-k nearest keyword search on large graphs. PVLDB **6**(10), 901–912 (2013)
6. Jiang, M., Fu, A., Wong, R.: Exact top-k nearest keyword search in large networks. In: SIGMOD, pp. 393–404 (2015)
7. Sibeyn, J.F., Abello, J., Meyer, U.: Heuristics for semi-external depth first search on directed graphs. In: SPAA, pp. 282–292 (2002)
8. Zhang, Z., Qin, L., Yu, J.X.: Contract & expand: I/O efficient sccs computing. In: ICDE, pp. 208–219 (2014)
9. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. OSDI **12**, 31–46 (2012)
10. Zhu, X., Han, W., Chen, W.: GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: USENIX Annual Technical Conference, pp. 375–386 (2015)
11. Roy, A., Bindschaedler, L., Malicevic, J., Zwaenepoel, W.: Chaos: Scale-out graph processing from secondary storage. In: SOSP, pp. 410–424 (2015)
12. Maheshwari, A., Zeh, N.: A survey of techniques for designing I/O-efficient algorithms. In: Algorithms for Memory Hierarchies, pp. 36–61. Springer, Berlin (2003)
13. Hutchinson, D.A., Maheshwari, A., Zeh, N.: An external memory data structure for shortest path queries. In: COCOON, pp. 51–60 (1999)
14. Hu, X., Tao, Y., Chung, C.: Massive graph triangulation. In: SIGMOD, pp. 325–336 (2013)
15. Chu, S., Cheng, J.: Triangle listing in massive networks and its applications. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, 21–24 August 2011, pp. 672–680 (2011)
16. Tao, Y., Papadopoulos, S., Sheng, C., Stefanidis, K.: Nearest keyword search in xml documents. In: SIGMOD, pp. 589–600 (2011)
17. Bender, M.A., Farach-colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) In Latin American Theoretical Informatics, pp. 88–94. Springer, Berlin (2000)
18. Michalis, P., Bonchi, F., Castillo, C., Gionis, A.: Fast shortest path distance estimation in large networks. In: CIKM, pp. 867–876 (2009)
19. Jon, K., Slivkins, A., Wexler, T.: Triangulation and embedding using small sets of beacons. In: IEEE Symposium on Foundations of Computer Science, pp. 444–453 (2004)
20. Vieira, M.V., Fonseca, B.M., Damazio, R., Golgher, P.B., Reis, D.D.C., Ribeiro-Neto, B.: Efficient search ranking in social networks. In: CIKM, pp. 563–572 (2007)
21. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
22. Sarma, A.D., Gollapudi, S., Najork, M., Panigrahy, R.: A sketch-based distance oracle for web-scale graphs. In: WSDM, pp. 401–410 (2010)
23. Spearman, C.: The proof and measurement of association between two things. Am. J. Psychol. **15**(1), 72–101 (1904)
24. Hristidis, V., Papakonstantinou, Y.: Discover: keyword search in relational databases. In: VLDB, pp. 670–681 (2002)
25. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using banks. In: ICDE, pp. 431–440 (2002)
26. Golenberg, K., Kimelfeld, B., Sagiv, Y.: Keyword proximity search in complex data graphs. In: SIGMOD, pp. 927–940 (2008)
27. Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L.: Ease: efficient and adaptive keyword search on unstructured, semi-structured and structured data. In: SIGMOD, pp. 903–914 (2008)
28. Qin, L., Yu, J.X., Chang, L., Tao, Y.: Querying communities in relational databases. In: ICDE, pp. 724–735 (2009)
29. Kargar, M., An, A.: Keyword search in graphs: finding r-cliques. PVLDB **4**(10), 681–692 (2011)
30. Yu, J.X., Qin, L., Chang, L.: Keyword search in relational databases: a survey. IEEE Data Eng. Bull. **33**(1), 67–78 (2010)
31. Yao, B., Tang, M., Li, F.: Multi-approximate-keyword routing in gis data. In: GIS, pp. 201–210 (2011)
32. Cao, X., Chen, L., Cong, G., Xiao, X.: Keyword-aware optimal route search. PVLDB **5**(11), 1136–1147 (2012)
33. Tao, Y.: A dynamic I/O-efficient structure for one-dimensional top-k range reporting. In: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, 22–27 June 2014, pp. 256–265 (2014)