8-2017

Pivot-based Metric Indexing

Lu CHEN
Zhejiang University

Yunjun GAO
Zhejiang University

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Christian S. JENSEN
Aalborg University

Hanyu YANG
Zhejiang University

See next page for additional authors

Citation

CHEN, Lu; GAO, Yunjun; ZHENG, Baihua; JENSEN, Christian S.; YANG, Hanyu; and YANG, Keyu. Pivot-based Metric Indexing. (2017). *Proceedings of the VLDB Endowment: 43rd International conference, Munich Germany, 2017 August 28 -September 1*. 1058-1069.
Available at: https://ink.library.smu.edu.sg/sis_research/3739

Pivot-based Metric Indexing: Experiments and Analyses

Lu Chen[†,‡] Yunjun Gao[†,*] Baihua Zheng[‡] Christian S. Jensen[§] Hanyu Yang[†] Keyu Yang[†]

[†]College of Computer Science, Zhejiang University, Hangzhou, China
[*]The Key Lab of Big Data Intelligent Computing of Zhejiang Province, Zhejiang University, Hangzhou, China
[‡]School of Information Systems, Singapore Management University, Singapore
[§]Department of Computer Science, Aalborg University, Denmark

[†]{luchen, gaoyj, hyy_zju, kyyang}@zju.edu.cn [‡]bhzheng@smu.edu.sg [§]csj@cs.aau.dk

ABSTRACT
The general notion of a metric space encompasses a diverse range of data types and accompanying similarity measures. Hence, metric search plays an important role in a wide range of settings, including multimedia retrieval, data mining, and data integration. With the aim of accelerating metric search, a collection of pivot-based indexing techniques for metric data has been proposed, which reduces the number of potentially expensive similarity comparisons by exploiting the triangle inequality for pruning and validation. However, no comprehensive empirical study of those techniques exists. Existing studies each offers only a narrower coverage, and they use different pivot selection strategies that affect performance substantially and thus render cross-study comparisons difficult or impossible. We offer a survey of existing pivot-based indexing techniques, and report a comprehensive empirical comparison of their construction costs, update efficiency, storage sizes, and similarity search performance. As part of the study, we provide modifications for two existing indexing techniques to make them more competitive. The findings and insights obtained from the study reveal different strengths and weaknesses of different indexing techniques, and offer guidance on selecting an appropriate indexing technique for a given setting.

1. INTRODUCTION
Search is a fundamental functionality in computer science, with similarity search being a prominent type of queries. Given a query object, similarity search finds similar objects according to a definition of similarity. This kind of functionality is useful in many settings. 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 based on users' preferences.

Considering the wealth of data types (e.g., images and strings), a generic model is desirable that is capable of accommodating a wide spectrum of data types rather than some specific data types. In addition, the distance metric used for comparing the similarity of objects goes beyond the Euclidean distance (i.e., the L_2-norm) and includes metrics such as the L_p-norm and earth mover's distance for images and the edit distance for strings. Hence, we consider metric spaces to accommodate a wide of data types and similarity notations.

Proceedings of the VLDB Endowment, Vol. 10, No. 10
Copyright 2017 VLDB Endowment 2150-8097/17/06.

A number of indexing techniques exist that aim to accelerate search in metric spaces. As an example, environment for developing KDD-applications supported by index-structures, termed as ELKI, is an open source data mining software that uses indexing (e.g., M-tree [13]) to improve efficiency [25]. Existing indexes can be classified into two categories, i.e., *compact partitioning techniques* [1], [3], [7], [10], [13], [14], [18], [21], [22], [27] and *pivot-based techniques* [5], [8], [11], [12], [17], [19], [20], [23], [24], [26]. The former divides data space into compact regions and tries to eliminate entire regions during search. The latter employs search that relies on pre-computed distances between each object in the database and each object in 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. Thus, it may be possible to prune an object o as a result object for q using the lower bound value $|d(q, p) - d(o, p)|$ rather than computing $d(q, o)$, which may be costly. This enables pivot-based techniques to outperform compact partitioning techniques in terms of the number of distance computations [2], one of the key performance criteria in metric spaces. For this reason and to be comprehensive within the chosen scope, we focus on the pivot-based techniques.

We aim to address limitations of existing empirical studies. First, the use of different pivot selection strategies renders the comparison of pivot-based indexing techniques challenging. For example, the OmniR-tree [17] utilizes the hull of foci algorithm (HF) to select outliers as pivots, while the spacing filing curve and pivot based B^+-tree (SPB-tree) [12] uses the HF based incremental pivot selection algorithm (HFI) to select pivots that maximize the similarity between the original metric space and the vector space achieved by using the pivots. Since the performance of similarity query processing depends highly on the pivots used [9], we compare pivot-based indexes using the same pivot selection strategy. Second, while studies [11], [30] survey metric indexing techniques pre-2006, the last dozen years have seen many proposals for new and better metric indexes (e.g., disk-based indexes), such as the OmniR-tree, the M-index [23], and the SPB-tree. We offer a comprehensive empirical study as of today.

In brief, the key contributions of this paper are as follows:

- We provide a compact survey of existing pivot-based indexing techniques, focusing on the underlying principles.
- We enhance two existing pivot-based metric indexes to give them better search performance. Specifically, we provide a better pivot selection strategy for the extreme pivot table, and integrate minimum bounding box information into the M-index.
- We present a comprehensive empirical comparison of existing pivot-based indexing techniques, considering index construction cost, update efficiency, index size, and query performance while ensuring an equal footing where the same pivot selection

strategy is employed. The findings and insights obtained from the empirical study offer new insights on the strengths and weaknesses of exiting techniques and aid in selecting an appropriate indexing technique for a given setting.

The rest of this paper is organized as follows. Section 2 presents the preliminaries of pivot-based indexes. Sections 3, 4, and 5 describe three categories of pivot-based metric index structures. Experimental results and our findings are reported in Section 6. Finally, Section 7 concludes the paper.

2. PRELIMINARIES

We proceed to define the core metric similarity queries. Then, we provide a brief overview of the pivot-based indexes, and describe the pivot-based filtering enabled by these indexing techniques.

2.1 Metric Similarity Search

A metric space is a two-tuple (M, d), in which M is an object domain and d is a distance function for measuring the "similarity" between 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 these properties, we define metric similarity search, including the metric range query and the metric k nearest neighbor query below.

DEFINITION 1 (METRIC RANGE QUERY). *Given an object set O, a query object q, and a search radius r in a metric space, a* metric range query (MRQ) *returns the objects in O that are within distance r of q*, i.e., $MRQ(q, r) = \{o \mid o \in O \wedge d(q, o) \leq r\}$.

DEFINITION 2 (METRIC *K* NEAREST NEIGHBOR QUERY). *Given an object set O, a query object q, and an integer k in a metric space, a* metric k nearest neighbor query (MkNNQ) *finds k objects in O that are most similar to q*, i.e., $MkNNQ(q, k) = \{S \mid S \subseteq O \wedge |S| = k \wedge \forall s \in S, \forall o \in O - S, d(q, s) \leq d(q, o)\}$.

Consider the English word set O = {"defoliates", "defoliation", "defoliating", "defoliated", "citrate"}, where edit distance is used to measure similarity between words. An example of metric range query finds the words from O with edit distances to the query word "defoliate" no larger than 1, i.e., *MRQ*("defoliate", 1) = {"defoliates", "defoliated"}. An example of metric k nearest neighbor query finds 2 words from O with the smallest edit distances to the query word "defoliate", i.e., *MkNNQ*("defoliate", 2) = {"defoliates", "defoliated"}.

An MkNNQ can be answered by an MRQ, if the distance from q to its kth nearest neighbor, denoted as ND_k, is known. However, ND_k is not known when a query is issued. Two typical methods exist for computing MkNNQ [6], [15]. One utilizes MRQ with incremental search radius. Specifically, an MRQ with a small search radius is performed first, and then the search radius is increased gradually until k nearest neighbors are found. Although this method tries to avoid visiting objects already verified, it still needs to traverse the metric index multiple times, resulting in high query cost. The other sets the search radius to infinity and then verifies the objects in order, where the search radius is tighten using verification.

2.2 Pivot-based Metric Index Structures

Pivot-based methods store pre-computed distances from every object in the database to a set of pivots and then use those distances to prune objects during search. Pivot-based methods can be clustered into three categories, namely, *pivot-based tables*, *pivot-based trees*, and *pivot-based external indexes*, according to the structures they use for storing the pre-computed distances, as listed in Table 1.

Indexes in the first category utilize tables to store pre-computed distances. Approximating Eliminating Search Algorithm (AESA) [28] uses a table to preserve the distances from each object to other objects. However, it incurs a high storage cost O(n^2), where n is the number of objects in the dataset. To save main memory storage for the table, Linear AESA (LAESA) [19] only keeps the distances from every object to selected pivots; Extreme Pivot Table (EPT) [24] selects a set of essential pivots covering the entire database; and Clustered Pivot Table (CPT) [20] clusters the pre-computed distances to further improve query efficiency.

Table 1. Pivot-based metric index structures

Category	Index	Storage	Distance Domain
Pivot-based tables	AESA[28], LAESA[19]	Main-memory	Continuous
	EPT[24]	Main-memory	Continuous
	CPT[20]	Main-memory	Continuous
Pivot-based trees	BKT[8]	Main-memory	Discrete
	FQT[4], FQA[11]	Main-memory	Discrete
	VPT[29], MVPT[5]	Main-memory	Continuous
Pivot-based external indexes	PM-tree[26]	Disk	Continuous
	Omni-family[17]	Disk	Continuous
	M-index[23]	Disk	Continuous
	SPB-tree[12]	Disk	Continuous

Indexes in the second category use tree structures to store pre-computed distances. Burkhard-Keller Tree (BKT) [8] is designed for discrete distance functions. It chooses a pivot p as the root, and inserts the objects having distance i to the pivot p in its ith sub-tree. Different from BKT that uses different pivots at individual levels, Fixed Queries Tree (FQT) [4] and Fixed Queries Array (FQA) [11] use the same pivot for all the nodes at the same tree level. Vantage-Point Tree (VPT) [29] is designed for continuous distance functions, and its generalization to m-ary trees is called MVPT [5].

Indexes in the third category utilize an existing disk-based index (e.g., the R-tree or the B$^+$-tree) to store pre-computed distances. The Omni-family [17] employs existing structures (e.g., the R-tree) to index pre-computed distances. The PM-tree [26] stores cut-regions defined by pivots in each node of an M-tree to accelerate search. The M-index [23] generalizes the iDistance [16] technique for general metric spaces, and uses the B$^+$-tree to store pre-computed distances. The SPB-tree [12] utilizes a space-filling curve to map pre-computed distances to integers, which are then indexed by the B$^+$-tree.

The index structures that belong to the first and the second categories refer to indexes stored in main memory, while index structures in the third category are disk-based.

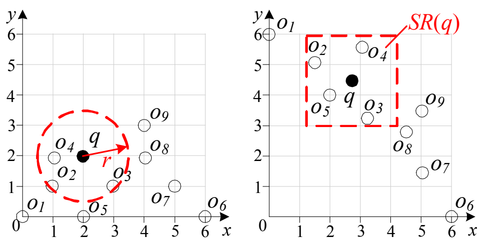
2.3 Pivot-based Filtering

Using well-chosen pivots, the objects in a metric space can be mapped to data points in a vector space. Given a pivot set $P = \{p_1, p_2, \ldots, p_l\}$, a metric space ($M$, d) can be mapped to a vector space (R^l, L_∞). Specifically, an object q in the metric space is represented as a point $\phi(q) = \langle d(q, p_1), d(q, p_2), \ldots, d(q, p_l)\rangle$ in the vector space. Consider the example in Fig. 1, where the *L_2-norm* is used as the distance function. If $P = \{o_1, o_6\}$, the object set in the original metric space (as illustrated in Fig. 1(a)) can be mapped to the data points in a two-dimensional vector space (as depicted in Fig. 1(b)), in which the x-axis denotes $d(o_i, o_1)$ and the y-axis represents $d(o_i, o_6)$ for any object o_i. As an example, object o_5 is mapped to point $\langle 2, 4\rangle$.

Based on the pivot mapping, the pivot-based filtering [12] can be used to avoid unnecessary similarity computations.

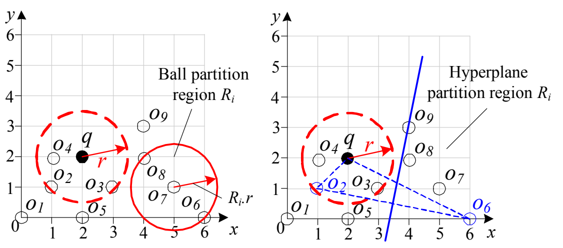
LEMMA 1 (PIVOT FILTERING). *Given a set P of pivots, a query object q, and a search radius r, let $SR(q)$ be a search region such that* $SR(q) = \{\langle v_1, v_2, \ldots, v_l\rangle \mid 1 \leq i \leq l \wedge v_i \geq 0 \wedge v_i \subseteq [d(q, p_i) - r, d(q, p_i) + r]\}$. *If $\phi(o)$ locates outside $SR(q)$, then $o \notin MRQ(q, r)$.*

PROOF. Assume, to the contrary, that there exists an object o ($\in MRQ(q, r)$) which satisfies $d(q, o) \leq r$, but $\phi(o) \notin SR(q)$ (i.e., $\exists\, p_i \in$



(a) Oringal metric space (b) Mapped vector space

Figure 1. Pivot mapping



Ball partition region R_i
$R_i.r$
Hyperplane partition region R_i

(a) Range-pivot filtering (b) Double-pivot filtering

Figure 2. Pivot filtering

Object table
Pivot table ***Distance table***

P	object	$d(o_i, o_1)$	$d(o_i, o_6)$
o_1	o_1	0	6
o_6	o_2	$\sqrt{2}$	$\sqrt{26}$
	o_3	$\sqrt{10}$	$\sqrt{10}$
	o_4	$\sqrt{5}$	$\sqrt{29}$
	o_5	2	4
	o_6	6	0
	o_7	$\sqrt{26}$	$\sqrt{2}$
	o_8	$2\sqrt{5}$	$2\sqrt{2}$
	o_9	5	$\sqrt{13}$

Figure 3. LEASA

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, $d(q, o) \geq |d(q, p_i) - d(o, p_i)| > r$, which contradicts our assumption. The proof completes. □

Since the pre-computed distances $\phi(o)$s are stored together with object o in a metric index, we can avoid distance computations involving object o if $\phi(o) \notin SR(q)$, based on Lemma 1. Consider the example in Fig. 1(b) where the dotted rectangle represents the search region $SR(q)$. Here, object o_1 can be pruned as $\phi(o_1) \notin SR(q)$. Also, Lemma 1 can be utilized to prune an entire region (i.e., a minimum bounding box that contains multiple $\phi(o)$) if it does not intersect $SR(q)$.

To obtain compact regions, two typical metric partitioning techniques are used [30], i.e., *ball partitioning* and *generalized hyperplane partitioning*. We proceed to introduce the partitioning and corresponding pivot filtering techniques that enable pruning of whole regions.

Definition 3 (Ball Partitioning). *Let $R_i.p$ be the corresponding pivot for a partition region R_i, and let $R_i.r$ be the radius of R_i. Then the set of objects o ($\in O$) in the partition R_i, obtained via ball partitioning, is defined as $\{o \mid o \in O \wedge d(o, p_i) \leq R_i.r\}$.*

Based on the definition of ball partitioning, a range-pivot filtering technique [30] can be developed as follows.

Lemma 2 (Range-Pivot Filtering). *Given a ball partitioning region R_i, a query object q, and a search radius r, if $d(q, R_i.p) > R_i.r + r$, then R_i can be pruned safely.*

Proof. For any object o in R_i, if $d(q, R_i.p) > R_i.r + r$, then $d(q, o) \geq d(q, R_i.p) - d(o, R_i.p) > R_i.r + r - d(o, R_i.p)$ due to the triangle inequality. As $d(o, R_i.p) \leq R_i.r$ according to Definition 3, then $d(q, o_j) > r$. Hence, any object o in R_i cannot be in the final result set, and R_i can be pruned safely, which completes the proof. □

Consider the ball partitioning example depicted in Fig. 2(a), where the red solid circle denotes the ball region R_i with $R_i.p = o_7$, $R_i.r = d(o_7, o_6)$, and $R_i = \{o_6, o_7, o_8\}$. As $d(q, R_i.p) > R_i.r + r$, R_i can be pruned away according to Lemma 2.

Definition 4 (Generalized Hyperplane Partitioning). *Given a set P of pivots, let p_i be the corresponding pivot for a partition region R_i. Then the set of objects o ($\in O$) in the partition R_i, obtained by the generalized hyperplane partitioning, is defined as $\{o \mid o \in O \wedge \forall p_j \neq p_i, d(o, p_i) \leq d(o, p_j)\}$.*

Based on the definition of generalized hyperplane partitioning, a double-pivot filtering technique [30] is developed as follows.

Lemma 3 (Double-Pivot Filtering). *Given two pivots p_i and p_j, a query object q, and a search radius r, if $d(q, p_i) - d(q, p_j) > 2 \times r$, then R_i can be pruned safely, as p_i is the corresponding pivot for the partition region R_i.*

Proof. For every o in R_i, according to the definition of R_i, $d(o, p_i) \leq d(o, p_j)$. Based on the triangle inequality, we have $d(q, p_i) \leq d(o, p_i) + d(q, o)$ and $d(q, p_j) \geq d(o, p_j) - d(q, o)$. Thus, we can derive that $d(q, p_i) - d(q, p_j) \leq d(o, p_i) + d(q, o) - d(o, p_j) + d(o, q) \leq 2 \times d(q, o)$ as $d(o, p_i) \leq d(o, p_j)$. If $d(q, p_i) - d(q, p_j) > 2 \times r$, then $d(q, o) > r$. Therefore, no object o ($\in R_i$) can be a real answer object (i.e., $o \notin MRQ(q, r)$), and R_i can be pruned safely. The proof completes. □

Consider the generalized hyperplane partitioning example shown in Fig. 2(b). Assume o_2 and o_6 are two pivots, and $R_i = \{o_6, o_7, o_8, o_9\}$ is the hyperplane partition region corresponding to pivot o_6. Since $d(q, o_6) - d(q, o_2) > 2 \times r$, R_i can be discarded safely according to Lemma 3.

Lemmas 1 through 3 are pivot filtering techniques. Nonetheless, a distance computation is still needed for verifying each object that cannot be pruned. Hence, a validation technique [12] is proposed to save unnecessary verifications, as stated in Lemma 4 below.

Lemma 4 (Pivot Validation). *Given a pivot set P, a query object q, and a search radius r, if there exists, for an object o in O, a pivot p_i ($\in P$) satisfying $d(o, p_i) \leq r - d(q, p_i)$, then o is validated to be an actual answer object for $MRQ(q, r)$.*

Proof. Given a query object q, an object o, and a pivot p_i, $d(q, o) \leq d(o, p_i) + d(q, p_i)$ because of the triangle inequality. If $d(o, p_i) \leq r - d(q, p_i)$, then $d(q, o) \leq r - d(q, p_i) + d(q, p_i) = r$. Thus, o is guaranteed to be contained in the final result set, which completes the proof. □

3. PIVOT-BASED TABLES

We proceed to describe the indexes that belong to the category of pivot-based tables, and present corresponding MRQ and M*k*NNQ processing, together with some discussions.

3.1 AESA and LAESA

AESA uses a table to store the distances from every object to other objects. If $|O|$ is the cardinality of a dataset O, the main-memory storage cost of AESA is $O(|O|^2)$, which is high for large datasets. That renders AESA a theoretical metric index. In order to reduce the storage cost of AESA, LAESA is proposed. It only stores the distances from each object to the pivots in a pivot set P, and thus, its storage cost is reduced to $O(|P| \times |O|)$, in which $|P|$ is the number of pivots in P. LAESA utilizes three tables to store the pivots, the real data, and the pre-computed distances to the pivots. Fig. 3 shows the LAESA on the object set O depicted in Fig. 1, where $P = \{o_1, o_6\}$.

MRQ processing. MRQ(q, r) processing using LAESA is simple. We compute the distances $d(q, p_i)$ between the query object q and the pivots p_i ($\in P$) and then verify the objects in the dataset O one by one. For every object o in O, if it cannot be pruned by Lemma 1, we compute $d(q, o)$ and insert o into the result set S_r if $d(q, o) \leq r$.

M*k*NNQ processing. M*k*NNQ(q, k) processing based on LAESA follows the second approach introduced in Section 2.1. It initializes the search radius to infinity, and computes the distances from the query object q to the pivots in P. Subsequently, objects in the dataset O are evaluated one by one. For each object o, if it cannot be pruned by Lemma 1, we compute $d(q, o)$ and update the search radius using the current k^{th} nearest neighbor distance.

Discussion. Although LAESA significantly reduces the main-memory storage cost of AESA, it still incurs high storage cost for a

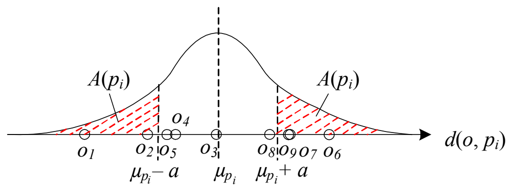


Figure 4. Illustration of $A(p_i)$

Pivot table	*Object table*	*Distance table*	
P	**object**	**(p_1, $d(o_i, p_1)$)**	**(p_2, $d(o_i, p_2)$)**
o_1	o_1	(o_1, 0)	(o_9, 5)
o_4	o_2	(o_1, $\sqrt{2}$)	(o_4, 1)
o_6	o_3	(o_1, $\sqrt{10}$)	(o_9, $\sqrt{5}$)
o_9	o_4	(o_6, $\sqrt{29}$)	(o_4, 0)
	o_5	(o_1, 2)	(o_9, $\sqrt{13}$)
	o_6	(o_6, 0)	(o_4, $\sqrt{29}$)
	o_7	(o_6, $\sqrt{2}$)	(o_4, $\sqrt{17}$)
	o_8	(o_1, 2 $\sqrt{5}$)	(o_9, 1)
	o_9	(o_1, 5)	(o_9, 0)

Figure 5. EPT

large dataset, which limits its applicability. For similarity search processing, although it utilizes Lemma 1 to avoid certain unnecessary distance computations, it still needs to scan the full dataset to find the result set. In addition, since the objects in the dataset are verified according to the order they are stored, M*k*NNQ processing using LAESA results in unnecessary distance computations.

3.2 EPT

Unlike LAESA that utilizes the same pivots for each object, EPT selects different pivots for different objects in order to achieve better search performance.

Extreme pivots (EP) consist of a set of pivot groups. Each group G contains m pivots p_i ($1 \leq i \leq m$), according to which the whole dataset O is partitioned into m parts $A(p_i)$, such that $A(p_i) \cap A(p_j) = \varnothing$ ($i \neq j$) and $\cup_{p_i \in G} A(p_i) = O$. An object o belongs to $A(p_i)$ iff $|d(o, p_i) - \mu_{pi}| \geq \alpha$, where μ_{pi} is the expected value of $d(o, p_i)$. Consider the example in Fig. 4, $A(p_i) = \{o_1, o_2, o_6, o_7, o_9\}$.

However, it is hard to obtain α, and hence, EPT tries to maximize α. In other words, EPT randomly selects m pivots as a pivot group G_j, and sets the pivot p_i in G_j to an object o having $max\{|d(o, p_i) - \mu_{pi}| \mid p_i \in G_j\}$. The processing is repeated l times, i.e., l groups G_j ($1 \leq j \leq l$) are selected. Thus, each object has corresponding l pivots.

Given the dataset shown in Fig. 1, and letting $m = 2$ and $l = 2$, two pivot groups are selected at random, i.e., $G_1 = \{o_1, o_6\}$ and $G_2 = \{o_4, o_9\}$. Fig. 5 depicts an example of EPT. The structure used by EPT is similar to that used by LAESA. However, since each object in EPT may have different pivots, EPT needs to store the corresponding pivot (i.e., the *id* of the pivot) with the pre-computed distance.

Let $X = d(p_i, o)$ and $Y = d(p_i, q)$, then the query cost in terms of the number of distance computations can be estimated as:

$$\begin{aligned} cost &= m \times l + |O| \times (1 - \Pr(|X - Y| > r))^l \\ &\geq m \times l + |O| \times \left(1 - \frac{\sigma_X^2 + \sigma_Y^2}{r^2}\right)^l \end{aligned} \quad (1)$$

Using Equation (1), we can approximate the optimal m by fixing l (to control the main-memory storage size), where σ_X^2, σ_Y^2, and r can be estimated. Nevertheless, EPT utilizes $Z = d(o, q)$ to estimate $Y = d(p_i, q)$, which is inaccurate. In addition, it is difficult to estimate r value which is specified by the user.

We therefore proceed to introduce a new pivot selection algorithm (PSA) to improve the efficiency of EPT. Let $D(q, o) = \max\{|d(q, p_i) - d(o, p_i)| \mid p_i \in P\}$, which is a lower bound of $d(q, o)$ according to the triangle inequality. Hence, the query cost can be estimated as:

$$cost = m \times l + |O| \times \Pr(D(q, u) \leq r) \quad (2)$$

To achieve the optimal query cost defined in Equation (2), $D(q, u)$ should approach $d(q, o)$ as much as possible in order to avoid unnecessary distance computations of $d(q, o)$. Thus, PSA tries to maximize the random variable $D(q, o)/d(q, o)$.

Algorithm 1 Pivot Selecting Algorithm (PSA)

```
Input: a set $O$ of objects, the number $l$ of pivots for each object
Output: EPT$^*$
1: obtain a sample set $S$ from $O$
2: $CP$ = HF($O$, $cp\_scale$)  // get a candidate pivot set $CP$ ($|CP| = cp\_scale$)
3: for each object $o$ in $O$ do
4:     $P = \varnothing$
5:     while $|P| < l$ do
6:         select a different $p_i$ from $CP$ with the maximal $\sum_{o' \in S} D(o', o)/d(o', o)$
7:         $P = P \cup \{p_i\}$
8:     update EPT$^*$ with $\langle(p_1, d(o, p_1)), (p_2, d(o, p_2)), \ldots, (p_l, d(o, p_l))\rangle$
9: return EPT$^*$
```

Algorithm 1 presents the pseudo-code of PSA. First, it samples the object set O as set S, and invokes HF algorithm [17] to obtain outliers as candidate pivots CP (lines 1-2). Here, *cp_scale* is set to 40 because this value yields enough outliers in our experiments. Then, for each object o in O, the algorithm incrementally selects effective pivots from CP (lines 4-7), and updates EPT* (line 8). Finally, EPT* is returned (line 9).

MRQ and M*k*NNQ processing. Like LAESA, EPT and EPT* use tables to store pre-computed distances. The only difference is that EPT and EPT* utilize different pivots for different objects, while LAESA uses the same pivots for every object. Hence, MRQ and M*k*NNQ processing on EPT or EPT* are the same as those on LAESA.

Discussion. EPT* achieves a better similarity search performance than EPT, contributed by the higher quality pivots selected by PSA. Nonetheless, it is costly to maximize $\sum_{o' \in S} D(o', o)/d(o', o)$.

3.3 CPT

LAESA and EPT store the distance table and the data file in main memory, and similarity query processing needs to scan the whole table. However, when the size of the dataset exceeds the capacity of the main memory, it is necessary to store the dataset on disk, and it becomes attractive to cluster the data to improve I/O efficiency.

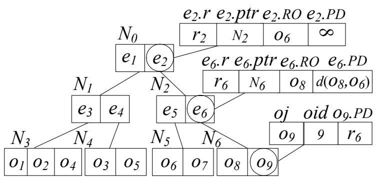
CPT uses an M-tree to cluster and store the objects on disk. Fig. 6(b) shows an M-tree for the object set $O = \{o_1, o_2, \ldots, o_9\}$ in Fig. 1. An *intermediate* (i.e., a *non-leaf*) entry e in a root node (e.g., N_0) or a non-leaf node (e.g., N_1, N_2) records the following: (i) A *routing object e.RO* that is a selected object in the subtree ST_e of e. (ii) A *covering radius e.r* that is the maximum distance between *e.RO* and the objects in ST_e. (iii) A *parent distance e.PD* that equals the distance from e to the routing object of its parent entry. Since a root entry e (e.g., e_6) has *no* parent entry, *e.PD* = ∞. (iv) An *identifier e.ptr* that points to the root node of ST_e. A leaf entry (i.e., a data object) o in a leaf node (e.g., N_3, N_4, N_5, N_6) records the following: (i) An *object oj* that stores the detailed information of o. (ii) An *identifier oid* that represents o's identifier. (iii) A *parent distance o.PD* that equals the distance from o to the routing object of o's parent entry.

An example of CPT is shown in Fig. 6. CPT consists of three parts, i.e., a pivot table, a distance table, and an M-tree. The distance table stores the pre-computed distances between objects and pivots in main memory. The M-tree stores the objects in the tree structure on disk (i.e., each M-tree entry contains one object). Note that, the distance table includes pointers to the leaf entries in the M-tree, in order to enable loading of the corresponding objects for verification.

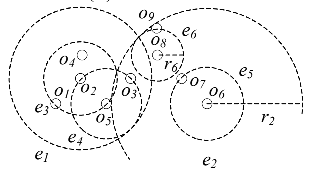
Pivot table **Distance table**

P	ptr	$d(o_i, o_1)$	$d(o_i, o_6)$
o_1	o_1	0	6
o_6	o_2	$\sqrt{2}$	$\sqrt{26}$
	o_4	$\sqrt{5}$	$\sqrt{29}$
	o_3	$\sqrt{10}$	$\sqrt{10}$
	o_5	2	4
	o_6	6	0
	o_7	$\sqrt{26}$	$\sqrt{2}$
	o_8	$2\sqrt{5}$	$2\sqrt{2}$
	o_9	5	$\sqrt{13}$

(a) Pivot and distance tables


$e_2.r$ $e_2.ptr$ $e_2.RO$ $e_2.PD$
r_2 N_2 o_6 ∞
N_0
e_1 e_2
$e_6.r$ $e_6.ptr$ $e_6.RO$ $e_6.PD$
r_6 N_6 o_8 $d(o_8,o_6)$
N_1 N_2
e_3 e_4 e_5 e_6
oj oid $o_9.PD$
o_9 9 r_6
N_3 N_4 N_5 N_6
o_1 o_2 o_4 o_3 o_5 o_6 o_7 o_8 o_9

(b) The M-tree


o_9 e_6 o_8 r_6 o_4 o_7 e_5 o_1 o_2 o_3 o_6 e_3 o_5 r_2 e_4 e_1 e_2

(c) Data distribution

Figure 6. CPT

MRQ and M*k*NNQ processing. MRQ and M*k*NNQ processing using CPT are similar as the processing using LAESA. The only difference is that, when an object cannot be pruned by Lemma 1, the object must be read from disk.

Discussion. Using CPT, we can avoid loading the whole dataset into main memory to perform query processing. However, CPT incurs CPU cost to load objects from disk. In addition, the distance table is stored in main memory, meaning that the applicability of CPT is only limited to the dataset whose distance table fits in main memory.

4. PIVOT-BASED TREES

We describe the indexes belonging to the category of pivot-based trees along with the corresponding MRQ and M*k*NNQ processing.

4.1 BKT

BKT is a tree structure designed for discrete distance functions. It chooses a pivot as the root, and maintains the objects having the distance i to the pivot in its ith sub-tree. If a sub-tree contains more than one object, it selects a pivot at random and partitions the sub-tree recursively. Fig. 7 gives an example BKT, constructed based on the objects from Fig. 1(a) and the discrete distance function L_∞-norm. The leaf nodes store the actual objects, while the non-leaf nodes store the corresponding pivots used to partition the sub-trees. To improve the efficiency of the pivot-based trees, we only store the identifiers in the tree structures, and store the objects in a separate table.

MRQ processing. In order to answer MRQ(q, r), the nodes in the BKT are traversed in depth-first fashion. When a non-leaf node is accessed, we identify its qualifying child entries using Lemma 1; and when a leaf node is accessed, we insert the corresponding object into the result set if it is not pruned by Lemma 1.

M*k*NNQ processing. In order to answer M*k*NNQ(q, k), the nodes in the BKT are traversed in best-first manner, i.e., in ascending order of their minimum distances to the query object q, where Lemma 1 is used to filter out unqualified nodes. Here, we first set the search radius to infinity and then update it using the visited objects.

Discussion. BKT is an unbalanced tree. To avoid empty sub-trees for large distance domains, every sub-tree covers the same range of distance values, which are stored together with each sub-tree. BKT randomly selects the pivots for sub-trees. If BKT uses the same pivots as other pivot-based metric indexes, it produces FQT as discussed below.

4.2 FQT

Unlike BKT, FQT utilizes the same pivot at the same level. Fig. 8 shows an example of FQT, where o_1 and o_6 are selected as the pivots for the first level and the second level, respectively.

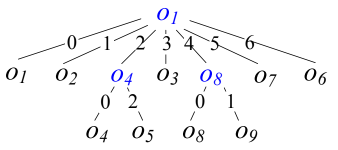


Figure 7. BKT

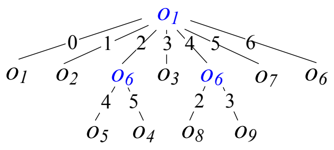
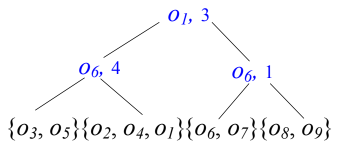
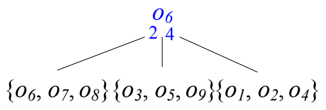


Figure 8. FQT



(a) VPT



(b) MVPT

Figure 9. VPT and MVPT

MRQ and M*k*NNQ processing. MRQ and M*k*NNQ processing using FQT are the same as that for BKT.

Discussion. FQT is also an unbalanced tree. In order to utilize the same set P of pivots as other pivot-based metric indexes, the tree-level is set to the number of pivots, and $p_i \in P$ is set as the pivot for the ith level. With well-chosen pivots, the performance of FQT is expected to be better than that of BKT.

4.3 MVPT

Unlike BKT and FQT that only support discrete distance functions, VPT and its variant MVPT are able to support continuous distance functions. VPT chooses a pivot p as the root, and selects a medium value v so that the objects o with $d(o, p) \leq v$ are put in the left sub-tree, while the remaining objects are put in the right sub-tree. If the number of objects in a sub-tree exceeds a threshold, the sub-tree is further partitioned. Fig. 9(a) depicts an example of VPT, where L_∞-norm is used. Note that, the pivots for the nodes at the same level can be different. In order to be able to compare the efficiency of different indexes using the same set of pivots, nodes of VPT at the same level share the same pivot.

VPT can be generalized to m-ary trees, yielding MVPT. Specifically, each time, MVPT selects $m - 1$ medium values $v_1, v_2, \ldots, v_{m-1}$ instead of one, such that the objects o with $d(o, p) \leq v_1$ are put in the first sub-tree, the objects o with $v_1 < d(o, p) \leq v_2$ are put in the second sub-tree, etc. Fig. 9(b) gives an example of MVPT, where L_∞-norm is used and m is set to 3.

MRQ and M*k*NNQ processing. MRQ and M*k*NNQ processing using VPT are similar to the processing using BKT.

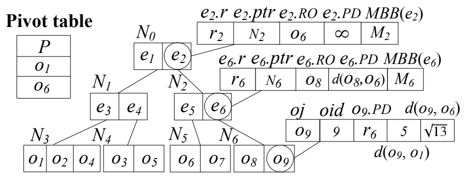
Discussion. Unlike BKT and FQT, MVPT is a balanced tree. As m grows, the pruning ability first increases and then drops. This occurs because, with larger m values, more compact sub-trees are obtained at every tree level. Nevertheless, larger m values also result in lower MVPT tree levels, indicating that fewer pivots are available for pruning. In this paper, we set m as 5 to achieve high query performance. In addition, it only needs to store medium values to partition the sub-trees, which incurs lower storage cost than BKT and FQT.

5. PIVOT-BASED EXTERNAL INDEXES

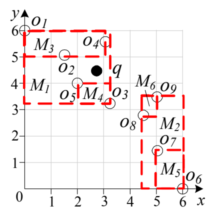
We proceed to detail the indexes belonging to the category of pivot-based external indexes, present corresponding MRQ and M*k*NNQ processing, and give some discussions.

5.1 PM-Tree

The PM-tree combines the pivot mapping and the M-tree, where the M-tree is used to cluster the objects, and the pivot mapping is utilized to avoid unnecessary distance computations. Hence, different from the M-tree introduced in Section 3.3, each leaf entry of the PM-



(a) PM-tree structure



(b) MBB

Figure 10. PM-tree

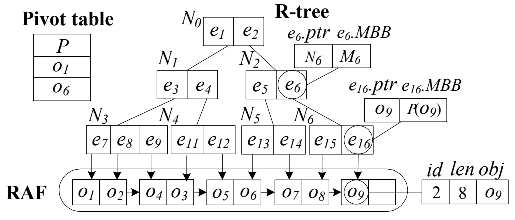
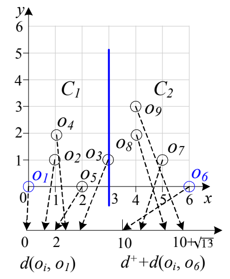
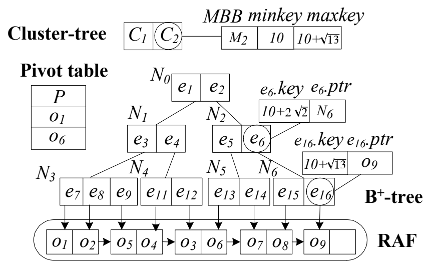


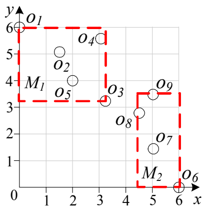
Figure 11. OmniR-tree



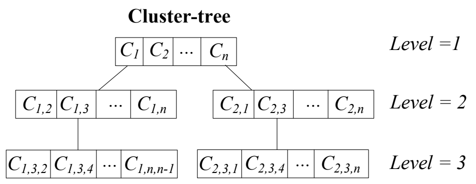
(a) Hyperplane partitioning



(b) M-index structure



(c) MBB



(d) Dynamic cluster-tree

Figure 12. M-index

tree stores the mapped vector (i.e., the pre-computed distances to the pivots) with the real data object. In each intermediate entry, the PM-tree stores a minimum bounding box (MBB) that bounds all the mapped vectors in its child leaf entries. Specifically, given a pivot set $P = \{p_i \mid 1 \leq i \leq n\}$, $MBB(e) = \{[a_i, b_i] \mid 1 \leq i \leq n\}$, where $a_i = min\{d(o, p_i) \mid o \in e\}$, and $b_i = max\{d(o, p_i) \mid o \in e\}$. Fig. 10 depicts an example of PM-tree, with the data distribution shown in Fig. 6(c).

MRQ processing. In order to answer MRQ(q, r), the entries in the PM-tree are traversed in depth-first fashion. When an intermediate entry is accessed, we verify its qualifying child entries using Lemmas 1 and 2; and when a leaf entry is accessed, we insert the corresponding object into the result set if it is not discarded by Lemma 1.

M*k*NNQ processing. In order to answer MkNNQ(q, k), the entries in the PM-tree are traversed in best-first manner, i.e., in ascending order of their minimum distances to the query object q, where Lemmas 1 and 2 are employed to eliminate unqualified entries. We first set the search radius to infinity, and then, we update the search radius during the search using the visited objects.

Discussion. The PM-tree stores the data objects in its entries instead of in a separate file, which limits its usability. In particular, for complex objects (e.g., the 282 dimensional vectors used in our experiments), the PM-tree needs a large page/node size.

5.2 Omni-Family

Unlike the PM-tree, Omni-family uses a separate random access file (RAF) to store the objects, in order to avoid the impact of the object size. The Omni-family utilizes an existing external index, e.g., the sequential file, the B$^+$-tree, or the R-tree, to index the vectors after the pivot mapping. A sequential file stores the pre-computed distances of objects in order of their identifiers; a B$^+$-tree is used to index the pre-computed distances for each pivot; and an R-tree is used to index the pre-computed distances for all the pivots together. An existing study [17] shows that the OmniR-tree performs the best in most cases. Fig. 11 depicts an example of OmniR-tree, including a pivot table that stores the pivots, an R-tree that indexes the pre-computed distances, and an RAF that stores the objects. The MBB of each R-tree node is shown in Fig. 10(b).

MRQ processing. To answer MRQ(q, r), the entries in the R-tree are traversed in depth-first fashion. When an intermediate entry is visited, we verify its qualifying child entries using Lemma 1; and when a leaf entry is accessed, we compute the actual distance and insert the corresponding object into the result set if it is in the answer.

M*k*NNQ processing. To answer MkNNQ(q, k), the entries in the R-tree are traversed in best-first manner, i.e., in ascending order of their minimum distances to the query object q, where Lemma 1 is used to eliminate unqualified entries. Here, we set the search radius to infinity and then update it using the visited objects.

Discussion. The Omni-family contains the Omni-sequential-file, the OmniB$^+$-tree, and the OmniR-tree. Omni-sequential-file can be regarded as LAESA stored on disk, which incurs substantial I/O during search as the data is not clustered. The OmniB$^+$-tree needs one B$^+$-tree for every pivot, resulting in redundant storage and I/O during search. The OmniR-tree utilizes MBBs to cluster the data, and uses the pivot filtering to achieve high query efficiency.

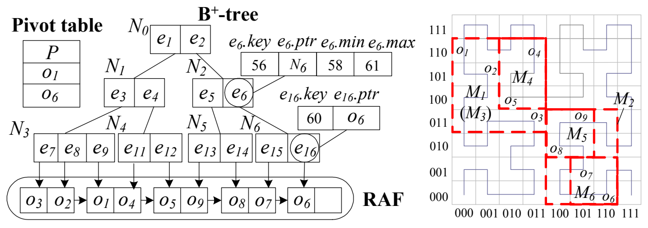
5.3 M-Index

Unlike the PM-tree that utilizes the ball partitioning technique, the M-index uses hyperplane partitioning (as discussed in Section 2.3) to cluster the data. Given a set P of pivots, each object o is mapped to the real number $key(o) = d(p_i, o) + (i - 1) \times d^+$, where p_i ($\in P$) is the pivot nearest to o and d^+ is the maximum distance in a certain metric space. Considering the example in Fig. 12, if $P = \{o_1, o_6\}$, we obtain two clusters C_1 and C_2. M-index consists of (i) a pivot table, (ii) a cluster tree that maintains the information of the clusters (i.e., the minimum and maximum mapped digits *minkey* and *maxkey* in each cluster), (iii) a B$^+$-tree that indexes the mapped real numbers, and (iv) an RAF that stores the data objects with all the pre-computed distances. If more pivots are used, the cluster-tree can be extended to a dynamic tree. Specifically, if the number of the objects in a certain cluster exceeds a threshold *maxnum* (set to 1,600 in this paper), it can be further partitioned using the left pivots, as shown in Fig. 12(d).

MRQ processing. To answer MRQ(q, r), the entries in the cluster tree are traversed in depth-first fashion. When an intermediate entry is visited, we evaluate its qualifying child entries using Lemma 3; and when a leaf entry is accessed, we obtain the objects that belong to this cluster from B$^+$-tree, and filter out the unqualified objects according to Lemma 1.

M*k*NNQ processing. To answer MkNNQ(q, k), a range query with a small search radius is performed first, and then, the search radius is increased gradually until k closest objects are found.

We add the MBB information for each cluster to the M-index, obtaining an M-index*. Based on the MBBs, the pivot filtering tech-



(a) SPB-tree structure (b) Hilbert mapping

Figure 13. SPB-tree

nique (i.e., Lemma 1) can be applied when traversing the cluster-tree in order to filter unqualified clusters in advance, and M*k*NNQ can traverse the cluster-tree in best-first manner. Specifically, clusters are visited in ascending order of their distances to the query object *q*. In addition, the data validation technique (i.e., Lemma 4) can also be integrated to avoid unnecessary verifications.

Discussion. By integrating the data validation and the MBB information in the cluster tree, the efficiency of MRQ and M*k*NNQ is improved. Since the M-index* can use Lemma 3 based on the hyperplane partitioning technique for pruning while others cannot, it can achieve a better performance in terms of distance computations.

5.4 SPB-Tree

To reduce the storage cost, the SPB-tree utilizes a space-filling curve (SFC) to map the pre-computed distances into SFC values (i.e., integers) while (to some extent) maintaining spatial proximity. SPB-tree consists of (i) a pivot table, (ii) a B$^+$-tree storing SFC values, and (iii) an RAF that stores data objects. Each non-leaf B$^+$-tree entry *e* stores SFC values *min* and *max* for $\langle a_1, a_2, \ldots, a_n\rangle$ and $\langle b_1, b_2, \ldots, b_n\rangle$ that represent $MBB(e) = \{[a_i, b_i] \mid 1 \le i \le n\}$. Fig. 13 depicts an example of SPB-tree, where Fig. 13(b) illustrates the Hilbert mapping.

MRQ processing. To answer MRQ(q, r), the entries in the B$^+$-tree are traversed in depth-first fashion. When an intermediate entry is visited, we identify its qualifying child entries using Lemma 1; and when a leaf entry is accessed, we utilize Lemma 4 or compute the actual distance to validate the object.

M*k*NNQ processing. To answer M*k*NNQ(q, k), the entries in the B$^+$-tree are traversed in best-first manner, i.e., in ascending order of their minimum distances to the query object *q*, where Lemma 1 is used to filter unqualified entries. Here, we set the search radius to infinity and then update it using the visited objects.

Discussion. We employ the SFC mapping to reduce the storage cost and meanwhile maintain spatial proximity, resulting in improved I/O and index storage costs. However, for continuous distance functions, the continuous distances are approximated as the discrete ones to perform the SFC mapping, which decreases the pruning power.

6. EXPERIMENTAL STUDY

We proceed to report an empirical study on the performance of the pivot-based metric indexes via experiments. To be more specific, we consider the index construction cost, study the efficiency of EPT* and the M-index*, and evaluate the performance of all the pivot-based metric indexes when varying pertinent parameters.

6.1 Experimental Setup

We implemented all the indexes and associated similarity search algorithms in C++. Further, all pivot-based metric indexes utilize the same set of pivots selected by the state-of-the-art algorithm [12]. This does, however, not apply to EPT, EPT*, and BKT. As discussed in Sections 3.1 and 4.1, EPT and EPT* utilize different pivots for different objects, while BKT needs to randomly select pivots in its sub-trees. All experiments were conducted on an Intel Xeon E5-2620 v3 2.4GHz PC with 8GB memory.

Table 2. Datasets used in the experiments

Dataset	Cardinality	Dim.	Int. Dim.	MaxD	Dis. Measure
LA	1,073,727	2	5.4	14000	L_2-norm
Words	611,756	1~34	1.2	34	*Edit distance*
Color	1,000,000	282	6.5	100000	L_1-norm
Synthetic	1,000,000	20	6.6	10000	L_∞-norm

Table 3. Parameter settings

Parameter	Value	Default
the number |*P*| of pivots	1, 3, 5, 7, 9	5
r	4%, 8%, 16%, 32%, 64%	16%
k	5, 10, 20, 50, 100	20

We employ three real datasets, namely, *LA*, *Words*, and *Color*. *LA*[1] consists of geographical locations in Los Angeles. *Words*[2] contains proper nouns, acronyms, and compound words taken from the Moby project. *Color*[3] consists of standard MPEG-7 image features extracted from *Flickr*. A synthetic dataset is also created, where five dimension values are generated randomly, and the remaining dimension values are linear combinations of the previous ones. Each dimension of *LA* and *Synthetic* is mapped to [0, 10000], while each dimension of *Color* is mapped to [-255, 255]. To study the performance of BKT and FQT that are designed for discrete distance functions, the values in *Synthetic* are generated as integers. Table 2 summarizes the statistics of the datasets, including the cardinality, the dimensionality (Dim.), the intrinsic dimensionality (Int. Dim.), the maximum distance (MaxD) between data objects, and the distance measure (Dis. Measure). To capture the distance distribution of the dataset, the Int. Dim. is calculated as $\mu^2/2\sigma^2$, where μ and σ^2 are the mean and variance of the pairwise distances in the dataset.

We investigate the similarity query performance of the indexes when varying the parameters listed in Table 3. The value of the radius *r* denotes the percentage of objects in the dataset that are result objects of a metric range query. In each experiment, one parameter is varied, and the others are fixed at their default values. The main performance metrics contain the number of page accesses (*PA*), the number of distance computations (*compdists*), and the CPU time. Each measurement we report is an average over 100 random queries.

To maintain consistency with the operating system, the indexes use a fixed page size of 4KB as default. However, the data size for high-dimensional datasets is relatively large. CPT and PM-tree store directly the data in the index structures, and hence, a larger page size is needed to ensure a proper tree height; while other indexes separate the data from the index structures, meaning that the tree height is independent of the data size. Thus, a larger page size 40KB is used for CPT and PM-tree on *Color* and *Synthetic* datasets. As stated in Section 5, M*k*NNQ using M-index traverses the index multiple times for every query, while M*k*NNQ using SPB-tree, M-index*, or OmniR-tree has duplicate RAF page accesses since it does not visit the data stored in a separate RAF in sequence. Therefore, a 128KB LRU cache is used in our experiments to improve M*k*NNQ efficiency.

6.2 Construction Cost

Table 4 details the construction costs and storage sizes for the indexes using real datasets, where **I** denotes a main-memory storage cost, and **D** indicates a disk storage cost. There are no values for BKT and FQT on *LA* and *Color*, as BKT and FQT assume discrete

[1] *LA* is available at http://www.dbs.informatik.uni-muenchen.de/~seidl.
[2] *Words* is available at http://icon.shef.ac.uk/Moby/.
[3] *Color* is available at http://cophir.isti.cnr.it/.

Table 4. Construction costs and storage sizes

	LA				*Words*				*Color*			
	PA	*Compdists*	*Time* (s)	*Storage* (KB)	*PA*	*Compdists*	*Time* (s)	*Storage* (KB)	*PA*	*Compdists*	*Time*(s)	*Storage* (KB)
LAESA	—	5,368,635	2	50,331 (I)	—	3,058,780	3	44,209 (I)	—	5,000,000	80	1,140,625 (I)
EPT	—	539,394,597	87	71,302 (I)	—	124,892,823	82	56,157 (I)	—	240,990,000	426	1,160,156 (I)
EPT*	—	5,806,935,663	6,375	71,302 (I)	—	1,895,345,425	2,545	56,157 (I)	—	5,040,204,959	11,742	1,160,156 (I)
CPT	12,057,791	92,141,339	263	54,525 (I) 73,836 (D)	1,708,696	65,742,690	113	31,066 (I) 96,880 (D)	12,651,989	75,938,556	390	50,782 (I) 2,035,599 (D)
BKT	—	—	—	—	—	3,036,382	1.6	22,896 (I)	—	—	—	—
FQT	—	—	—	—	—	3,058,780	1.3	22,770 (I)	—	—	—	—
MVPT	—	5,368,635	2.7	21,054 (I)	—	3,058,780	1.8	22,729 (I)	—	5,000,000	117	1,105,552 (D)
PM-tree	4,342,461	32,828,371	167	240,424 (D)	4,643,434	60,087,931	230	213,552 (D)	4,803,502	93,930,402	609	2,605,440 (D)
OmniR-tree	171,648	5,368,635	291	90,956 (D)	1,441,537	3,058,780	68	57,104 (D)	3,690,582	5,000,000	495	1,400,752 (D)
M-index$^{(*)}$	93,904	5,368,635	15	76,775 (D)	53,866	3,058,780	10	45,140 (D)	416,897	5,000,000	101	1,389,174 (D)
SPB-tree	33,867	5,368,635	8	33,844 (D)	15,397	3,058,780	7	18,228 (D)	360,952	5,000,000	95	1,349,168 (D)

Table 5. Ranking according to construction and storage costs

	1st	**2nd**	**3rd**	**4th**	**5th**
PA	SPB-tree	M-index$^{(*)}$	OmniR-tree	PM-tree	CPT
Compdists	{LAESA, BKT, FQT, MVPT, OmniR-tree, M-index$^{(*)}$, SPB-tree}	PM-tree	CPT-tree	EPT	EPT*
Time	{LAESA, BKT, FQT, MVPT }	{SPB-tree, M-index$^{(*)}$}	EPT	{CPT, PM-tree, OmniR-tree}	EPT*
Storage	{BKT, FQT, MVPT, SPB-tree}	LAESA	EPT$^{(*)}$	{M-index$^{(*)}$, OmniR-tree}	{CPT-tree, PM-tree}

distance functions; and there are also no *PA* values for LAESA, EPT, EPT*, BKT, FQT, and MVPT, since they are in-memory indexes. To summarize our findings, Table 5 provides the ranking for all pivot-based metric indexes. Here, the M-index and the M-index* are listed together as M-index$^{(*)}$ because they have similar construction costs.

I/O cost. The SPB-tree performs the best in terms of I/O cost, followed by the M-index$^{(*)}$, while the PM-tree and CPT perform the worst. The SPB-tree and the M-index achieve high I/O efficiency via the B^+-tree, while the SPB-tree uses SFC to further reduce I/O cost.

Compdists. As observed, LAESA, BKT, FQT, MVPT, the OmniR-tree, the M-index, and the SPB-tree achieve the same performance in terms of *compdists*. This is because *compdists* for these indexes depend on the number of pivots and the number of data objects. However, the PM-tree and CPT incur additional distance computations for constructing the M-tree to store actual data, while EPT and EPT* need more *compdists* to select pivots for every data.

CPU time. The first observation is that LAESA, BKT, FQT, and MVPT perform the best in terms of CPU time, since they operate in main memory. Although EPT and EPT* are also in-memory indexes, EPT ranks 3rd and EPT* performs the worst in terms of CPU time. This is because they need to select different pivots for different objects while selecting pivots for EPT* is costly as analyzed in Section 3.2. Second, the SPB-tree and the M-index$^{(*)}$ can achieve high CPU efficiency (i.e., ranking 2nd) because of the B^+-tree used, while the OmniR-tree, CPT, and the PM-tree need more CPU time (i.e., ranking 4th) as they use an R-tree or an M-tree instead of a B^+-tree.

Storage. The storage cost for a pivot-based metric index includes two parts, i.e., the storage for the pre-computed distances and the storage needed for the objects themselves. First, the SPB-tree performs the best, since it utilizes an SFC to reduce the storage cost of the pre-computed distances. However, on the *Color* dataset, the storage cost of the SPB-tree is relatively larger. The reason is that, each object in *Color* needs 1,136 bytes, and that the size of the pages used to store the real objects is 4KB, thus incurring a waste of storage in every page. Second, the storage costs of the pivot-based trees are relatively smaller, because they only store the distance values used to partition the sub-tree instead of all the pre-computed distances. Third, the storage costs of LAESA and EPT$^{(*)}$ are smaller than those of the OmniR-tree and the M-index, since the latter two require additional storage to index the pre-computed distances. In addition, the storage cost of EPT$^{(*)}$ exceeds that of LAESA, as EPT$^{(*)}$ selects different pivots for every object and hence needs additional storage to indicate the corresponding pivot for each object. Finally, the storage costs of CPT and the PM-tree are the largest. This is because they store the real objects directly in the tree structures, while the other indexes use separate files to store the real objects.

Table 6. Update Costs

	LA			*Words*			*Color*		
	PA	*Comp.*	*Time*(s)	*PA*	*Comp.*	*Time*(s)	*PA*	*Comp.*	*Time*(s)
LAESA	—	5	0.15	—	5	0.14	—	5	2.29
EPT	—	1.5E7	2.5	—	3.3E6	2.1777	—	7.5E6	11.7
EPT*	—	5448	0.3999	—	3138	0.2612	—	5080	2.94
CPT	15.2	78.3	0.6559	1052	93	0.1772	16.4	80.9	0.5698
BKT	—	—	—	—	9.56	0.0001	—	—	—
FQT	—	—	—	—	10	0.0004	—	—	—
MVPT	—	10	0.0001	—	10	0.0001	—	10	0.0001
PM-tree	32	48	0.0023	2004	4167	0.035	119	227	0.1033
OmniR-tree	16.1	10	0.0042	52.9	10	0.0029	15.9	10	0.0045
M-index$^{(*)}$	13	10	0.0014	20.5	10	0.0016	11.8	10	0.0027
SPB-tree	12.9	10	0.0003	12.6	10	0.0014	12.5	10	0.0009

In conclusion, the pivot-based trees (including BKT, FQT, MVPT), LAESA, and the SPB-tree achieve the highest construction and storage efficiency, followed by the M-index, EPT, and the OmniR-tree, while EPT*, CPT, and the PM-tree perform the worst.

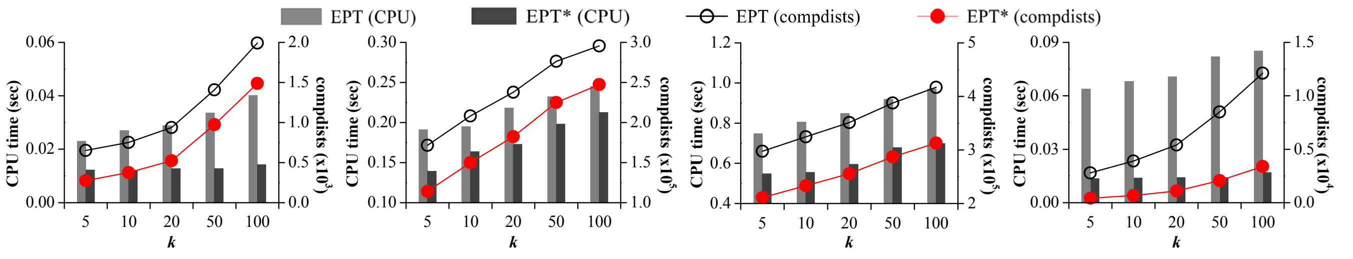
Discussion. Index construction can be accelerated using parallelization in several ways: (i) as the pivots are independent of each other, the pre-computed distances to each pivot can be computed in parallel; (ii) since objects are independent of each other, the pre-computed distances for each object can be computed in parallel; and (iii) as the data can be partitioned into disjoint parts, multiple index structures (e.g., multiple B^+-trees, M-trees, R-trees, BKTs, FQTs, MVPTs) instead of one can be constructed in parallel.

6.3 Update Cost

Table 6 details the update costs when using real datasets, while Table 7 provides the ranking. Here, an update operation first deletes a specific data object and then inserts it back. First, we observe that BKT, FQT, and MVPT can achieve high update efficiency. This is

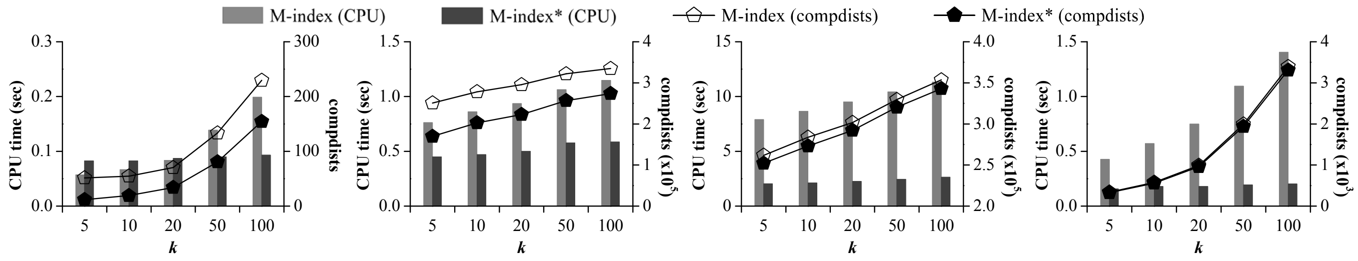
Table 7. Ranking according to update costs

	1st	2nd	3rd	4th	5th
PA	SPB-tree	M-index(*)	OmniR-tree	CPT	PM-tree
Compdists	LAESA	{BKT, FQT, MVPT, OmniR-tree, M-index(*), SPB-tree}	PM-tree, CPT-tree	EPT*	EPT
Time	{BKT, FQT, MVPT }	{ SPB-tree, M-index(*), OmniR-tree}	PM-tree	{LAESA, EPT*, CPT}	EPT

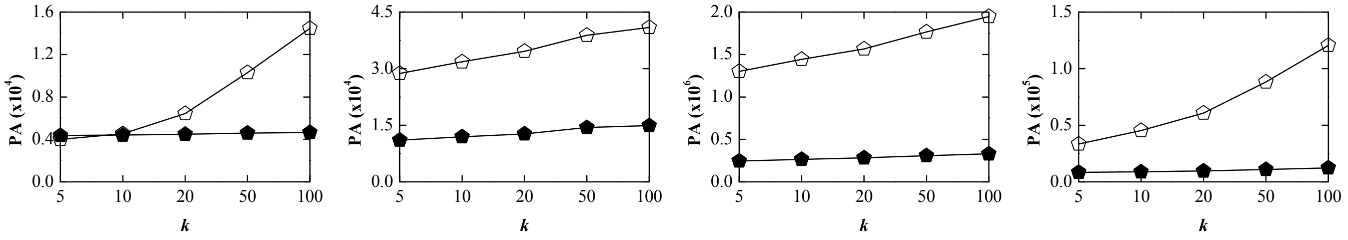


(a) *LA* (b) *Words* (c) *Color* (d) *Synthetic*

Figure 14. Comparison between EPT and EPT*



(a) *LA* (b) *Words* (c) *Color* (d) *Synthetic*



(e) *LA* (f) *Words* (g) *Color* (h) *Synthetic*

Figure 15. Comparison between M-index and M-index*

because they are stored in main memory and the positions for inserting/deleting can be found quickly using the tree structures. Second, the update costs of the PM-tree and CPT are relatively larger since they store the data objects directly in the trees. Third, the CPU time of LEASA, EPT*, and CPT is relatively high (i.e., ranking 4th) as they employ sequential scans to perform deletions. Fourth, the update costs of EPT and EPT* are high, because they need additional cost when selecting pivots for each data to be inserted. Note that, EPT* has better update efficiency than EPT, because EPT incurs high estimation costs when selecting pivots.

6.4 Efficiency of EPT* and M-Index*

We proceed to consider the efficiency of EPT*, as compared against EPT. In doing so, we only employ M*k*NNQs to observe the effect of parameters on the indexes, due to the space limitation and because MRQs yield similar findings. Fig. 14 depicts the results, where *PA* is omitted, since EPT and EPT* are in-memory indexes. As observed, EPT* performs better than EPT. This is because the quality of the pivots selected by EPT* is higher. Nonetheless, as shown in Table 4, the construction cost of EPT* is much higher in order to select pivots with higher quality. Since EPT* can be built in advance and has better update efficiency, it represents a useful improvement over EPT. Also, the construction efficiency of EPT* can be further improved, which is left as a direction of future work.

Next, we consider the efficiency of M-index*, as compared with M-index. Fig. 15 plots the results. As observed, the M-index* performs better than the M-index, while the *compdists* of the M-index* and the M-index are similar on *Color* and *Synthetic*. The reason is that M*k*NNQs using M-index visit the index multiple times, resulting in redundant *PA* and CPU cost, while M*k*NNQs based on M-index* traverse the index only once by using MBB information. However, the number of unnecessary distance computations depends on the increased radius value and the distance distribution of the dataset, which makes it possible that the *compdists* of M-index* and M-index are similar on *Color* and *Synthetic*. The second observation is that, on *LA*, the CPU time and *PA* of the M-index* are slightly larger than those of the M-index for smaller *k* values. This is because, for smaller *k* values on *LA*, the M-index based M*k*NNQ processing algorithm needs fewer MRQs to find the results, incurring little redundant cost.

6.5 Similarity Search Performance

We compare the efficiency of the pivot-based metric indexes under various parameters, including (i) the search radius for MRQ, (ii) the desired number *k* for M*k*NNQ, and (iii) the number |*P*| of pivots.

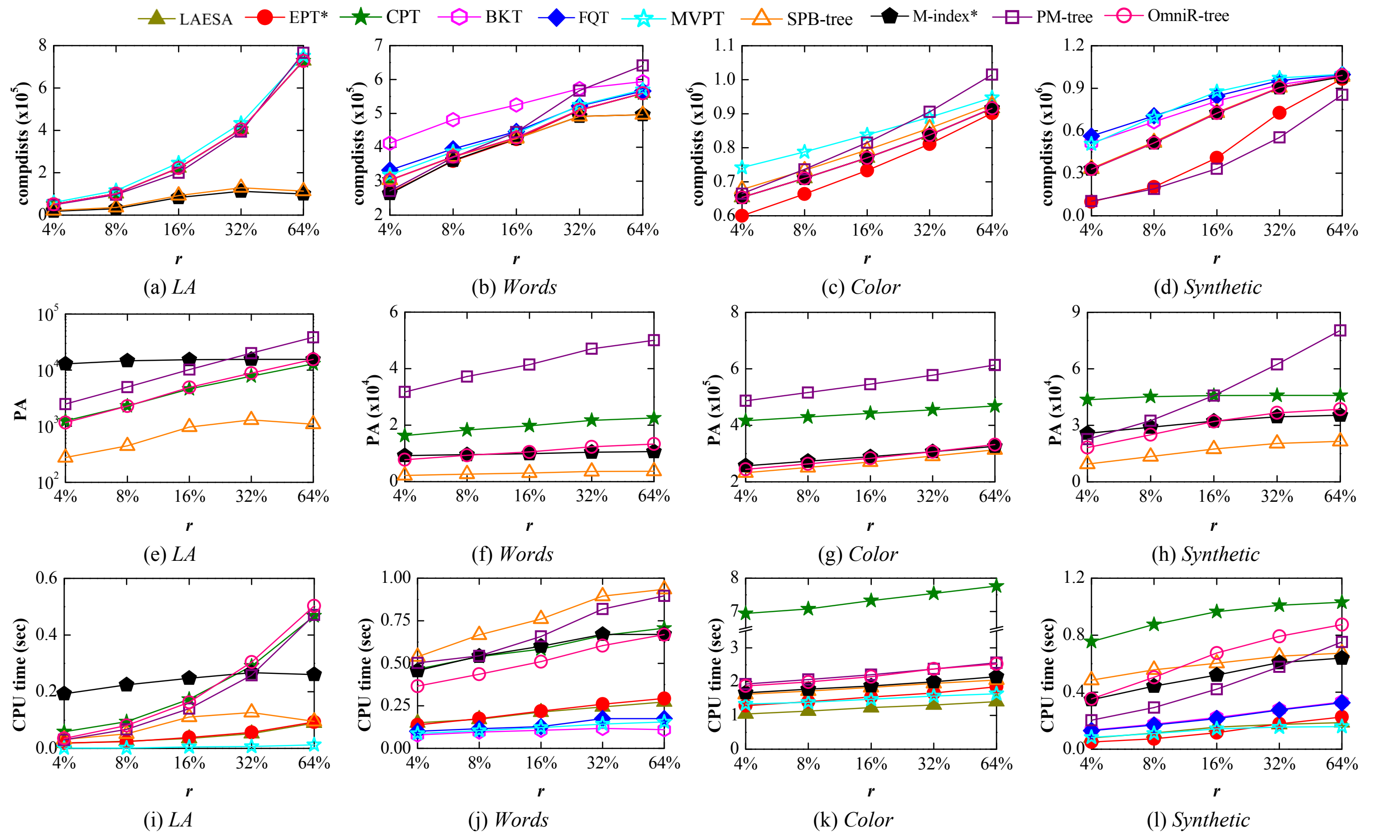


Figure 16. MRQ performance vs. radius *r*

6.5.1 *Effect of R*

We first compare the performance of the pivot-based metric indexes by using MRQ. Fig. 16 illustrates the corresponding query costs including *compdists*, *PA*, and CPU time for varying *R* values. As expected, the query cost increases with the growth of *R* due to a larger search space. However, on *LA*, the query costs of the SPB-tree and the M-index* drop at the value of 64% due to the use of pivots with stronger validation capabilities as achieved by larger *R* values.

Compdists. We observe that (i) the M-index* and the SPB-tree achieve high search performance in terms of *compdists* on *LA* and *Words*, because they utilize the pivot validation technique to avoid unnecessary distance computations; (ii) the PM-tree achieves high computational efficiency on *Synthetic* due to its range-pivot filtering, i.e., the routing objects of the PM-tree can be regarded as an additional pivot used for pruning; and (iii) EPT* has the smallest *compdists* on *Color*, as it selects different pivots for each object to achieve high pruning power. In addition, the *compdists* of the pivot-based trees (i.e., BKT, FQT, and MVPT) are slightly higher. This is because only some of the pre-computed distances used for the pivot filtering are stored. In addition, MVPT is slightly better than BKT and FQT in most cases, since BKT and FQT are unbalanced trees. Finally, the remaining indexes share similar *compdists*, as their pruning power relies on the pivot filtering based on the same set of pivots.

I/O cost. As can be seen, the SPB-tree has the lowest I/O cost, followed by the OmniR-tree and the M-index*, while CPT and the PM-tree perform the worst. The reasons are that, (i) the SPB-tree uses an SFC to compact the pre-computed distances while preserve the similarity proximity, thus incurring lower I/O cost; (ii) the OmniR-tree and the M-index* store all the pre-computed distances, resulting in larger I/O costs; and (iii) CPT and the PM-tree store the real objects directly in the tree structure instead of in a separate file, leading to low I/O efficiency. The I/O cost of the M-index* is high on *LA*, because MBBs do not cluster well on *LA* with the i-Distance technique.

CPU time. The first observation is that, the CPU costs of the in-memory indexes (viz., BKT, FQT, MVPT, LAESA, and EPT*) are relatively lower than those of the disk-based indexes (viz. CPT, the SPB-tree, the M-index*, the OmniR-tree, and the PM-tree). The reason is that the disk-based indexes need additional work to transform data read from disk into the formats required for further processing. In addition, the CPU cost of CPT on low dimensional datasets (e.g., *LA* and *Words*) is better than that on high dimensional datasets (e.g., *Color* and *Synthetic*) due to the additional CPU time needed to read objects from disk. It is observed that, the in-memory pivot-based trees (i.e., BKT, FQT, and MVPT) have lower CPU costs than the in-memory pivot-based tables (i.e., LAESA and EPT*), especially on *LA* and *Words*. This is because LAESA and EPT* need to scan the whole pivot table of the dataset, while the pivot-based trees can prune sub-trees via the pivot-based filtering.

6.5.2 *Effect of k*

Then, we compare the performance of indexes by using M*k*NNQs. Fig. 17 shows the query costs for different *k* values. As expected, query costs increase with the growth of *k* due to larger search space.

Compdists. It can be seen that, (i) the PM-tree and EPT* achieve the highest computational efficiency on *Color* and *Words*, and that (ii) the *compdists* of the pivot-based trees (viz., BKT, FQT, and MVPT) are the largest. The reasons are already discussed in Section 6.4.1. The second observation is that the *compdists* of the SPB-tree is higher than that of the M-index* and the OmniR-tree on *LA* and *Color*. This is because, for continuous distance functions, the SPB-tree uses approximated discrete distances in order to perform its SFC mapping,

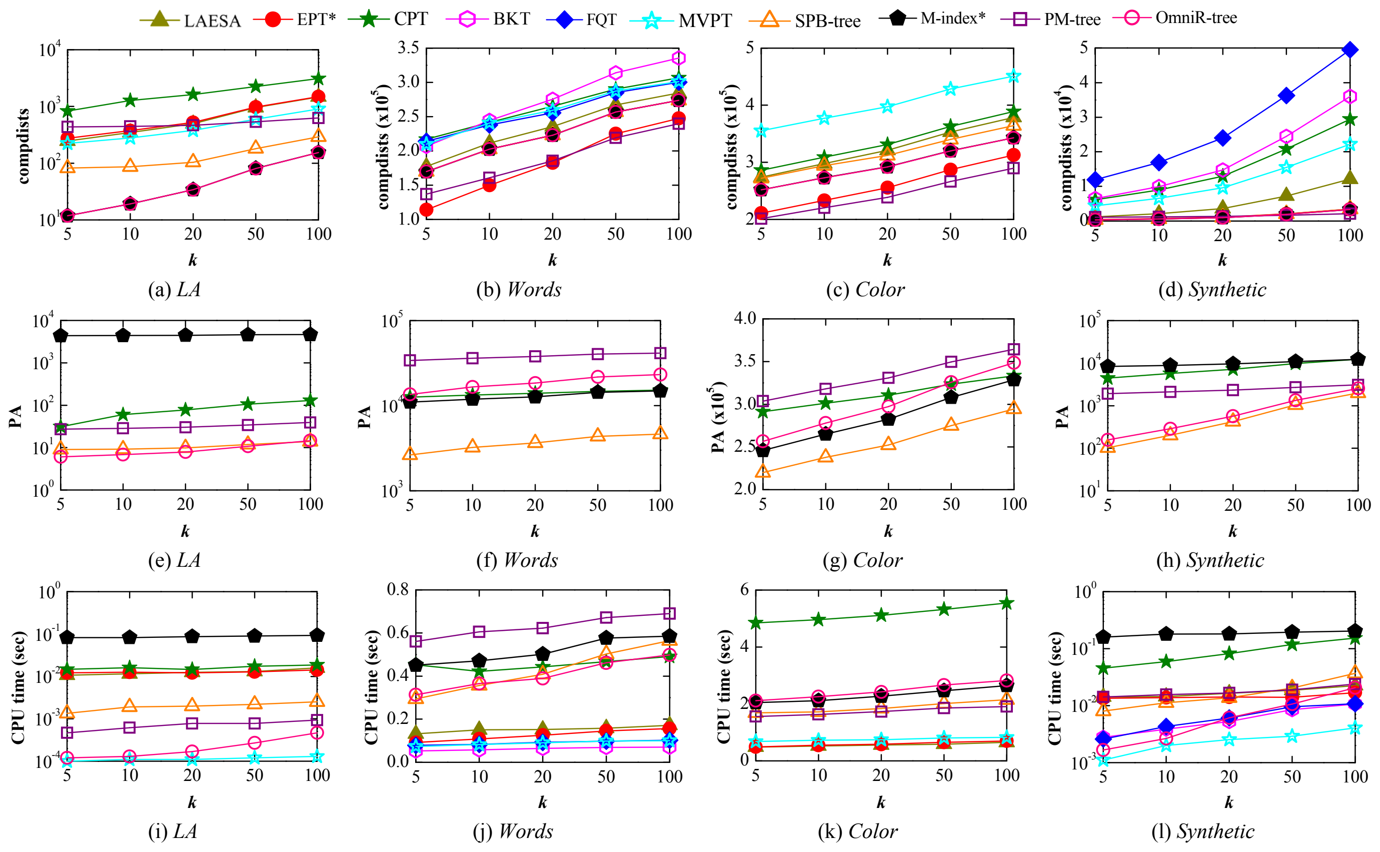


Figure 17. M*k*NNQ performance vs. *k*

resulting in less effective pivot-based filtering. In addition, the *compdists* of LAESA and CPT are relatively larger, because their M*k*NNQ algorithms traverse the objects in the dataset in the same order as they appear, which is suboptimal in terms of *compdists*.

I/O cost. First, we see that the SPB-tree achieves the highest I/O efficiency, as covered in Section 6.4.1. Second, the *PA* of the M-index* is the largest on *LA* and *Synthetic*, due to the unbalanced partitions caused by the data distribution. Finally, the *PA* of the PM-tree is the largest on *Color* and *Words* datasets. As the PM-tree stores objects directly in its tree structure instead of in a separate file, the high dimensional and variable sized data incurs low page utilization.

CPU time. As expected, the in-memory indexes have lower CPU costs than the disk-based indexes. Further, the CPU costs of EPT* and LAESA generally exceed those of the pivot-based trees in most cases. Because the M*k*NNQ algorithm that uses EPT* and LAESA verifies the objects in the order as they appear, resulting in many unnecessary verifications. In addition, although the computational cost of the SPB-tree is slightly higher than that of the M-index* on datasets using continuous distance functions, the CPU time of the M-index* is larger, due to the additional CPU cost caused by larger *PA*.

6.5.3 Effect of $|P|$

Next, we explore the influence of $|P|$ on the performance of the indexes. Here, M*k*NNQs are used due to the space limitation and similar findings for MRQs. Fig. 18 depicts the query costs using *LA* and *Synthetic*. Values for the M-index* are absent, as more than one pivot is needed for the generalized hyperplane partitioning. Next, the *compdists* drops as $|P|$ grows. This is because having more pivots yields better pivot filtering. The second observation is that, the *PA* and CPU time first drop and then stay stable or increase with $|P|$. The reason is that, (i) the number of verified objects drops as *compdists* decreases, incurring smaller I/O and CPU costs; and that (ii) the storage size increases due to more pre-computed distances being stored, resulting in more I/O and higher CPU costs. We can see that an appropriate number of the selected pivots is related to the intrinsic dimensionality, which is consistent with the observation made in [11].

7. CONCLUSIONS

We classify existing pivot-based metric indexes into three categories, i.e., pivot-based tables, pivot-based trees, and pivot-based external indexes, and we study their performance empirically on an equal footing. The resulting findings and insights, summarized below, enable users to select the indexes that best support the intended uses:

- Although the storage sizes of the indexes in our experiments are under 3 GB, which can be loaded into main-memory, the pivot-based external indexes can achieve better scalability than the pivot-based tables and trees for the cases when the available main memory is small or the dataset is extremely large.
- For the pivot-based tables, (i) CPT tries to improve LAESA by utilizing an M-tree to store the data, in order to handle the case when the dataset does not fit into main memory, resulting in high construction, update, and query costs; and (ii) EPT* tries to improve EPT by trading index construction efficiency for query efficiency. Although the construction cost of EPT* is high, it can be built in advance and has fewer distance computations for a query. As the computational cost is the dominant cost in the case of complex distance functions, EPT* is a good candidate for small datasets with complex distance functions.
- The pivot-based trees can achieve high construction and update efficiency. Although they incur more distance computations during search, they have smaller CPU time due to the tree

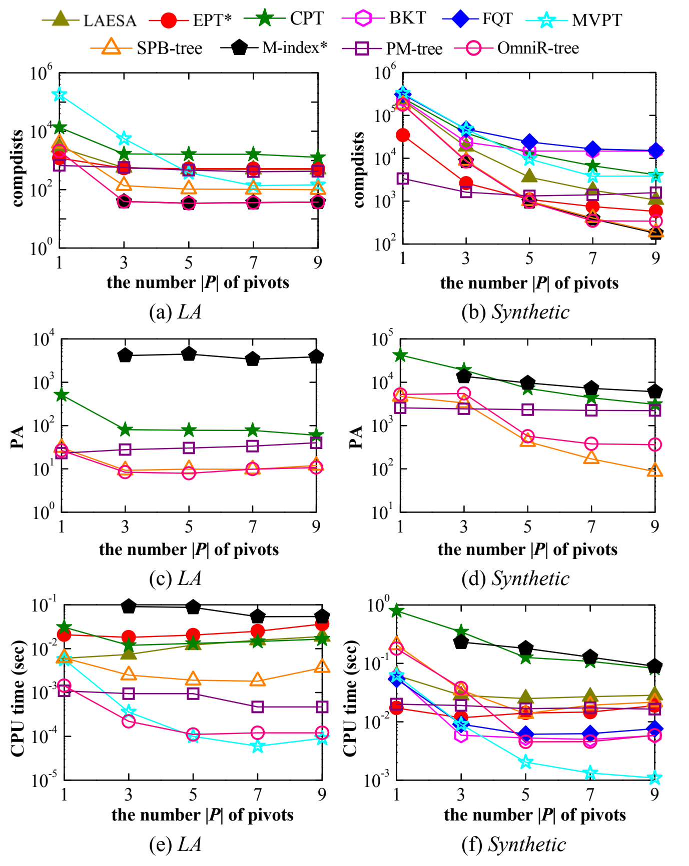


Figure 18. MkNNQ performance vs. |P|

structures and the absence of I/O costs. In addition, MVPT performs the best among the indexes of this category, because it uses a balanced tree. Thus, for small datasets with simple distance computation functions, MVPT is a good candidate.

- For pivot-based external indexes, (i) the PM-tree stores the data with pre-computed distances in the index structure, which incurs relatively large construction, update, and query costs; (ii) the SPB-tree and the M-index* achieve high construction, update, and query efficiency by using a B$^+$-tree with MBB information; and (iii) the SPB-tree outperforms the OmniR-tree, since it utilizes an SFC to reduce the storage cost and while to some extent preserving similarity locality. Hence, for large datasets, the SPB-tree and the M-index* are good candidates.

The study suggests that extension of EPT$^{(*)}$ to a disk-based metric index with a low construction cost is a promising direction. Also, comparisons between pivot-based metric indexes and compact partitioning metric indexes are an interesting research direction.

8. ACKNOWLEDGMENTS

This work was supported in part by the 973 Program of China Grant No. 2015CB352502, NSFC Grant No. 61522208 and 61379033, the NSFC-Zhejiang Joint Fund Grant No. U1609217, and a grant from the Obel Family Foundation. Yunjun Gao is a corresponding author of this work.

9. REFERENCES

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

[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*, pages 74–80, 2009.

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

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

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

[6] T. Bozkaya and M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Datab. Syst.*, 24(3):361–404, 1999.

[7] S. Brin. Near neighbor search in large metric spaces. In *VLDB*, pages 574–584, 1995.

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

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

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

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

[12] L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen. Efficient metric indexing for similarity search. In *ICDE*, pages 591–602, 2015.

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

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

[15] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[16] 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):364–397, 2005.

[17] 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):483–505, 2007.

[18] C. T. Jr, A. J. M. Traina, B. Seeger, and C. Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *ICDE*, pages 51–65, 2000.

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

[20] J. Mosko, J. Lokoc, and T. Skopal. Clustered pivot tables for I/O-optimized similarity search. In *SISAP*, pages 17–24, 2011.

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

[22] H. Noltemeier, K. Verbarg, and C. Zirkelbach. Monotonous bisector* Trees —A tool for efficient partitioning of complex scenes of geometric objects. In *Data Struc. and Efficient Algo.*, pages 186–203, 1992.

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

[24] G. Ruiz, F. Santoyo, E. Chavez, K. Figueroa, and E.S. Tellez. Extreme pivots for faster metric indexes. In *SISAP*, pages 115–126, 2013.

[25] E. Schubert, A. Koos, T. Emrich, A. Zufle, K. A. Schmid, and A. Zimek. A framework for clustering uncertain data. *PVLDB*, 8(12):1976–1979, 2015.

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

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

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

[29] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311–321, 1993.

[30] P. Zezula, G. Amato, V. Dohnal, and M. Batko. Similarity search: The metric space approach. *Springer US*, 2006.