Efficient Metric Indexing for Similarity Search

Lu Chen [#1], Yunjun Gao [#2], Xinhan Li [#3], Christian S. Jensen [†4], Gang Chen [#5]

[#] *College of Computer Science*, *Zhejiang University*, *Hangzhou*, *China*

[†] *Department of Computer Science*, *Aalborg University*, *Denmark*

{[1] luchen, [2] gaoyj, [3] lixh, [5] cg}@cs.zju.edu.cn [4] csj@cs.aau.dk

***Abstract*—The goal in similarity search is to find objects similar to a specified query object given a certain similarity criterion. Although useful in many areas, such as multimedia retrieval, pattern recognition, and computational biology, to name but a few, similarity search is not yet supported well by commercial DBMS. This may be due to the complex data types involved and the needs for flexible similarity criteria seen in real applications. We propose an efficient disk-based metric access method, the *Space-filling curve and Pivot-based B$^+$-tree* (SPB-tree), to support a wide range of data types and similarity metrics. The SPB-tree uses a small set of so-called pivots to reduce significantly the number of distance computations, uses a space-filling curve to cluster the data into compact regions, thus improving storage efficiency, and utilizes a B$^+$-tree with minimum bounding box information as the underlying index. The SPB-tree also employs a separate random access file to efficiently manage a large and complex data. By design, it is easy to integrate the SPB-tree into an existing DBMS. We present efficient similarity search algorithms and corresponding cost models based on the SPB-tree. Extensive experiments using real and synthetic data show that the SPB-tree has *much lower* construction cost, *smaller* storage size, and can support *more efficient* similarity queries with *high* accuracy cost models than is the case for competing techniques. Moreover, the SPB-tree scales sublinearly with growing dataset size.**

I. Introduction

The objective of similarity search is to find objects similar to a given query object under a certain similarity criterion. This kind of functionality has been used in many areas of computer science as well as in many application areas. For instance, in pattern recognition, similarity queries can be used to classify a new object according to the labels of already classified nearest neighbors; in multimedia retrieval, similarity queries can be utilized to identify images similar to a specified image; and in recommender systems, similarity queries can be employed to generate personalized recommendations for users based on their own preferences.

Considering the wide range of data types in the above application scenarios, e.g., images, strings, and protein sequences, a generic model is desirable that is capable of accommodating not just a single type, but a wide spectrum. In addition, the distance metrics for comparing the similarity of objects, such as cosine similarity used for vectors, and edit distance used for strings, are not restricted to the Euclidean distance (i.e., the L_2-norm). To accommodate a wide range of similarity notions, we consider similarity queries in generic metric spaces, where *no* detailed representations of objects are required and where any similarity notion that satisfies the *triangle inequality* can be accommodated.

A number of metric access methods exist that are designed to accelerate similarity search in generic metric spaces. They can be generally classified into two categories, namely, compact partitioning methods [12], [15], [17], [21] and pivot-based methods [9], [20], [27], [42]. Compact partitioning methods divide the space into compact regions and try to discard unqualified regions during search, while pivot-based methods store pre-computed distances from each object in the database to a set of pivots. Given two objects q and o, the distance $d(q, o)$ cannot be smaller than $|d(q, p) - d(o, p)|$ for any pivot p, due to the triangle-inequality. Hence, it may be possible to prune an object o as a match for q using the lower bound value $|d(q, p) - d(o, p)|$ instead of calculating $d(q, o)$. This capability makes pivot-based approaches outperform compact partitioning methods in terms of the number of distance computations, one of the key performance criteria in metric spaces. Nonetheless, pivot-based approaches need large space to store pre-computed distances, and their I/O costs are often high because the data needed to process a similarity query is not well clustered. Due to the above, we propose a hybrid method that integrates the compact partitioning into a pivot-based approach.

To design an efficient metric access method (MAM), three challenging issues have to be addressed. The first is how to support efficient similarity retrieval in terms of the number of distance computations (i.e., CPU cost) and the number of page accesses (i.e., I/O cost). We tackle this by identifying and using a small set of effective pivots for reducing significantly the number of distance computations during search, and we utilize a space-filling curve (SFC) to cluster objects into compact regions, to further boost performance. The second challenge is to achieve low-cost index storage, construction, and manipulation. To reduce the storage cost, we store multi-dimensional pre-computed distances as one-dimensional integers using the SFC, and we employ a B$^+$-tree to support efficient index construction and manipulation. The third challenge is how to efficiently manage a large set of complex objects (e.g., DNA, images). Towards this, we develop a disk-based MAM that maintains the index and the data separately, to ensure the efficiency of the index. The resulting proposal is called the *Space-filling curve and Pivot-based B$^+$-tree* (SPB-tree). It keeps complex objects in a separate random access file (RAF) and uses a B$^+$-tree with additional minimum bounding box (MBB) to index objects after a two-stage pivot and SFC mapping. The SPB-tree is generic, as it does not rely on the detailed representations of objects, and it can support any distance notion that satisfies the triangle inequality. To sum up, the key contributions are as follows:

- We develop the SPB-tree, which integrates the compact partitioning with a pivot-based approach. The tree utilizes a space-filling curve and a B^+-tree to efficiently index pre-computed distances and to cluster objects into compact regions.
- We propose an efficient pivot selection algorithm for identifying a small set of effective pivots in order to reduce significantly the number of distance computations during the search.
- We present efficient similarity search algorithms, including range and k nearest neighbor (kNN) queries, and we provide corresponding cost models.
- We conduct extensive experiments using both real and synthetic data sets to compare the SPB-tree against other MAMs, finding that the SPB-tree has much lower construction and storage cost, and supports more efficient similarity queries with high accuracy cost models. Also, the SPB-tree scales well with the data size.

The rest of this paper is organized as follows. Section II reviews related work. Section III presents the problem statement. Section IV describes the SPB-tree and the pivot selection algorithm. Section V details the similarity query algorithms and their corresponding cost models. Considerable experimental results and our findings are reported in Section VI. Finally, Section VII concludes the paper with some directions for future work.

II. Related Work

In this section, we survey previous work on metric access methods and pivot selection algorithms.

A. Metric Access Methods

Two broad categories of MAM exist, namely, compact partitioning methods and pivot-based methods.

Compact partitioning methods partition the space as compact as possible and try to prune unqualified regions during search. BST [22], [32] is a binary tree built recursively. It uses a center with a covering radius to represent a partition. GHT [38] uses two centers for each tree node, and it divides the space according to which of the two centers is closer to every object. GANT [6] is an m-way generalization of GHT. It uses a Voronoi-like partitioning of the space, and a dynamic structure EGANT has also been proposed [30]. SAT [29] is also based on Voronoi diagrams, but unlike GHT and GNAT, it attempts to approximate the structure of a Delaunay graph. Dynamic and secondary memory extensions of SAT are also available [8], [31]. Next, the M-tree [15] is a height-balanced tree that is optimized for secondary memory. It is the first dynamic MAM, and it supports insertion and deletion. Several variants of M-trees, such as the Slim-tree [21], DBM-tree [41], and CM-tree [3], try to reduce the overlap among nodes and to further compact each partition. The D-index [17] is a multi-level structure that hashes objects into buckets, which are search-separable. LC [12] employs a list of clusters, which trades construction time for query time. Since LC is an unbalance structure, to index an object set O in a metric space, its construction cost increases to $O(|O|^2)$ from $O(|O| \log|O|)$ in high dimensional spaces. Finally, BP [1] is an unbalanced tree index that integrates disjoint and non-disjoint paradigms.

Pivot-based methods store pre-computed distances from every object in the database to a set of pivots and then utilize these distances and the triangle inequality to prune objects during search. AESA [40] uses a pivot table to preserve the distances from each object to other objects. In order to save main-memory storage for the pivot table, several variants have been proposed. For example, LAESA [27] only keeps the distances from every object to selected pivots. EP [35] selects a set of essential pivots (without redundancy) covering the entire database. Clustered Pivot-table [28] clusters the pre-computed distances to further improve the query efficiency. BKT [9] is a tree structure designed for discrete distance functions. It chooses a pivot as the root, and puts the objects at distance i to the pivot in its i-th sub-tree. In contrast to BKT, where pivots at individual levels are different, FQT [4] and FQA [13] use the same pivot for all nodes at the same level of the tree. VPT [42] is designed for continuous distance functions, and it has also been generalized to m-ary trees, yielding MVPT [5]. The Omni-family [20] employs selected pivots together with existing structures (e.g., the R-tree) to index pre-computed distances.

Recently, hybrid methods that combine compact partitioning with the use of pivots have appeared. The PM-tree [36] uses cut-regions defined by pivots to accelerate similarity queries on the M-tree. In particular, cut-regions can be used to improve the performance of metric indexes with simple ball-regions [25]. The M-Index [33] generalizes the iDistance [19] technique for general metric spaces, which compacts the objects by using pre-computed distances to their closest pivots.

For similarity queries, although pivot-based methods clearly outperform compact partitioning methods in terms of the number of distance computations (i.e., the CPU cost) [2], [13], [20], [28], pivot-based approaches generally have high I/O cost since objects are not well clustered on disk. Moreover, the space requirements for both pivot-based and hybrid methods to store pre-computed distances are *high*, resulting in large indexes and considerable I/O.

B. Pivot Selection Algorithms

The efficiency of pivot-based methods depends on the pivots used. Existing work is based on two observations: (1) good pivots are far away from other objects, and (2) good pivots are far away from each other. For instance, FFT [16] tries to maximize the minimum distance between pivots. HF [20] selects pivots near the hull of a dataset. SSS [7], [11] dynamically selects pivots, if their distances to already selected pivots exceed $\alpha \times d^+$, where d^+ is the maximal distance between any two objects and parameter α controls the density of pivots with which the space is covered. These pivot selection approaches have low time complexities, but they do not perform the best. The reason is that, as pointed out in [10], good pivots are outliers, but outliers are not always good pivots.

To achieve strong pruning power using a small set of pivots, several criteria have been proposed for pivot selection. Bustos et al. [10] maximize the mean of the distance distribution in

TABLE I
SYMBOLS AND DESCRIPTION

Notation	Description
q, O	a query object, the set of objects in a generic metric space
P	the set/table of pivots
o, p	an object in O, a pivot in P
$\|O\|$, $\|P\|$	the cardinality of O, the cardinality of P
$d(\ )$	the distance function for the generic metric space
$D(\ )$	the L_∞-norm metric for the mapped vector space
d^+	the maximal distance in a generic metric space
$\phi(o)$	the data point for o after mapping to the vector space
ε	the value used to approximate $\phi(o)$ for the real numeric domain of $d(\ )$
$SFC(\phi(o))$	the space-filling curve value of an object o
$RQ(q, r)$	the result set of a range query with a search radius r
$kNN(q, k)$	the result set of a kNN query w.r.t. q
$RR(r)$	the range region with a range radius r

the mapped vector space. Hennig and Latecki [18] select pivots using a loss measurement, i.e., the nearest neighbor distance in the mapped vector space. Venkateswaran et al. [39] choose pivots that maximize pruning for a sample of queries. Leuken and Veltkamp [24] select pivots with minimum correlation to ensure that objects are evenly distributed in the mapped vector space. More recently, PCA [26] has been developed for pivot selection. As will be discussed in Section IV.B, query efficiency relies on the similarity between the original metric space and the mapped vector space. Therefore, we aim to maximize the similarity in order to achieve better search performance. The success of this approach is studied in Section VI.A.

III. PROBLEM FORMULATION

In this section, we present the characteristics of the metric space and the definitions of similarity queries. Table I lists the notations frequently used throughout this paper.

A metric space is a tuple (M, d), in which M is the domain of objects and d is a distance function which defines the similarity between the objects in M. In particular, the distance function d has four properties: (1) *symmetry*: $d(q, o) = d(o, q)$, (2) *non-negativity*: $d(q, o) \geq 0$, (3) *identity*: $d(q, o) = 0$ iff $q = o$, and (4) *triangle inequality*: $d(q, o) \leq d(q, p) + d(p, o)$. Based on the properties of the metric space, we formally define two types of similarity queries: range query and k nearest neighbor (kNN) query.

Definition 1 (Range Query). Given an object set O, a query object q, and a search radius r in a generic metric space, *a range query* finds the objects in O that are within distance r of q, i.e., $RQ(q, r) = \{o | o \in O \wedge d(q, o) \leq r\}$.

Definition 2 (kNN Query). Given an object set O, a query object q, and an integer k in a generic metric space, a kNN *query* finds k objects in O most similar to q, i.e., $kNN(q, k) = \{R \mid R \subseteq O \wedge |R| = k \wedge \forall r \in R, \forall o \in O - R, d(q, r) \leq d(q, o)\}$.

Consider an English word set O = {"citrate", "defoliates", "defoliated", "defoliating", "defoliation"}, for which the edit distance is the similarity measurement. The range query RQ("defoliate", 1) retrieves the words in O with distances to "defoliate" bounded by 1. The query result is {"defoliates", "defoliated"}. Next, the kNN query kNN("defoliate", 2) retrieves two words in O that are most similar to "defoliate", yielding the result {"defoliates", "defoliated"}. It is worth noting that $kNN(q, k)$ may be not unique due to the distance tie. Nonetheless, the target of our proposed algorithms is to find one possible instance.

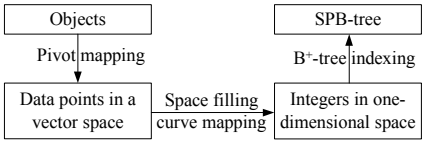


Fig. 1. The construction framework of an SPB-tree

The behavior of similarity search on a dataset can be estimated using the dimensionality of the dataset. Since metric datasets do not always have an embedded dimensionality (e.g., a word set), the *intrinsic dimensionality* can be employed. The intrinsic dimensionality of a metric dataset can be calculated as $\rho = \mu^2/2\sigma^2$, in which μ and σ are the mean and variance of the pairwise distances in the dataset [13]. We shall see in Section IV.B that the appropriate number of the pivots selected is also related to the intrinsic dimensionality.

IV. THE SPB-TREE

In this section, we first present the construction framework for the SPB-tree, and then propose a pivot selection algorithm and an index structure with bulk-loading, insertion, and deletion operations.

A. Construction Framework

As shown in Figure 1, the construction framework of the SPB-tree is based on a two-stage mapping. In the first stage, we map the objects in a metric space to data points in a vector space using well-chosen pivots. The vector space offers more freedom than the metric space when designing search approaches, since it is possible to utilize the geometric and coordinate information that is unavailable in the metric space. In the second stage, we use the SFC to map the data points in the vector space into integers in a one-dimensional space. Finally, a B$^+$-tree with MBB information is employed to index the resulting integers.

The SPB-tree utilizes the B$^+$-tree with MBB information to index the SFC values of objects after a pivot mapping. This is attractive because (1) the use of an SFC can cluster objects into compact regions, reducing the amount of storage needed for pre-computed distances, and because (2) bulk-loading, insertion, and deletion operations on the SPB-tree are simple and effective since they rely on the manipulation of the B$^+$-tree. Although the ZBtree [23], [34] that combines a Z-curve and a B$^+$-tree can be used to index objects after the pivot mapping, the ZB-tree is designed with a special SFC suitable for skyline queries; whereas any SFC (e.g., a Hilbert curve which offers better proximity preservation than Z-curve, as to be verified in Section VI.A) is applicable for the SPB-tree.

Pivot Mapping. Given a pivot set $P = \{p_1, p_2, \ldots, p_n\}$, a general metric space (M, d) can be mapped to a vector space (R^n, L_∞). Specifically, an object o in a metric space is represented as a point $\phi(o) = \langle d(o, p_1), d(o, p_2), \ldots, d(o, p_n)\rangle$ in

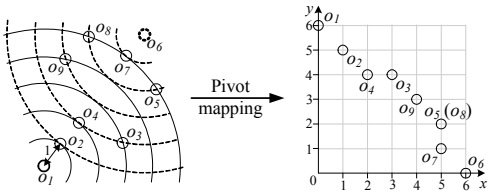
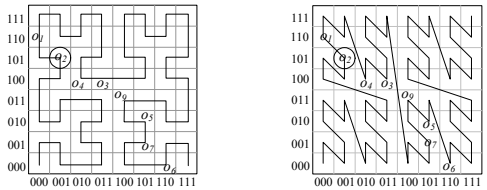


Fig. 2. Pivot mapping



(a) Hilbert curve mapping (b) Z-curve mapping

Fig.3. Space-filling curve mapping

the vector space. For instance, consider the example in Figure 2, where $O = \{o_1, o_2, \ldots, o_9\}$ and L_2-*norm* is used. If $P = \{o_1, o_6\}$, O can be mapped to a two-dimensional vector space, in which the x-axis represents $d(o_i, o_1)$ and the y-axis represents $d(o_i, o_6)$, $1 \le i \le 9$.

Given objects o_i, o_j, and p in a metric space, $d(o_i, o_j) \ge |d(o_i, p) - d(o_j, p)|$ according to the triangle inequality. Hence, for a pivot set P, $d(o_i, o_j) \ge \max\{|d(o_i, p_i) - d(o_j, p_i)| \mid p_i \in P\} = D(\phi(o_i), \phi(o_j))$, in which $D(\ )$ is the L_∞-norm. Clearly, we can conclude that the distance in the mapped vector space is a *lower bound* on that in the metric space. Take the example depicted in Figure 2 again, $d(o_2, o_3) > D(\phi(o_2), \phi(o_3)) = 2$.

Space-Filling Curve Mapping. Given a vector $\phi(o)$ after pivot mapping and assuming that the range of $d(\ )$ in the metric space is *discrete* integers (e.g., edit distance), then SFC can directly map $\phi(o)$ to an integer $SFC(\phi(o))$. Consider the SFC mapping examples in Figure 3, where $SFC(\phi(o_2)) = 18$ for the Hilbert curve and $SFC(\phi(o_2)) = 19$ for the Z-curve. Note that, for simplicity, we use the Hilbert curve in the rest of this paper.

Considering the range of $d(\ )$ in a metric space may be *continuous* real numbers, *ε-approximation* is utilized to partition the real range into discrete integers, i.e., 0, 1, …, $\lfloor d^+/\varepsilon \rfloor$, where d^+ is the maximal distance in the metric space. Thus, the whole vector space can be partitioned into $(\frac{d^+}{\varepsilon})^{|P|}$ cells. Then, given an ε, $\phi(o)$ can be approximated as $\langle \lfloor d(o, p_1)/\varepsilon \rfloor, \lfloor d(o, p_2)/\varepsilon \rfloor, \ldots, \lfloor d(o, p_n)/\varepsilon \rfloor \rangle$. As to be verified in Section VI.A, the value of ε might affect the query efficiency. If the value of ε is too big, the average collision probability $\mathbf{Pr}(collision) = |O|/(\frac{d^+}{\varepsilon})^{|P|}$ that different objects share the same approximation increases, resulting in more distance computations during the search. Instead, if a small ε is used, the transformed vector space becomes too sparse, i.e., the vast majority of the cells must be empty, incurring large search space. In the rest of this paper, for simplify, we assume that the range of $d(\ )$ is *discrete* integers, as the techniques can be easily adapted to a *continuous* real range using *ε-approximation*.

To design efficient metric access methods for similarity queries, we have identified two important issues that have to be addressed: (1) How should we pick pivots to perform a pivot mapping? (2) Which index structures can be used to support metric similarity queries? We discuss the first issue in Section IV.B and turn to the second issue in Section IV.C.

B. Pivot Selection

The selected pivots influence the search performance because the lower-bound distances computed using the pivots are utilized for pruning during the search. In order to achieve high performance, the lower-bound distances should be close to the actual distances, i.e., the mapping to the vector space should preserve the proximity from the metric space. Hence, the quality of a pivot set can be evaluated as the similarity between the mapped vector space and the original metric space, as stated in Definition 3.

Definition 3 (Precision). Given a set OP of object pairs in a metric space, the *quality* of a pivot set P is evaluated as the average ratio between the distances in the vector space and the distances in the metric space, i.e.,

$$precison(P) = \frac{1}{|OP|} \sum_{<o_i, o_j> \in OP} \frac{D(\phi(o_i), \phi(o_j))}{d(o_i, o_j)}.$$

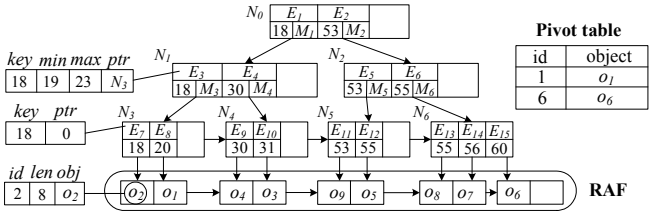
The more pivots in P, the better the pruning capability; however, the cost of using the transformed objects also increases, as presented in Section IV.A. The more pivots there are in P, the larger $D(\phi(o_i), \phi(o_j))$ will be, then $D(\phi(o_i), \phi(o_j))$ approaches $d(o_i, o_j)$, and hence, $precision(P)$ approaches 1. Therefore, we can discard more objects using a larger pivot set. On the other hand, the number of distance computations between the query object and the pivots increases as the number of pivots grows. Also, the cost (e.g., $D(\phi(o_i), \phi(o_j))$ computation cost) to prune unqualified objects increases. Thus, as pointed out in related work [13], [20], to achieve high query efficiency, the appropriate number of pivots is related to the intrinsic dimensionality of the dataset, which is also confirmed in Section VI.A.

Determining a pivot set P (from O) with a fixed size that maximizes $precision(P)$ has time complexity $O(\frac{|O|!}{|P|!(|O|-|P|)!})$, which is costly. To reduce significantly the time complexity, we propose an *HF based Incremental pivot selection algorithm* (HFI), which first employs the HF algorithm [20] to obtain outliers as candidate pivots CP and then incrementally selects effective pivots from CP. The underlying rationale is that good pivots are usually outliers, but outliers are not always good pivots [10]. Hence, the time complexity of HFI is $O(|P| \times |CP|)$, in which the cardinality of CP is small and is only related to the distribution of the object set. We fix $|CP|$ at 40 (as in reference [26]), which is enough to find all outliers in our experiments.

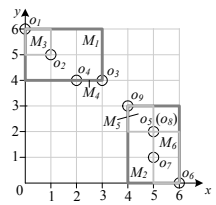
Algorithm 1 depicts the pseudo-code of HFI. First, HFI invokes HF algorithm to obtain a candidate pivot set CP from O (line 1). Thereafter, it picks incrementally pivots from CP (lines 2-5). Initially, the pivot set P is empty. Then, a *while-loop* is performed until $|P| = n$, i.e., n pivots are selected. In each iteration, HFI chooses a pivot p from CP to maximize

Algorithm 1 HF based Incremental Pivot Selection Algorithm (HFI)

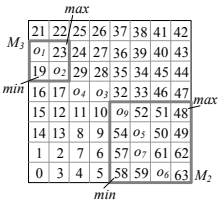
Input: a set O of objects, the number n of pivots
Output: a set P of pivots // $|P| = n$
1: CP = HF(O, cp_scale) // get a candidate pivot set CP with $|CP| = cp\_scale$
2: $P = \emptyset$
3: **while** $|P| < n$ **do**
4: select p from CP with the maximal $precision(P \cup \{p\})$
5: $P = P \cup \{p\}$ and $CP = CP - \{p\}$
6: return P



(a) The SPB-tree structure



(b) MBBs of an SPB-tree



(c) The MBB representation

Fig. 4. Example of an SPB-tree

$precision(P \cup \{p\})$ (line 4) and updates P to $P \cup \{p\}$ and CP to $CP - \{p\}$ (line 5). Finally, the pivot set P is returned.

Note that the HFI algorithm does not need to take the whole object set as an input, and it works well using only a sampled object set. Moreover, theoretically, pivots do not need to be part of the object set. Consequently, objects can be inserted or deleted without changing the pivot set.

C. Indexing Structure

An SPB-tree used to index an object set in a generic metric space contains three parts, i.e., the pivot table, the B$^+$-tree, and the RAF. Figure 4 shows an SPB-tree example to index an object set $O = \{o_1, \ldots, o_9\}$ in Figure 2. A pivot table stores selected objects (e.g., o_1 and o_6) to map a metric space into a vector space. A B$^+$-tree is employed to index the SFC values of objects after a pivot mapping. Each leaf entry in the leaf node (e.g., N_3, N_4, N_5, and N_6) of the B$^+$-tree records (1) the SFC value *key*, and (2) the pointer *ptr* to a real object, which is the address of the actual object kept in the RAF. As an example, in Figure 4, the leaf entry E_7 associated with the object o_2 records the Hilbert value 18 and the storage address 0 of o_2. Each non-leaf entry in the root or intermediate node (e.g., N_0, N_1, and N_2) of the B$^+$-tree records (1) the minimum SFC value *key* in its subtree, (2) the pointer *ptr* to the root node of its subtree, and (3) the SFC values *min* and *max* for $\langle a_1, a_2, \ldots, a_{|P|}\rangle$ and $\langle b_1, b_2, \ldots, b_{|P|}\rangle$, to represent the MBB M_i (= $\{[a_i, b_i] \mid i \in [1, |P|]\}$) of the root node N_i of its subtree. Specifically, a MBB M_i denotes the axis aligned *minimum bounding box* to contain all $\phi(o)$ with $SFC(\phi(o)) \in N_i$. For

Algorithm 2 Bulkload SPB-tree Algorithm (BA)

Input: a set O of objects, the number n of pivots
Output: an SPB-tree
1: P = HFI(O, n) // see Algorithm 1
2: compute $\{\phi(o) \mid o \in O\}$ using P
3: compute and sort SFC values $\{SFC(\phi(o)) \mid o \in O\}$
4: build_RAF(O) // build the RAF in ascending order of SFC values
5: bulkload-B$^+$-tree($\{\langle SFC(\phi(o)), ptr(o)\rangle \mid o \in O\}$)
6: return SPB-tree

instance, the non-leaf entry E_3 uses *min* (= 19) and *max* (= 23) to represent the M_3 of N_3.

Unlike compact partitioning methods (e.g., M-tree), which store actual objects in the index directly since the routing objects are needed for pruning unqualified partitions, SPB-tree uses a RAF to keep objects separately and supports both random access and sequential scan, in order to enhance the efficiency for managing the complex objects. Note that, RAF is sorted to store the objects in ascending order of SFC values as they appear in the B$^+$-tree. Each RAF entry records (1) an object identifier *id*, (2) the length *len* of the object, and (3) the real object *obj*. Here, *len* is recorded to support efficient storage management, because the object size may be *different* in generic metric spaces. As an example, words in a dictionary may have different lengths, e.g., the length of "word" is 4, and the length of "dictionary" is 10. Also, in Figure 4, the RAF entry associated with an object o_2 records the object identifier 2, the object length 8, and the real object o_2, respectively.

Bulk-loading Operation. We develop a bulk-loading operation, with the pseudo-code of *Bulkload SPB-tree Algorithm* (BA) shown in Algorithm 2. First, HFI (Algorithm 1) is called to get a pivot table P. Then, BA computes $\phi(o)$ for every object $o \in O$ using P. After that, BA computes and sorts the SFC values for all objects, and then, it invokes *build_RAF* function to build the RAF. Next, the bulkload operation of the B$^+$-tree can be directly employed to build B$^+$-tree for $\{\langle SFC(\phi(o)), ptr(o)\rangle \mid o \in O\}$) and meanwhile compute the MBB for every node. Finally, an SPB-tree is returned.

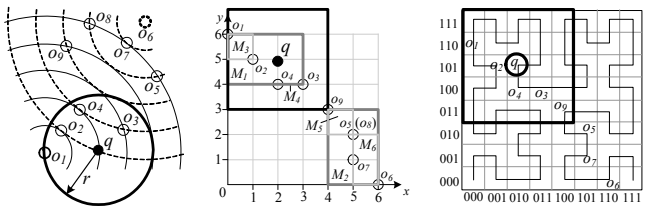
Insertion/Deletion Operation. For a new object o to be inserted or deleted, we first compute $\phi(o)$ using P, and then get its corresponding $SFC(\phi(o))$. Thereafter, the leaf entry e (= $\langle SFC(\phi(o)), ptr(o)\rangle$) and o are inserted into or deleted from the B$^+$-tree and the RAF, respectively. Finally, the MBBs of e's ancestors are updated if necessary. Note that, the insertion or deletion may result in page splits or merges of the B$^+$-tee and RAF. Thus, the corresponding MBBs are updated if necessary.

V. Similarity Search

In this section, we propose efficient algorithms for processing similarity queries based on the SPB-tree, and then derive their corresponding cost models.

A. Range Query

Given a metric object set O, a range query finds the objects in O with their distances to a specified query object q bounded by a threshold r, i.e., a range query retrieves the objects enclosed in the range region that is an area centered at q with a radius r. Consider, for example, Figure 5(a), where a *circle*



(a) Metric space (b) Vector space (c) Hilbert curve

Fig. 5. Illustration of $RQ(q, r)$

denotes a range region, and $RQ(q, 2) = \{o_1, o_2, o_3, o_4\}$. Given a pivot set P, the range region can also be mapped into a vector space. For instance, the *thick black rectangle* in Figure 5(b) represents the *mapped range region* using $P = \{o_1, o_6\}$. To obtain $RQ(q, r)$, we only need to verify the objects o whose $\phi(o)$ are contained in the mapped range region, as stated below.

Lemma 1. Given a pivot set P, if an object o is enclosed in $RQ(q, r)$, then $\phi(o)$ is certainly contained in the mapped range region $RR(r)$, where $RR(r) = \{\langle s_1, s_2, \ldots, s_{|P|}\rangle \mid 1 \le i \le |P| \wedge s_i \ge 0 \wedge s_i \in [d(q, p_i) - r, d(q, p_i) + r]\}$.

Proof. Assume, to the contrary, that there exists an object $o \in RQ(q, r)$ but $\phi(o) \notin RR(r)$, i.e., $\exists\, p_i \in P$, $d(o, p_i) > d(q, p_i) + r$ or $d(o, p_i) < d(q, p_i) - r$. According to the triangle inequality, $d(q, o) \ge |d(q, p_i) - d(o, p_i)|$. If $d(o, p_i) > d(q, p_i) + r$ or $d(o, p_i) < d(q, p_i) - r$, then $d(q, o) \ge |d(o, p_i) - d(q, p_i)| > r$, which contradicts with our assumption. Consequently, the proof completes. □

Based on Lemma 1, if the MBB of a node N does not intersect with $RR(r)$, we can discard N, in which MBB can be easily obtained by using SFC values *min* and *max* stored in the SPB-tree. Considering the example range query depicted in Figure 5 with its corresponding SPB-tree illustrated in Figure 4, N_6 can be pruned as $M_6 \cap RR(r) = \emptyset$.

Lemma 1 is used to avoid distance computations for the objects not contained in $RQ(q, r)$. Nonetheless, we still have to verify all the objects o whose $\phi(o)$ are enclosed in $RR(r)$. To this end, we develop Lemma 2 to further avoid unqualified distance computations during the verification.

Lemma 2. Given a pivot set P, for an object o in O, if there exists a pivot p_i ($\in P$) satisfying $d(o, p_i) \le r - d(q, p_i)$, then o is certainly included in $RQ(q, r)$.

Proof. Given a query object q, an object o, and a pivot p_i, $d(q, o) \le d(o, p_i) + d(q, p_i)$ due to the triangle inequality. If $d(o, p_i) \le r - d(q, p_i)$, then $d(q, o) \le r - d(q, p_i) + d(q, p_i) = r$. Thus, o is for sure contained in $RQ(q, r)$, which completes the proof. □

Back to the example shown in Figure 5, where $O = \{o_1, \ldots, o_9\}$ and $P = \{o_1, o_6\}$. Suppose $r = 3$, for an object o_2, there exists a pivot p_1 ($= o_1$), which holds that $d(o_2, p_1) = r - d(q, p_1)$. Hence, o_2 is certainly included in $RQ(q, 3)$ without any further distance computation of $d(q, o_2)$.

The pseudo-code of *Range Query Algorithm* (RQA) is depicted in Algorithm 3. First, RQA computes $\phi(q)$ using a pivot table P. Then, it calls a function ComputeRR to obtain $RR(r)$. Next, it pushes the root node of a B$^+$-tree into a heap H,

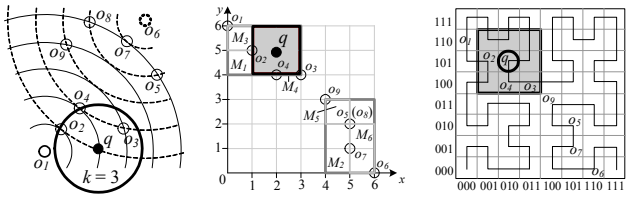
Algorithm 3 Range Query Algorithm (RQA)

Input: a query object q, a radius r, an object set O indexed by a SPB-tree
Output: the result set $RQ(q, r)$ of a range query
1: compute $\phi(q)$ using a pivot table P // $\phi(q) = \langle d(q, p_i) \mid p_i \in P\rangle$
2: $RR(r)$ = ComputeRR($\phi(q)$, r) // get $RR(r)$
3: push the root node of a B$^+$-tree into a min-heap H
4: **while** $H \neq \emptyset$ **do**
5: de-heap the top node N from H
6: **if** N is a non-leaf node **then**
7: **for** each entry e in N **do**
8: **if** $MBB(e.ptr) \cap RR(r) \neq \emptyset$
9: push $e.ptr$ into H
10: **else** // N is a leaf node
11: **if** $MBB(N) \subseteq RR(r)$ **then**
12: **for** each entry e in N **do**
13: VerifyRQ(e, *false*) // verify e
14: **else if** $|RR(r) \cap MBB(N)| < |N|$ **then**
15: S = computeSFC($RR(r) \cap MBB(N)$)
16: $s = S.get\_first()$ and $e = N.get\_first()$
17: **while** $s \neq$ NULL and $e \neq$ NULL **do**
18: **if** $e.key = s$ **then** VerifyRQ(e, *false*) and $e = N.get\_next()$
19: **else if** $e.key > s$ **then** $s = S.get\_next()$
20: **else** $e = N.get\_next()$
21: **else** // $|RR(r) \cap MBB(N)| \ge |N|$
22: **for** each entry e in N **do**
23: VerifyRQ(e, *true*)
24: **return** $RQ(q, r)$

Function: VerifyRQ(e, *flag*)
25: **if** *flag* and $\phi(o) \notin RR(r)$ **then return** // $\phi(o)$ is obtained by $e.key$
26: **if** the condition of Lemma 2 satisfies **then**
27: insert $e.ptr$ into $RQ(q, r)$ and **return**
28: **if** $d(q, e.ptr) \le r$ **then**
29: insert $e.ptr$ into $RQ(q, r)$

and a *while-loop* (lines 4-23) is performed until H is empty. Each time, RQA pops the top node N from H. If N is a non-leaf node, RQA pushes all its sub nodes $e.ptr$ ($e \in N$) with $MBB(e.ptr) \cap RR(r) \neq \emptyset$ into H (lines 7-9). Otherwise (i.e., N is a leaf node), if $MBB(N) \subseteq RR(r)$, for each entry in N, VerifyRQ is utilized to determine whether RQA inserts the corresponding object into $RQ(q, r)$. In order to achieve the lowest CPU time, i.e., minimize the cost of the transformation between $\phi(o)$ and $SFC(\phi(o))$, if the number of SFC values contained in the intersected region $RR(r) \cap MBB(N)$ is smaller than that of entries in N, RQA first invokes a function computeSFC to obtain S that includes all SFC values in the intersected region in ascending order (line 15), and then calls VerifyRQ for each entry e ($\in N$) with $e.key \in S$ (lines 16-20); otherwise, VerifyRQ is invoked for every entry in N (lines 22-23), where unqualified objects (i.e., $\phi(o) \notin RR(r)$) need not be verified due to Lemma 1 (line 25). Finally, the final query result set $RQ(q, r)$ is returned (line 24).

Example 1. We illustrate RQA using the example depicted in Figure 5 with its SPB-tree in Figure 4, and suppose $r = 2$. Initially, RQA computes $\phi(q) = (2, 5)$ using P and gets $RR(2)$. It then pushes a root node N_0 into H, and pops N_0. As N_0 is a non-leaf node, its sub-nodes N_1 and N_2 are pushed into H due to M_1 and M_2 are intersected with $RR(2)$. Next, similarly, it pops N_1 and pushes N_3 and N_4 into H. Thereafter, N_3 is popped. Since N_3 is a leaf node and $M_3 \subseteq RR(2)$, RQA calls *VerifyRQ* to insert o_1 and o_2 into $RQ(q, 2)$ due to Lemma 2 and $d(q, o_2) < 2$, respectively. The algorithm proceeds in the same manner



(a) Metric space (b) Vector space (c) Hilbert curve

Fig. 6. Illustration of $kNN(q, k)$

until H is empty, with the final query result set $RQ(q, 2) = \{o_1, o_2, o_3, o_4\}$. □

B. k Nearest Neighbour Search

Given an object set O in a metric space, a kNN query finds from O the k NNs of a specified query object q. For instance, in Figure 6, the result set $kNN(q, 3) = \{o_3, o_2, o_4\}$ of a 3NN query. In general, the kNN query is a little tricker than range query. In order to minimize the kNN query cost, our *kNN Query Algorithm* (NNA) follows an incremental traversal paradigm, i.e., NNA visits the B$^+$-tree entries and verifies corresponding objects in ascending order of their minimum distances to q in the mapped vector space until k NNs are found. Moreover, to avoid unnecessary entry accesses, a pruning rule is developed as follows.

Lemma 3. Given a query object q and a B$^+$-tree entry E, E can be safely pruned if $MIND(q, E) \geq curND_k$, where $MIND(q, E)$ denotes the minimum distance between q and E in the mapped vector space and $curND_k$ represents the distance from q to the current k-th NN.

Proof. As pointed out in Section IV.A, the distance in the mapped vector space is the lower bound of that in the original metric space. Thus, $mind(q, E) \geq MIND(q, E)$, with $mind(q, E)$ denoting the minimal distance between q and any object contained in E under the original metric space. If $MIND(q, E) \geq curND_k$, we can get that $mind(q, E) \geq curND_k$. Hence, E can be discarded safely, and the proof completes. □

Note that, for Lemma 3, $curND_k$ is obtained and updated during kNN search. For example, as depicted in Figure 6 with its corresponding SPB-tree in Figure 4, assume that $curND_k = 1$, E_2 can safely pruned as $MIND(q, E_2) > 1$. Due to Lemma 3, kNN search can stop when visiting the entry E that satisfies the early termination condition, i.e., $MIND(q, E) \geq curND_k$. Hence, NNA is optimal in the number of distance computations, since it only needs to search in $RR(ND_k)$, as stated in Lemma 4.

Lemma 4. NNA has to evaluate the objects o having $\phi(o) \in RR(ND_k)$ only once, in which ND_k is the k-th NN distance.

Proof. Assume, to the contrary, NNA visits an object o having $\phi(o) \notin RR(ND_k)$, i.e., $MIND(q, o) > ND_k$. Since NNA follows an incremental traversal manner, all the objects o' contained in $RR(ND_k)$ are retrieved before o due to $MIND(q, o) > MIND(q, o')$. According to Lemma 1, we know that $\phi(o) \in RR(ND_k)$ if $o \in kNN(q, k)$, as $kNN(q, k)$ can be regarded as $RQ(q, ND_k)$. Thus, $curND_k$ has been already updated to ND_k before accessing o. Hence, NNA can be terminated, due to the stop condition $MIND(q, o) > curND_k$ (= ND_k), without visiting o, which contradicts with our assumption. In order to complete the proof, we still need to show that the objects in $RR(ND_k)$ are not visited multiple times, which is straightforward as every entry is visited a single once. □

Algorithm 4 kNN Query Algorithm (NNA)

Input: a query object q, an integer k, an object set O indexed by a SPB-tree
Output: the result set $kNN(q, k)$ of a kNN query
1: $curND_k = \infty$, $H = \emptyset$ // H stores the intermediate entries of B$^+$-tree in ascending order of $MIND(q, E)$
2: compute $\phi(q)$ using P and push the root entries of B$^+$-tree into H
3: **while** $H \neq \emptyset$ **do**
4: de-heap the top entry E from H
5: **if** $MIND(q, E) \geq curND_k$ **then break**
6: **if** E is a non-leaf entry **then**
7: **for** each sub entry $e \in E$ **do**
8: **if** $MIND(e, D) < curND_k$ **then** // Lemma 3
9: push e into H
10: **else** // E is a leaf entry
11: **if** $d(q, e.ptr) < curND_k$ **then**
12: insert $e.ptr$ into $kNN(q, k)$ and update $curND_k$ if necessary
13: **return** $kNN(q, k)$

The pseudo-code of NNA is presented in Algorithm 4. First of all, NNA sets $curND_k$ to infinity, and initializes the min-heap H. Then, it computes $\phi(q)$ using P and pushes the root entries of a B$^+$-tree into H. Next, a *while-loop* (lines 3-12) is performed until H is empty or the early termination condition satisfied (line 5). In each while-loop, NNA deheaps the top entry E from H. If E is a non-leaf entry, it pushes all the qualified sub entries of E into H (lines 7-9) based on Lemma 3; otherwise, for a leaf entry E, it verifies whether its corresponding object is an actual answer object and updates $curND_k$ if necessary (lines 11-12). In the end, the final query result set $kNN(q, k)$ is returned (line 13).

Example 2. We illustrate NNA using the $kNN(q, 3)$ (k = 3) example shown in Figure 6 with its SPB-tree in Figure 4. First, $curND_k$ and the min-heap H are initialized to infinity and empty, respectively. Then, NNA computes $\phi(q) = (2, 5)$ using P and pushes the root entries into H (= $\{E_1, E_2\}$). Next, it performs a while-loop. In the first loop, NNA pops the top entry E_1 from H, and then pushes its qualified sub entries into H (= $\{E_4, E_3, E_2\}$) as E_1 is a non-leaf entry. In the second loop, it pops E_4 and pushes the qualified sub leaf entries into H (= $\{E_9, E_{10}, E_3, E_2\}$). Then, NNA pops the leaf entry E_9 and inserts o_4 into $kNN(q, 3)$ due to $d(q, o_4) < curND_k$. Thereafter, it pops and evaluates entries in H similarly until $MIND(q, E_2) > curND_k$, after which $kNN(q, 3) = \{o_3, o_2, o_4\}$. Finally, NNA stops, and returns $kNN(q, 3)$ as the final result set. □

NNA evaluates objects contained in $RR(ND_k)$ in ascending order of their $MIND$ to q, incurring the random page accesses in RAF. Since SFC preserves the spatial proximity, the objects to be verified are supposed to be kept close to each other in RAF. Thus, with a small cache, we can avoid duplicated RAF page accesses, as to be confirmed in Section VI.A. However, for the kNN query that needs to retrieve a large portion of the dataset, a small cache is not enough. To this end, a greedy traversal paradigm can be utilized, i.e., when visiting a B$^+$-tree entry pointing to a leaf node, instead of re-inserting the

qualified sub leaf entries into the min-heap, the objects pointed by leaf entries can be immediately evaluated. Although the greedy traversal paradigm will result in unnecessary distance computations for verifying the objects not contained in $RR(ND_k)$, it can still boost the computational efficiency, because the objects in the same leaf node satisfy the spatial proximity, as also demonstrated in Section VI.A.

C. Cost Models

In this section, we develop cost models for similarity search, including range and kNN queries, to estimate their I/O and CPU costs. With the help of cost models, we can choose promising execution strategies. For instance, if the estimated cost of our MAM is lower than that of other MAMs, it would be better to pick our similarity query algorithms.

In order to estimate the CPU cost in terms of the number of distance computations, we need to utilize the distance distribution, as it is the *natural* way to characterize metric datasets. The overall distribution of distances from objects in O to a pivot p_i is defined as:

$$F_{pi}(r) = \mathbf{Pr}\{d(o, p_i) \le r\} \quad (1)$$

where o is a random object in O. Nevertheless, distance distributions $F_{pi}(r)$ for pivots in a pivot set P are not independent, because pivots are not selected randomly, and the distances in a metric space are also not independent due to the triangle inequality. Thus, we introduce the *union* distance distribution function for P, since it can be obtained using the sampled dataset:

$$F(r_1, r_2, \ldots, r_{|P|}) = \mathbf{Pr}\{d(o, p_1) \le r_1, d(o, p_2) \le r_2, \ldots, d(o, p_{|P|}) \le r_{|P|}\} \quad (2)$$

To determine the *estimated number of distance computations* (*EDC*) for a similarity query, it is enough to sum two parts, including the number of distance computations for computing $\phi(q)$ and the number of distance computations for verifying whether an object o is contained in the final result set, i.e.,

$$EDC = |P| + |O| \times \mathbf{Pr}(d(q, o) \text{ is needed to compute}) \quad (3)$$

For the range query algorithm, $\mathbf{Pr}(d(q, o)$ is needed to compute) in equation (3) can be estimated as the probability that $\phi(o)$ is contained in $RR(r)$, which can be computed as:

$$\begin{aligned}\mathbf{Pr}(\phi(o) \in RR(r)) &= \mathbf{Pr}(d(q, p_1) - r \le d(o, p_1) \le d(q, p_1) + r)), \ldots, \\ &\quad d(q, p_{|P|}) - r \le d(o, p_{|P|}) \le d(q, p_{|P|}) + r)) \\ &= F(u_1, u_2, \ldots, u_{|P|}) - F(l_1, u_2, \ldots, u_{|P|}) - \\ &\quad F(u_1, l_2, \ldots, u_{|P|}) - \cdots - F(u_1, u_2, \ldots, l_{|P|}) + \\ &\quad F(l_1, l_2, \ldots, u_{|P|}) + \cdots + F(u_1, u_2, \ldots, l_{|P|-1}, \\ &\quad l_{|P|}) - \cdots + (-1)^{|P|} \times F(l_1, l_2, \ldots, l_{|P|}) \quad (4)\end{aligned}$$

where $l_i = d(q, p_i) - r - 1$ and $u_i = d(q, p_i) + r$.

A kNN query can be regarded as the range query with a search radius $r = ND_k$, in which ND_k denotes the distance from q to its farthest NN. Hence, in order to drive *EDC* for kNN retrieval, the first step is to determine the ND_k value. Using the distance distribution function $F_q(r)$, ND_k can be estimated as eND_k, the minimal r that has at least k objects with their distances to q bounded by r:

$$eND_k = \min\{r \mid |O| \times F_q(r) \le k\} \quad (5)$$

However, $F_q(r)$ is not known in advance. In this paper, we employ a simple but efficient method [14] to estimate $F_q(r)$ using $F_{pi}(r)$, where p_i is the nearest neighbor of q.

Thus, to obtain *EDC* for kNN search, according to Lemma 4, $\mathbf{Pr}(d(q, o)$ is needed to compute) in equation (3) equals to the probability that $\phi(o)$ is contained in $RR(ND_k)$, which can be calculated via using equation (4) with $r = eND_k$ (computed by equation (5)).

Since the I/O cost for similarity queries on an SPB-tree includes two parts, i.e., the B$^+$-tree page accesses and the RAF page accesses. To obtain the number of B$^+$-tree page accesses, it is sufficient to sum all the nodes whose MBBs are intersected with the search region, i.e., $RR(r)$ for a range query or $RR(ND_k)$ for a kNN query. In addition, since the objects accessed in RAF are supposed to be stored close to each other, the number of RAF page accesses can be estimated as $\frac{EDC}{f}$, in which *EDC* is used to estimate the total number of the objects visited, and f represents the average number of the objects accessed per RAF page. Thus, to sum up, the *expected number of page accesses* (*EPA*) of a similar query can be calculated as:

$$EPA = \sum_{M_i \text{ for } N_i \in B^+\text{-tree}} I(M_i) + \frac{EDC}{f} \quad (6)$$

where $I(M_i) = \begin{cases} 1 & M_i \text{ intersects with the search region} \\ 0 & \text{otherwise} \end{cases}$.

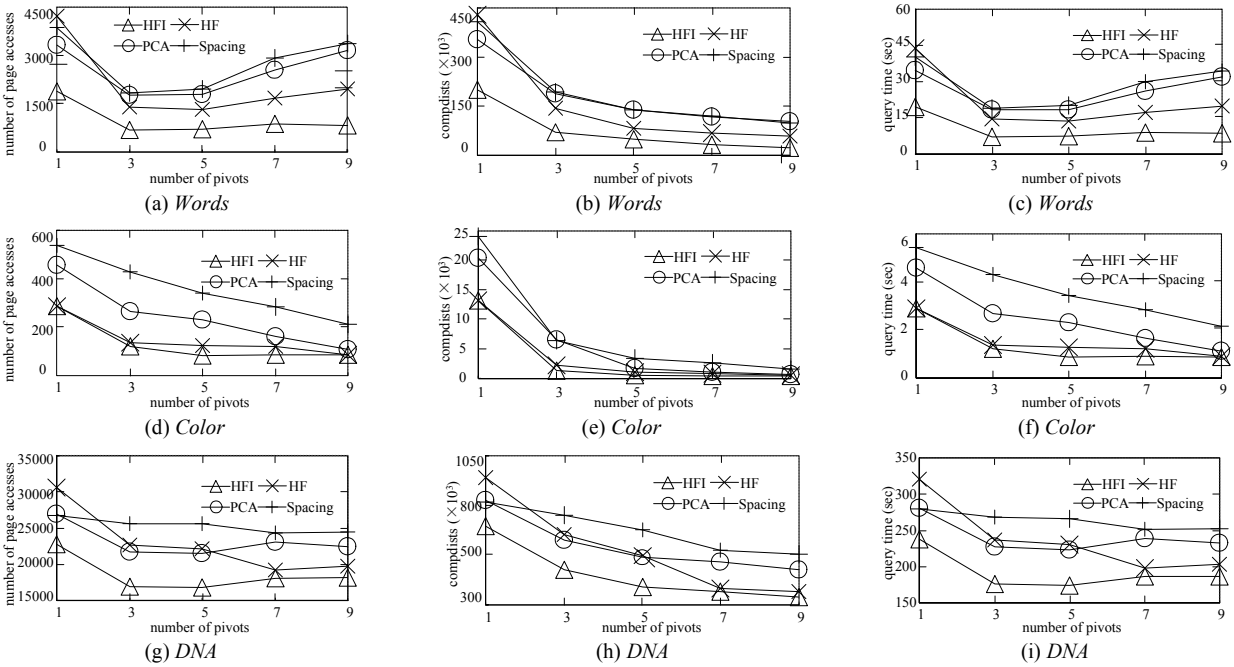
VI. Experimental Evaluation

In this section, we experimentally evaluate the performance of SPB-tree. First, we study the effect of parameters for the SPB-tree. Then, we compare the SPB-tree against several representative MAMs. Next, the scalability of the SPB-tree is explored. Finally, we verify the accuracy of cost models. We implemented the SPB-tree and associated similarity search algorithms in C++. All experiments were conducted on an Intel Core 2 Duo 2.93GHz PC with 3GB RAM.

We employ three real datasets, namely, *Words*, *Color*, and *DNA*. *Words* [1] contains proper nouns, acronyms, and compound words taken from the Moby Project, and the edit distance is used to compute the distance between two words. *Colors*[2] denotes the color histograms extracted from an image database, and L_5-norm is utilized to compare the color image features. *DNA*[3] consists of 1 million DNA data, and the cosine similarity is used to measure its similarity under the tri-gram counting space. Following the experimental setup in [37], we generate a *Signature* dataset, and the Hamming distance is employed. *Synthetic* datasets are also created, where the first five dimensional values are generated randomly, and each remaining dimension is the linear combination of previous ones. Without loss of generality, L_2-norm is utilized for *Synthetic* datasets. Table II summarizes the statistics of the datasets used in our experiments. All MAMs to index the datasets are configured to use a fixed disk page size of 4KB.

We investigate the efficiency of the SPB-tree and the performance of similarity search algorithms under various

[1] *Words* is available at http://icon.shef.ac.uk/Moby/
[2] *Color* is available at http://www.sisap.org/Metric_Space_Library.html
[3] *DNA* is available at http://www.ncbi. nlm.nih.gov/genome



(a) *Words* (b) *Words* (c) *Words*

(d) *Color* (e) *Color* (f) *Color*

(g) *DNA* (h) *DNA* (i) *DNA*

Fig. 7. Efficiency of pivot selection methods vs. the number of pivots $|P|$

TABLE II
STATISTICS OF DATASETS USED

Dataset	Cardinality	Dim.	Ins. Dim.	Measurement
Words	611,756	1~34	4.9	*Edit distance*
Colors	112,682	16	2.9	L_5-norm
DNA	1,000,000	108	6.9	*Cosine similarity under tri-gram counting space*
Signature	49,740	64	14.8	*Hamming distance*
Synthetic	[200K, 1000K]	20	4.76	L_2-norm

TABLE III
PARAMETER SETTINGS

Parameter	Value	Default
the number of pivots $\|P\|$	1, 3, 5, 7, 9	5
cache size (pages)	0, 8, 16, 32, 64, 128	32
ε	0.001, 0.003, 0.005, 0.007, 0.009	0.005
r (% of d^+)	2, 4, 8, 16, 32, 64	8
k	1, 2, 4, 8, 16, 32	8
cardinality	200K, 400K, 600K, 800K, 1000K	600K

parameters, which are listed in Table III. Note that, in every experiment, only one factor varies, whereas the others are fixed to their default values. The main performance metrics include the number of page accesses (*PA*), the number of distance computations (*compdists*), and the total query time (i.e., the sum of the I/O time and CPU time, where the I/O time is computed by charging 10ms for each page access [37]). Each measurement we report is the average of 500 queries.

A. Effect of Parameters

The first set of experiments studies the effect of parameters for the SPB-tree. Note that, we only employ *k*NN queries to demonstrate the effect of parameters on the efficiency of the SPB-tree, due to space limitation and similar performance behavior on range queries.

First, we evaluate the efficiency of the SPB-tree under different SFCs, with the results illustrated in Table IV. As

TABLE IV
SPB-TREE EFFICIENCY UNDER DIFFERENT SFCS

	Hilbert Curve			Z-Curve		
	Words	*Color*	*DNA*	*Words*	*Color*	*DNA*
PA	703.22	82.198	16,789	812.08	200.25	19,430
compdists	49,746	522.8	391,411	49,782	522.8	558,580
time (sec)	0.228	0.02	6.157	0.215	0.021	8.961

TABLE V
*k*NN SEARCH WITH DIFFERENT TRAVERSAL STRATEGIES

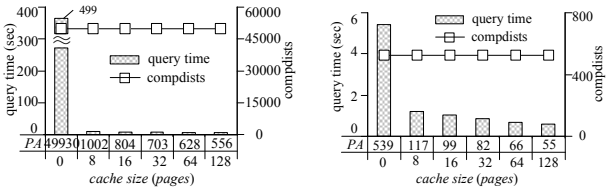
	Incremental Traversal			Greedy Traversal		
	Words	*Color*	*DNA*	*Words*	*Color*	*DNA*
PA	703.218	82.198	309,765	469.784	57.686	16,789
compdists	49,746	522.802	391,215	51,188	740.574	391,411
time (sec)	0.2282	0.0203	9.126	0.259	0.0198	6.157

observed, the query cost (including the number of page accesses and the number of distance computations) of Hilbert curve is lower than that of Z-curve. This is because a Hilbert curve is a continuous SFC, which achieves better clustering property than a Z-curve. Thus, in the rest of experiments, the Hilbert curve is used to build the SPB-tree.

Then, we investigate the effectiveness of our pivot selection algorithm (i.e., HFI). Figure 7 depicts the experimental results, using real datasets. The first observation is that, HFI performs better than existing pivot selection algorithms, viz., HF [20], Spacing [24], and PCA [26]. The reason is that, similarity search performance is highly related with *precision* defined in Definition 3, and HFI tries to maximize *precision*. The second observation is that the number of distance computations decreases as the number of pivots grows. This is because, using more pivots, the query efficiency improves as *precision* becomes larger, incurring less distance computations. The number of page accesses and the total query time first drop and then stay stable or increase as the number of pivots ascends. The reason is that, the cost for filtering unqualified objects grows as well with more pivots. Hence, similarity

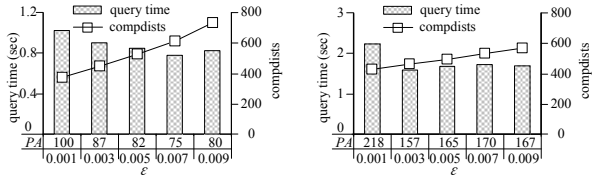
TABLE VI
THE CONSTRUCTION COSTS AND STORAGE SIZES OF MAMS

	Words				***Color***				***DNA***			
	PA	*Compdists*	*Time*(sec)	*Storage*(KB)	*PA*	*Compdists*	*Time*(sec)	*Storage*(KB)	*PA*	*Compdists*	*Time*(sec)	*Storage*(KB)
M-tree	5,896,000	54,303,500	186.88	69,772	1,286,500	4,694,000	22.9	34,364	11,665,125	76,430,441	1027.33	133,748
OmniR-tree	—	—	—	—	335,002	450,728	7.52	13,290	—	—	—	—
M-Index	49,493	12,235,310	213.43	242,469	81,920	2,253,830	12.89	30,264	104,776	20,000,190	1433.23	499,106
SPB-tree	**13,577**	**3,058,780**	**10.17**	**13,462**	**4,864**	**563,410**	**2.494**	**9,858**	**52,204**	**5,000,000**	**77.944**	**130,120**



(a) *Word* (b) *Color*

Fig. 8. Effect of *cache size*



(a) *Color* (b) *Synthetic*

Fig. 9. Effect of ε

search achieves high efficiency in all performance metrics, when the number of pivots approaches the dataset's *instinct dimensionality* (Ins. Dim. for short in Table II).

Next, we explore the influence of cache size on the efficiency of *k*NN query algorithms, as shown in Figure 8. As expected, the number of page accesses and the total query time decrease as cache size ascends, and stay stable when cache size reaches 32 pages. Thus, a small cache is enough. However, as discussed in Section V.B, if a *k*NN query needs to retrieve a large portion of the dataset, i.e., $\frac{Compdists}{\text{dataset cardinality}}$ is large, a small cache is not enough for the incremental traversal strategy. As shown in Table V, on *DNA*, it needs huge I/O cost (i.e., *PA*) with a small default cache, while the greedy traversal strategy is optimal in terms of *PA*, and it achieves the computational efficiency accordingly. It is worth noting that, in our remaining experiments, the cache size is set as the default (i.e., 32 pages) for all the indexes.

Finally, in order to inspect the impact of ε on the efficiency of the SPB-tree, we use *Color* and *Synthetic* datasets since the range of their distance functions is real numeric. Figure 9 plots the experimental results with respect to various ε values. As observed, the number of distance computations increases with the growth of ε. The reason is that, for larger ε, the collision probability $|O|/(\frac{d^+}{\varepsilon})^{|P|}$ that different objects can be approximated as the same vectors ascends, resulting in more distance computations. However, the total query time first drops and then stays stable. This is because, for smaller ε, the search space becomes sparse as the collision probability decreases, leading to high query cost.

B. Comparisons with Other MAMs

The second set of experiments compares the SPB-tree with three representative MAMs, namely, M-tree [15], OmniR-tree [20], and M-Index [33]. It is worth mentioning that, OmniR-tree utilizes HF algorithm to select (*instinct dimension* + 1) pivots, while M-Index randomly chooses 20 pivots.

Table VI depicts the construction costs and storage sizes for all MAMs using real datasets. Note that, on *Words* and *DNA*, OmniR-tree cannot run because of the large cardinality of the dataset. Clearly, SPB-tree has much lower construction cost, in terms of the number of page accesses (i.e., *PA*), the number of distance computations (i.e., *Compdists*), and the construction time (denoted by *Time*). The reason is that, the SPB-tree uses a B^+-tree as the underlying index to achieve its construction efficiency. In addition, the storage size (denoted by *Storage*) of the SPB-tree is also much smaller than that of other MAMs, due to the dimensionality reduction performed using SFC.

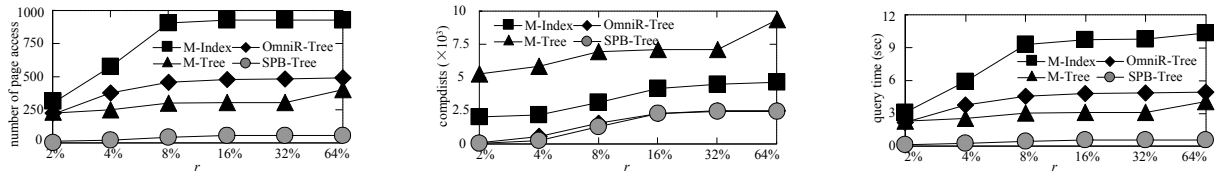
Figures 10 and 11 show the performance of range and *k*NN queries, using *Signature* and real datasets. It is observed that, SPB-tree performs the best in terms of the number of page accesses, including both B^+-tree node accesses and RAF page accesses, due to two reasons below. First, SPB-tree uses SFC to cluster objects into compact regions, and hence it achieves the I/O efficiency as both B^+-tree entries and RAF objects to be visited are stored close to each other. Second, SPB-tree preserves multi-dimensional pre-computed distances as one-dimensional SFC values, resulting in smaller index storage size and less page accesses. In addition, SPB-tree performs better or comparable to existing MAMs, in terms of the number of distance computations, which equals to the number of the objects accessed during similar search. The reason is that, our pivot selection algorithm selects effective pivots to avoid significant number of distance computations, and our similarity search algorithms only compute qualified distances, as stated in Lemmas 1 to 4. Consequently, SPB-tree has the lowest query time, which is used to evaluate the query at once, i.e., not separately expressed by the number of page accesses and the distance computations.

C. Scalability of the SPB-tree

The third set of experiments aims to verify the scalability of the SPB-tree. Figure 12 plots the performance of range and *k*NN queries as a function of cardinality, using *Synthetic* datasets. Obviously, the query costs including the number of page accesses, the number of distance computations, and the total query time ascend linearly with cardinality, because the search space grows as cardinality increases.

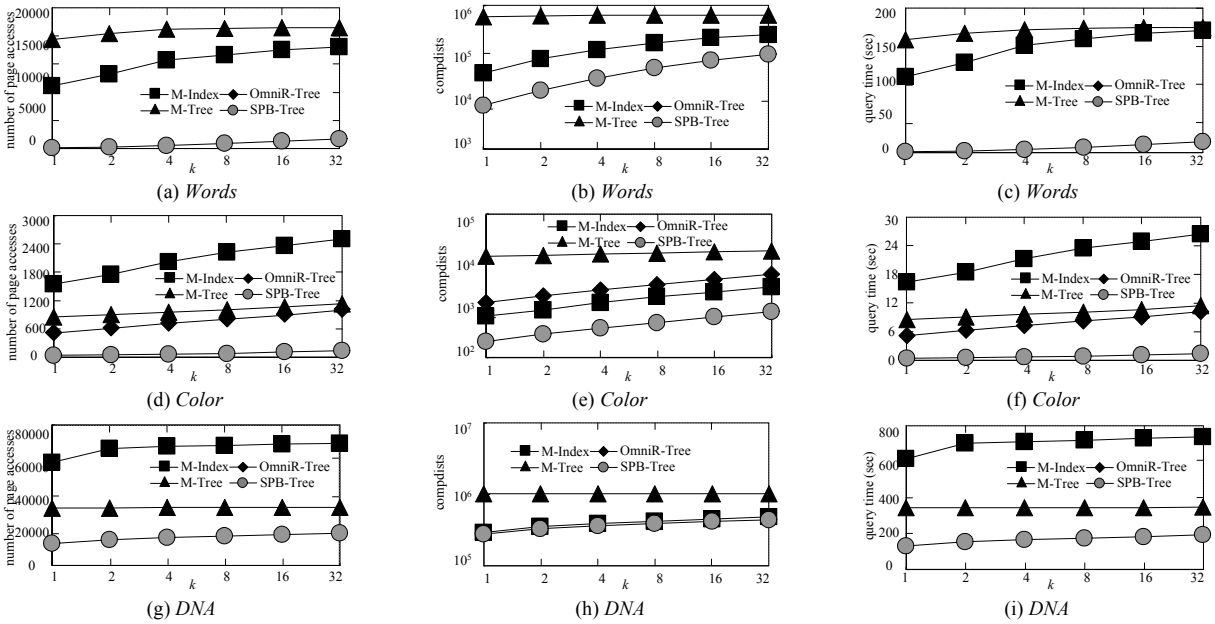
D. Accuracy of Cost Models

The last set of experiments evaluates the accuracy of our cost models for similarity queries. Figures 13 and 14 illustrate



(a) *Signature* (b) *Signature* (c) *Signature*

Fig. 10. Range query performance vs. *r*

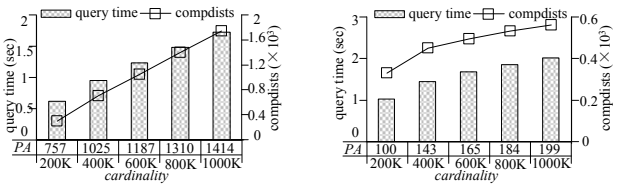


(a) *Words* (b) *Words* (c) *Words*

(d) *Color* (e) *Color* (f) *Color*

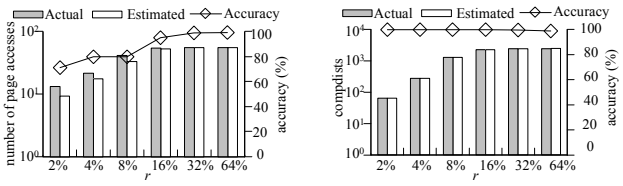
(g) *DNA* (h) *DNA* (i) *DNA*

Fig. 11. *k*NN query performance vs. *k*



(a) Range query (b) *k*NN query

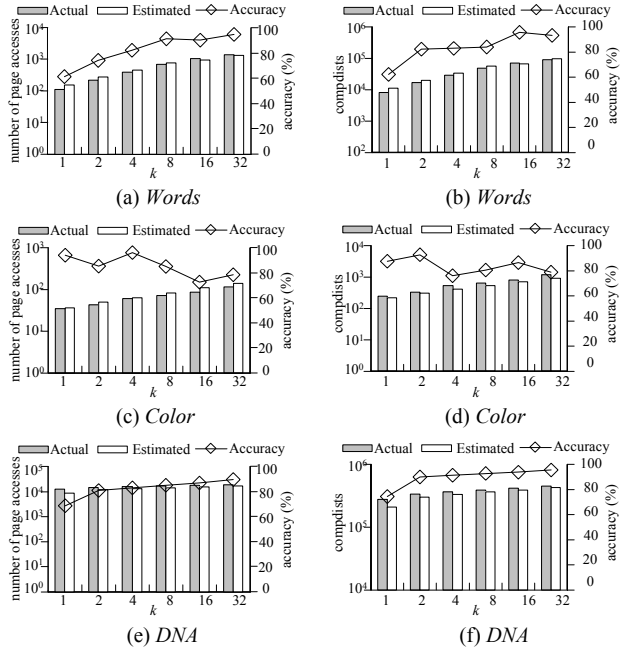
Fig. 12. Scalability of SPB-tree vs. *cardinality*



(a) *Signature* (b) *Signature*

Fig. 13. Range query cost model vs. *r*

the I/O overhead (i.e., the number of page accesses) and CPU cost (i.e., the number of distance computations) for range and *k*NN queries, respectively. In particular, every diagram contains (1) the actual costs *Actual*, (2) the estimated costs *Estimated* computed by our cost models, and (3) the accuracy between actual and estimated values *Accuracy* (i.e., 1 – |*Actual* – *Estimated*| / *Actual*). It is observed that, our cost models to estimate I/O and CPU costs are very accurate, with the average accuracy over 80%.



(a) *Words* (b) *Words*

(c) *Color* (d) *Color*

(e) *DNA* (f) *DNA*

Fig. 14. *k*NN query cost model vs. *k*

VII. CONCLUSIONS

Similarity queries are useful in many areas of computer science, such as pattern recognition, computational biology, multimedia retrieval, and so forth. In this paper, we develop a

new metric index, namely, *Space-filling curve and Pivot-based B^+-tree* (SPB-tree), for similarity search in a generic metric space, which supports a wide range of data types and any similarity metric. The SPB-tree picks few but effective pivots to reduce significantly the number of distance computations; uses SFC to cluster data objects into compact regions, thus improving storage efficiency; utilizes a B^+-tree with MBB information as the underlying index that can be easily applicable to existing DBMS; and employs a separate RAF to store a large set of complex data. In addition, we propose efficient similarity search algorithms and derive their corresponding cost models based on the SPB-tree. Extensive experiments show that, compared with other MAMs, the SPB-tree has *lower* construction and storage costs, and supports *more efficient* similarity queries. In the future, we intend to extend the SPB-tree to various distributed environments.

ACKNOWLEDGMENT

Yunjun Gao was supported in part by NSFC Grant No. 61379033, the National Key Basic Research and Development Program (i.e., 973 Program) No. 2015CB352502, the Cyber Innovation Joint Research Center of Zhejiang University, and the Key Project of Zhejiang University Excellent Young Teacher Fund (Zijin Plan). We would like to thank Prof. D. Novak and Prof. R. Mao for useful feedback on the source-codes of their proposed algorithms in [26, 33].

REFERENCES

[1] J. Almeida, R. D. S. Torres, and N. J. Leite, "BP-tree: An efficient index for similarity search in high-dimensional metric spaces," in *CIKM*, 2010, pp. 1365–1368.

[2] L. G. Ares, N. R. Brisaboa, M. F. Esteller, O. Pedreira, and A. S. Places, "Optimal pivots to minimize the index size for metric access methods," in *SISAP*, 2009, pp. 74–80.

[3] L. Aronovich and I. Spiegler, "CM-tree: A dynamic clustered index for similarity search in metric databases," *Data Knowl. Eng.*, 63(3), pp. 919–946, 2007.

[4] R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. "Proximity matching using fixed-queries trees," in *CPM*, 1994, pp. 198–212.

[5] T. Bozkaya and M. Ozsoyoglu, "Distance-based indexing for high-dimensional metric spaces," in *SIGMOD*, 1997, pp. 357–368.

[6] S. Brin, "Near neighbor search in large metric spaces," in *VLDB*, 1995, pp. 574–584.

[7] N. R. Brisaboa, A. Farina, O. Pedreira, and N. Reyes, "Similarity search using sparse pivots for efficient multimedia information retrieval," in *ISM*, 2006, pp. 881–888.

[8] L. Britos, A. M. Printista, and Nora Reye, "DSACL$^+$-tree: A dynamic data structure for similarity search in secondary memory," in *SISAP*, pp. 116–131, 2012.

[9] W. Burkhard and R. Keller, "Some approaches to best-match file searching," *Commun. ACM*, 16(4), pp. 230–236, 1973.

[10] B. Bustos, G. Navarro, and E. Chavez, "Pivot selection techniques for proximity searching in metric spaces," *Pattern Recognition Letters*, 24(14), pp. 2357–2366, 2003.

[11] B. Bustos, O. Pedreira, and N. R. Brisaboa, "A dynamic pivot selection technique for similarity search in metric spaces," in *SISAP*, 2008, pp. 105–112.

[12] E. Chavez and G. Navarro, "A compact space decomposition for effective metric indexing," *Pattern Recognition Letters*, 26(9), pp. 1363–1376, 2005.

[13] E. Chavez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquin, "Searching in metric spaces," *ACM Comput. Surv.*, 33(3), pp. 273–321, 2001.

[14] P. Ciaccia and A. Nanni, "A query-sensitive cost model for similarity queries with M-tree," in *ADC*, 1999, pp. 65–76.

[15] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB*, 1997, pp. 426–435.

[16] S. Dasgupta, "Performance guarantees for hierarchical clustering," *J. Comput. Syst. Sci.*, 70(4), pp. 555–569, 2005.

[17] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula, "D-index: Distance searching index for metric data sets," *Multimedia Tools Appl.*, 21(1), pp. 9–33, 2003.

[18] C. Hennig and L. J. Latecki, "The choice of vantage objects for image retrieval," *Pattern Recognition*, 36(9), pp. 2187–2196, 2003.

[19] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, 30(2), pp. 364–397, 2005.

[20] C. T. Jr., R. F. S. Filho, A. J. M. Traina, M. R. Vieira, and C. Faloutsos, "The Omni-family of all-purpose access methods: A simple and effective way to make similarity search more efficient," *VLDB J.*, 16(4), pp. 483–505, 2007.

[21] C. T. Jr., A. Traina, B. Seeger, and C. Faloutsos, "Slim-trees: High performance metric trees minimizing overlap between nodes," in *ICDE*, 2000, pp. 51–65.

[22] I. Kalantari and G. McDonald, "A data structure and an algorithm for the nearest point problem," *IEEE Trans. Software Eng.*, 9(5), pp. 631–634, 1983.

[23] K.C. K. Lee, B. Zheng, H. Li, and W.-C. Lee, "Approaching the skyline in Z order," in: *VLDB*, 2007, pp. 279-290.

[24] R. H. V. Leuken and R. C. Veltkamp, "Selecting vantage objects for similarity indexing," *ACM Trans. Multimedia Comput., Commun., and Appl.*, 7(3), article 16, 2011.

[25] J. Lokoc, J. Mosko, P. Cech, and T. Skopal, "On indexing metric spaces using cut-regions," *Inf. Syst.*, 43, pp. 1–19, 2014.

[26] R. Mao, W. L. Mirankerb, and D. P. Mirankerc, "Pivot selection: Dimension reduction for distance-based indexing," *J. Discrete Algorithms*, 13, pp. 32–46, 2012.

[27] L. Mico, J. Oncina, and R. C. Carrasco, "A fast branch & bound nearest neighbour classifier in metric spaces," *Pattern Recognition Letters*, 17(7), pp. 731–739, 1996.

[28] J. Mosko, J. Lokoc, and T. Skopal, "Clustered pivot tables for I/O-optimized similarity search," in *SISAP*, 2011, pp. 17–24.

[29] G. Navarro, "Searching in metric spaces by spatial approximation," *VLDB J.*, 11(1), pp. 28–46, 2002.

[30] G. Navarro and R. U. Paredes, "Fully dynamic metric access methods based on hyperplane partitioning," *Inf. Syst.*, 36(4), pp. 734–747, 2011.

[31] G. Navarro and N. Reyes, "Dynamic spatial approximation trees for massive data," in *SISAP*, 2009, pp. 81–88.

[32] H. Noltemeier, K. Verbarg, and C. Zirkelbach, "Monotonous bisector* Trees — A tool for efficient partitioning of complex scenes of geometric objects," in *Data Structures and Efficient Algorithms*, 1992, pp. 186–203.

[33] D. Novak, M. Batko, and P. Zezula, "Metric Index: An efficient and scalable solution for precise and approximate similarity search," *Inf. Syst.*, 36(4), pp. 721–733, 2011.

[34] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integrating the UB-tree into a database system kernel", in *VLDB*, pp. 263–272, 2000.

[35] G. Ruiz, F. Santoyo, E. Chavez, K. Figueroa, and E. S. Tellez, "Extreme pivots for faster metric indexes," in *SISAP*, 2013, pp. 115–126.

[36] T. Skopal, J. Pokorny, and V. Snasel, "PM-tree: Pivoting metric tree for similarity search in multimedia databases," in *ADBIS*, 2004, pp. 803–815.

[37] Y. Tao, M. L. Yiu, and N. Mamoulis, "Reverse nearest neighbor search in metric spaces," *IEEE Trans. Knowl. Data Eng.*, 18(9), pp. 1239–1252, 2006.

[38] J. K. Uhlmann, "Satisfying general proximity/similarity queries with metric trees," *Inf. Process. Lett.*, 40(4), pp. 175–179, 1991.

[39] J. Venkateswaran, T. Kahveci, C. M. Jermaine, and D. Lachwani, "Reference-based indexing for metric spaces with costly distance measures," *VLDB J.*, 17(5), pp. 1231–1251, 2008.

[40] E. Vidal, "An algorithm for finding nearest neighbors in (approximately) constant average time," *Pattern Recognition Letters*, 4(3), pp. 145–157, 1986.

[41] M. R. Vieira, C. T. Jr., F. J. T. Chino, and A. J. M. Traina, "DBM-Tree: A dynamic metric access method sensitive to local density data," *J. Inf. Data Management*, 1(1), pp. 111–128, 2010.

[42] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA*, 1993, pp. 311-321.