Efficient Metric Indexing for Similarity Search and Similarity Joins

Lu Chen, Yunjun Gao, *Member, IEEE,* Xinhan Li, Christian S. Jensen, *Fellow, IEEE,* Gang Chen

Abstract—Spatial queries including similarity search and similarity joins are useful in many areas, such as multimedia retrieval, data integration, and so on. However, they are not supported well by commercial DBMSs. This may be due to the complex data types involved and the needs for flexible similarity criteria seen in real applications. In this paper, we propose a versatile and efficient disk-based index for metric data, the *Space-filling curve and Pivot-based B^+-tree* (SPB-tree). This index leverages the B^+-tree, and uses space-filling curve to cluster data into compact regions, thus achieving storage efficiency. It utilizes a small set of so-called pivots to reduce significantly the number of distance computations when using the index. Further, it makes use of a separate random access file to support a broad range of data. By design, it is easy to integrate the SPB-tree into an existing DBMS. We present efficient algorithms for processing similarity search and similarity joins, as well as corresponding cost models based on SPB-trees. Extensive experiments using both real and synthetic data show that, compared with state-of-the-art competitors, the SPB-tree has *much lower* construction cost, *smaller* storage size, and supports *more efficient* similarity search and similarity joins with *high* accuracy cost models.

Index Terms—Indexing technique, Metric space, Query processing, Cost model

1 Introduction

SPATIAL queries have numerous uses, and have been studied extensively. For instance, similarity search, one important spatial query, finds objects similar to a query object under a certain similarity criterion. In multimedia settings, similarity search can be utilized to retrieve images similar to a specified image. In computational biology, similarity search can also be employed to identify similar protein sequences. Similarity join is another common spatial query type, which finds pairs of similar objects. In data integration, similarity joins can help to identify near-duplicate records.

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 data type, but a wide spectrum of data types. 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). We investigate similarity search and similarity joins in generic metric spaces, and provide solutions that make no assumptions on the detailed representations of the objects and that support any similarity notion satisfying the *triangle inequality*.

A number of metric access methods exist that are designed to accelerate query processing in metric spaces. They can be generally classified into two categories, namely, compact partitioning methods [1], [2], [3], [4] and pivot-based methods [5], [6], [7], [8]. The former methods partition the space into compact partitions, and try to discard unqualified regions during search; while the latter 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, a key performance criterion. Nonetheless, pivot-based approaches need larger space to store pre-computed distances, and their I/O costs are often high because the data is not clustered well [7], [9]. In view of this, we propose a hybrid method that integrates compact partitioning into a pivot-based approach.

We design a versatile and efficient metric access method (MAM) to support similarity search and similarity joins. To do so, we address three challenging issues. *Challenge I: How to achieve low-cost index storage, construction, and manipulation?* We use the space-filling curve (SFC) dimensionality reduction to reduce the storage cost, and employ a B^+-tree to enable efficient index construction and manipulation operations. *Challenge II: How to support efficient query processing in metric spaces?* The query cost in metric spaces can be measured as the number of distance computations (i.e., CPU cost) and the number of page accesses (i.e., I/O cost). To achieve query efficiency, we identify and use a small set of effective pivots for reducing significantly the number of distance computations during search, and we utilize an SFC to cluster objects into compact regions, to further boost performance. *Challenge III: How to manage efficiently 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, in order to ensure the efficiency of the index.

The resulting proposal is called the *Space-filling curve and*

- *L. Chen, Y. Gao (corresponding author), X. Li, and G. Chen are with the College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, P. R. China. E-mail: {luchen, gaoyj, xhli, cg}@zju.edu.cn.*
- *Christian S. Jensen is with the Department of Computer Science, Aalborg University, DK-9220 Aalborg, Denmark. E-mail: csj@cs.aau.dk.*

TABLE 1: Symbols and description

Notation	Description
q	a query object
Q or O	the set of objects in generic metric spaces
P	the set/table of pivots
o or p	an object o in O, a pivot p in P
$\|Q\|, \|O\|, \|P\|$	the cardinality of set Q, or O, or P
$d()$	the distance function for the generic metric space
$D()$	the L_∞-norm metric for the mapped vector space
d^+	the maximum distance in the generic metric space
$\phi(o)$	the data point for o in the mapped vector space
δ	the value used to approximate $\phi(o)$ for the real
$SFC(o)$	the space-filling curve value of an object o
$RQ(q, O, r)$	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
$SJ(Q,O,\varepsilon)$	the result set of a similarity join w.r.t. Q, O, and ε
$RR(q,r)$	the mapped range region of $RQ(q, O, r)$ using P

P̲ivot-based B̲$^+$-tree (SPB-tree). It stores complex objects in a separate random access file (RAF), and uses a B$^+$-tree with minimum bounding boxes (MBB) to index objects after a two-stage pivot-and-SFC mapping. The SPB-tree is generic: 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 compact partitioning into a pivot-based approach. It utilizes an SFC and a B$^+$-tree to efficiently cluster objects into compact regions and to index pre-computed distances.
- We propose an efficient pivot selection algorithm for identifying a small set of effective pivots.
- We present efficient algorithms for similarity search and similarity joins in metric spaces, and we also derive corresponding cost models.
- We conduct extensive experiments with real and synthetic data to compare the SPB-tree against other MAMs, finding that the SPB-tree has much lower construction and storage costs and supports more efficient similarity search and similarity joins with high accuracy cost models.

A preliminary report of this work is published in [10]. We extend mainly it in this paper, by (1) studying similarity joins under metric spaces based on the SPB-tree, with the corresponding cost models; and (2) conducting enhanced experimental evaluation that investigates the efficiency of the SPB-tree update operation, and incorporates the new classes of metric similarity joins and their corresponding cost models.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the SPB-tree and pivot selection algorithm. Section 4 details similarity query algorithms and their corresponding cost models. Section 5 covers similarity joins. Section 6 gives empirical study. Finally, Section 7 concludes and provides directions for future work.

2 Related Work

We survey existing work on metric access methods, pivot selection algorithms, and querying metric spaces. Table 1 summarizes the notations used throughout the paper.

2.1 Metric Access Methods

Two broad categories of MAMs exist that aim to accelerate query processing in metric spaces, namely, compact partitioning methods and pivot-based approaches. The former methods partition the space as compactly as possible. They try to prune unqualified partitions during search. BST [11], [12], GHT [13], [14], SAT [15], M-tree family [2], [4], [16], D-index [3], eD-index [17], LC family [1], [18], [19], and BP [20] all belong to this category. Methods of the other category store pre-computed distances from every object in the database to a set of pivots. They utilize the distances and triangle inequality to prune unqualified objects during search. AESA [7], [9], EP [21], FQT [22], VPT [8], [23], and Omni-family [6] all belong to this category.

Recently, hybrid methods that combine compact partitioning with the use of pivots have appeared. The PM-tree [24] uses cut-regions defined by pivots to accelerate query processing on the M-tree. Furthermore, the cut-regions can be used to improve the performance of metric indexes with simple ball-regions [25]. The M-Index [26] generalizes the iDistance technique for metric spaces, which compacts the objects by using pre-computed distances to their closest pivots.

Although pivot-based methods clearly outperform compact partitioning approaches in terms of the number of distance computations (i.e., CPU cost) [6], [27], [28], [29], they generally have high I/O cost because pivot-based methods usually only use the pre-computed distances to accelerate the search and thus objects are stored without clustering [7], [9]. Hence, hybrid methods are needed to accelerate metric query processing. However, the existing hybrid methods (PM-tree and M-Index) are far from enough. Their space requirements to store all the pre-computed distances are *high*, resulting in large indexes and considerable I/O cost during search, which is also confirmed empirically in Sections 6.2 and 6.3. Motivated by these, in order to achieve the efficiency in both I/O and CPU costs, we utilize the pivot mapping to avoid unnecessary distance computations, and use the SFC with the B$^+$-tree to cluster data and cut down the storage cost.

2.2 Pivot Selection Algorithms

The efficiency of pivot-based methods depends on the pivots. Two broad categories of pivot selection algorithms exist. The first category methods are based on two observations: good pivots are far from other objects, and are far away from each other. For instance, FFT [30] tries to maximize the minimum distance between pivots. HF [6] selects pivots near the hull of a dataset. SSS [31], [32] 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 to be covered. The second category approaches propose several criteria to achieve strong pruning power using a small set of pivots. Bustos et al. [33] maximize the mean of the distance distribution in the mapped vector space. Hennig and Latecki [34] select pivots using a loss measurement, i.e., the nearest neighbor distance in the mapped vector space. Venkateswaran et al. [35] choose pivots that maximize pruning for a sample of queries. Leuken and Veltkamp [36] select pivots with the minimum correlation to ensure that objects are evenly distributed in the mapped vector space. Recently, PCA [37] has been developed for pivot selection.

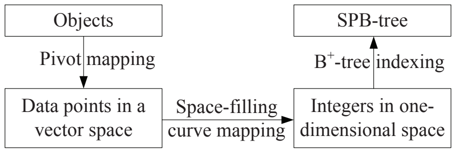

Objects
Pivot mapping
Data points in a vector space
Space-filling curve mapping
Integers in one-dimensional space
B+-tree indexing
SPB-tree

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

Methods of the first category have lower time complexities, but they also have lower quality pivots than the second category methods. The reason is that good pivots are outliers, but outliers are not always good pivots [33]. Thus, we present a hybrid pivot selection method using a newly introduced criterion to achieve both the efficiency and the effectiveness.

2.3 Querying Metric Spaces

A metric space is a tuple (M, d), in which M is the domain of objects and d is a distance function that defines the similarity between the objects in M. In particular, the distance function d satisfies 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, several spatial queries in metric spaces have been explored.

Similarity search (including range and nearest neighbor (NN) queries) in metric spaces has been summarized well in the literature [28], [38], [39], and has been studied for each MAM discussed in Section 2.1. In addition, cost models [40], [41] are also derived for metric similarity queries.

A similarity join retrieves pairs of objects that are within a distance ε of each other. This operator has been investigated in metric spaces, and efficient solutions exist [42]. Recently, Paredes and Reyes [18] handle similarity joins using LTC, which indexes jointly two sets. Fredriksson and Braithwaite [43] improve the Quickjoin algorithm [42] for similarity joins. Silva and Pearson [44], [45] develop a non-blocking similarity join operator, DBSimJoin, and explore index-based similarity joins. Nevertheless, existing solutions are all partition based methods. In this paper, we utilize the pivot mapping and the space-filling curve to further improve the efficiency of metric similarity joins, which is also confirmed in Section 6.4, compared with state-of-the-art approaches [43], [44].

3 The SPB-tree

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.

3.1 Construction Framework

As shown in Fig. 1, the construction of an 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 can utilize geometric information unavailable in the metric space. In the second stage, we utilize the SFC to map the data points in the vector space into integers in an one-dimensional space, since SFC preserves the spatial proximity when performing the dimensionality reduction. Finally, a B$^+$-tree with MBB information is employed to index the resulting integers.

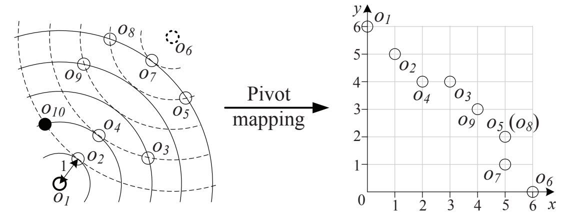
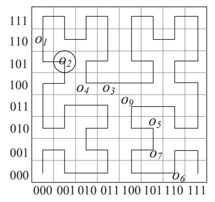
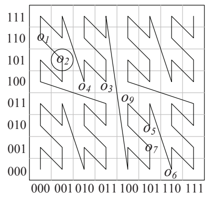


Fig. 2: Pivot mapping



(a) Hilbert curve mapping



(b) Z curve mapping

Fig. 3: Space-filling curve mapping

The use of a B$^+$-tree with MBB information is attractive, because bulk-loading, insertion, and deletion operations on the B$^+$-tree are simple and effective. The use of an SFC can cluster objects into compact regions, and reduce the amount of storage needed for pre-computed distances. Although the ZB-tree [46] and UB-tree [47] that combine a Z-curve and a B$^+$-tree can be employed to index objects after the pivot mapping, they are designed to use a specific SFC, and the ZB-tree is only suitable for skyline queries. In contrast, any SFC (e.g., a Hilbert curve that offers better proximity preservation than a Z-curve, to be verified in Section 6.1) can be used in the SPB-tree.

Pivot Mapping. Given a pivot set $P = \{p_1, \ldots, p_n\}$, a metric space (M, d) can be mapped to a vector space (R^n, L_∞). Specifically, an object o in the metric space is represented as a point $\phi(o) = \langle d(o, p_1), \ldots, d(o, p_n)\rangle$ in the vector space. Consider the example in Fig. 2, where $O = \{o_1, \ldots, o_9\}$ and the 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 denotes $d(o_i, o_1)$ and the y-axis denotes $d(o_i, o_6)$ $(1 \leq i \leq 9)$.

Given objects o_i, o_j, and p in a metric space, $d(o_i, o_j) \geq |d(o_i, p) - d(o_j, p)|$ according to the triangle inequality. Hence, for a pivot set P, $d(o_i, o_j) \geq \max\{|d(o_i, p_t) - d(o_j, p_t)| \mid p_t \in P\} = D(\phi(o_i), \phi(o_j))$, in which $D(\ )$ is the L_∞-norm. Consequently, the distance in the mapped vector space is a *lower bound* on that in the metric space. For example, in Fig. 2, $d(o_2, o_3) > D(\phi(o_2), \phi(o_3)) = 2$.

Space-Filling Curve Mapping. Given a vector $\phi(o)$ after pivot mapping, and assume that the range of $d(\ )$ in the metric space is *discrete* integers (e.g., $d()$ is edit distance), SFC can map $\phi(o)$ to an integer $SFC(\phi(o))$. Consider the SFC mapping example in Fig. 3, where $SFC(\phi(o_2)) = 18$ for the Hilbert curve and $SFC(\phi(o_2)) = 19$ for the Z-curve. Without loss of generality, we use the Hilbert curve in the rest of the paper.

Considering the range of $d(\ )$ in a metric space may be *continuous* real numbers, δ-*approximation* is utilized to partition the real range into discrete integers, i.e., $0, \ldots, \lfloor d^+/\delta \rfloor$. Thus, the whole vector space can be partitioned into $(\frac{d^+}{\delta})^{|P|}$ cells. Then, given an δ, $\phi(o)$ can be approximated as $\langle \lfloor d(o, p_1)/\delta \rfloor, \ldots, \lfloor d(o, p_n)/\delta \rfloor \rangle$. In the rest of the paper, for simplification, we assume that the range of $d(\ )$ is *discrete* integers, even though the presented techniques can be easily adapted to a *continuous* real range using δ-*approximation*.

To design efficient indexing for querying metric spaces, two important issues have to be addressed: (1) How should we pick pivots? (2) Which index structures can be used to support metric queries? We discuss the first issue in Section 3.2, and turn to the second issue in Section 3.3.

3.2 Pivot Selection

The pivots used can significantly affect search performance. In order to achieve strong pruning power, the mapping to the vector space should preserve the proximity in the metric space, i.e., the lower-bound distances should be close to the actual distances. 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 1.

Definition 1: 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.,

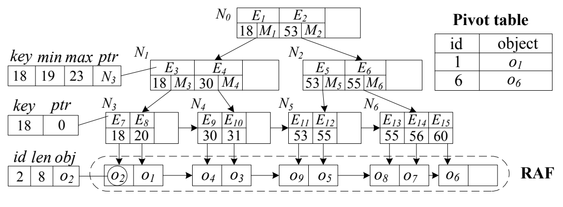
$$precison(P) = \frac{1}{|OP|} \sum_{\langle o_i, o_j \rangle \in OP} \frac{D(\phi(o_i), \phi(o_j))}{d(o_i, o_j)}$$

The more pivots in P, the better the pruning capability; however, the cost of using the transformed objects also increases. More specifically, 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 thus, *precision*(P) approaches 1. Therefore, we can discard more objects when 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., the $D(\phi(o_i),\ \phi(o_j))$ computation cost) to prune unqualified objects increases. Thus, the appropriate number of the pivots needed to achieve high query efficiency is related to the intrinsic dimensionality of the dataset [6], [28], which is also verified in Section 6.1. The intrinsic dimensionality of a metric dataset can be calculated as $\rho = \mu^2/2\sigma^2$, where μ and σ are the mean and variance of the pairwise distances in the dataset [28].

Determining a pivot set P (from O) with a fixed size that maximizes *precision*(P) takes $O(\frac{|O|!}{|P|!(|O|-|P|)!})$ time, which is costly. To reduce the pivot selection cost, we propose an *HF based Incremental pivot selection algorithm* (HFI) (see Appendix A [48]) that first employs HF algorithm [6] 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 [33]. Since the time complexity of HF is $O(|O|)$, the time complexity of HFI is $O(|O| + |P||CP|)$, 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 [37]), which is enough to find all the outliers in our experiments.

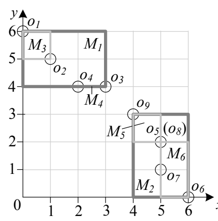
3.3 Index Structure

An SPB-tree has three parts, a pivot table, a B$^+$-tree, and an RAF. Fig. 4 depicts an SPB-tree for an object set $O = \{o_1, \ldots, o_9\}$ in Fig. 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

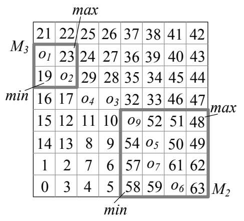


N_0 E_1 E_2 18 M_1 53 M_2
Pivot table
id object
1 o_1
6 o_6
key min max ptr
18 19 23 N_3
N_1 E_3 E_4 18 M_3 30 M_4
N_2 E_5 E_6 53 M_5 55 M_6
key ptr
18 0
N_3 E_7 E_8 18 20
N_4 E_9 E_{10} 30 31
N_5 E_{11} E_{12} 53 55
N_6 E_{13} E_{14} E_{15} 55 56 60
id len obj
2 8 o_2
o_2 o_1 o_4 o_3 o_9 o_5 o_8 o_7 o_6
RAF

(a) The SPB-tree



(b) MBBs of an SPB-tree



max
M_3
21 22 25 26 37 38 41 42
o_1 23 24 27 36 39 40 43
19 o_2 29 28 35 34 45 44
min 16 17 o_4 o_3 32 33 46 47
15 12 11 10 o_9 52 51 48 max
14 13 8 9 54 o_5 50 49
1 2 7 6 57 o_7 61 62
0 3 4 5 58 59 o_6 63 M_2
min

(c) The MBB representation

Fig. 4: Example of an SPB-tree

indexes the SFC values of objects after the pivot mapping. Each entry in a 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) a pointer *ptr* to the actual object in the RAF. For example, in Fig. 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 node 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 $\langle L_1, \ldots, L_{|P|}\rangle$ and $\langle U_1, \ldots, U_{|P|}\rangle$, to represent the MBB M (= $\{[L_i, U_i] \mid i \in [1, |P|]\}$) of the root node N of its subtree. Specifically, an MBB M denotes the axis aligned *minimum bounding box* of all $\phi(o)$ with $SFC(\phi(o)) \in N$, and thus, L_i and U_i represent the minimum and maximum values of $\phi(o)$ in dimension i. For instance, the non-leaf entry E_3 uses *min* (= 19) and *max* (= 23) to represent M_3 of N_3.

Unlike the compact partitioning methods (e.g., the M-tree), which store actual objects directly in the index, the SPB-tree utilizes an RAF to store objects separately. The RAF stores the objects in ascending order of their SFC values. 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 for supporting 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. In Fig. 4, the RAF entry associated with object o_2 contains object identifier 2, object length 8, and the real object o_2.

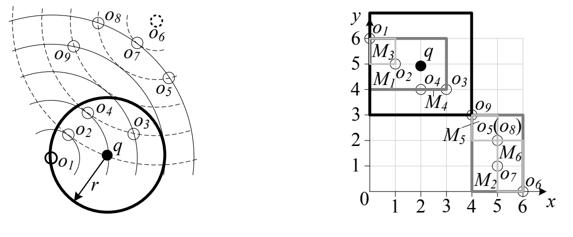
Also, we develop bulk-loading (see Appendix B) and insertion/deletion (see Appendix C) operations of the SPB-tree.

4 Similarity Search

We first formalize similarity search in metric spaces, and then present efficient algorithms for processing similarity queries. Finally, we derive corresponding cost models.

4.1 Problem Definitions

Similarity search is useful in many settings. For example, in pattern recognition, similarity queries can be used to classify a



(a) Metric space (b) Vector space

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

new object according to the labels of already classified nearest neighbors. We study the problem of similarity search in metric spaces, which includes range and kNN queries.

Definition 2: Given an object set O, a query object q, and a search radius r in a metric space M, a *range query* finds the objects in O that are within distance r to q, i.e., $RQ(q, O, r) = \{o | o \in O \wedge d(q, o) \leq r\}$.

Definition 3: Given an object set O, a query object q, and an integer k in a metric space M, a kNN *query* retrieves 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 a word set O = {“citrate”, “defoliates”, “defoliated”, “defoliating”, “defoliation”}, for which the edit distance is the metric. The range query RQ(“defoliate”, O, 1) returns the words in O with distances to “defoliate” bounded by 1. The range query result is {“defoliates”, “defoliated”}. Then, the nearest neighbor 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 distance ties. The target of our proposed algorithms is to find one possible instance.

4.2 Range Query Processing

Our indexing approach applies two mappings to the original data objects. We then need to apply corresponding mappings to a circular range query. Consider, Fig. 5(a), where the *circle* denotes a query range, and $RQ(q, O, 2) = \{o_1, o_2, o_3, o_4\}$. Given a pivot set P, the range is mapped into the corresponding vector space. The *thick black rectangle* in Fig. 5(b) represents the *mapped range region* using $P = \{o_1, o_6\}$. To compute $RQ(q, O, r)$, we only need to consider 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 in $RQ(q, O, r)$, then $\phi(o)$ is contained in the mapped range region $RR(q, r)$, where $RR(q, r) = \{(s_1, \ldots, s_{|P|}) | \ 1 \leq i \leq |P| \wedge s_i \geq 0 \wedge s_i \in [d(q, p_i) - r, d(q, p_i) + r]\}$.

Proof. Please refer to Appendix D. □

Based on Lemma 1, if the MBB of a node N in the SPB-tree does not intersect $RR(q, r)$, we can discard N. Note that, MBB can be easily obtained by using SFC values *min* and *max* stored in the SPB-tree. Considering the range query in Fig. 5 and the corresponding SPB-tree in Fig. 4, N_6 can be discarded because $M_6 \cap RR(q, r) = \emptyset$.

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

Algorithm 1 Range Query Algorithm (RQA)

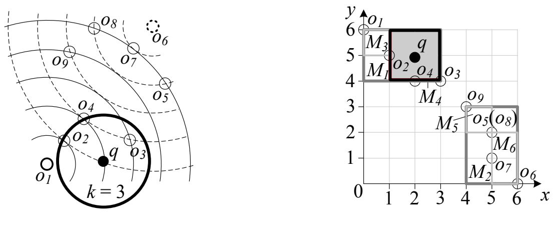
```
 Input: a query object q, a radius r, an object set O indexed by an SPB-tree
 Output: the result set RQ(q, O, r) of a range query
 1: compute $\phi(q)$ using a pivot table P
 2: RR(q, r) = ComputeRR($\phi(q)$, r)  // comprute the range region
 3: push the root node of a B$^+$-tree onto heap H
 4: while H ≠ ∅ 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) ∩ RR(q, r) ≠ ∅ then  // Lemma 1
 9:         push e.ptr into H
10:   else  // N is a leaf node
11:     if MBB(N) ⊆ RR(q, r) then  // Lemma 1
12:       for each entry e in N do
13:         VerifyRQ(e, false)
14:     else if |RR(q, r) ∩ MBB(N) | < |N|
        // only subset of entries in N included in the intersected region
15:       S = computeSFC(RR(q, r) ∩ MBB(N))
          // compute SFC values in the intersected region
16:       s = S.get_first() and e = N.get_first()
17:       while s ≠ NULL and e ≠ 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(q, r) ∩ MBB(N) | ≥ |N|
22:       for each entry e in N do
23:         VerifyRQ(e, true)
24: return RQ(q, O, r)

 Function: VerifyRQ(e, flag)
25: if flag and $\phi(o)$ ∉ RR(q, r) then return  // Lemma 1
26: if the condition of Lemma 2 is satisfied then  // Lemma 2
27:   insert e.ptr into RQ(q, O, r) and return
28: if d(q, e.ptr) ≤ r then
29:   insert e.ptr into RQ(q, O, r)
```

Lemma 2: Given a pivot set P, for an object o in O, if there exists a pivot p_i ($\in P$) satisfying $d(o, p_i) \leq r - d(q, p_i)$, then o belongs to $RQ(q, O, r)$.

Proof. Please refer to Appendix E. □

Consider again the example in Fig. 5, where $O = \{o_1, \ldots, o_9\}$ and $P = \{o_1, o_6\}$, and we suppose $r = 3$. Then, for an object o_2, there exists a pivot p_1 (= o_1) such that $d(o_2, p_1) = r - d(q, p_1)$. Hence, o_2 is included in $RQ(q, O, 3)$, and the computation of the distance $d(q, o_2)$ is unnecessary.

The pseudo-code of the *Range Query Algorithm* (RQA) is presented in Algorithm 1. First, RQA computes $\phi(q)$ using a pivot table P. Then it calls a ComputeRR function to obtain $RR(q, r)$. Next, it pushes the root node of the B$^+$-tree into a heap H, 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 $MBB(e.ptr) \cap RR(q, r) \neq \emptyset$ into H (lines 7–9). Otherwise (i.e., N is a leaf node), if $MBB(N) \subseteq RR(q, r)$, for each entry in N, VerifyRQ is utilized to determine whether RQA inserts the corresponding object into $RQ(q, O, r)$. 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 $RR(q, r) \cap MBB(N)$ is smaller than that of entries in N, RQA first invokes a computeSFC function 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., $\phi(o) \notin RR(q, r)$) do not need to be verified by Lemma 1 (line



(a) Metric space (b) Vector space

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

25). Finally, the query result $RQ(q,\ O,\ r)$ is returned (line 24).

Example 1. Please refer to Appendix F.

4.3 *k*NN Query Processing

Given an object set O in a metric space, a *k*NN query finds from O the k NNs of a specified query object q. For example, in Fig. 6, the result set $kNN(q,\ 3) = \{o_3,\ o_2,\ o_4\}$. The *k*NN query can be regarded as a range query with radius ND_k, where ND_k is the distance from q to its k-th NN. However, ND_k is not known in advance, which makes the *k*NN query tricker than the range query. In order to reduce query cost, our *kNN Query Algorithm* (NNA) follows an incremental traversal paradigm, i.e., NNA visits 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 unqualified entry accesses, a pruning rule is developed.

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. Please refer to Appendix G. □

Note that, for Lemma 3, $curND_k$ is obtained and updated during *k*NN search. An example of Lemma 3 is depicted in Fig. 6 and Fig. 4. Assume that $curND_k = 1$, E_2 can be pruned as $MIND(q,\ E_2) > 1$. Due to Lemma 3, *k*NN retrieval can stop when the entry E visited satisfies the early termination condition, i.e., $MIND(q,\ E) \geq curND_k$. Therefore, NNA is optimal in terms of the number of distance computations, since it only needs to consider the objects in $RR(q,\ ND_k)$, as stated in Lemma 4 below.

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

Proof. Please refer to Appendix H. □

The pseudo-code of NNA is depicted in Algorithm 2. First of all, NNA sets $curND_k$ as 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 is satisfied (line 5). In every 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 query result $kNN(q,\ k)$ is returned (line 13).

Example 2. Please refer to Appendix I.

Algorithm 2 *k*NN Query Algorithm (NNA)

```
Input: a query object $q$, an integer $k$, an object set $O$ indexed by an 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 onto $H$
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   // Lemma 3
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 $kNN(q, k)$
```

NNA evaluates the objects contained in $RR(q,\ ND_k)$ based on ascending order of their $MIND$ to q, incurring random page accesses in RAF. Since SFC preserves spatial proximity, the objects to be verified are expected to be close to each other in RAF. Thus, with a small cache, we can avoid duplicate RAF page accesses, as to be confirmed in Section 6.1. Nevertheless, for a *k*NN 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: 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 to the leaf entries can be evaluated immediately. Although the greedy traversal paradigm may result in unnecessary distance computations for verifying the objects not contained in $RR(q,\ ND_k)$, it might still achieve computational efficiency because the objects in the same leaf node satisfy spatial proximity, as to be demonstrated in Section 6.1 as well.

4.4 Cost Models

We proceed to derive cost models for range and *k*NN queries, to estimate their I/O and CPU costs. With the help of cost models, we can choose promising execution strategies.

We use the number of distance computations as a proxy for the CPU cost [2], [43]. To determine this cost, we utilize the distance distribution, which is the natural way to characterize metric datasets. The overall distribution of the distances from objects in O to a pivot p_i is defined as:

$$F_{p_i}(r) = \mathbf{Pr}\{d(o,p_i) \leq r\}, \tag{1}$$

where o is a random object in O. Distance distributions $F_{p_i}(r)$ 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 sampling:

$$F(r_1,\ldots,r_{|P|}) = \mathbf{Pr}\{d(o,p_1) \leq r_1,\ldots,d(o,p_{|P|}) \leq r_{|P|}\} \tag{2}$$

The *union* distance distribution function $F(r_1,\ldots,r_{|P|})$ can be statistically obtained during SPB-tree construction, when distances $d(o,\ p_i)$ $(1 \leq i \leq |P|)$ are computed.

To determine the *estimated number of distance computations* (*EDC*) for a similarity query, it is enough to sum two terms: 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) can be estimated as the probability that $\phi(o)$ is contained in $RR(q,\ r)$ according to Lemma 1:

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

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

The kNN query can be regarded as a range query with search radius $r = ND_k$, in which ND_k denotes the distance from q to its k-th nearest neighbor. Hence, in order to drive EDC for kNN retrieval, the first step is to determine ND_k. 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:

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

However, $F_q(r)$ is not known in advance. We therefore employ a simple but efficient method [40] to estimate $F_q(r)$ using $F_{p_i}(r)$, where p_i is the nearest neighbor of q. Thus, to obtain EDC for kNN search, due 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 $RR(q,\ ND_k)$, which can be calculated using equation (4) with $r = eND_k$ (computed by equation (5)).

As the objects accessed in the RAF are expected to be stored close to each other, the number of RAF page accesses can be estimated as $\frac{EDC}{f}$, in which EDC is an estimation of the total number of the objects visited and f represents the average number of the objects accessed per RAF page. Hence, the *expected number of page accesses* (EPA) of a similarity query can be calculated as:

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

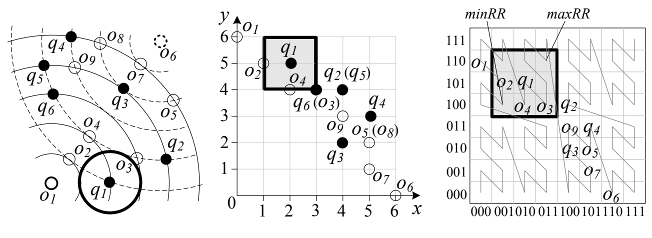
$$\text{where } I(M_i) = \begin{cases} 1 & M_i \text{ intersects the search region} \\ 0 & \text{otherwise} \end{cases}$$

5 Similarity Join

We first formalize the similarity join in metric spaces and then propose an efficient algorithm for processing similarity joins. Finally, we derive the corresponding cost models.

5.1 Problem Definition

The similarity join is an important operator in many applications, including data cleaning, web page deduplication, click fraud detection, entity resolution, and data integration. As an example, a very important data cleaning operator [49] is that of joining similar data. In a sales data warehouse, due to typing mistakes and differences in conventions, errors in the data result in product and customer names in sales records may not be matching exactly with those in the master product catalog and reference customer registration records,



(a) Metric space (b) Vector space (c) Z-order curve

Fig. 7: Illustration of $SJ(Q, O, \varepsilon)$

respectively. Similarity joins can be used to eliminate such errors. Similarity join applications involve the identification of *close* objects, and the closeness is evaluated using similarity functions specific to the particular domain and application. To this end, we study similarity joins in metric spaces that support a wide range of data types and similarity metrics.

Definition 4: Given two object sets Q and O in a metric space M and a distance threshold ε, *a similarity join* finds the object pairs $\langle q,o\rangle$ in $Q \times O$ with distances $d(q,o)$ within ε, i.e., $SJ(Q,\ O,\ \varepsilon) = \{\langle q,o\rangle | q \in Q \wedge o \in O \wedge d(q,\ o) \leq \varepsilon\}$.

Consider two word sets Q = {"defoliate", "defoliates", "defoliation"} and O = {"citrate", "defoliated", "defoliating"}, for which the edit distance is the metric. The similarity join $SJ(Q,\ O,\ 1)$ retrieves the word pairs in $Q \times O$ with distances bounded by 1, yielding the result {⟨"defoliate", "defoliated"⟩, ⟨"defoliates", "defoliated"⟩}.

5.2 Similarity Join Processing

Consider the similarity join running example shown in Fig. 7, where $Q = \{q_i \mid 1 \leq i \leq 6\}$ and $O = \{o_j \mid 1 \leq j \leq 9\}$. Assume that ε = 1 and the L_2-norm is utilized. The result of $SJ(Q,\ O,\ 1)$ is $\{\langle q_1,o_3\rangle, \langle q_2,o_3\rangle, \langle q_3,o_7\rangle, \langle q_4,o_8\rangle, \langle q_4,o_9\rangle, \langle q_5,o_9\rangle, \langle q_6,o_9\rangle\}$. A similarity join $SJ(Q,\ O,\ \varepsilon)$ can be regarded as multiple range queries $RQ(q,\ O,\ \varepsilon)$ on O for every object q in Q, i.e., $SJ(Q,\ O,\ \varepsilon) = \{\cup_{q\in Q}\{\langle q,o\rangle \mid o \in RQ(q,\ O,\ \varepsilon)\}$. Hence, to obtain $SJ(Q,\ O,\ \varepsilon)$, we only need to consider objects o ($\in O$) with their corresponding vector $\phi(o)$ in the mapping vector space contained in the mapped range region $RR(q,\ \varepsilon)$ (introduced in Section 4.2) for each q, as stated in Lemma 5.

Lemma 5: Given a pivot set P, if an object pair $\langle q,\ o\rangle$ belongs to $SJ(Q,\ O,\ \varepsilon)$, then $\phi(o)$ (= $\langle d(o,p_i) \mid p_i \in P\rangle$) is contained in $RR(q,\ \varepsilon)$.

Proof. Assume that there exists an object pair $\langle q,\ o\rangle \in SJ(Q,\ O,\ \varepsilon)$ but $\phi(o) \notin RR(q,\ \varepsilon)$, i.e., $\exists\ p_i \in P$, $(d(o,\ p_i) > d(q,\ p_i) + \varepsilon \vee d(o,\ p_i) < d(q,\ p_i) - \varepsilon)$. According to the triangle inequality, $d(q,\ o) \geq |d(q,\ p_i) - d(o,\ p_i)|$. If $d(o,\ p_i) > d(q,\ p_i) + \varepsilon$ or $d(o,\ p_i) < d(q,\ p_i) - \varepsilon$, then $d(q,\ o) \geq |d(o,\ p_i) - d(q,\ p_i)| > \varepsilon$, which contradicts our assumption. □

Based on Lemma 5, if any object o is not contained in $RR(q,\ \varepsilon)$, we can discard o for q. For instance, in Fig. 7, object o_1 can be pruned for q_1 as $\phi(o_1) \notin RR(q_1,\ \varepsilon)$. Note that, the similarity join is symmetric, i.e., $SJ(Q,\ O,\ \varepsilon) = SJ(O,\ Q,\ \varepsilon)$. If an object pair $\langle q,o\rangle$ is included in $SJ(Q,\ O,\ \varepsilon)$, then $\phi(o) \in RQ(q,\ O,\ \varepsilon)$ and $\phi(q) \in RQ(q,\ Q,\ \varepsilon)$ according to Lemma 5.

Based on the property of Z-order curve, given two points $\phi(o) = \langle s_1,\ \ldots,\ s_{|P|}\rangle$ and $\phi(o') = \langle s'_1,\ \ldots\ ,\ s'_{|P|}\rangle$ in the vector space, if $s_i \leq s'_i$ for all $1 \leq i \leq |P|$, then $SFC(\phi(o)) \leq SFC(\phi(o'))$. Let $minRR(q,\ \varepsilon)$ and $maxRR(q,\ \varepsilon)$ be the SFC

Algorithm 3 Similarity Join Algorithm (SJA)

```
Input: SPB-trees $SPB_Q$ and $SPB_O$ to index $Q$ and $O$, a distance threshold $\epsilon$
Output: the result set $SJ(Q, O, \epsilon)$ of a similarity join
1: $L_Q = L_Q = \varnothing$
2: traverse $SPB_Q$ and $SPB_O$ to get the first leaf entries $E_Q$ and $E_O$
3: while $E_Q \neq \varnothing$ or $E_O \neq \varnothing$ do
4:   if $E_O = \varnothing$ or $E_Q.key < E_O.key$ then   // all the entries in $SPB_O$ are visited
                                     // or the SFC value of the current $E_Q$ is smaller than that of $E_O$
5:     Verify($E_Q.ptr$, $L_O$)
6:     insert $E_Q.ptr$ into $L_Q$
7:     $E_Q = E_Q$.get_next()   // get the next leaf entry $E_Q$ in $SPB_Q$
8:   else // $E_Q = \varnothing$ or $E_Q.key \geq E_O.key$
9:     Verify($E_O.ptr$, $L_Q$)
10:    insert $E_O.ptr$ into $L_O$
11:    $E_O = E_O$.get_next()   // get the next leaf entry $E_O$ in $SPB_O$
12: return $SJ(Q, O, \epsilon)$

Function: Verify($q$, $L$)
13: $o = L$.get_last( )   // get the last entry in $L$
14: while $o \neq \varnothing$ do
15:   if $maxRR(o, \epsilon) < SFC(\phi(q))$ then   // Lemma 6
16:     delete $o$ from $L$ and continue
17:   if $SFC(\phi(o)) \geq minRR(q, \epsilon)$   // Lemma 6
18:     if $\phi(o) \in RR(q, \epsilon)$ then   // Lemma 5
19:       if $d(q, o) \leq \epsilon$ then
20:         insert $\langle q, o\rangle$ into $SJ(Q, O, \epsilon)$
21:   $o = o$.get_pre( )   // get the previous entry in $L$
```

values for the left-lower and right-upper points in $RR(q, \varepsilon)$, i.e., $minRR(q, \varepsilon) = SFC(\phi(\langle d(q, p_1) - \varepsilon, \ldots, d(q, p_{|P|}) - \varepsilon\rangle))$ and $maxRR(q, \varepsilon) = SFC(\phi(\langle d(q, p_1) + \varepsilon, \ldots, d(q, p_{|P|}) + \varepsilon\rangle))$. To get the result set, we only need to verify the objects o whose $SFC(\phi(o))$ are contained in the range $[minRR(q, \varepsilon), maxRR(q, \varepsilon)]$, as stated below.

Lemma 6: Assume that the Z-order curve and a pivot set P are used. If an object pair $\langle q, o\rangle$ is enclosed in $SJ(Q, O, \varepsilon)$, then $SFC(\phi(o)) \in [minRR(q, \varepsilon), maxRR(q, \varepsilon)]$.

Proof. According to the definition of $RR(q, \varepsilon)$, $RR(q, \varepsilon) = \{\langle s_1, \ldots, s_{|P|}\rangle | \ d(q, p_i) - \varepsilon \leq s_i \leq d(q, p_i) + \varepsilon \ (1 \leq i \leq P)\}$. Thus, for $\phi(o) \in RR(q, \varepsilon)$, $minRR(q, \varepsilon) \leq SFC(\phi(o)) \leq maxRR(q, \varepsilon)$, according to the property of the Z-order curve. Based on Lemma 5, if an object pair $\langle q, o\rangle$ is included in the query result, then $\phi(o)$ is certainly contained in $RR(q, \varepsilon)$, and hence, $minRR(q, \varepsilon) \leq SFC(\phi(o)) \leq maxRR(q, \varepsilon)$. □

Consider the example shown in Fig. 7, where $minRR$ $(q_1, 1)$ and $maxRR(q_1, 1)$ equal to 18 and 30, respectively. According to Lemma 6, objects o_i $(5 \leq i \leq 9)$ can be pruned since $SFC(\phi(o_i)) > maxRR(q_1, 1)$. Based on Lemma 6, we can stop evaluating objects o for q, when $SFC(\phi(o))$ exceeds $maxRR(q, \varepsilon)$ if the objects o are retrieved in ascending order of their SFC values, or when $SFC(\phi(o))$ is smaller than $minRR(q, \varepsilon)$ if the objects o are retrieved in descending order.

A naive solution for similarity join $SJ(Q, O, \varepsilon)$ is to perform $|Q|$ range queries $RQ(q, O, \varepsilon)$ $(q \in Q)$. However, it is inefficient because it has to scan the object set O multiple times, resulting in high costs, especially for larger $|Q|$. In view of this, we develop an efficient algorithm, termed *Similarity Join Algorithm* (SJA), which scans the object sets Q and O only once. In particular, SPB-trees are assumed on the two object sets Q and O. As the trees index the object sets on the objects' SFC values, the leaf levels contain the objects in ascending SFC order. A merge join is then performed on the leaf levels of two SPB-trees. More specifically, objects q and o stored in the leaf levels of two SPB-trees are visited in ascending order of SFC values, and two lists are used to keep the objects q or o visited, respectively. When an object q is visited, the algorithm finds answer object pairs $\langle q,o\rangle$ for q in objects o visited before q; when an object o is visited, SJA finds answer object pairs $\langle q,o\rangle$ for o in objects q visited before o. Note that, Lemmas 5 and 6 can be utilized to improve query efficiency by avoiding unnecessary verifications, and Lemma 6 is employed to prune unqualified q and o from the lists.

Lemma 7: The presented algorithm SJA can return exactly actual result set $SJ(Q, O, \varepsilon)$, i.e., the algorithm has no missing and duplicated answer object pairs.

Proof. To prove the algorithm has no missing and duplicated answer object pairs, we only need to prove that all the objects o in O are verified once and only once for each q. First, we show that all the objects o in O are verified for every q, i.e., there is no missing object pairs. For objects o with $SFC(\phi(o)) \leq SFC(\phi(q))$, they are verified for an object q when visiting q; For objects o with $SFC(\phi(o)) > SFC(\phi(q))$, they are verified for an object q when visiting each o. Second, all the objects o in O are verified only once for every q, i.e., there is no duplicated answer pairs, because objects are visited in order of SFC values, and are only verified for objects in the other object set already visited. The proof completes. □

The pseudo-code of SJA is depicted in Algorithm 3. First, SJA initializes two lists L_Q and L_O to empty, and gets the first leaf entries E_Q and E_O of the SPB-trees SPB_Q and SPB_O, respectively (lines 1–2). Then, the algorithm performs a *while loop* to visit the leaf entries in ascending order of SFC values (i.e., *key*s stored in the SPB-trees), until all the leaf entries of both SPB_Q and SPB_O are evaluated (i.e., E_Q and E_O are empty) (lines 3–11). Each time, if all the leaf entries of SPB_Q are visited (i.e., $E_O = \emptyset$) or $E_Q.key \leq E_O.key$, *Verify* function is invoked (line 5) to find the objects stored in L_O with their distances to $E_Q.ptr$ within ε and to delete unqualified o from L_O. After that, SJA inserts $E_Q.ptr$ into the list L_Q, and gets the next leaf entry E_Q in SPB_Q (lines 6–7). Otherwise, if E_O is empty (i.e., all leaf entries in SPB_O have been visited) or $E_Q.key > E_O.key$, the algorithm invokes *Verify* function (line 9) to find the objects stored in L_Q with their distances to $E_O.ptr$ within ε and to prune unqualified q in L_Q. Next, SJA inserts $E_O.ptr$ in the list L_O and gets the next leaf entry E_O in SPB_O (lines 10–11). Finally, the query result $SJ(Q, O, \varepsilon)$ is returned.

The memory complexity of SJA is $O(|L_Q| + |L_O| + |SJ(Q, O, \varepsilon)|)$, where $|L_Q|$ and $|L_O|$ denote the maximum sizes of lists L_Q and L_O, and $|SJ(Q, O, \varepsilon)|$ represents the cardinality of the final result set. Here, $|L_Q|$ equals to the maximum number of objects q in Q with $SFC(\phi(q)) \in [minRR(o, \varepsilon), maxRR(o, \varepsilon)]$ $(o \in O)$, as derived below. Assume that q_i is the first entry in the current L_Q, then objects o in the current L_O satisfy that $SFC(\phi(o)) \leq maxRR(q_i, \varepsilon)$, because q_i can be removed from L_Q when o' with $SFC(\phi(o')) > maxRR(q_i, \varepsilon)$ is inserted into L_O due to Lemma 6. For the possible last entry o_j in L_O with $SFC(\phi(o_j)) = maxRR(q_i, \varepsilon)$, we can get that $SFC(\phi(q_i)) = minRR(o_j, \varepsilon)$, and thus objects q in L_Q satisfy that $minRR(o_j, \varepsilon) \leq SFC(\phi(q)) \leq maxRR(o_j, \varepsilon)$. Then $|L_O|$ can be derived similarly. Therefore, $|L_Q|$ and $|L_O|$ depend on ε and the data distribution, and they can grow up to $|Q|$ and $|O|$ in the worst case respectively.

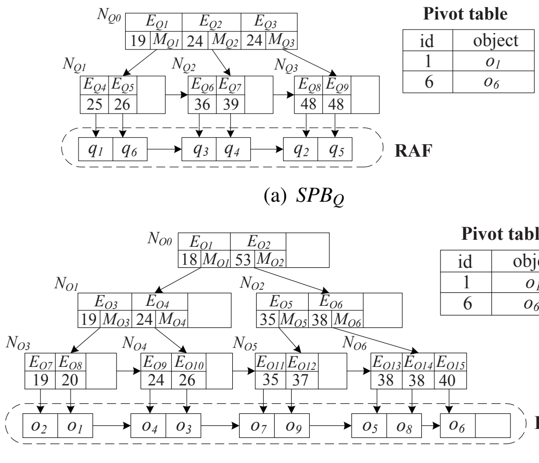


Fig. 8: SPB-trees built on Q and O

Example 3. We illustrate SJA using the example depicted in Fig. 7 and Fig. 8, and we suppose ε = 1. Initially, SJA sets L_Q and L_O to empty, and traverses SPB_Q and SPB_O to get the first leaf entries E_{Q_4} and E_{O_7}. Then, it performs a while loop. In the first iteration, since $E_{Q_4}.key > E_{O_7}.key$, SJA invokes *Verfiy*($o_2$, L_Q), inserts o_2 (i.e., $E_{O_7}.ptr$) into L_O, and gets the next leaf entry E_{O_8} in SPB_O. In the following two iterations, the leaf entries E_{O_8} and E_{O_9} are processed similarly, after which L_O = $\{o_2, o_1, o_4\}$, and the next leaf entry $E_{O_{10}}$ in SPB_O is obtained. In the fourth iteration, as $E_{Q_4}.key > E_{O_{10}}.key$, *Verify*($q_1$, L_O) is invoked, where o_2, o_1, and o_4 are pruned for $E_{Q_4}.ptr$ (= q_1). Then, SJA inserts q_1 into L_Q, and gets the next leaf entry E_{Q_5} in SPB_Q. In the fifth iteration, as $E_{Q_5}.key = E_{O_{10}}.key$, the algorithm calls *Verify*(o_3, L_Q) to add $\langle q_1, o_3\rangle$ to *SJ*(Q, O, 1), inserts o_3 into L_O, and gets the next leaf entry $E_{O_{11}}$ in SPB_O. The algorithm proceeds in the same manner until all the leaf entries in SPB_Q and SPB_O are visited, and it returns the final result set *SJ*(Q, O, 1) = $\{\langle q_1,o_3\rangle, \langle q_2,o_3\rangle, \langle q_3,o_7\rangle, \langle q_4,o_8\rangle, \langle q_4,o_9\rangle, \langle q_5,o_9\rangle, \langle q_6,o_9\rangle\}$. □

5.3 Cost Models

We proceed to derive cost models for similarity joins, in order to estimate I/O and CPU costs. With the help of cost models, we can choose promising execution strategies for queries that involve similarity joins.

We use the number of distance computations as a proxy for the CPU cost [2], [43]. To determine the *estimated number of distance computations* (*EDC*) for a similarity join, it is sufficient to sum all the distance computations needed when finding, for each q in Q, the objects o in O with distances to q bounded by ε. According to Lemma 5, for every q, the number of distance computations needed can be estimated as the number of the objects o having $\phi(o)$ contained in $RR(q,\varepsilon)$, i.e., $|O| \times \mathbf{Pr}(\phi(o) \in RR(q, \varepsilon))$. We thus get

$$EDC = \sum_{q\in Q} |O| \times \mathbf{Pr}(\phi(o) \in RR(q,\varepsilon)) \tag{7}$$

Here, $\mathbf{Pr}(\phi(o) \in RR(q, \varepsilon))$ can be computed using equation (4), and *EDC* is degraded to $|Q| \times |O|$ in the worst case.

The I/O cost for the similarity join includes two parts: B$^+$-tree page accesses and RAF page accesses. Since SJA traverses SPB-trees built on two object sets Q and O only once, it is sufficient to sum the number of B$^+$-tree leaf pages and RAF pages in the SPB-trees. Let $|SPB_Q|$ ($|SPB_O|$) denote the total number of B$^+$-tree leaf pages of SPB_Q (SPB_O), and let f_Q (f_O) represent the average number of the objects per RAF page for SPB_Q (SPB_O). The *expected number of page accesses* (*EPA*) of a similarity join can be calculated as:

$$EPA = |SPB_Q| + |SPB_O| + \frac{|Q|}{f_Q} + \frac{|O|}{f_O} \tag{8}$$

TABLE 2: Statistics of the datasets used

Dataset	Cardinality	Dim.	Ins.	Measurement	Prec.
Words	611,756	1∼34	4.9	*Edit distance*	0.582
Color	112,682	16	2.9	*L_5-norm*	0.935
DNA	1,000,000	108	6.9	*Cosine similarity under tri-gram counting space*	0.470
Signature	49,740	64	14.8	*Hamming distance*	0.424
Synthetic	1,000,000	20	4.76	*L_2-norm*	0.713

TABLE 3: Parameter ranges and default values

Parameter	Setting	Default
number of pivots $\|P\|$	1, 3, 5, 7, 9	5
cache size (pages)	0, 8, 16, 32, 64, 128	32
δ	0.001, 0.003, 0.005, 0.007, 0.009	0.005
cardinality	200K, 400K, 600K, 800K, 1000K	600K
r (% of d^+)	2, 4, 6, 8, 16, 32, 64	8
k	1, 2, 4, 8, 16, 32	8
ε (% of d^+)	2, 4, 6, 8, 10	6

6 Performance Study

We experimentally evaluate the performance of our methods. First, we study the effect of parameters. Then, we compare the SPB-tree construction cost against those of several representative MAMs. Next, the efficiency of similarity search and similarity join algorithms using SPB-trees as well as the accuracy of the corresponding cost models are explored. We implemented the SPB-trees and associated similarity search, similarity join algorithms in C++. All experiments were conducted on an Intel Core 2 Duo 2.93GHz PC with 3GB RAM.

Table 2 depicts the statistics of the datasets. Please refer to Appendix J for detailed description. All MAMs to index the datasets use a fixed disk page size of 4KB. We investigate the efficiency of the SPB-tree and query algorithms under varying parameters, as listed in Table 3. Note that, in every experiment, only one parameter is varied, while the other parameters are fixed at their default values. The main performance metrics include the number of page accesses (*PA*), the number of distance computations (*compdists*), and CPU time (i.e., wall time). Each measurement we report is the average of 500 queries for the first 500 objects in every dataset.

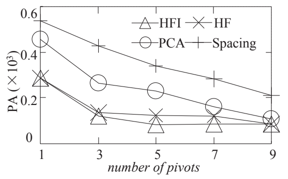
6.1 Effect of Parameters

The first set of experiments explores the effect of parameters. We employ *k*NN search to obtain insights into the effect of parameters on the efficiency of the SPB-tree, where k is set to 8 as default. The other queries offer similar insights.

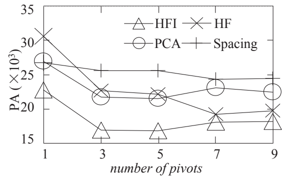
First, we verify the efficiency of the SPB-tree under different SFCs. The results are depicted in Table 4. As observed, the query cost (including the number of page accesses and the number of distance computations) of the Hilbert curve is lower

TABLE 4: SPB-tree efficiency under different SFCs

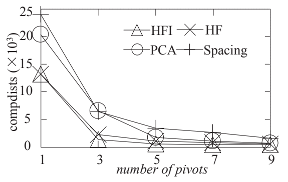
	Hilbert Curve			Z-Curve		
	Color	*Words*	*DNA*	*Color*	*Words*	*DNA*
PA	82.2	703.22	16,789	200.25	812.08	19,430
compdists	522.8	49,746	391,411	522.8	49,782	558,580
Time(sec)	0.02	0.23	6.16	0.02	0.22	8.96



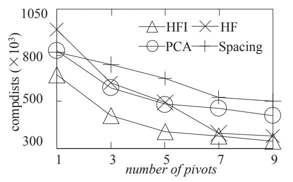
(a) *Color*



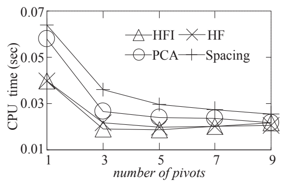
(b) *DNA*



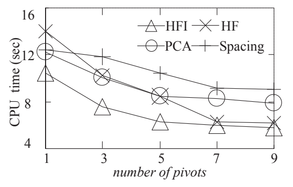
(c) *Color*



(d) *DNA*



(e) *Color*



(f) *DNA*

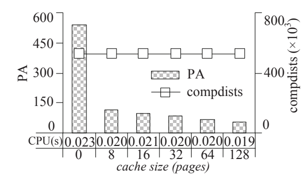
Fig. 9: Efficiency of pivot selection methods vs. $|P|$

than that of Z-curve. The reason is that the Hilbert curve has better clustering properties than the Z-curve. In most cases, the query time of Hilbert curve is less than that of Z-order curve due to fewer page accesses and distance computations. Nonetheless, the transformation cost between SFC values and vectors for the Hilbert curve is higher, making it possible that the Z-order curve has less CPU time, e.g., on *Words*. However, the Hilbert curve is better for all performance metrics in most cases. Thus, in the rest of experiments, the Hilbert curve is used except for similarity joins. This is because similarity join algorithm utilizes the property of the Z-order curve as discussed in Section 5.2.

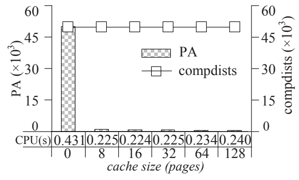
Then, we investigate the effectiveness of our pivot selection algorithm (i.e., HFI). Fig. 9 shows the results obtained using real datasets. The first observation is that HFI performs better than the existing pivot selection algorithms considered, i.e., HF [6], Spacing [36], and PCA [37]. The reason is that the search performance is highly related with the *precision* as defined in Definition 1, and HFI tries to maximize *precision*. The second observation is that the number of distance computations decreases as the number of pivots grows. Using more pivots, query efficiency improves as *precision* increases, incurring fewer distance computations. The number of page accesses and CPU time first drop and then stay stable or increase as the number of pivots is increased. This is because the cost for filtering unqualified objects grows as well with more pivots. The appropriate number of pivots needed to achieve high query efficiency in all performance metrics approaches the dataset's *intrinsic dimensionality* (Ins. for short

TABLE 5: *k*NN search with different traversal strategies

	Incremental Traversal			Greedy Traversal		
	Color	*Words*	*DNA*	*Color*	*Words*	*DNA*
PA	82.2	703.22	309,765	57.69	469.78	16,789
compdists	522.8	49,746	391,215	740.57	51,188	391,411
Time(sec)	0.02	0.23	9.13	0.02	0.26	6.16



(a) *Word*



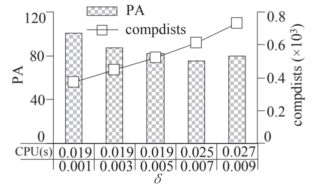
(b) *Color*

Fig. 10: Effect of *cache size (pages)*

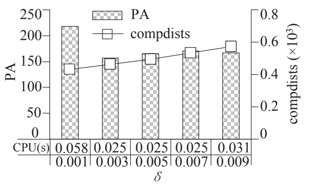
in Table 2), which keeps consistency with our observation. Since the intrinsic dimensionality for the datasets used in our experiments is among 3∼6, the number of pivots is set to 5 as default. Also, *precision* (prec. for short) of 5 pivots on each dataset is reported in Table 2.

Next, we study the influence of the cache size on the efficiency of *k*NN query algorithms, as illustrated in Fig. 10. The cache aims to improve the I/O efficiency of a single query (i.e., to reduce the number of duplicated RAF page accesses). Here, it is flushed before each of the 500 queries. As expected, the number of page accesses and CPU time decrease as the cache size is increased, and a small cache is enough to achieve query efficiency. However, as discussed in Section 4.3, if a *k*NN query needs to retrieve a large portion of the dataset, i.e., $\frac{Compdists}{datset\ cardinality}$ is large, a small cache is not enough for the incremental traversal strategy. Note that, $\frac{Compdists}{datset\ cardinality}$ depends on *precision*, and DNA has the lowest *precision* among three real datasets, as depicted in Table 2. Hence, on *DNA*, the algorithm has a high I/O cost (i.e., *PA*) using the incremental traversal, as shown in Table 5. However, the greedy traversal strategy is optimal in terms of *PA* (i.e., no duplicated RAF page accesses), and achieves high computational efficiency accordingly. Consequently, the greedy traversal is used as default on *DNA*, while the incremental traversal is used as default for other datasets in the remaining experiments.

Finally, in order to observe the impact of δ on the efficiency of the SPB-tree, we use *Color* and *Synthetic* datasets since the range of their distance functions is real numeric. Fig. 11 plots the experimental results under various δ values. As observed, the number of distance computations increases with δ. The reason is that for larger δ, the average collision probability $|O|/(\frac{d^+}{\delta})^{|P|}$ that different objects are approximated as the same vectors grows, resulting in more distance computations. Nevertheless, the query costs (including *PA* and CPU time)



(a) *Color*



(b) *Syntetic*

Fig. 11: Effect of δ

TABLE 6: The construction costs and storage sizes of MAMs

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

TABLE 7: The update cost of SPB-tree on *Words*

	M-tree	**OmniR-tree**	**M-Index**	**SPB-tree**
PA	9.87	4.05	3.38	9.16
Compdists	82.32	6	20	5
Time (sec)	0.045	0.02	0.013	0.0004

first drop and then stay stable in most cases. This is because, for smaller δ, the search space becomes sparse as the collision probability decreases, leading to higher query costs.

6.2 SPB-Tree Construction and Update

The second set of experiments considers the construction and update costs of the SPB-tree.

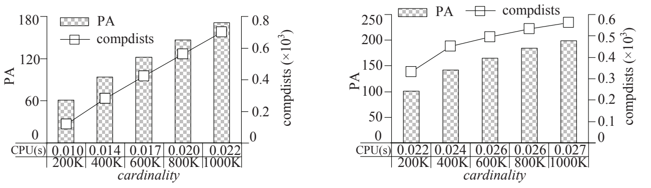
First, we compare the construction cost and storage size of the SPB-tree with three representative MAMs, namely, M-tree [2], OmniR-tree [6], and M-Index [26]. All these MAMs are built by using their corresponding bulk-loading methods. It is worth mentioning that, the OmniR-tree utilizes HF algorithm to select (*intrinsic dimensionality* + 1) pivots, while the M-Index randomly chooses 20 pivots. Table 6 lists the construction costs and storage sizes for all MAMs using real datasets. The OmniR-tree cannot run on *Words* and *DNA* because of the large cardinality of the datasets. Clearly, the SPB-tree has much lower construction cost, in terms of the number of page accesses , the number of distance computations, and the construction 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 of the SPB-tree is also much smaller than that of other MAMs, due to the SFC dimensionality reduction.

Further, we evaluate the SPB-tree update performance on real datasets. Table 7 depicts the average update cost of inserting 100 random objects. As expected, the update operation of the SPB-tree is more efficient than that of other competitors, because it relies on the manipulation of a B$^+$-tree. However, the I/O cost of the SPB-tree is relatively high as both B$^+$-tree and RAF page accesses are needed.

6.3 Results on Similarity Search

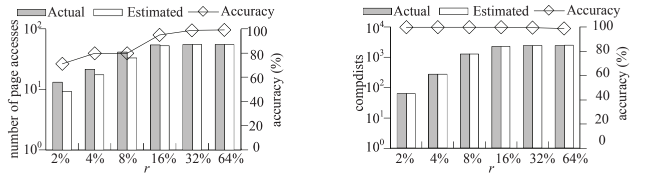
The third set of experiments concerns the performance of the similarity search algorithms and the accuracy of the corresponding cost models.

First, we investigate the efficiency of similarity search using SPB-trees, compared with those based on three other MAMs, viz., M-tree [2], OmniR-tree [6], and M-Index [26]. Figs. 12 and 13 show the performance of range and *k*NN queries, using *Signature* and real datasets. The OmniR-tree cannot run on *Words* and *DNA* because of the large cardinality of the datasets. It is observed that the SPB-tree performs the best in terms of the number of page accesses, including both B$^+$-tree node accesses and RAF page accesses. This is due to two



(a) *Range query* (b) *kNN query*

Fig. 14: Scalability of similarity search vs. cardinality



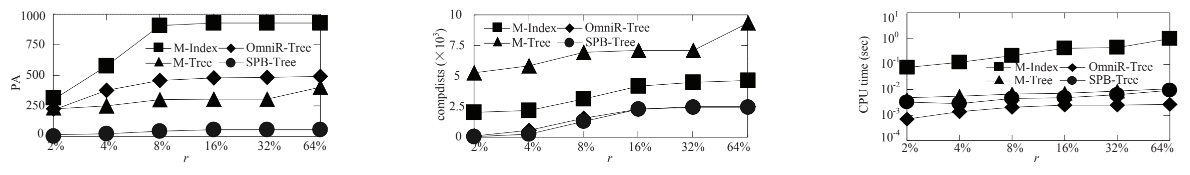
(a) *Signature* (b) *Signature*

Fig. 15: Range query cost model vs. *r*

reasons. First, the SPB-tree uses an SFC to cluster objects into compact regions, and thus achieves I/O efficiency since both B$^+$-tree entries and RAF objects to be visited are stored close to each other. Second, the SPB-tree preserves multi-dimensional pre-computed distances as one-dimensional SFC values, resulting in a smaller index storage size and fewer page accesses. In addition, the SPB-tree performs better or comparable to existing MAMs in terms of the number of distance computations. The reason is that our pivot selection algorithm selects effective pivots, thus avoiding significant number of distance computations, and our similarity search algorithms only compute qualified distances, as stated in Lemmas 1 to 4. Consequently, the SPB-tree has the lowest CPU time in most cases, due to having the fewest distance computations and page accesses. Nonetheless, on *Signature* and *Color* datasets, the CPU cost of SPB-tree exceeds that of the OmniR-tree and the M-tree. This is because, during similarity search using the SPB-tree, the transformation between SFC values and vectors results in additional computational costs, and hence, the SPB-tree may have larger CPU time on datasets using simple similarity metrics.

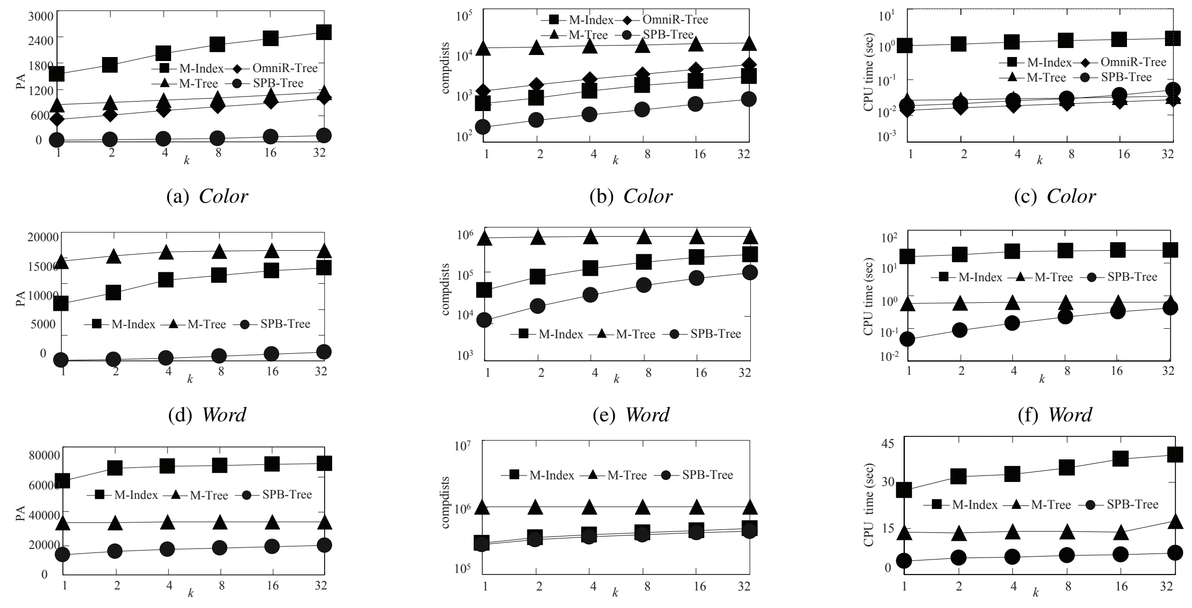
Next, we study the scalability of similarity search. Fig. 14 plots the performance of range and *k*NN queries as a function of cardinality, using *Synthetic* datasets. The query costs, including the number of page accesses, the number of distance computations, and CPU time, grow linearly with the cardinality, because the search space grows as cardinality increases.

Finally, we explore the accuracy of our cost models for similarity queries. Figs. 15 and 16 illustrate 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, each diagram contains (1) the actual



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

Fig. 12: Range query performance vs. *r*

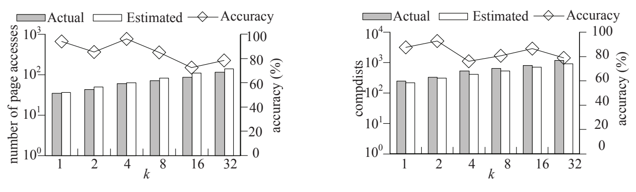


(a) *Color* (b) *Color* (c) *Color*

(d) *Word* (e) *Word* (f) *Word*

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

Fig. 13: *k*NN query performance vs. *k*



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

Fig. 16: *k*NN cost model vs. *k*

costs, *Actual*, (2) the estimated costs, *Estimated*, computed by our cost models, and (3) the accuracy of the estimated values, *Accuracy*, i.e., 1 − |*Actual* − *Estimated*|/ *Actual*. As can be seen, our cost models are able to estimate I/O and CPU costs accurately, with the average accuracy over 80%.

6.4 Results on Similarity Joins

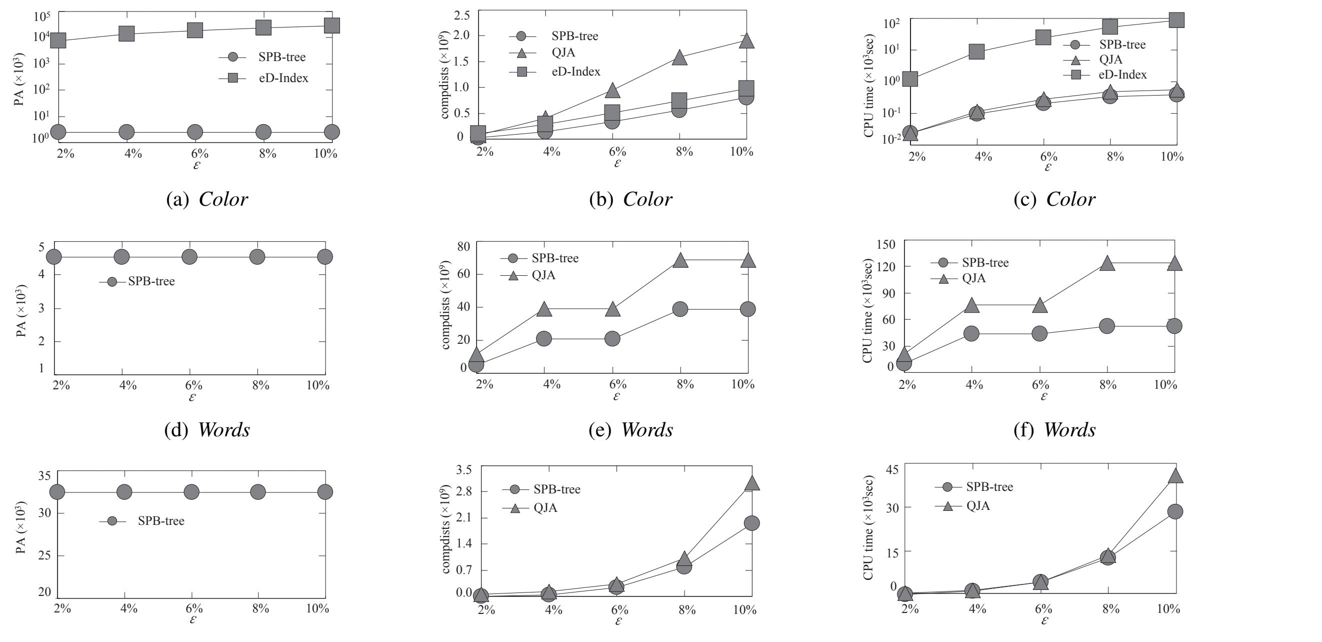
The last set of experiments verify the performance of the similarity join algorithm and the accuracy of the corresponding cost models.

First, we inspect the efficiency of our similarity join algorithm using the SPB-tree, compared with the state-of-the-art competitors, including eD-index based method [44] and improved Quickjoin Algorithm (QJA) [43]. Fig. 17 plots the performance of similarity join algorithms as a function of ε, using real datasets. The number of page accesses is not reported for QJA because it is an in-memory algorithm. The first observation is that the SPB-tree outperforms the competitors, and preforms several orders of magnitude better than eD-index based method due to two reasons below. First, our similarity join algorithm traverses the SPB-trees only once, while eD-index based method incurs lots of duplicated page accesses. Second, QJA is designed without any underlying index built in advance, and thus needs additional cost to partition the dataset. Hence, eD-index is omitted on the *Words* and *DNA* datasets. In addition, eD-index is only applicable for similarity joins with smaller ε values, and the index has to be rebuilt for larger ε values, which limits its applicability. To sum up, our new similarity join algorithm can support any ε value, and can also benefit from the SPB-trees built in advance. The second observation is that the query cost increases with increasing ε. This is because the search space grows with the growth of ε.

Then, we study the accuracy of our cost models for similarity joins. Fig. 18 depicts the I/O overhead (i.e., the number of page accesses) and CPU cost (i.e., the number of distance computations). It is observed that the cost models are quite accurately, with the average accuracy over 90%.

7 Conclusions

Spatial queries, including similarity queries and similarity joins, are useful in many areas of computer science, such as multimedia retrieval, data integration, and computational biology, to name but a few. We present a new metric index, the *Space-filling curve and Pivot-based B^+-tree* (SPB-tree), for query processing in generic metric spaces that supports

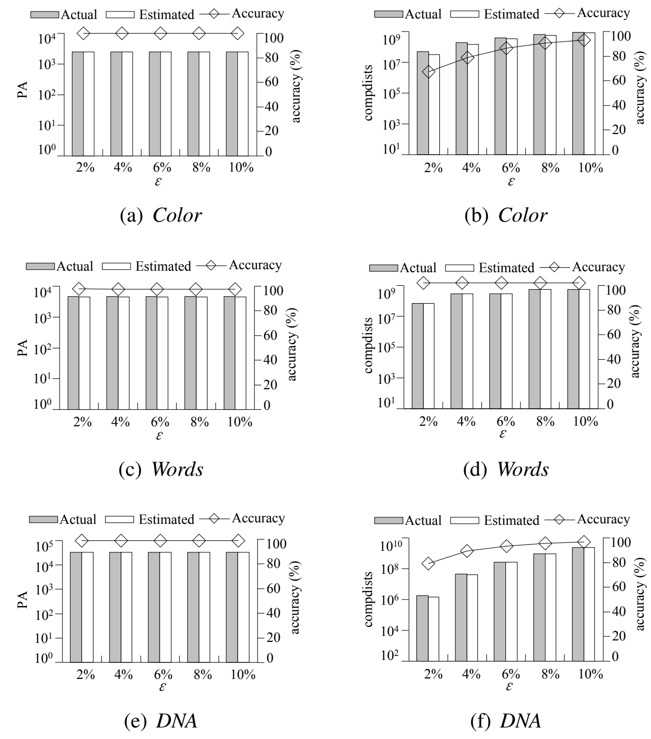


(a) *Color* (b) *Color* (c) *Color*

(d) *Words* (e) *Words* (f) *Words*

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

Fig. 17: Similarity join performance vs. ε



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

(c) *Words* (d) *Words*

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

Fig. 18: Similarity join cost model vs. ε

a wide range of data types and similarity metrics. The SPB-tree picks few but effective pivots to reduce significantly the number of distance computations; it uses a space-filling curve to cluster data objects into compact regions, thus improving storage efficiency; it utilizes a B$^+$-tree with MBB information as the underlying index, which enables easy integration into an existing DBMS; and it employs a separate random-access file to support large sets of complex data. In addition, we propose efficient algorithms for similarity search and similarity joins, and we derive their corresponding cost models based on SPB-trees. Extensive experiments with both real and synthetic data sets show that, compared with other MAMs, the SPB-tree has *lower* construction and storage costs, and supports *more efficient* similarity queries and similarity joins with *high accuracy* cost models. In future research, it is relevant to extend the SPB-tree to different distributed environments.

Acknowledgments. This work was supported in part by 973 Program 2015CB352502, NSFC Grants 61522208, 61379033, and 61472348, and the Fundamental Research Funds for the Central University under Grant 2015XZZX005-07. We also would like to thank Prof. D. Novak, Prof. R. Mao, and Prof. Y. N. Silva for useful feedback on the source-codes of their proposed algorithm in [37], [26], [44].

References

[1] E. Chavez and G. Navarro, “A compact space decomposition for effective metric indexing,” *Pattern Recognition Letters*, vol. 26, no. 9, pp. 1363–1376, 2005.

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

[3] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula, “D-Index: Distance searching index for metric data sets,” *Multimedia Tools Appl.*, vol. 21, no. 1, pp. 9–33, 2003.

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

[5] W. Burkhard and R. Keller, “Some approaches to best-match file searching,” *Commun. ACM*, vol. 16, no. 4, pp. 230–236, 1973.

[6] 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.*, vol. 16, no. 4, pp. 483–505, 2007.

[7] L. Mico, J. Oncina, and R. C. Carrasco, “A fast branch & bound nearest neighbour classifier in metric spaces,” *Pattern Recognition Letters*, vol. 17, no. 7, pp. 731–739, 1996.

[8] P. N. Yianilos, “Data structures and algorithms for nearest neighbor search in general metric spaces,” in *SODA*, pp. 311–321, 1993.

[9] E. Vidal, "An algorithm for finding nearest neighbors in (approximately) constant average time," *Pattern Recognition Letters*, vol. 4, no. 3, pp. 145–157, 1986.
[10] L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen, "Efficient metric indexing for similarity search," in *ICDE*, pp. 591–602, 2015.
[11] I. Kalantari and G. McDonald, "A data structure and an algorithm for the nearest point problem," *IEEE TSE*, vol. 9, no. 5, pp. 631–634, 1983.
[12] 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*, pp. 186–203, 1992.
[13] J. K. Uhlmann, "Satisfying general proximity/similarity queries with metric trees," *Inf. Process. Lett.*, vol. 40, no. 4, pp. 175–179, 1991.
[14] S. Brin, "Near neighbor search in large metric spaces," in *VLDB*, pp. 574–584, 1995.
[15] G. Navarro, "Searching in metric spaces by spatial approximation," *VLDB J.*, vol. 11, no. 1, pp. 28–46, 2002.
[16] L. Aronovich and I. Spiegler, "CM-tree: A dynamic clustered index for similarity search in metric databases," *Data Knowl. Eng.*, vol. 63, no. 3, pp. 919–946, 2007.
[17] V. Dohnal, C. Gennaro, and P. Zezula, "Similarity join in metric spaces using eD-Index," *Database and Expert Systems Applications*, vol. 2736, pp. 484–493, 2003.
[18] R. Paredes and N. Reyes, "Solving similarity joins and range queries in metric spaces with the list of twin clusters," *J. Discrete Algorithms*, vol. 7, no. 1, pp. 18–35, 2009.
[19] G. Navarro and N. Reyes, "Dynamic list of clusters in secondary memory," in *SISAP*, pp. 94–105, 2014.
[20] 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*, pp. 1365–1368, 2010.
[21] G. Ruiz, F. Santoyo, E. Chavez, K. Figueroa, and E. S. Tellez, "Extreme pivots for faster metric indexes," in *SISAP*, pp. 115–126, 2013.
[22] R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu, "Proximity matching using fixed-queries trees," in *CPM*, pp. 198–212, 1994.
[23] T. Bozkaya and M. Ozsoyoglu, "Distance-based indexing for high-dimensional metric spaces," in *SIGMOD*, pp. 357–368, 1997.
[24] T. Skopal, J. Pokorny, and V. Snasel, "PM-tree: Pivoting metric tree for similarity search in multimedia databases," in *ADBIS*, pp. 803–815, 2004.
[25] J. Lokoc, J. Mosko, P. Cech, and T. Skopal, "On indexing metric spaces using cut-regions," *Inf. Syst.*, vol. 43, pp. 1–19, 2014.
[26] D. Novak, M. Batko, and P. Zezula, "Metric Index: An efficient and scalable solution for precise and approximate similarity search," *Inf. Syst.*, vol. 36, no. 4, pp. 721–733, 2011.
[27] 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*, pp. 74–80, 2009.
[28] E. Chavez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquin, "Searching in metric spaces," vol. 33, pp. 273–321, 2001.
[29] J. Mosko, J. Lokoc, and T. Skopal, "Clustered pivot tables for I/O-optimized similarity search," in *SISAP*, pp. 17–24, 2011.
[30] S. Dasgupta, "Performance guarantees for hierarchical clustering," *J. Comput. Syst. Sci.*, vol. 70, no. 4, pp. 555–569, 2005.
[31] N. R. Brisaboa, A. Farina, O. Pedreira, and N. Reyes, "Similarity search using sparse pivots for efficient multimedia information retrieval," in *ISM*, pp. 881–888, 2006.
[32] B. Bustos, O. Pedreira, and N. R. Brisaboa, "A dynamic pivot selection technique for similarity search in metric spaces," in *SISAP*, pp. 105–112, 2008.
[33] B. Bustos, G. Navarro, and E. Chavez, "Pivot selection techniques for proximity searching in metric spaces," *Pattern Recognition Letters*, vol. 24, no. 14, pp. 2357–2366, 2003.
[34] C. Hennig and L. J. Latecki, "The choice of vantage objects for image retrieval," *Pattern Recognition*, vol. 36, no. 9, pp. 2187–2196, 2003.
[35] J. Venkateswaran, T. Kahveci, C. M. Jermaine, and D. Lachwani, "Reference-based indexing for metric spaces with costly distance measures," *VLDB J.*, vol. 17, no. 5, pp. 1231–1251, 2008.
[36] R. H. V. Leuken and R. C. Veltkamp, "Selecting vantage objects for similarity indexing," *ACM TOMCCAP*, vol. 7, no. 3, article 16, 2011.
[37] R. Mao, W. L. Mirankerb, and D. P. Mirankerc, "Pivot selection: Dimension reduction for distance-based indexing," *J. Discrete Algorithms*, vol. 13, pp. 32–46, 2012.
[38] G. R. Hjaltason and H. Samet, "Index-driven similarity search in metric spaces," *ACM TODS*, vol. 28, no. 4, pp. 517–580, 2003.
[39] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA: Morgan Kaufmann, 2006.
[40] P. Ciaccia and A. Nanni, "A query-sensitive cost model for similarity queries with M-tree," in *ADC*, pp. 65–76, 1999.
[41] P. Ciaccia, M. Patella, and P. Zezula, "A cost model for similarity queries in metric spaces," in *PODS*, pp. 59–68, 1998.
[42] E. H. Jacox and H. Samet, "Metric space similarity joins," *ACM TODS*, vol. 33, no. 2, article 7, 2008.
[43] K. Fredriksson and B. Braithwaite, "Quicker similarity joins in metric spaces," in *SISAP*, pp. 127–140, 2013.
[44] S. S. Pearson and Y. N. Silva, "Index-based R-S similarity joins," in *SISAP*, pp. 106–112, 2014.
[45] Y. N. Silva and S. Pearson, "Exploiting database similarity joins for metric spaces," *PVLDB*, vol. 5, no. 12, pp. 1922–1925, 2012.
[46] K. K. Lee, B. Zheng, H. Li, and W.-C. Lee, "Approaching the skyline in Z order," in *VLDB*, pp. 279–290, 2007.
[47] 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.
[48] https://drive.google.com/file/d/0B7FJy6T_Tv3FMUt0RUR5U2MtQk0/view?usp=sharing.
[49] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, pp. 5–16, 2006.

Lu Chen received the BS degree in computer science from Southeast University, China, in 2011. She is currently working toward the PhD degree in the College of Computer Science, Zhejiang University, China. Her research interests include indexing and querying metric spaces.

Yunjun Gao received the PhD degree in computer science from Zhejiang University, China, in 2008. He is currently an associate professor in the College of Computer Science, Zhejiang University, China. His research interests include spatial and spatio-temporal databases, metric and incomplete/uncertain data management, and spatio-textual data processing. He is a member of the ACM and the IEEE, and a senior member of the CCF.

Xinhan Li received the BS degree in computer science from Zhejiang University, China, in 2013. He is currently working toward the MS degree in the College of Computer Science, Zhejiang University, China. His research interests include indexing and querying metric spaces.

Christian S. Jensen is an Obel professor of computer science at Aalborg University, Denmark. He was a professor at Aarhus University during 2010 to 2013, and he was previously at Aalborg University for two decades. He recently spent a 1-year sabbatical at Google Inc., Mountain View, CA. His research concerns data management and data-intensive systems, and its focus is on temporal and spatiotemporal data management. He is an editor-in-chief of ACM TODS, and a fellow of the ACM and the IEEE.

Gang Chen received the PhD degree in computer science from Zhejiang University, China. He is currently a professor in the College of Computer Science, Zhejiang University, China. He has successfully led the investigation in research projects that aim at building Chinese indigenous database management systems. His research interests range from relational database systems to large-scale data management technologies. He is a member of the ACM and senior member of the CCF.