This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

Nearest Keyword Set Search in Multi-dimensional Datasets

Vishwakarma Singh, Bo Zong, Ambuj K. Singh

Abstract—Keyword-based search in text-rich multi-dimensional datasets facilitates many novel applications and tools. In this paper, we consider objects that are tagged with keywords and are embedded in a vector space. For these datasets, we study queries that ask for the tightest groups of points satisfying a given set of keywords. We propose a novel method called ProMiSH (Projection and Multi Scale Hashing) that uses random projection and hash-based index structures, and achieves high scalability and speedup. We present an exact and an approximate version of the algorithm. Our experimental results on real and synthetic datasets show that ProMiSH has up to 60 times of speedup over state-of-the-art tree-based techniques.

Index Terms—Querying, Multi-dimensional data, Indexing, Hashing

1 Introduction

Objects (*e.g.,* images, chemical compounds, documents, or experts in collaborative networks) are often characterized by a collection of relevant features, and are commonly represented as points in a multi-dimensional feature space. For example, images are represented using color feature vectors, and usually have descriptive text information (*e.g.,* tags or keywords) associated with them. In this paper, we consider multi-dimensional datasets where each data point has a set of keywords. The presence of keywords in feature space allows for the development of new tools to query and explore these multi-dimensional datasets.

In this paper, we study *nearest keyword set* (referred to as NKS) queries on text-rich multi-dimensional datasets. An NKS query is a set of user-provided keywords, and the result of the query may include k sets of data points each of which contains all the query keywords and forms one of the top-k tightest cluster in the multi-dimensional space. Fig. 1 illustrates an NKS query over a set of 2-dimensional data points. Each point is tagged with a set of keywords. For a query $Q = \{a, b, c\}$, the set of points $\{7, 8, 9\}$ contains all the query keywords $\{a, b, c\}$ and forms the tightest cluster compared with any other set of points covering all the query keywords. Therefore, the set $\{7, 8, 9\}$ is the top-1 result for the query Q.

NKS queries are useful for many applications, such as photo-sharing in social networks, graph pattern search, geo-location search in GIS systems[1] [1], [2], and so on. The following are a few examples.

1) Consider a photo-sharing social network (*e.g.,* Facebook), where photos are tagged with people names and locations. These photos can be embedded in a high-dimensional feature space of texture, color, or shape [3], [4]. Here an NKS query can find a group of similar photos which contains a set of people.
2) NKS queries are useful for graph pattern search, where labeled graphs are embedded in a high dimensional space (*e.g.,* through Lipschitz embedding [5]) for scalability. In this case, a search for a subgraph with a set of specified labels can be answered by an NKS query in the embedded space [6].
3) NKS queries can also reveal geographic patterns. GIS can characterize a region by a high-dimensional set of attributes, such as pressure, humidity, and soil types. Meanwhile, these regions can also be tagged with information such as diseases. An epidemiologist can formulate NKS queries to discover patterns by finding a set of similar regions with all the diseases of her interest.

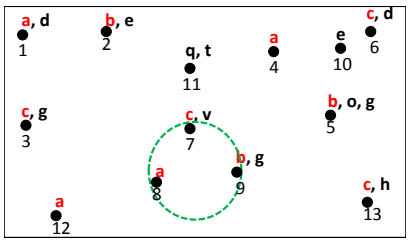


Fig. 1. An example of an NKS query on a keyword tagged multi-dimensional dataset. The top-1 result for query $\{a, b, c\}$ is the set of points $\{7, 8, 9\}$.

We formally define NKS queries as follows.

Nearest Keyword Set. Let $\mathcal{D} \subset \mathcal{R}^d$ be a d-dimensional dataset with N points. For any $o \in \mathcal{D}$, it is tagged with a set of keywords $\sigma(o)$=$\{v_1, .., v_t\} \subseteq \mathcal{V}$, where $\mathcal{V}$ is a dictionary of U unique keywords. For any $o_i, o_j \in \mathcal{D}$, the distance between o_i and o_j is measured by their L_2-norm (*i.e.,* Euclidean distance) as $dist(o_i, o_j) = \|o_i - o_j\|_2$. Given a set of data points $A \subset \mathcal{D}$, $r(A)$ is the diameter A and is defined by the maximum distance between any two points in A,

$$r(A) = \max_{\forall o_i, o_j \in A} \|o_i - o_j\|_2.$$

- *Vishwakarma Singh is an alumnus of the Department of Computer Science, University of California, Santa Barbara.*
  E-mail: vsingh014@gmail.com
- *Bo Zong is with the Department of Computer Science, University of California, Santa Barbara.*
  E-mail: bzong@cs.ucsb.edu
- *Ambuj K. Singh is with the Department of Computer Science, University of California, Santa Barbara.*
  E-mail: ambuj@cs.ucsb.edu

1. http://www.geabios.com

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

$\mathcal{D}$: A dataset	$\mathcal{V}$: A dictionary of unique keywords in $\mathcal{D}$	Q : A set of keywords comprising a query
o : A point in $\mathcal{D}$	v : A keyword	B : Hashtable size
N : Number of points in $\mathcal{D}$	U : Number of unique keywords in $\mathcal{D}$	q : Number of keywords in query Q
d : Number of dimensions of a point	t : Average number of keywords per point	k : Number of top results
w_0 : Initial bin-width for hashtable	m : Number of unit random vectors used for projection	L : Index level
s : A scale value	r : Diameter of a set of points	z : A d-dimensional unit random vector

TABLE 1
A glossary of notations used in the paper.

A smaller $r(A)$ implies the points in A are more similar to each other.

Given an NKS query with q keywords $Q = \{v_1, \cdots, v_q\}$, $A \subseteq \mathcal{D}$ is a candidate result of Q if it covers all the keywords in Q by $Q \subseteq \bigcup_{o \in A} \sigma(o)$. Let $\mathcal{S}$ be the set including all candidates of Q. The top-1 result A^* of Q is obtained by

$$A^* = \arg\min_{A \in \mathcal{S}} r(A).$$

Similarly, a top-k NKS query retrieves the top-k candidates with the least diameter. If two candidates have equal diameters, then they are further ranked by their cardinality.

Although existing techniques using tree-based indexes [2], [7], [8], [9] suggest possible solutions to NKS queries on multi-dimensional datasets, the performance of these algorithms deteriorates sharply with the increase of size or dimensionality in datasets. Our empirical results show that these algorithms may take hours to terminate for a multi-dimensional dataset of millions of points. Therefore, there is a need for an efficient algorithm that scales with dataset dimension, and yields practical query efficiency on large datasets.

In this paper, we propose ProMiSH (short for Projection and Multi-Scale Hashing) to enable fast processing for NKS queries. In particular, we develop an exact ProMiSH (referred to as ProMiSH-E) that always retrieves the optimal top-k results, and an approximate ProMiSH (referred to as ProMiSH-A) that is more efficient in terms of time and space, and is able to obtain near-optimal results in practice. ProMiSH-E uses a set of hashtables and inverted indexes to perform a localized search. The hashing technique is inspired by Locality Sensitive Hashing (LSH) [10], which is a state-of-the-art method for nearest neighbor search in high-dimensional spaces. Unlike LSH-based methods that allow only approximate search with probabilistic guarantees, the index structure in ProMiSH-E supports accurate search. ProMiSH-E creates hashtables at multiple bin-widths, called index levels. A single round of search in a hashtable yields subsets of points that contain query results, and ProMiSH-E explores each subset using a fast pruning-based algorithm. ProMiSH-A is an approximate variation of ProMiSH-E for better time and space efficiency. We evaluate the performance of ProMiSH on both real and synthetic datasets and employ state-of-the-art VbR*-Tree [2] and CoSKQ [8] as baselines. The empirical results reveal that ProMiSH consistently outperforms the baseline algorithms with up to 60 times of speedup, and ProMiSH-A is up to 16 times faster than ProMiSH-E obtaining near-optimal results.

Our main contributions are summarized as follows. (1) We propose a novel multi-scale index for exact and approximate NKS query processing. (2) We develop efficient search algorithms that work with the multi-scale indexes for fast query processing. (3) We conduct extensive experimental studies to demonstrate the performance of the proposed techniques.

The paper is organized as follows. We start with the related work in Section 2. Next, we present the index structure for exact search (ProMiSH-E) in Section 3, the exact search algorithm in Section 4, and its optimization techniques in Section 5. In addition, we introduce the approximate algorithm (ProMiSH-A) and provide an analysis for its approximation ratio in Section 6. The time and space complexity for ProMiSH are analyzed in Section 7. Experimental results are presented in Section 8. Finally, We conclude this paper with future work in Section 9. A glossary of the notations is shown in Table 1.

2 Related Work

A variety of related queries have been studied in literature on text-rich spatial datasets.

Location-specific keyword queries on the web and in the GIS systems [11], [12], [13], [14] were earlier answered using a combination of R-Tree [15] and inverted index. Felipe et al. [16] developed IR^2-Tree to rank objects from spatial datasets based on a combination of their distances to the query locations and the relevance of their text descriptions to the query keywords. Cong et al. [17] integrated R-tree and inverted file to answer a query similar to Felipe et al. [16] using a different ranking function. Martins et al. [18] computed text relevancy and location proximity independently, and then combined the two ranking scores. Cao et al. [7] and Long et al. [8] proposed algorithms to retrieve a group of spatial web objects such that the group's keywords cover the query's keywords and the objects in the group are nearest to the query location and have the lowest inter-object distances. Other related queries include aggregate nearest keyword search in spatial databases [19], top-k preferential query [20], top-k sites in a spatial data based on their influence on feature points [21], and optimal location queries [22], [23].

Our work is different from these techniques. First, existing works mainly focus on the type of queries where the coordinates of query points are known [7], [8]. Even though it is possible to make their cost functions same to the cost function in NKS queries, such tuning does not change their techniques. The proposed techniques use location information as an integral part to perform a best-first search on the IR-Tree, and query coordinates play a fundamental role in almost every step of the algorithms to prune the search space. Moreover, these techniques do not provide concrete guidelines on how to enable efficient processing for the type of queries where query coordinates are missing. Second, in multi-dimensional spaces, it is difficult for users to provide meaningful coordinates, and our work deals with another type of queries where users can

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

only provide keywords as input. Without query coordinates, it is difficult to adapt existing techniques to our problem. Note that a simple reduction that treats the coordinates of each data point as possible query coordinates suffers poor scalability. Third, we develop a novel index structure based on random projection with hashing. Unlike tree-like indexes adopted in existing works, our index is less sensitive to the increase of dimensions and scales well with multi-dimensional data.

Another track of related works deal with m-closest keywords queries [9]. In [9], bR*-Tree is developed based on a R*-tree [24] that stores bitmaps and minimum bounding rectangles (MBRs) of keywords in every node along with points MBRs. The candidates are generated by the *apriori* algorithm [25]. Unwanted candidates are pruned based on the distances between MBRs of points or keywords and the best found diameter. However, the pruning techniques become ineffective with an increase in the dataset dimension as there is a large overlap between MBRs due to the curse of dimensionality. This leads to an exponential number of candidates and large query times. A poor estimation of starting diameter further worsens the performance of their algorithm. bR*-Tree also suffers from a high storage cost; therefore, Zhang et al. modified bR*-Tree to create Virtual bR*-Tree [2] in memory at run time. Virtual bR*-Tree is created from a pre-stored R*-Tree, which indexes all the points, and an inverted index which stores keyword information and path from the root node in R*-Tree for each point. Both bR*-Tree and Virtual bR*-Tree, are structurally similar, and use similar candidate generation and pruning techniques. Therefore, Virtual bR*-Tree shares similar performance weaknesses as bR*-Tree.

Tree-based indexes, such as R-Tree [15] and M-Tree [26], have been extensively investigated for nearest neighbor search in high-dimensional spaces. These indexes fail to scale to dimensions greater than 10 because of the curse of dimensionality [27]. Random projection [28] with hashing [10], [29], [30], [31], [32] has come to be the state-of-the-art method for nearest neighbor search in high-dimensional datasets. Datar et al. [10] used random vectors constructed from p-stable distributions to project points, computed hash keys for the points by splitting the line of projected values into disjoint bins, and then concatenated hash keys obtained for a point from m random vectors to create a final hash key for the point. Our problem is different from nearest neighbor search. NKS queries provide no coordinate information, and aim to find the top-k tightest clusters that cover the input keyword set. Meanwhile, nearest neighbor queries usually require coordinate information for queries, which makes it difficult to develop an efficient method to solve NKS queries by existing techniques for nearest neighbor search.

In addition, multi-way distance joins for a set of multi-dimensional datasets have been studied in [33], [34]. Tree-based index is adopted, but suffers poor scalability with respect to the dimension of the dataset. Furthermore, it is not straightforward to adapt these algorithms since every query requires a multi-way distance join only on a subset of the points of each dataset.

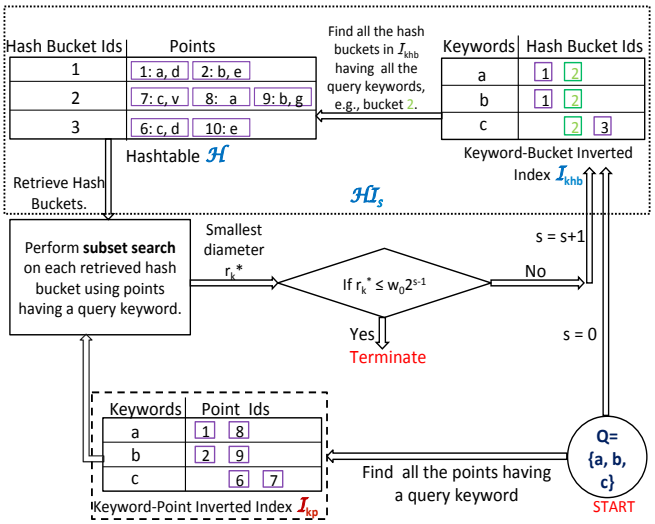


Fig. 2. Index structure and flow of execution of ProMiSH.

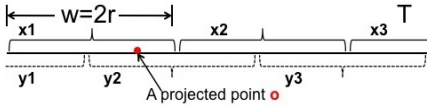


Fig. 3. The projection space (a segment) of a random unit vector is partitioned into overlapping bins of equal width: point o is included in bin $x1$ and $y2$.

3 Index Structure for Exact ProMiSH

We start with the index for exact ProMiSH (ProMiSH-E). This index consists of two main components.

Inverted Index $\mathcal{I}_{kp}$. The first component is an inverted index referred to as $\mathcal{I}_{kp}$. In $\mathcal{I}_{kp}$, we treat keywords as keys, and each keyword points to a set of data points that are associated with the keyword. Let $\mathcal{D}$ be a set of data points and $\mathcal{V}$ be a dictionary that contains all the keywords appearing in $\mathcal{D}$. We build $\mathcal{I}_{kp}$ for $\mathcal{D}$ as follows. (1) For each $v \in \mathcal{V}$, we create a key entry in $\mathcal{I}_{kp}$, and this key entry points to a set of data points $\mathcal{D}_v = \{o \in \mathcal{D} \mid v \in \sigma(o)\}$ (*i.e.,* a set includes all data points in $\mathcal{D}$ that contain keyword v). (2) We repeat (1) until all the keywords in $\mathcal{V}$ are processed. In Fig. 2, an example for $\mathcal{I}_{kp}$ is shown in the dashed rectangle at the bottom.

Hashtable-Inverted Index Pairs $\mathcal{HI}$. The second component consists of multiple hashtables and inverted indexes referred to as $\mathcal{HI}$. $\mathcal{HI}$ is controlled by three parameters: (1) (*Index level*) L, (2) (*Number of random unit vectors*) m, and (3) (*hashtable size*) B. All the three parameters are non-negative integers. Next, we describe how these three parameters control the construction of $\mathcal{HI}$.

In general, $\mathcal{HI}$ contains L hashtable-inverted index pairs, characterized by $\{(\mathcal{H}^{(s)}, \mathcal{I}_{khb}^{(s)}) \mid s \in \{0, 1, 2, ..., L-1\}\}$, where $\mathcal{H}^{(s)}$ and $\mathcal{I}_{khb}^{(s)}$ are the s-th hashtable and inverted index, respectively.

First, given a set of d-dimensional data points $\mathcal{D}$, we create hashtable $\mathcal{H}^{(s)}$ as follows.

1) We randomly sample m d-dimensional unit vectors $z_1, z_2, \cdots, z_m$ (*i.e.,* $\|z_i\|_2 = 1$ for $i = 1, 2, \cdots, m$);
2) For each $o \in \mathcal{D}$, we compute its projection on each of the unit vectors $o_{z_i} = o \cdot z_i$ where $i = 1, 2, \cdots, m$;

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

3) Let $pMax$ be the maximum projected value for data points in $\mathcal{D}$ on any of the m random unit vectors and $w_0 = \frac{pMax}{2^L}$. For each z_i, we consider its projection space as a segment $[0, pMax]$, and partition the segment into $2^{(L-s+1)}+1$ overlapping bins, where each bin has width $w = w_0 2^s$ and is equally overlapped with two other bins as shown in Fig. 3. We conduct the projection space partition on all the m random unit vectors.
4) For each z_i and $o \in \mathcal{D}$, since its projection space is partitioned into overlapping bins, o_{z_i} falls into two bins; therefore, we get two bin ids $\{b_1(o, z_i), b_2(o, z_i)\}$, and we can compute $b_1(o, z_i)$ and $b_2(o, z_i)$ as below,

$$b_1(o, z_i) = \lfloor \frac{o_{z_i}}{w} \rfloor, \quad (1)$$

$$b_2(o, z_i) = \lfloor \frac{o_{z_i} - \frac{w}{2}}{w} \rfloor + 2^{L-s}. \quad (2)$$

Fig. 3 shows an example where we partition the project space into overlapping bins $\{x1, x2, x3, y1, y2, y3\}$, and point o lies in bins $x1$ and $y2$.
5) For each $o \in \mathcal{D}$, we generate its signatures based on the bins into which its projections on random unit vectors fall. With m random unit vectors, we obtain m pairs of bin ids for each data point o. Next, we take a cartesian product over these m pairs of bin ids and generate 2^m signatures for each point o, where each signature $\{b_{i_1}(o, z_1), \cdots, b_{i_m}(o, z_m)\}$ contains a bin id from each of the m pairs. For example, let z_1 and z_2 be two random unit vectors, and the bin ids for a point o be $\{x1, y1\}$ from z_1 and $\{x2, y2\}$ from z_2. We create four signatures as $\{x1, x2\}$, $\{x1, y2\}$, $\{y1, x2\}$, and $\{y1, y2\}$.
6) For each point $o \in \mathcal{D}$, we hash it into 2^m buckets in $\mathcal{H}^{(s)}$ using its 2^m signatures. For each signature $\{b_{i_1}(o, z_1), \cdots, b_{i_m}(o, z_m)\}$, we convert it into a hashtable bucket id by a standard hash function,$(\sum_{j=1}^{m} b_{i_j} \cdot pr_j)\%B$, where B is hashtable size (*i.e.,* the number of buckets in $\mathcal{H}^{(s)}$) and pr_j is a random prime number.

Second, given a dictionary $\mathcal{V}$ and hashtable $\mathcal{H}^{(s)}$, we create the inverted index $\mathcal{I}_{khb}^{(s)}$. In this inverted index, keys are still keywords. For each $v \in \mathcal{V}$, v points to a set of buckets, each of which contains at least one data point o such that $v \in \sigma(o)$. Fig. 2 demonstrates an example about $\mathcal{HI}$ with one pair of hashtable and inverted index shown in the dotted rectangle.

In next section, we show how to conduct exact search using ProMiSH-E.

4 Search Algorithm for ProMiSH-E

In this section, we present the search algorithms in ProMiSH-E that finds top-k results for NKS queries. First, we introduce two lemmas that guarantee ProMiSH-E always retrieves the optimal top-k results. Then, we describe the details in ProMiSH-E.

We start with some theoretic results for ProMiSH-E.

Lemma 1: Let $\mathcal{R}^d$ be a d-dimensional Euclidean space and z be a random unit vector uniformly sampled from $\mathcal{R}^d$ such that $\|z\|_2 = 1$. For any two points $o_1 \in \mathcal{R}^d$ and $o_2 \in \mathcal{R}^d$, we have $\|o_1 - o_2\|_2 \geq \|o_{1z} - o_{2z}\|_2$, where o_{1z} and o_{2z} are the projection of o_1 and o_2 on z, respectively.

Proof: Since Euclidean space with dot product is an inner product space, we have

$$\begin{aligned} \|o_{1z} - o_{2z}\|_2 &= |(o_1 - o_2)_z| \\ &\leq \|z\|_2 \cdot \|o_1 - o_2\|_2 \\ &= \|o_1 - o_2\|_2, \text{ since } \|z\|_2 = 1 \end{aligned}$$

The inequality follows the Cauchy-Schwarz inequality. □

Lemma 2: Given $A = \{o_1, \cdots, o_n\} \subset \mathcal{R}^d$ with diameter r is projected onto a d-dimensional random unit vector z and the projection space of z is partitioned into overlapping bins with equal width w, there exists at least one bin containing all the points in A if $w \geq 2r$.

Proof: According to Lemma 1 and the definition of diameter, we have $\forall o_i, o_j \in A, |o_{iz} - o_{jz}| \leq \|o_i - o_j\|_2 \leq r$. Thus, we can further derive $\max(o_{1z}, \cdots, o_{nz}) - \min(o_{1z}, \cdots, o_{nz}) \leq r$. Since the projection space of z is partitioned into overlapping bins of width $w \geq 2r$, it follows from the construction that any line segment of width r is fully contained in one of the bins as shown in Fig. 3. Hence, all the points in A will fall into the same bin. □

We use an example to show how Lemma 2 guarantees the retrieval of the optimal top-1 results. Given a query Q, we assume the diameter of its top-1 result is r. We project all the data points in $\mathcal{D}$ on a unit random vector and partition the projected values into overlapping bins of bin-width $w \geq 2r$. If we perform a search in each of the bins independently, then Lemma 2 guarantees that the top-1 result of query Q will be found in one of the bins.

Based on Lemma 1 and Lemma 2, we propose ProMiSH-E as shown in Fig. 2. A search starts with the $\mathcal{HI}$ structure at index level $s = 0$. ProMiSH-E finds the buckets in hashtable $\mathcal{H}^{(0)}$, each of which contains all the query keywords by inverted index $\mathcal{I}_{khb}^{(0)}$. Then, ProMiSH-E explores each selected bucket using an efficient pruning based technique to generate results. ProMiSH-E terminates after exploring $\mathcal{HI}$ structure at the smallest index level s such that all the top-k results have been found.

Algorithm 1 details the steps in ProMiSH-E. It maintains a bitset BS. For each $v_Q \in Q$, ProMiSH-E retrieves the list of points corresponding to v_Q from $\mathcal{I}_{kp}$ in step 4. For each point o in the retrieved list, ProMiSH-E marks the bit corresponding to o's identifier in BS as true in step 5. Thus, ProMiSH-E finds all the points in $\mathcal{D}$ which are tagged with at least one query keyword. Next, the search continues in the $\mathcal{HI}$ structures, beginning at $s = 0$. For any given index level s, ProMiSH-E works with $\mathcal{H}^{(s)}$ and $\mathcal{I}_{khb}^{(s)}$ in $\mathcal{HI}$ at step 8. ProMiSH-E retrieves all the lists of hash bucket ids corresponding to keywords in Q from the inverted index $\mathcal{I}_{khb}^{(0)}$ at steps (10-11). An intersection of these lists yields a set of hash buckets each of which contains all the query keywords in steps (12-16) (*e.g.,* In Fig. 2, this intersection yields the bucket id 2). For each selected hash bucket, ProMiSH-E retrieves all the points in the bucket from the hashtable $\mathcal{H}$, and filters these points using bitset BS to get a subset of points F' in steps (17-22). Subset F' contains only those points which are tagged with at least

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

Algorithm 1 ProMiSH-E

In: Q: query keywords; k: number of top results
In: w_0: initial bin-width
1: $PQ \leftarrow [e([\,], +\infty)]$: priority queue of top-$k$ results
2: HC: hashtable to check duplicate candidates
3: BS : bitset to track points having a query keyword
4: **for all** $o \in \cup_{\forall v_Q \in Q} \mathcal{I}_{kp}[v_Q]$ **do**
5: $BS[o] \leftarrow$ true /* Find points having query keywords*/
6: **end for**
7: **for all** $s \in \{0, ..., L-1\}$ **do**
8: Get $\mathcal{HI}$ at s
9: $E[\,] \leftarrow 0$ /* List of hash buckets */
10: **for all** $v_Q \in Q$ **do**
11: **for all** $bId \in \mathcal{I}_{khb}[v_Q]$ **do**
12: $E[bId] \leftarrow E[bId] + 1$
13: **end for**
14: **end for**
15: **for all** $i \in (0, ..., SizeOf(E))$ **do**
16: **if** $E[i] = SizeOf(Q)$ **then**
17: $F' \leftarrow \emptyset$ /* Obtain a subset of points */
18: **for all** $o \in \mathcal{H}[i]$ **do**
19: **if** $BS[o] =$ true **then**
20: $F' \leftarrow F' \cup o$
21: **end if**
22: **end for**
23: **if** checkDuplicateCand(F', HC) = false **then**
24: searchInSubset(F', PQ)
25: **end if**
26: **end if**
27: **end for**
28: /* Check termination condition */
29: **if** $PQ[k].r \leq w_0 2^{s-1}$ **then**
30: Return PQ
31: **end if**
32: **end for**
33: /* Perform search on $\mathcal{D}$ if algorithm has not terminated */
34: **for all** $o \in \mathcal{D}$ **do**
35: **if** $BS[o] =$ true **then**
36: $F' \leftarrow F' \cup o$
37: **end if**
38: **end for**
39: searchInSubset(F', PQ)
40: Return PQ

one query keyword and is explored further.

Subset F' is checked whether it has been explored earlier or not using *checkDuplicateCand* (Algorithm 2) in step 23. Since each point is hashed using 2^m signatures, duplicate subsets may be generated. If F' has not been explored earlier, then ProMiSH-E performs a search on it using *searchInSubset* (Algorithm 3) at step 24 (We discuss in this algorithm in detail in Section 5). Results are inserted into a priority queue PQ of size k. Each entry of PQ is a tuple containing a set of points and their diameter. PQ is initialized with k entries, each of whose set is empty and the diameter is $+\infty$. Entries of PQ are ordered by their diameters, and entries with equal diameters are further ordered by their set sizes. A new result is inserted into PQ only if its diameter is smaller than the k-th smallest diameter in PQ. If ProMiSH-E does not terminate after exploring the $\mathcal{HI}$ structure at index level s, then the search proceeds to $\mathcal{HI}$ at index level $(s+1)$.

ProMiSH-E terminates when the k-th smallest diameter r_k has been found during steps (29-31). Since $r_k \leq \frac{w_0 2^s}{2}$, Lemma 2 guarantees that all the possible candidates are fully contained in one of the bins of the hashtable, and therefore, must have been explored. If ProMiSH-E fails to terminate after exploring $\mathcal{HI}$ at all the index levels $s \in \{0, ..., L-1\}$, then it performs a search in the complete dataset $\mathcal{D}$ during steps (34-39).

Algorithm 2 checkDuplicateCand

In: F': a subset; HC: hashtable of subsets
1: $F' \leftarrow sort(F')$
2: $pr1$: list of prime numbers; $pr2$: list of prime numbers;
3: **for all** $o \in F'$ **do**
4: $pr_1 \leftarrow$ randomSelect($pr1$); $pr_2 \leftarrow$ randomSelect($pr2$)
5: $h_1 \leftarrow h_1 + (o \times pr_1)$; $h_2 \leftarrow h_2 + (o \times pr_2)$
6: **end for**
7: $h \leftarrow h_1 h_2$;
8: **if** isEmpty($HC[h]$)=false **then**
9: **if** elementWiseMatch(F', $HC[h]$) = true **then**
10: Return true;
11: **end if**
12: **end if**
13: $HC[h]$.add(F');
14: Return false;

Algorithm *checkDuplicateCand* (Algorithm 2) uses a hashtable HC to check duplicates for a subset F'. Points in F' are sorted by their identifiers. Two separate standard hash functions are applied to the identifiers of the points in the sorted order to generate two hash values in steps (2-6). Both of the hash values are concatenated to get a hash key h for the subset F' in step 7. The use of multiple hash functions helps to reduce hash collisions. If HC already has a list of subsets at h, then an element-wise match of F' is performed with each subset in the list in steps (8-9). Otherwise, F' is stored in HC using key h in step 13.

As shown in Algorithm 1, the efficiency of ProMiSH-E highly depends on an efficient search algorithm that finds top-k results from a subset of data points. In next section, we propose a search algorithm that provides such efficiency.

5 Search in a Subset of Data Points

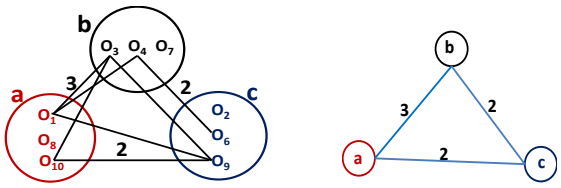
We present an algorithm for finding top-k tightest clusters in a subset of points. A subset is obtained from a hashtable bucket as explained in section 4. Points in the subset are grouped based on the query keywords. Then, all the promising candidates are explored by a multi-way distance join of these groups. The join uses r_k, the diameter of the kth result obtained so far by ProMiSH-E, as the distance threshold.

We explain a multi-way distance join with an example. A multi-way distance join of q groups $\{g_1, ..., g_q\}$ finds all the tuples $\{o_{1,i}, ..., o_{x,j}, o_{y,k}, ..., o_{q,l}\}$ such that $\forall x, y$: $o_{x,j} \in g_x$, $o_{y,k} \in g_y$, and $||o_{x,j} - o_{y,k}||_2 \leq r_k$. Figure 4(a) shows groups $\{a, b, c\}$ of points obtained for a query Q=$\{a, b, c\}$ from a subset F'. We show an edge between a pair of points of two groups if the distance between the points is at most r_k, e.g, an edge between point o_1 in group a and point o_3 in group b. A multi-way distance join of these groups finds tuples $\{o_1, o_3, o_9\}$ and $\{o_{10}, o_3, o_9\}$. Each tuple obtained by a multi-way join is a promising candidate for a query.

5.1 Group Ordering

A suitable ordering of the groups leads to an efficient candidate exploration by a multi-way distance join. We first perform a pairwise inner joins of the groups with distance threshold r_k. In inner join, a pair of points from two groups are joined only

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering



(a) Pairwise inner joins (b) A graph representation

Fig. 4. (a) a, b, and c are groups of points of a subset F' obtained for a query Q=$\{a, b, c\}$. A point o in a group g is joined to a point o' in another group g' if $||o-o'|| \leq r_k$. The groups in the order $\{a, c, b\}$ generates the least number of candidates by a multi-way join. (b) A graph of pairwise inner joins. Each group is a node in the graph. The weight of an edge is the number of point pairs obtained by an inner join of the corresponding groups.

if the distance between them is at most r_k. Figure 4(a) shows such a pairwise inner joins of the groups $\{a, b, c\}$. We see from figure 4(a) that a multi-way distance join in the order $\{a, b, c\}$ explores 2 true candidates $\{\{o_1, o_3, o_9\}, \{o_{10}, o_3, o_9\}\}$ and a false candidate $\{o_1, o_4, o_6\}$. A multi-way distance join in the order $\{a, c, b\}$ explores the least number of candidates 2. Therefore, a proper ordering of the groups leads to an effective pruning of false candidates. Optimal ordering of groups for the least number of candidates generation is NP-hard [35].

We propose a greedy approach to find the ordering of groups. We explain the algorithm with a graph in figure 4(b). Groups $\{a, b, c\}$ are nodes in the graph. The weight of an edge is the count of point pairs obtained by an inner join of the corresponding groups. The greedy method starts by selecting an edge having the least weight. If there are multiple edges with the same weight, then an edge is selected at random. Let the edge ac, with weight 2, be selected in figure 4(b). This forms the ordered set $(a-c)$. The next edge to be selected is the least weight edge such that at least one of its nodes is not included in the ordered set. Edge cb, with weight 2, is picked next in figure 4(b). Now the ordered set is $(a-c-b)$. This process terminates when all the nodes are included in the set. $(a-c-b)$ gives the ordering of the groups.

Algorithm 3 shows how the groups are ordered. The kth smallest diameter r_k is retrieved form the priority queue PQ in step 1. For a given subset F' and a query Q, all the points are grouped using query keywords in steps (2-5). A pairwise inner join of the groups is performed in steps (6-18). An adjacency list AL stores the distance between points which satisfy the distance threshold r_k. An adjacency list M stores the count of point pairs obtained for each pair of groups by the inner join. A greedy algorithm finds the order of the groups in steps (19-30). It repeatedly removes an edge with the smallest weight from M till all the groups are included in the order set $curOrder$. Finally, groups are sorted using $curOrder$ in step 30.

Algorithm 3 searchInSubset

In: F': subset of points; Q: query keywords; q: query size
In: PQ: priority queue of top-k results
1: $r_k \leftarrow PQ[k].r$ /* kth smallest diameter */
2: $SL \leftarrow [(v, [\,])]$: list of lists to store groups per query keyword
3: **for all** $v \in Q$ **do**
4: $SL[v] \leftarrow \{\forall o \in F' : o$ is tagged with $v\}$ /* form groups */
5: **end for**
6: /* Pairwise inner joins of the groups*/
7: AL: adjacency list to store distances between points
8: $M \leftarrow 0$: adjacency list to store count of pairs between groups
9: **for all** $(v_i, v_j) \in Q$ such that $i \leq q,\ j \leq q,\ i < j$ **do**
10: **for all** $o \in SL[v_i]$ **do**
11: **for all** $o' \in SL[v_j]$ **do**
12: **if** $||o - o'||_2 \leq r_k$ **then**
13: $AL[o, o'] \leftarrow ||o - o'||_2$
14: $M[v_i, v_j] \leftarrow M[v_i, v_j] + 1$
15: **end if**
16: **end for**
17: **end for**
18: **end for**
19: /* Order groups by a greedy approach */
20: $curOrder \leftarrow [\,]$
21: **while** $Q \neq \emptyset$ **do**
22: $(v_i, v_j) \leftarrow$ removeSmallestEdge(M)
23: **if** $v_i \notin curOrder$ **then**
24: $curOrder$.append(v_i); $Q \leftarrow Q \setminus v_i$
25: **end if**
26: **if** $v_j \notin curOrder$ **then**
27: $curOrder$.append(v_j); $Q \leftarrow Q \setminus v_j$
28: **end if**
29: **end while**
30: sort(SL, $curOrder$) /* order groups */
31: *findCandidates*(q, AL, PQ, Idx, SL, $curSet$, $curSetr$, r_k)

5.2 Nested Loops with Pruning

We perform a multi-way distance join of the groups by nested loops. For example, consider the set of points in figure 4. Each point $o_{a,i}$ of group a is checked against each point $o_{b,j}$ of group b for the distance predicate, i.e., $||o_{a,i} - o_{b,j}||_2 \leq r_k$. If a pair $(o_{a,i}, o_{b,j})$ satisfies the distance predicate, then it forms a tuple of size 2. Next, this tuple is checked against each point of group c. If a point $o_{c,k}$ satisfies the distance predicate with both the points $o_{a,i}$ and $o_{b,j}$, then a tuple $(o_{a,i}, o_{b,j}, o_{c,k})$ of size 3 is generated. Each intermediate tuple generated by nested loops satisfies the property that the distance between every pair of its points is at most r_k. This property effectively prunes false tuples very early in the join process and helps to gain high efficiency. A candidate is found when a tuple of size q is generated. If a candidate having a diameter smaller than the current value of r_k is found, then the priority queue PQ and the value of r_k are updated. The new value of r_k is used as distance threshold for future iterations of nested loops.

We find results by nested loops as shown in Algorithm 4 (*findCandidates*). Nested loops are performed recursively. An intermediate tuple $curSet$ is checked against each point of group $SL[Idx]$ in steps (2-23). First, it is determined using AL whether the distance between the last point in $curSet$ and a point o in $SL[Idx]$ is at most r_k in step 3. Then, the point o is checked against each point in $curSet$ for the distance predicate in steps (5-15). The diameter of $curSet$ is updated in steps (9-11). If a point o satisfies the distance predicate with each point of $curSet$, then a new tuple $newCurSet$ is formed in step 17 by appending o to $curSet$. Next, a recursive call is made to *findCandidates* on the next group $SL[Idx+1]$ with $newCurSet$ and $newCurSetr$. A candidate is found if $curSet$ has a point from every group. A result is inserted into PQ after checking for duplicates in steps (26-33). A duplicate check is done by a sequential match with the results in PQ. For a large value of k, a method similar to Algorithm 2 can be used for a duplicate check. If a new result gets inserted into

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

Algorithm 4 findCandidates

```
In: q: query size; SL: list of groups
In: AL: adjacency list of distances between points
In: PQ: priority queue of top-k results
In: Idx: group index in SL
In: curSet: an intermediate tuple
In: curSetr: an intermediate tuple's diameter
 1: if Idx ≤ q then
 2:   for all o ∈ SL[Idx] do
 3:     if AL[curSet[Idx-1], o] ≤ r_k then
 4:       newCurSetr ← curSetr
 5:       for all o' ∈ curSet do
 6:         dist ← AL[o, o']
 7:         if dist ≤ r_k then
 8:           flag ← true
 9:           if newCurSetr < dist then
10:             newCurSetr ← dist
11:           end if
12:         else
13:           flag ← false; break;
14:         end if
15:       end for
16:       if flag = true then
17:         newCurSet ← curSet.append(o)
18:         r_k ← findCandidates(q, AL, PQ, Idx+1, SL,
            newCurSet, newCurSetr, r_k)
19:       else
20:         Continue;
21:       end if
22:     end if
23:   end for
24:   return r_k
25: else
26:   if checkDuplicateAnswers(curSet, PQ) = true then
27:     return r_k
28:   else
29:     if curSetr < PQ[k].r then
30:       PQ.Insert([curSet, curSetr])
31:       return PQ[k].r
32:     end if
33:   end if
34: end if
```

PQ, then the value of r_k is updated in step 18.

6 Approximate Search: ProMiSH-A

In this section, we discuss the approximate version of ProMiSH referred to as ProMiSH-A. We start with the algorithm description of ProMiSH-A, and then analyze its approximation quality.

Algorithm overview. In general, ProMiSH-A is more time and space efficient than ProMiSH-E, and is able to obtain near-optimal results in practice. The index structure and the search method of ProMiSH-A are similar to ProMiSH-E; therefore, we only describe the differences between them.

The index structure of ProMiSH-A differs from ProMiSH-E in the way of partitioning projection space of random unit vectors. ProMiSH-A partitions projection space into non-overlapping bins of equal width, unlike ProMiSH-E which partitions projection space into overlapping bins. Therefore, each data point o gets one bin id from a random unit vector z in ProMiSH-A. Only one signature is generated for each point o by the concatenation of its bin ids obtained from each of the m random unit vectors. Each point is hashed into a hashtable using its signature.

The search algorithm in ProMiSH-A differs from ProMiSH-E in the termination condition. ProMiSH-A checks for a termination condition after fully exploring a hashtable at a given index level: It terminates if it has k entries with non-empty data point sets in its priority queue PQ.

Approximation quality analysis. In the following, we analyze the approximation quality for the top-1 result returned by ProMiSH-A. In particular, we use approximation ratio $\rho \geq 1$ as the metric to evaluate approximation quality. This ratio is defined as the ratio of the diameter of the result reported by ProMiSH-A r to the diameter of the optimal result r^*: $\rho = r/r^*$. Let $\mathcal{D}$ be a d-dimensional dataset and Q=$\{v_1, \cdots, v_q\}$ be an NKS query for top-1 result. Assume that each data point in $\mathcal{D}$ has t keywords, and each keyword is independently sampled by a uniform distribution over a dictionary $\mathcal{V}$ with U unique keywords. We define $f(v) = 1 - (1 - \frac{1}{U})^t$ as the probability that a data point has keyword $v \in \mathcal{V}$. Thus, we can estimate the expected number of points that have keyword v as $E[N_v] = Nf(v)$. To this end, the expected number of candidates for query Q in $\mathcal{D}$ is estimated by $N^q \prod_{i=1}^{q} f(v_i)$.

Let $g(r)$ be the probability that a candidate has a diameter no more than r. Then, the expected number of candidates for query Q with diameter no more than r is estimated by $N_r = g(r)N^q \prod_{i=1}^{q} f(v_i)$.

We index data points in $\mathcal{D}$ by ProMiSH-A, where each data point is projected onto m random unit vectors. The projection space of each random unit vector is partitioned into non-overlapping bins of equal width w. Let $Pr(A, r \mid w)$ be the conditional probability for random unit vectors that a candidate A of query Q having diameter r is fully contained within a bin with width w. For m independent unit random vectors, the joint probability that a candidate A is contained in a bin in each of the m vectors is $Pr(A, r \mid m)^m$, and the probability that no candidate of diameter r is retrieved by ProMiSH-A from the hashtable, created using m unit random vectors, is $(1 - Pr(A, r \mid w)^m)^{N_r}$. Let the diameter of the top-1 result of query Q be r^*. Then, the probability $P(r')$ of at least one candidate of any diameter r, where $r^* \leq r \leq r'$, being retrieved by ProMiSH-A is given by

$$P(r') = 1 - \prod_{r=r^*}^{r'} (1 - Pr(A, r \mid w)^m)^{N_r}. \quad (3)$$

For a given constant λ, $0 \leq \lambda \leq 1$, we can compute the smallest value of r' using Equation (3) such that $\lambda \leq P(r')$. The value ρ=$\frac{r'}{r^*}$ gives the approximation ratio of the results returned by ProMiSH-A with the probability λ.

We empirically computed ρ for queries of 3 keywords for different values of λ using this model. We used a 32-dimensional real dataset having 1 million points described in Section 8 for our investigation, and computed the values of N_r and $Pr(A, r \mid w)^2$, where we use 2 random unit vectors with bin-width of $w = 100$. In this way, we obtained the approximation ratio bound of $\rho = 1.4$ and $\rho = 1.5$ for $\lambda = 0.8$ and $\lambda = 0.95$, respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

7 Complexity Analysis of ProMiSH

In this section, we first analyze the query time complexity and index space complexity in ProMiSH. Then we discuss how ProMiSH prunes the search space.

7.1 Query time complexity

Given a set of d-dimensional data points $\mathcal{D}$, we assume data points are uniformly distributed in the buckets of a hashtable, and keywords of each data point are uniformly sampled from the dictionary.

Suppose $\mathcal{D}$ has N data points, each data point has t keywords, and the keywords are sampled from a dictionary of U unique keywords. Let N_v be the number of data points with keyword v. The expectation of N_v is computed as follows,

$$E[N_v] = \sum_{i=1}^{N}(1-(1-\frac{1}{U})^t) = N(1-(1-\frac{1}{U})^t).$$

Time complexity of ProMiSH-E. Let L be the index level applied in the index structure of ProMiSH-E, $\mathcal{H}^{(s)}$ be the hashtable at scale $s \in \{0, 1, \cdots, L-1\}$, B be hashtable size, and $N_{b,v}$ be the number of data points with keyword v lying in a bucket b among B buckets. Suppose ProMiSH-E applies m random unit vectors. Since ProMiSH-E generates 2^m signatures for each data point, the expectation of $N_{b,v}$ under uniformity assumptions is estimated as below,

$$E[N_{b,v}] = \frac{2^m}{B}E[N_v].$$

Searching a bucket b in $\mathcal{H}^{(s)}$ includes inner group joins and nested loops. Let q be the number of keywords in a query. First, inner group joins for d-dimensional data points are computed in $O(d(E[N_{b,v}])^2 + (qE[N_{b,v}])^2)$. Second, nested loops are computed in $O((E[N_{b,v}])^q)$. Thus, the total complexity of searching a bucket b is $O(d(E[N_{b,v}])^2 + (qE[N_{b,v}])^2 + (E[N_{b,v}])^q)$. In the worst case, we may need to check all the buckets at all scales; therefore, the overall complexity is $O(dLB(E[N_{b,v}])^2 + LB(qE[N_{b,v}])^2 + LB(E[N_{b,v}])^q)$.

Time complexity of ProMiSH-A. Let L be the index level applied in the index structure of ProMiSH-A, $\mathcal{H}^{(s)}$ be the hashtable at scale $s \in \{0, 1, \cdots, L-1\}$, B be hashtable size, and $N'_{b,v}$ be the number of data points with keyword v lying in a bucket b. Since ProMiSH-A only generates one signature per data point, the expectation of $N'_{b,v}$ under uniformity assumptions is estimated as

$$E[N'_{b,v}] = \frac{E[N_v]}{B}.$$

Similarly, we can derive the overall complexity of ProMiSH-A is $O(dLB(E[N'_{b,v}])^2 + LB(qE[N'_{b,v}])^2 + LB(E[N'_{b,v}])^q)$.

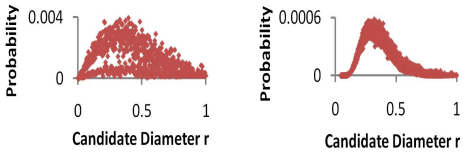
7.2 Index space complexity

Let N be the number of data points to index, d be the dimension of data points, t be the number of keywords per data point, U be dictionary size, m be the number of random unit vectors for point projection, L be index level, and B be hashtable size.

Space complexity of ProMiSH-E. The Indexes of ProMiSH-E includes keyword-point inverted index $\mathcal{I}_{kp}$, hashtable $\mathcal{H}$, and keyword-bucket inverted index $\mathcal{I}_{khb}$. First, we need $O(tN+U)$ space for $\mathcal{I}_{kp}$. Second, in the worst case, we need to include $O(2^mNL + BL)$ points in $\mathcal{H}$. Finally, $\mathcal{I}_{khb}$ takes $O(UBL)$ space. Thus, at index level L, the overall space complexity is $O(tN + 2^mNL + UBL)$.

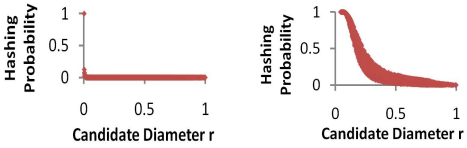
Space complexity of ProMiSH-A. The Indexes of ProMiSH-A also includes keyword-point inverted index $\mathcal{I}_{kp}$, hashtable $\mathcal{H}$, and keyword-bucket inverted index $\mathcal{I}_{khb}$. Unlike ProMiSH-E, $\mathcal{H}$ in ProMiSH-A takes at most $O(NL + BL)$. Thus, at index level L, the overall space complexity is $O(tN + NL + UBL)$.

7.3 Pruning intuition



(a) d=2 (b) d=16

Fig. 5. Candidate diameter distributions for queries with 3 keywords over a 2-dimensional and a 16-dimensional real datasets



(a) d=2 (b) d=16

Fig. 6. Distributions of $Pr(A, r|w)^2$ for candidates of queries with 3 keywords over a 2-dimensional and a 16-dimensional real datasets

Dataset Dimension d	2	4	8	16	32
Percentage ratio ($\frac{N_p}{N_Q}$)	0.007	0.3	5.8	22	47

TABLE 2
Ratios of the expected number of candidates N_p to the expected number of candidates N_Q

Let $\mathcal{D}$ be a d-dimensional dataset of N data points, U be dictionary size (*i.e.,* the number of unique keywords) in $\mathcal{D}$, L be index level used in ProMiSH, and Q=$\{v_1, v_2, \cdots, v_q\}$ be an NKS query of q keywords. For ease of demonstration, we assume each data point is associated with only one keyword.

Suppose node set $A^* \subset \mathcal{D}$ with diameter r^* is the top-1 result for query Q. Let $f(v)$ denote the probability that a data point has keyword v and $g(r)$ denote the probability that a candidate of Q has diameter no more than r. Given query Q, the expected number of candidates N_Q and the expected number of candidates $N_{Q,r}$ with diameter no more than r are calculated as follows,

$$N_Q = \prod_{i=1}^{q}(f(v_i)N), N_{Q,r} = g(r)N_Q.$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

We select all the points in $\mathcal{D}$ which contain at least one query keyword v_i, project these points on a random unit vector, and split the line of projected values into overlapping bins of equal width $w = 2r^*$. Let $Pr(A, r \mid w)$ be the conditional probability for random unit vectors that a candidate A with diameter r is fully contained within a bin of width w. For m independent random unit vectors, the probability that a candidate A is contained in a bin in each of the m vectors is $Pr(A, r \mid w)^m$. Ideally, the expected number of candidates explored by ProMiSH in a hashtable is

$$N_p = \sum_{i=0}^{L-1} Pr(A, r \mid w_i)^m \cdot N_Q.$$

We empirically measured keyword distribution $f(\cdot)$, $Pr(A, r \mid w)^m$, and the ratio of N_p to N_Q by real datasets of one million data points with varied dimensions (more details about the dataset are described in Section 8).

Candidate diameter distributions and the distributions of $Pr(A, r \mid w)^2$ are demonstrated in Fig. 5 and Fig. 6, where candidate diameters are scaled to between 0 and 1. We make following observations. (1) Candidate diameters follow a heavy-tailed distribution, which suggests a large number of candidates have diameters much larger than r^*. (2) The distributions of $Pr(A, r \mid w)^2$ decreases exponentially with candidate diameter, which implies that the candidates with diameter larger than r^* have much smaller chance of falling in a bin and being probed by ProMiSH, compared with A^*. Therefore, most of candidates with diameters larger than r^* are effectively pruned out by ProMiSH using its index.

Table 2 presents the ratios of N_p to N_Q. Each ratio is computed as an average of 50 random queries. We observe that ProMiSH prunes more than 99% and 50% of false candidates for $d = 2$ and $d = 32$, respectively.

8 Experimental Results

In this section, we evaluate the effectiveness and efficiency of ProMiSH by both real and synthetic data.

8.1 Setup

Dataset	Dataset size N	Dictionary size U	Keywords per point t
Real-1	10,000	5,661	12
Real-2	30,000	6,753	13
Real-3	50,000	7,101	13
Real-4	70,000	7,902	14
Real-5	1,000,000	24,874	11

TABLE 3
Statistics of datasets used in experiments

Dataset. Our evaluation employs real and synthetic datasets.

The real datasets are collected from photo-sharing websites. As discussed in Section 1, one of the applications in NKS queries is to find tight clusters of photos which contain all the keywords provided by a user in a photo-sharing social network. We crawl images with descriptive tags from Flickr[2], and then these images are transformed into grayscale. Let d be the desired dimensionality. We convert each image into a d-dimensional point by extracting its color histogram, and associate each data point with a set of keywords that are derived from its tags. In total, we collect five datasets (referred to as Real-1, Real-2, Real-3, Real-4, and Real-5) with up to 1 million data points. Their statistics are shown in Table 3.

We also generate synthetic datasets to evaluate the scalability of ProMiSH. In particular, the data generation process is governed by the following parameters: (1) Dimension d specifies the dimensionality of each data point; (2) Dataset size N indicates the total number of multi-dimensional points in a synthetic dataset; (3) Keywords per point t suggests the number of keywords for each data point; and (4) Dictionary size U denotes the total number of keywords in a dataset. For each data point, its coordinate in each dimension is randomly sampled between 0 and 10,000, and its keyword is randomly selected following a uniform distribution. We create multiple synthetic datasets to investigate how these parameters affect the performance of ProMiSH.

Query. We generate NKS queries for real and synthetic datasets. In general, the query generation process is controlled by two parameters: (1) Keywords per query q decides the number of keywords in each query; and (2) Dictionary size U indicates the total number of keywords in a target dataset. For a real dataset, the probability that a keyword will be sampled in a query is proportional to the keyword's frequency in the dataset. For a synthetic dataset, a keyword of a query is randomly sampled following a uniform distribution.

Implementation. In addition to the exact ProMiSH (ProMiSH-E) and the approximate ProMiSH (ProMiSH-A), we also implement Virtual bR*-Tree (VbR*-Tree) [2] and CoSKQ [7], [8] as baselines.

For VbR*-Tree, we fix the leaf node size to be 1,000 entries and other node size to be 100 entries, as it demonstrate the best performance under this parameter setting.

CoSKQ is designed to handle the type of queries with query coordinates. To adapt CoSKQ to our problem, we transform an NKS query into a set of CoSKQ queries. Given a data point from a target dataset and an NKS query, we build a CoSKQ query by using the coordinates of the data point and the keywords in the NKS query. To ensure the correctness, if a dataset has N data points, we have to build N CoSKQ queries that enumerate all possible query coordinates.

All the algorithms are implemented in C++ with GCC 4.8.2, and all the experiments are conducted on a server with Ubuntu 14.04, powered by an Intel Core i7-2620M 2.7GHz CPU and 64GB of RAM. Each experiment is repeated 10 times, and their average results from 100 queries are presented.

8.2 Effectiveness

We apply real datasets to demonstrate the effectiveness of ProMiSH-A. In particular, we use the metric *approximation ratio* [30], [32] for evaluation. Let Q be an NKS query, r_i be the i-th smallest diameter from the top-k results returned by ProMiSH-A, and r_i^* be the i-th smallest diameter returned by ProMiSH-E. The approximation ratio of ProMiSH-A with

2. http://www.flickr.com/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

respect to Q is defined by $\rho(Q) = \frac{1}{k}\sum_{i=1}^{k}\frac{r_i}{r_i^*}$. It is easy to see $\rho(Q) \geq 1$; moreover, the smaller $\rho(Q)$ is, the better the algorithm will be with respect to Q. In the following, we report the average approximation ratio (AAR), which is the mean of the approximation ratios over all evaluated queries.

Fig. 7 shows the effectiveness of ProMiSH-A under different input real data. In this set of experiments, the index parameters are fixed as $m = 4$, $L = 5$, and $B = 10,000$. We range the dataset among Real-1, Real-2, Real-3, Real-4, and Real-5 with Real-3 as the default dataset, the number of keywords per query q from 3 to 15 with 9 as the default q, and the number of dimensions for data points d from 2 to 128 with 16 as the default d. All the algorithms focus on top-1 result. The results demonstrate that the AAR of ProMiSH-A is no more than 1.6 in all circumstances, and is no more than 1.2 in most cases.

Fig. 8 reports the effectiveness of ProMiSH-A under different index parameters over the real dataset Real-3. In this set of experiments, we fix the dimensions of the data points in Real-3 to be 16, and the number of keywords in queries to be 9. For index parameters, we vary the number of random unit vectors m from 2 to 6 with 4 as the default m, index level L from 5 to 13 with 5 as the default m, and hashtable size B from $1,000$ to $100,000$ with $10,000$ as the default B. All the algorithms focus on top-1 result. In general, the AAR of ProMiSH-A is no more than 1.3 in all circumstances. Moreover, we observe the following trends: (1) when m increases, the AAR increases; (2) when L increases, the AAR decreases; and (3) when B increases, the AAR increases.

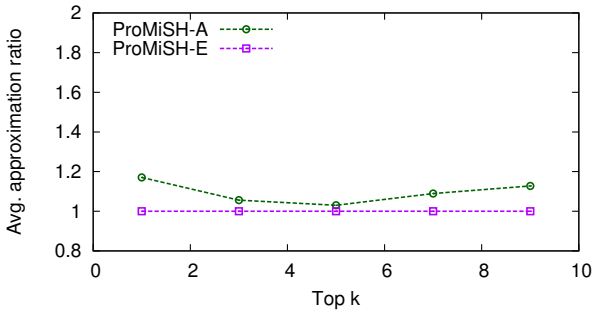


Fig. 9. Average approximation ratio of ProMiSH-A on searching top-k results over Real-3

Fig. 9 reports the effectiveness of ProMiSH-A in different top-k search over the real dataset Real-3. In this experiment, the input parameters are fixed as $d = 16$ and $q = 9$; and the index parameters are fixed as $m = 4$, $L = 5$, and $B = 10,000$. As k is varied from 1 to 9, the AAR of ProMiSH-A is no more than 1.2.

In sum, Fig 7, Fig. 8, and Fig. 9 consistently suggest the high effectiveness of ProMiSH-A.

8.3 Efficiency

We employ response time as the metric to evaluate the efficiency of different algorithms. Given a set of queries, the response time of an algorithm is defined as the average amount of time that the algorithm spends in processing one query.

Fig. 10 presents the response time of ProMiSH-E, ProMiSH-A, and VbR*-Tree under different input real data. In this set of experiments, the index parameters are fixed as $m = 4$, $L = 5$, and $B = 10,000$. We range the dataset among Real-1, Real-2, Real-3, Real-4, and Real-5 with Real-3 as the default dataset, the number of keywords per query q from 3 to 15 with 9 as the default q, and the number of dimensions for data points d from 2 to 128 with 16 as the default d. All the algorithms focus on top-1 result. Note that the result of CoSKQ based method is not shown in Fig. 10, as it cannot finish this experiment within one day. We make the following observations based the results. (1) As the number of keywords per query q increases, the response time of all algorithms increases. Compared with VbR*-Tree, ProMiSH-E and ProMiSH-A are up to 30 and 60 times faster, respectively. (2) In all real datasets, ProMiSH-E and ProMiSH-A consistently outperform VbR*-Tree with up to 18 and 25 times of speedup, respectively. Moreover, VbR*-Tree cannot process the workload for Real-5 within one day. (3) When d is ranged from 2 to 128, ProMiSH-E and ProMiSH-A can finish the computation within one second. As one has to transform an NKS query into thousands of CoSKQ queries for the correctness and evaluate them all, CoSKQ based method processes a query in 2 to 10 seconds (not shown) even when d is 2 or 4, which is up to 100 times slower than our methods. In terms of VbR*-Tree, it finishes the computation in more than one minute for $d = 32$ (not shown), but cannot finish this experiment within one day. (4) ProMiSH-A outperforms ProMiSH-E with up to 16 times of speedup.

Fig. 11 shows the efficiency of ProMiSH-E and ProMiSH-A under different index parameters over the real dataset Real-3. In this set of experiments, we fix the dimensions of the data points in Real-3 to be 16, and the number of keywords in queries to be 9. For index parameters, we vary the number of random unit vectors m from 2 to 6 with 4 as the default m, index level L from 5 to 13 with 5 as the default m, and hashtable size B from $1,000$ to $100,000$ with $10,000$ as the default B. All the algorithms focus on top-1 result. From the results, we observe that (1) $m = 4$ empirically renders the best response time for both ProMiSH-E and ProMiSH-A; (2) as L increases, the response time of both algorithms decreases; and (3) hashtable size has minor influence on the response time of the two algorithms.

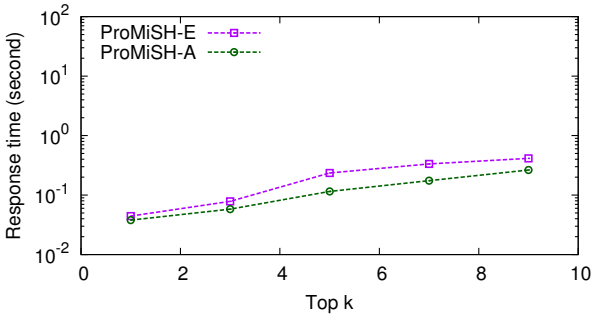


Fig. 12. Response time of ProMiSH-E and ProMiSH-A on searching top-k results over Real-3

Fig. 12 reports the response time of the algorithms on searching top-k results over Real-3. In this experiment, the input parameters are fixed as $d = 16$ and $q = 9$; and the index

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

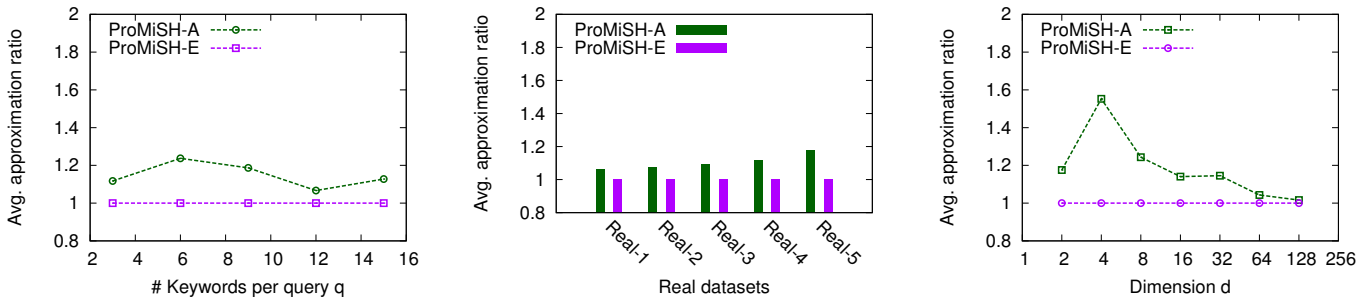


Fig. 7. Average approximation ratio of ProMiSH-A under different input real data: (left) varying the number of keywords per query q; (middle) varying real datasets; and (right) varying the number of dimensions in data points d

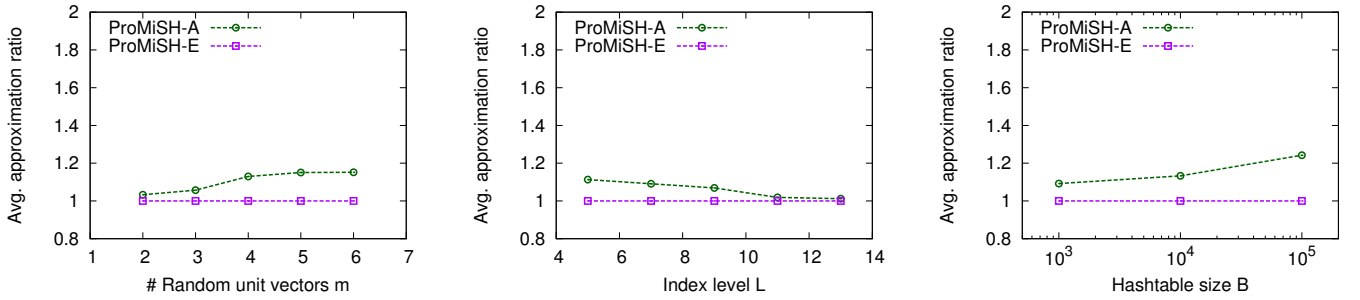


Fig. 8. Average approximation ratio of ProMiSH-A under different index parameters over Real-3: (left) varying the number of random unit vectors m; (middle) varying index level L; and (right) varying hashtable size B

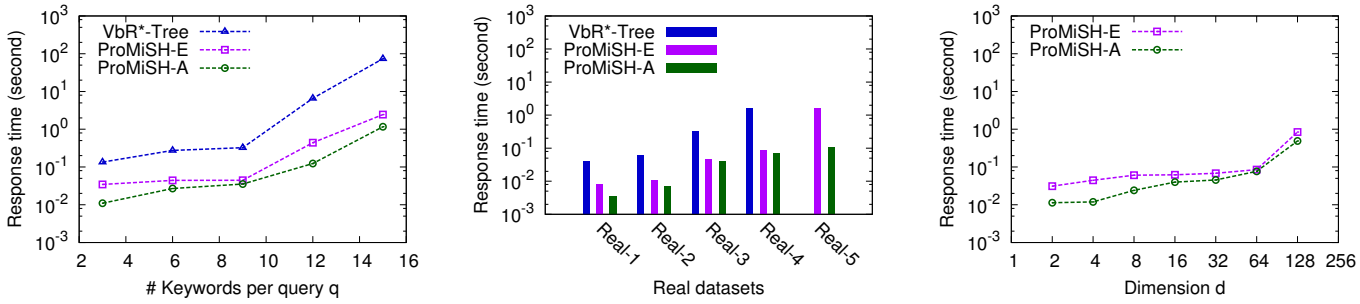


Fig. 10. Response time of ProMiSH-E, ProMiSH-A, and VbR*-Tree under different input real data: (left) varying the number of keywords per query q; (middle) varying real datasets; and (right) varying the number of dimensions in data points d

parameters are fixed as $m = 4$, $L = 5$, and $B = 10,000$. Note that VbR*-Tree and the CoSKQ based method are excluded from this experiment since they mainly support top-1 search. The results indicate that (1) as k increases, the response time of both algorithms increases; and (2) ProMiSH-A is consistently faster than ProMiSH-E.

Fig. 13 presents the response time of the algorithms under different synthetic data. In this set of experiments, the index parameters are fixed as $m = 4$, $L = 5$, and $B = 10,000$, and we apply 6 parameters to control synthetic data generation: (1) the number of keywords per query q, ranging from 3 to 15 with 9 as the default q; (2) dataset size N, ranging from $10,000$ to $10,000,000$ with $1,000,000$ as the default N; (3) data point dimension d, ranging from 2 to 128 with 16 as the default d; (4) the number of keywords per data point t, ranging from 1 to 16 with 4 as the default t; (5) dictionary size U, ranging from 100 to $10,000$ with $1,000$ as the default U; and (6) the k in top-k search, ranging from 1 to 9, with 1 as the default k. Note that the results of VbR*-Tree and the CoSKQ based method are not shown here since they cannot finish this experiment within one day. We draw the following observations based on the results. (1) As q, N, d, t, or k increases, the response time of ProMiSH-E and ProMiSH-A increases. (2) As U increases, the response time of both algorithms decreases. (3) ProMiSH-A can process one query in 2 minutes in all cases, while ProMiSH-E processes one query in 20 minutes in average. (4) ProMiSH-A outperforms ProMiSH-E in terms of response time with up to 100 times of speedup.

Fig. 14 reports the response time of ProMiSH-E and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

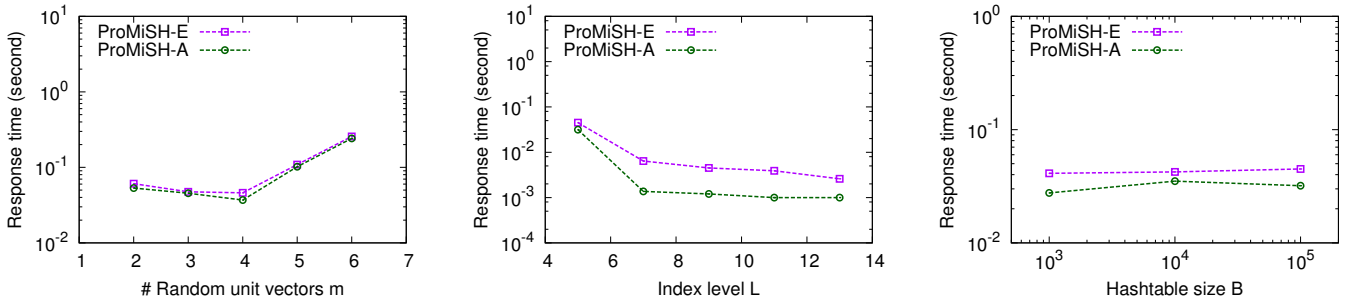


Fig. 11. Response time of ProMiSH-E and ProMiSH-A under different index parameters over Real-3: (left) varying the number of random unit vectors m; (middle) varying index level L; and (right) varying hashtable size B

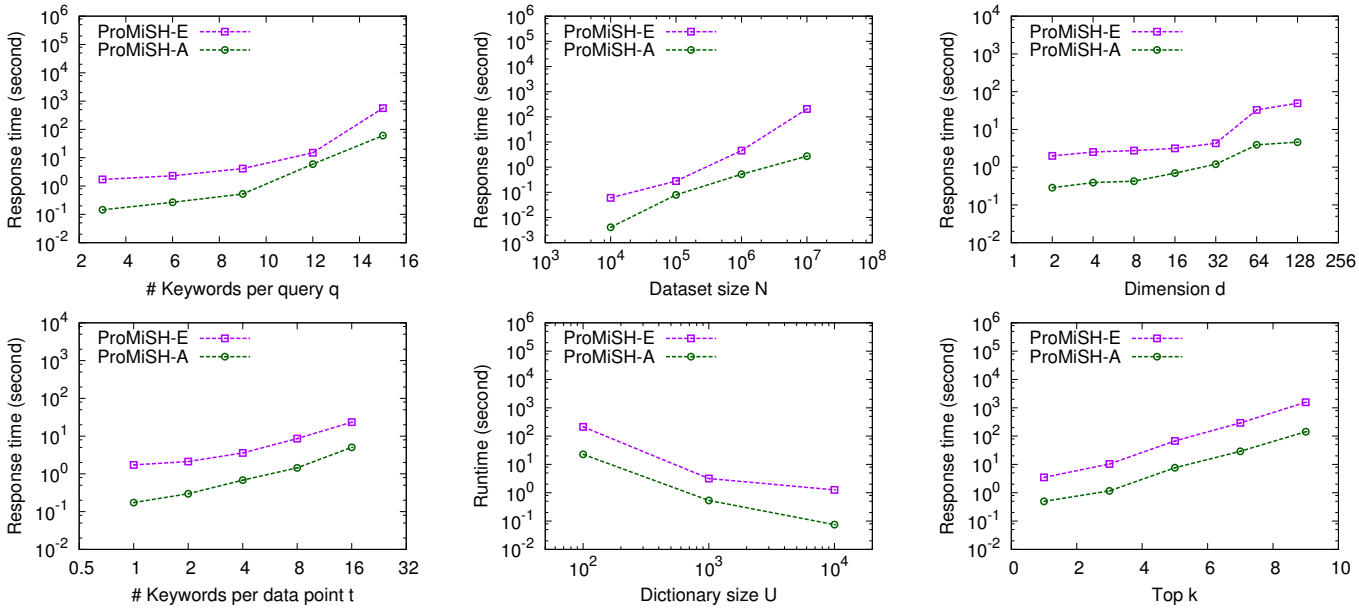


Fig. 13. Response time of ProMiSH-E and ProMiSH-A under different input synthetic data: (top-left) varying the number of keywords per query q; (top-middle) varying real datasets; (top-right) varying the number of dimensions in data points d; (bottom-left) varying the number of keywords per data point t; (bottom-middle) varying the dictionary size U; and (bottom-right) varying k for top-k search

ProMiSH-A under different index parameters over synthetic data. In this set of experiments, the synthetic data are generated with a parameter setting $q = 9$, $N = 1,000,000$, $d = 16$, $t = 4$, $U = 1000$, and $k = 1$. For index parameters, we range the number of random unit vectors m from 2 to 6 with 4 as the default m, index level L from 5 to 13 with 5 as the default L, and hashtable size B from $1,000$ to $100,000$ with $10,000$ as the default B. We observe that (1) $m = 4$ renders the best response time for ProMiSH-E; (2) as L increases, ProMiSH-A obtains significant improvement in terms of efficiency; (3) hashtable size B has minor influence on both algorithms' response time; and (4) ProMiSH-A is up to 200 times faster than ProMiSH-E.

8.4 Index efficiency

We use memory usage and indexing time as the metrics to evaluate the index size for ProMiSH-E and ProMiSH-A. In particular, Indexing time indicates the amount of time used to build ProMiSH variants.

Fig. 15 presents the memory usage and indexing time of ProMiSH-E and ProMiSH-A under different input real data. In this set of experiments, the index parameters are fixed as $m = 4$, $L = 5$, and $B = 10,000$. We vary the number of dimensions in data points m from 2 to 128 with 16 as the default m, and the datasets among Real-1, Real-2, Real-3, Real-4, and Real-5 with Real-3 as the default dataset. From the results, we make the following observations. (1) Memory usage grows slowly in both ProMiSH-E and ProMiSH-A when the number of dimensions in data points increases. (2) ProMiSH-A is more efficient than ProMiSH-E in terms of memory usage and indexing time: it takes 80% less memory and 90% less time, and is able to obtain near-optimal results as shown in Fig. 7. (3) Over all cases, the memory usage ratio of ProMiSH to raw data is no more than 13.4 for ProMiSH-E and no more than 2.4 for ProMiSH-A.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

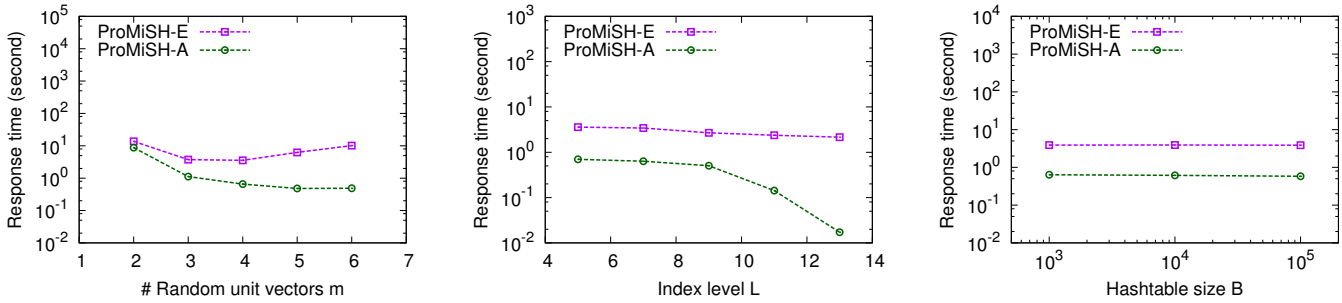


Fig. 14. Response time of ProMiSH-E and ProMiSH-A under different index parameters over synthetic data: (left) varying the number of random unit vectors m; (middle) varying index level L; and (right) varying hashtable size B

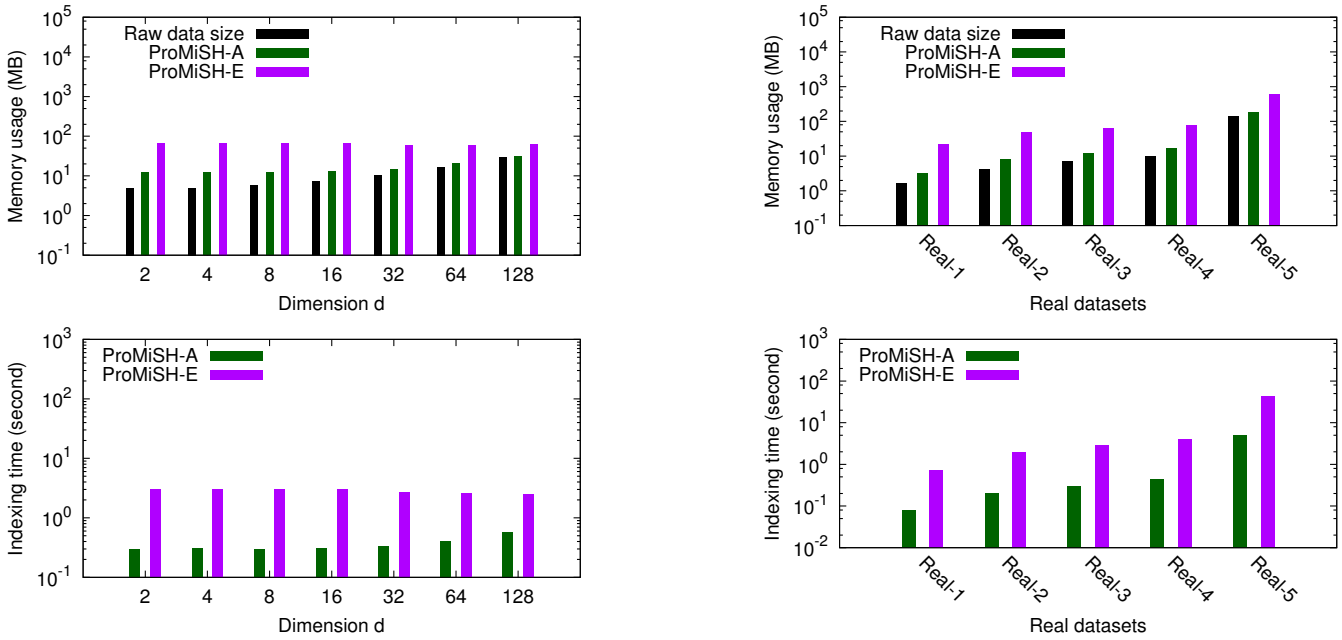


Fig. 15. Memory usage (upper row) and indexing time (lower row) of ProMiSH-E and ProMiSH-A under different input real data: (left column) varying the number of dimension in data points d; and (right column) varying real datasets

8.5 Summary

We summarize the experimental results as follows. First, ProMiSH-E and ProMiSH-A consistently outperform the baseline methods in terms of efficiency with up to 60 times of speedup. Second, ProMiSH-A is up to 16 times faster than ProMiSH-E, and can obtain near-optimal results. Third, ProMiSH-A is more space-efficient: compared with ProMiSH-E, it takes 80% less memory and 90% less indexing time.

9 Conclusions and Future Work

In this paper, we proposed solutions to the problem of top-k nearest keyword set search in multi-dimensional datasets. We proposed a novel index called ProMiSH based on random projections and hashing. Based on this index, we developed ProMiSH-E that finds an optimal subset of points and ProMiSH-A that searches near-optimal results with better efficiency. Our empirical results show that ProMiSH is faster than state-of-the-art tree-based techniques, with multiple orders of magnitude performance improvement. Moreover, our techniques scale well with both real and synthetic datasets.

Ranking functions. In the future, we plan to explore other scoring schemes for ranking the result sets. In one scheme, we may assign weights to the keywords of a point by using techniques like tf-idf. Then, each group of points can be scored based on distance between points and weights of keywords. Furthermore, the criteria of a result containing all the keywords can be relaxed to generate results having only a subset of the query keywords.

Disk extension. We plan to explore the extension of ProMiSH to disk. ProMiSH-E sequentially reads only required buckets from $\mathcal{I}_{kp}$ to find points containing at least one query keyword. Therefore, $\mathcal{I}_{kp}$ can be stored on disk using a directory-file structure. We can create a directory for $\mathcal{I}_{kp}$. Each bucket of $\mathcal{I}_{kp}$ will be stored in a separate file named after its key in the directory. Moreover, ProMiSH-E sequentially probes $\mathcal{HI}$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2492549, IEEE Transactions on Knowledge and Data Engineering

data structures starting at the smallest scale to generate the candidate point ids for the subset search, and it reads only required buckets from the hashtable and the inverted index of a $\mathcal{HI}$ structure. Therefore, all the hashtables and the inverted indexes of $\mathcal{HI}$ can again be stored using a similar directory-file structure as $\mathcal{I}_{kp}$, and all the points in the dataset can be indexed into a B+-Tree [36] using their ids and stored on the disk. In this way, subset search can retrieve the points from the disk using B+-Tree for exploring the final set of results.

10 Acknowledgments

This research was supported partially by the National Science Foundation under grant IIS-1219254.

References

[1] W. Li and C. X. Chen, "Efficient data modeling and querying system for multi-dimensional spatial data," in *GIS*, 2008, pp. 58:1–58:4.
[2] D. Zhang, B. C. Ooi, and A. K. H. Tung, "Locating mapped resources in web 2.0," in *ICDE*, 2010, pp. 521–532.
[3] V. Singh, S. Venkatesha, and A. K. Singh, "Geo-clustering of images with missing geotags," in *GRC*, 2010, pp. 420–425.
[4] V. Singh, A. Bhattacharya, and A. K. Singh, "Querying spatial patterns," in *EDBT*, 2010, pp. 418–429.
[5] J. Bourgain, "On lipschitz embedding of finite metric spaces in hilbert space," *Israel J. Math.*, vol. 52, pp. 46–52, 1985.
[6] H. He and A. K. Singh, "Graphrank: Statistical modeling and mining of significant subgraphs in the feature space," in *ICDM*, 2006, pp. 885–890.
[7] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi, "Collective spatial keyword querying," in *SIGMOD*, 2011.
[8] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu, "Collective spatial keyword queries: a distance owner-driven approach," in *SIGMOD*, 2013.
[9] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa, "Keyword search in spatial databases: Towards searching by document," in *ICDE*, 2009, pp. 688–699.
[10] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *SCG*, 2004.
[11] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma, "Hybrid index structures for location-based web search," in *CIKM*, 2005.
[12] R. Hariharan, B. Hore, C. Li, and S. Mehrotra, "Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems," in *SSDBM*, 2007.
[13] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson, "Spatio-textual indexing for geographical search on the web," in *SSTD*, 2005.
[14] A. Khodaei, C. Shahabi, and C. Li, "Hybrid indexing and seamless ranking of spatial and textual features of web documents," in *DEXA*, 2010, pp. 450–466.
[15] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *ACM SIGMOD*, 1984, pp. 47–57.
[16] I. De Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," in *ICDE*, 2008, pp. 656–665.
[17] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *PVLDB*, vol. 2, pp. 337–348, 2009.
[18] B. Martins, M. J. Silva, and L. Andrade, "Indexing and ranking in geo-ir systems," in *workshop on GIR*, 2005, pp. 31–34.
[19] Z. Li, H. Xu, Y. Lu, and A. Qian, "Aggregate nearest keyword search in spatial databases," in *Asia-Pacific Web Conference*, 2010.
[20] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis, "Top-k spatial preference queries," in *ICDE*, 2007, pp. 1076–1085.
[21] T. Xia, D. Zhang, E. Kanoulas, and Y. Du, "On computing top-t most influential spatial sites," in *VLDB*, 2005, pp. 946–957.
[22] Y. Du, D. Zhang, and T. Xia, "The optimal-location query," in *SSTD*, 2005, pp. 163–180.
[23] D. Zhang, Y. Du, T. Xia, and Y. Tao, "Progressive computation of the min-dist optimal-location query," in *VLDB*, 2006, pp. 643–654.
[24] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD*, 1990, pp. 322–331.
[25] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB*, 1994, pp. 487–499.
[26] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB*, 1997.
[27] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and perfomance study for similarity-search methods in high-dimensional spaces," in *VLDB*, 1998, pp. 194–205.
[28] W. Johnson and J. Lindenstrauss, *Extensions of Lipschitz mappings into a Hilbert space. Contemporary Mathematics*, 1984.
[29] J. M. Kleinberg, "Two algorithms for nearest-neighbor search in high dimensions," in *STOC*, 1997, pp. 599–608.
[30] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB*, 1999, pp. 518–529.
[31] V. Singh and A. K. Singh, "Simp: accurate and efficient near neighbor search in high dimensional spaces," in *EDBT*, 2012, pp. 492–503.
[32] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Quality and efficiency in high dimensional nearest neighbor search," in *SIGMOD*, 2009.
[33] H.-H. Park, G.-H. Cha, and C.-W. Chung, "Multi-way spatial joins using r-trees: Methodology and performance evaluation," in *SASD*, 1999.
[34] D. Papadias, N. Mamoulis, and Y. Theodoridis, "Processing and optimization of multiway spatial joins using r-trees," in *PODS*, 1999.
[35] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing n-relational joins," *ACM Trans. Database Syst.*, vol. 9, 1984.
[36] D. Comer, "The ubiquitous b-tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

Vishwakarma Singh Vishwkarma is a data scientist professional. He received a B.Tech. degree from the Indian Institute of Technology (BHU), Varanasi, and a PhD degree from the University of California at Santa Barbara. His research interests are broadly in in the areas of databases and data mining.

Bo Zong Bo Zong is a PhD candidate at University of California, Santa Barbara. Before joining UCSB, he obtained BS and MS degree in Computer Science at Nanjing University, China. He is interested in general data mining and data management problems. His recent research work focuses on mining and managing large-scale temporal graph data for real-life applications in cybersecurity, system performance analysis, online social networks, cloud data-centers, stream processing systems, and mobile networks.

Ambuj K. Singh Ambuj K. Singh is a Professor of Computer Science at the University of California, Santa Barbara, with part-time appointments in the Biomolecular Science and Engineering Program and Technology Management Program. He received a B.Tech. degree from the Indian Institute of Technology, Kharagpur, and a PhD degree from the University of Texas at Austin. His research interests are broadly in in the areas of databases, data mining, and bioinformatics. He has published approximately 200 technical papers over his career. He has led a number of multidisciplinary projects in the past including UCSBs Information Network Academic Research Center funded by the US Army. He is currently directing UCSBs Interdisciplinary Graduate Education Research and Training (IGERT) program on Network Science funded by the National Science Foundation (NSF). Besides the NSF and Army, his research has also been funded by the National Institute of Health.