KTree: a Kernel-Based Index for Scalable Exact Similarity Search

Khaoula Abdenouri
UM6P

Karima Echihabi
UM6P

ABSTRACT

We propose the KTree, a balanced tree-based index for exact similarity search, with $O(d\ log(n))$ average-case and worst-case search complexity, on a collection of n d-dimensional vectors. We present novel index building and query answering algorithms that leverage a new tight lower-bounding distance, a rich class of kernel-based splitting strategies, and a data-adaptive variance-guided segmentation that considers the best dimensions to join into segments regardless of their order. We establish theoretically the subsumption property of KTree's splitting (i.e., that it subsumes all popular existing splitting strategies), the correctness of the lower-bounding distance, and the logarithmic average-case and worst-case search complexity. We demonstrate empirically the superiority of the KTree with an exhaustive experimental evaluation against state-of-the-art techniques, using query workloads of varying difficulty and four real datasets, including two out-of-core datasets containing 1 billion dense vector embeddings. The results show that the KTree outperforms the second-best competitor (which is not always the same) by up to a 5.7x speed-up, a 2.2x better pruning ratio and a 15% tighter lower bound. We also conduct an ablation study that shows the impact of each key design choice on KTree's performance.

PVLDB Reference Format:
Khaoula Abdenouri and Karima Echihabi. KTree: a Kernel-Based Index for Scalable Exact Similarity Search, 15(10): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

1 INTRODUCTION

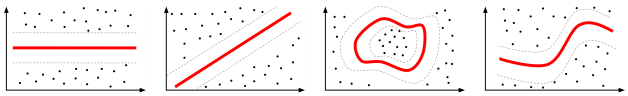
Data drives innovation in virtually all scientific and business domains [12, 71, 72, 88, 93, 118, 130, 133, 138, 152]. Rapid technological advancements, coupled with the explosion of interconnected devices, have produced massive datasets of high-dimensional objects. To extract meaningful value from these data collections, analysts must perform analytical tasks such as personalized recommendation, fraud detection, drug discovery, and cross-modal media retrieval [13, 25, 29, 39, 46–48, 87, 95, 115, 116, 125, 128, 131, 161]. At the core of these tasks lies a subroutine called similarity search, which consists of finding the data instances most similar to a query, based on some measure of similarity. The similarity search problem is often abstracted as a k-nearest neighbor (kNN) search, which returns the k closest objects to a query object according to some distance, such as the Euclidean distance.

Motivation. Similarity search has been studied for half a century, with many techniques being proposed in the literature. They are typically based on scans [57, 151], trees [4, 22, 27, 89, 97, 110, 120, 121, 146, 148, 156], hashing [69, 73, 141], graphs [58, 60, 102, 112, 139, 143], inverted indexes [11, 76, 79, 154], or a combination of these data structures [5, 28, 58, 82]. Some techniques compute exact answers, while others ng-approximate answers, i.e., without theoretical guarantees on query accuracy, or ϵ- and δ-ϵ-approximate answers, i.e., with deterministic and probabilistic guarantees, respectively, on the accuracy of the answers. While ng-approximate search techniques have seen a surge in adoption [6, 144] thanks to their excellent empirical performance, they may not be appropriate for critical applications in regulated fields such as law, aviation, and medicine that could require guarantees on the quality of the answers. Besides, a recent work [75] has identified adversarial instances where popular ng-approximate search techniques such as HNSW [102] would require linear time, i.e. the cost of an exact kNN brute force scan, to find true nearest neighbors, despite their strong average-case performance. In addition, a recent survey [55] has demonstrated that the recall of HNSW search is linked to the dataset's intrinsic dimensionality and is significantly affected by the data insertion order, in contrast to exact kNN search recall which is invariant to both.Many exact solutions have been proposed. Since those based on scans [49, 57, 106, 147] do not scale to large dataset collections, indexes were introduced to improve search efficiency [2, 23, 32, 45, 66, 85, 122, 126, 149]. The most popular being the k-d-tree [17], the R-tree [15], and the M-tree [33]. These indexes exploit different splitting strategies and summarization techniques to organize the data into a tree, with the goal of storing similar objects in the same leaf. During search, unpromising subtrees are pruned, typically based on a lower-bounding distance [56] to guarantee correctness. Recently, it has been shown that the data series techniques DSTree [150] and Hercules [45] outperform the popular indexes developed for generic high-dimensional vectors [45, 49].

Challenges. State-of-the art (SotA) tree-based indexes, whether designed for data series [45, 148] or generic vectors [15, 17, 33], rely on predefined splitting strategies such as hyperplanes for the k-d tree (Fig. 1a), hyperrectangles for the R-tree, and hyperspheres for the M-trees. These strategies often produce suboptimal partitions because they do not adapt the splitting strategy at each node, thus degrading pruning efficiency during search, as they may separate similar points while grouping dissimilar ones. In contrast, DStree and Hercules support two different splitting strategies, based on either the average (Fig. 1b) or standard deviation (Fig. 1c) which we demonstrate for the first time to be based on a diagonal hyperplane, and an irregular quadratic shape, respectively, and rely on heuristics to choose the best strategy at each node. However, both methods are still limited to only two splitting options. Besides, their summarization always consider adjacent dimensions, which is useful for data series because the dimensions are ordered and the neighboring dimensions are correlated, but not necessarily so for generic vector data such as embeddings, where non-consecutive dimensions could reveal more meaningful patterns. In addition,

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:XX.XX/XXX.XX

both the DSTree and Hercules are unbalanced trees; therefore, lacking guarantees on average-case complexity, and prone to a linear worst-case complexity. Note that although the R-tree, M-tree and k-d-tree offer average-case guarantees, they empirically lag behind DSTree and Hercules, which we believe is due to their single splitting strategy, leading to similar points being scattered across many leaves, affecting negatively query performance.



(a) k-d tree (b) DSTree (μ) (c) DSTree (σ) (d) KTree

Figure 1: Splitting Strategies (note that only one of KTree's splitting options is indicated. The other options include hyperplanes and hyperspheres.)

Contributions. In this paper, we make the following contributions.
• We propose the KTree, a novel balanced tree-based index for exact similarity search, with logarithmic average-case and worst-case complexity, that addresses the three key aforementioned challenges in SotA works, namely the predefined splitting strategies, the summarizations that exploit only neighboring dimensions, and the lack of average-case guarantees on search performance.
• We propose a novel rich class of non-linear node splitting strategies that exploits kernels, and to the best of our knowledge, has not been proposed before in the context of similarity search (Fig. 1d). We demonstrate theoretically that it subsumes all popular existing strategies and we show experimentally that it leads to superior performance. Besides, we share for the first time some new insights that explain how the splitting strategies adopted by the DSTree (and similarly Hercules) lead to their superior performance.
• We propose a new variance-guided data-adaptive segmentation technique that considers the best dimensions to join into segments regardless of their original order.
• We provide a complexity analysis for KTree's worst-case and average-case indexing and query answering, and prove the correctness of the lower-bounding distance and exact search.
• We show empirically the superiority of the KTree against popular baselines (R-tree, M-tree, k-d tree, DSTree, Hercules, VA+file and serial scan) on billion-scale real datasets. Our comparison considers different metrics such as wall-clock-time, pruning, tightness of the lower bound, and footprint. We also conduct an ablation study that shows the impact of KTree's design choices on its performance.
Scope. This paper addresses the problem of exact k-nearest neighbor using the Euclidean distance, on large collections of dense high-dimensional vectors, which is relevant for real applications that would not tolerate unpredictable recall gaps [55] or require sublinear search time performance [75]. Extending the work to other distance measures, exploiting non-trivial parallel algorithms, and supporting dynamic datasets with frequent updates are out-of-scope and part of our future work.

2 PRELIMINARIES AND RELATED WORK

2.1 Definitions

Exact similarity search for high-dimensional objects is typically abstracted as an exact k-NN search problem in high-dimensional vector space, where the (dis)similarity between objects is measured using the distance between their vector representations [33, 45, 66, 148]. Table 1 summarizes the notations used in the rest of the paper.

Notation	Description
$\mathbb{D}, n, d$	The dataset of n d-dimensional vectors
$\mathcal{N}$	Node
$\mathbb{V}_{\mathcal{N}}$	Set of vectors in $\mathcal{N}$
$\overline{\mathbb{V}}_{\mathcal{N}}$	Set of segmentations for all vectors in $\mathbb{V}_{\mathcal{N}}$
$\widehat{\mathbb{V}}_{\mathcal{N}}$	Set of summaries (segmented and quantized) for all vectors in $\mathbb{V}_{\mathcal{N}}$
$\widetilde{\mathbb{V}}_{\mathcal{N}}$	Set of DFT coefficients for all vectors in $\mathbb{V}_{\mathcal{N}}$
$V[i,j]$	The value of the jth dimension for the ith vector ($\in \mathbb{R}$)
	Notation also used with any vector transformation, e.g.,
	$\overline{V}[i,j]$ refers to the jth value for the ith segmented vector
$V[i,:]$	The ith vector (e.g., the embedding with id i)
	Notation also used with any vector transformation, e.g.,
	$\widehat{V}[i,:]$ refers to the ith summarized vector
$V[:,j]$	The vector of the jth dimensions for all vectors in $\mathbb{D}$
	Notation also used with any vector transformation, e.g.,
	$\widetilde{V}[:,j]$ refers to the jth dimensions for all DFT summaries
r_s	right endpoint of segment s, $r_0 = 1$
$\mathcal{SG}$	Segmentation of a node (vector of the right endpoints)
ψ	Synopsis of a node (vector of min/max pairs)
$Q, \overline{Q}, \widehat{Q}$	Query and its segmentation and quantization respectively
	Q[j] refers to the jth dimension of Q
$d_{ed}(.,.)$	Euclidean distance
$d_{lb}(.,.)$	Lower-bounding distance
$\mathcal{K}$	Kernel function
α	Number of most variant dimensions
α'	Subset of the α dimensions within the refined segment ($\alpha' \leq \alpha$)
α''	Number of kernels used for data augmentation

Table 1: Notations

Definition 1. *The **Euclidean distance** between two vectors $V[i,:]$ and $V[i',:]$ measures their dissimilarity:*

$$d_{ed}(V[i,:], V[i',:]) = \sqrt{\sum_{j=1}^{d} (V[i,j] - V[i',j])^2}$$

Definition 2. *Given a positive integer $0 \leq k \leq n$, an **exact kNN query** Q, over a dataset $\mathbb{D}$, retrieves the set $\mathbb{A} \subseteq \mathbb{D}$ of k vectors $V[i,:]$ such that $0 \leq i \leq n$, $|\mathbb{A}| = k$, $\forall V[i,:] \in \mathbb{A}$ and $\forall V[i',:] \notin \mathbb{A}$, $d_{ed}(V[i,:], Q) \leq d_{ed}(V[i',:], Q)$*

Definition 3 ([132]). *A function $\mathcal{K} : \mathbb{R}^\beta \times \mathbb{R}^\beta \rightarrow \mathbb{R}$ (where β is the dimension of the input space) is a kernel if it computes an inner product in some feature space $\mathcal{F}$ via a mapping $\phi : \mathbb{R}^\beta \rightarrow \mathcal{F}$, such that:*

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle_{\mathcal{F}}$$

2.2 Similarity Search Primer

2.2.1 Overview. Similarity search consists of finding data objects within a collection similar to a given query object, according to some definition of similarity. There exist different variations for

the similarity search problem, one of the most popular being the k-nearest neighbor (kNN), where the (dis)similarity between points is measured using some distance, often the Euclidean distance. Over the past 50 years, a large number of kNN methods have been proposed. They can be categorized into: 1) exact methods that return the k points closest to a query; 2) ng-approximate methods that return k close points to a query without theoretical guarantees on their quality but with good empirical accuracy; 3) ϵ-approximate methods that return k points that are respectively within $1+\epsilon$ from the k exact matches to a query; and 4) δ-ϵ-approximate methods that return ϵ-approximate answers with probability $1-\delta$.

These similarity search methods exploit scans, trees, hash tables, inverted indexes, graphs, or a mix of these structures. Exact search methods are typically based on scans [7, 18, 30, 40, 49, 57, 106, 147] or trees [2, 15, 17, 20, 23, 32–34, 85, 108, 122, 126, 134, 149, 157, 159]. Methods that are *ng*-approximate usually exploit graphs [43, 59, 61, 94, 103, 140] or inverted indexes [10, 77, 80, 155], while deterministic/probabilistic approaches (δ-ϵ-approximate) leverage hashing [19, 26, 41, 62, 70, 74, 99–101, 107, 114] or trees [31, 52]. Some methods that exploit more than one data structure [5, 45, 112].

2.2.2 Summarization Techniques. Summarization techniques reduce the complexity of the data while preserving some of its important characteristics. We now discuss the main summarization techniques relevant to our work.

Segmentation reduces the vector dimensionality by dividing it into segments with each segment represented by a synopsis of the data belonging to it. Piecewise Aggregate Approximation (PAA) [86] and Adaptive Piecewise Constant Approximation (APCA) [24] consider equi-length and variable-length segments, respectively, and the mean as the synopsis. Extended Adaptive Piecewise Constant Approximation (EAPCA) [148] further enhances APCA by incorporating both the mean and standard deviation for each segment, which leads to tighter distance bounds and thus improved efficiency.

Quantization converts a continuous range of values using a finite set of codewords. Scalar Quantization converts each individual dimension separately [63]. Vector Quantization [98] converts the entire vector. Product Quantization [78] divides the vector into a set of smaller subvectors, then applies vector quantization on the latter. This reduces the complexity of applying quantization on the full vector both in time and space. Additive Quantization (AQ) [8] further improves upon product quantization by representing data points as a sum of multiple codewords, enabling more accurate approximations. Optimized Product Quantization [9] enhances the precision of product quantization by adding a transformation to decorrelate dimensions and optimize space decomposition.

Symbolic Aggregate Approximation (SAX) [96] converts data series using PAA and Scalar Quantization, representing the PAA values in a symbolic form using an alphabet of a given cardinality. The iSAX [21] technique allows the indexability of SAX by supporting comparisons of SAX representations with different cardinalities.

Discrete Fourier Transform (DFT) [3, 56, 123, 124] a transforms data series into the frequency domain, capturing periodic patterns while discarding noise, making it effective for trend detection in data series. It is used as a dimensionality reduction by representing each data series with only some of the frequency coefficients.

2.2.3 Exact Similarity Search Methods. While ng-approximate search techniques are popular [43, 53, 59, 61, 94, 103, 113, 140, 145] they may not be appropriate for critical applications in regulated fields such as law, aviation, and medicine [44, 90, 142] that require guarantees on the quality of the answers. A number of exact solutions have been proposed. Scans [109, 127] typically compare a query to every single candidate, which is not practical for large data collections. Indexes were introduced to improve search efficiency. Skip-sequential scans [151] prune distance calculations by using a filter file. Tree-based indexes are considered the methods of choice for exact search [50]. They exploit different splitting strategies and summarization techniques to organize the data into a tree hierarchy, with the goal of storing similar objects in the same leaf. During search, entire subtrees can be pruned avoiding costly CPU and I/O operations. This is achieved thanks to a lower-bounding distance [56] that guarantees search correctness. Recently, it has been shown that the data series techniques DSTree [150] and Hercules [45] outperform the popular indexes developed for generic high-dimensional vectors [45, 49], including the M-tree [33], the R-tree [66] and k-d tree [16]. The only method developed for generic vectors that is competitive is the VA+file [50, 151]. Next, we describe briefly these popular exact search techniques.

K-d tree [135] is a binary tree that partitions the data along axis-aligned hyperplanes. While different variations [111] exist, the original k-d tree splits the data based on a median value along a specific dimension, cycling through dimensions as the tree grows deeper. The k-d tree can be competitive in smaller dimensions, but its effectiveness decreases when the dimensionality exceeds 12.

R-tree [66] is designed for multi-dimensional spatial data, organizing it into minimum-bounding rectangles (MBR). Each internal node represents an MBR that encloses its children. During search, the MBRs help prune the search space by eliminating nodes that cannot possibly contain the query result. This makes R-trees particularly well-suited for applications such as geographical information systems (GIS), where spatial indexing is critical. Variants like the R*-tree [15] and Hilbert R-trees [84] optimize the placement of the MBRs, further enhancing query efficiency and robustness.

M-tree [33] is a balanced tree-based index for similarity search in metric spaces. It typically uses hyper-spheres to divide the data entries according to their relative distances. The leaf nodes contain data objects, and the internal nodes routing objects; with both storing distances from each object to its parent. During query answering, the M-tree uses these distances to prune the search space. The search correctness is guaranteed by the triangle inequality that holds for metric distance functions.

DSTree [148] is a data-series tree-based index that exploits EAPCA, a data-adaptive dynamic segmentation strategy. Each node stores segmentation information and maintains (upper/lower) bounds for distance computations. Empirical studies [50, 52] demonstrate DSTree's ability to outperform the state-of-the-art indexes designed both for data series and generic vectors on exact similarity search.

Hercules [45] is a hybrid index that exploits both SAX and EAPCA to support disk-based similarity search for data series. It also leverages a two-level buffer management technique to optimize memory accesses and parallelism to accelerate CPU-intensive calculations. Hercules also proposes a more efficient scheduling of costly EAPCA and SAX operations compared to related works [22, 148].

VA+file [151] is an extension of the VA-file [151]. Both techniques quantize the input vectors and store their quantized values in a filter file. During search, the filter file is used to prune the space, and the results are further refined using the original vectors to return the exact similarity search results. Unlike the VA-file, the VA+file considers that dimensions could exhibit some correlation, and thus allocates a different quantization budget to each dimension.

2.2.4 Kernels and Similarity Search. Per Definition 3, a kernel is a function that calculates the similarity score between two points by computing the inner product of their mappings in a higher-dimensional space, without computing the coordinates in that space directly. This is often useful when data in the original space is not linearly separable, but becomes separable in the higher space. Kernels have been exploited in the context of similarity search over dense vector embeddings for both exact and approximate search. In exact search, several works have proposed branch-and-bound methods that operate directly in the kernel-induced space [37, 38, 129]. However, both methods in [37, 38] focus on the generalized max-kernel search problem in the presence of unnormalized kernels, when it cannot be reduced to nearest neighbor search, and the approach in [129] focuses on maximum inner product search. In approximate search, a variety of methods leverage kernelized hashing and explicit kernel feature maps for efficient sublinear-time query answering [68, 81, 92, 153, 158]. To the best of our knowledge, KTree is the first method to combine per-node canonical (PCA/LDA or exact-kernel) splits with per-node quantization refinement and guarantee exact similarity search. The closest methods are PCA-based tree indices for exact search on dense data [42, 104, 105] and trees with per-node quantization but only approximate search [36].

3 THE KTREE APPROACH

3.1 Overview

Our approach consists of two key phases, an indexing phase, which builds a tree top-down and stores it on-disk, and a search phase which exploits the built tree to answer kNN queries. During indexing, the KTree builds a balanced tree, where each internal node contains a summary of the data belonging to its subtree, and each leaf node is associated with two files, an *RFile* that stores the raw data and an *SFile* that stores the Discrete Fourier Transform (DFT) [35] summarizations for the leaf's data. Then, the entire index tree and leaf files are materialized to disk. During search, the tree and *SFiles* are pre-loaded into memory, while the *RFiles* are kept on disk until a query requires some of them for processing because their corresponding leaves (or their ancestors) could not be pruned by the algorithm. First, the query returns an initial answer by heuristically visiting one leaf. Then, the tree is traversed in a Breadth-First Search, with the best-so-far (bsf) answers being periodically updated, and nodes pruned whenever their lower-bounding distance to the query indicates that their subtrees cannot improve the answers. Once all nodes are either pruned or visited, the exact k answers are returned.

3.2 The KTree Index

Figure 2 illustrates an example KTree (depth = 3, fanout = 2), where the $\mathcal{L}_{121}$ leaf is associated with an *RFile* consisting of 8-dimensional raw vectors, and an *SFile* containing their 4-dimensional DFT summaries. At the root level, the entire dataset is summarized as a single segment. Then, the data is evenly split into the two children nodes $\mathcal{N}_1$ and $\mathcal{N}_2$. The dimensions are then reordered before refining the summaries within each of $\mathcal{N}_1$ and $\mathcal{N}_2$ node individually ($\mathcal{SG} = [4, 8]$). Since both children exceeded their capacity, further splits are necessary. Node $\mathcal{N}_2$ is recursively divided into $\mathcal{N}_{21}$ and $\mathcal{N}_{22}$, which are again split: after summarizing the node, the most variant dimensions, specifically d1, d3 and d8, are identified. The first segment is then selected to utilize its dimensions for the split. Furthermore, the segmentation is further refined ($\mathcal{SG} = [2, 4, 8]$). We assume that, at this stage, the resulting subsets no longer exceed the leaf size constraint, making the children of $\mathcal{N}_{12}$ and $\mathcal{N}_{22}$ leaf nodes. For each leaf node, the data it contains is first summarized using DFT coefficients. The original high-dimensional data corresponding to each leaf is mapped to them on disk.

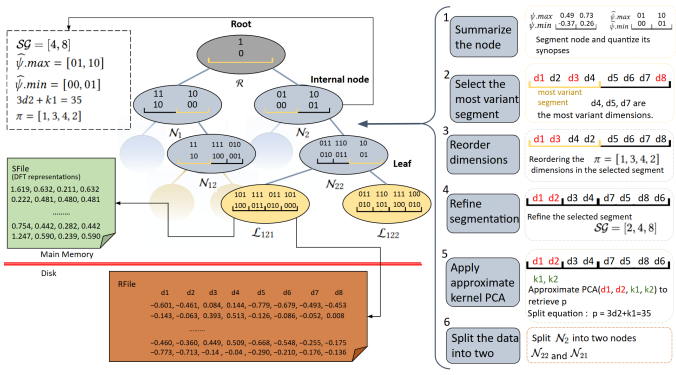


Figure 2: KTree indexing structure and workflow

3.3 Index Construction

Algorithm 1 illustrates KTree's index construction phase. Note that the algorithm assumes the following global variables: 1) $\mathcal{K}$, the kernel function used for splitting, 2) α, the maximum number of most variant original dimensions considered during segmentation and splitting ($\alpha' \leq \alpha$, the actual number of original dimensions used at each node, is deduced by the algorithm), 3) α'', the number of augmented dimensions generated by $\mathcal{K}$ on top of the α' original dimensions, 4) the maximum leaf size τ, and 5) the number of DFT coefficients to summarize the raw vectors. Note that KTree can support any fanout and any combination of different kernel functions, but for simplicity and without loss of generality, we consider the polynomial kernel function (we will demonstrate in §4.1.2 that it is sufficient to subsume all popular splitting functions), and a fanout of two. The algorithm recursively inserts data top-down starting at the root node $\mathcal{R}$. It first summarizes the current node's data by reducing its dimensionality using a variance-guided segmentation technique (line 2) and compressing this segmentation using multi-resolution quantization. Then the minimum and maximum quantized values are computed for each node segment and stored in the node's quantized synopsis $\widehat{\psi}$ (line 3). Note that only these quantized synopses are stored (to help pruning during query answering), not the full segmentations and quantizations of the node's

data. If the current node does not need to be split (line 10) because its size has not yet reached the maximum leaf size, its data is stored in its *RFile*, and its DFT summaries are computed and stored in its *SFile* (lines 11-12) . However, if a node should be split (line 4), its children are initialized (line 5), its segmentation is refined (line 6), and the kernel function $\mathcal{K}$ is leveraged to find an effective splitting strategy for this node (different nodes can have different splitting criteria) and redistribute its data into its children (line 7). Then, the algorithm follows the same approach to recursively build the current node's left and right subtrees (lines 8-9). We now describe in detail each of KTree's building blocks.

Algorithm 1: BuildIndex

```
   Input: Node* N, Float** V_N
1  Node* L, R; Float** V̄_N, V̂_N, V_L, V_R;
2  V̄_N ← SegmentVectors(N, V_N);
3  N.ψ̂ ← QuantizeVectors(N, V̄_N);    ▷ Note that V̄_N and
     V̂_N are used to compute the synopis but are not
     stored in N;
4  if N.toSplit() then
5  |  initialize N's children L and R;
6  |  SegmentNode(N, V_N);
7  |  SplitNode(N, V_N, V_L, V_R);
8  |  BuildIndex(L, V_L);
9  |  BuildIndex(R, V_R);
10 else
11 |  Ṽ_N ← DFT(V_N);
12 |  Store V_N and Ṽ_N in N's RFile and SFile
   |    respectively;
```

3.3.1 Variance-Guided Segmentation. To efficiently support similarity search on high-dimensional vector data, KTree adopts a novel segmentation technique for dimensionality reduction. Each node $\mathcal{N}$ in the KTree index represents vectors in $\mathbb{V}_\mathcal{N}$ with the same segmentation $\mathcal{N}.\mathcal{SG}$, which is a refinement of $Parent(\mathcal{N}).\mathcal{SG}$. For simplicity and without loss of generality, the root's segmentation $\mathcal{R}.\mathcal{SG}$ consists of a single segment. The segmentation is refined when a node is split by subdividing a selected segment in two, thus increasing the number of total segments by 1.

Applying a Node's Segmentation to its Vectors. Considering that a node has a segmentation $\mathcal{SG}$, the set of its segmented vectors $\overline{\mathbb{V}}_\mathcal{N}$ is represented as follows:

$$\overline{\mathbb{V}}_\mathcal{N} = \{\overline{V}[i,s],\ \forall\, V[i,s] \in \mathbb{V}_\mathcal{N},\ \forall\, s \in \mathcal{SG},\ \forall\, i < n\}$$
such that
$$\overline{V}[i,s] = \frac{1}{r_s - r_{s-1}} \sum_{j=r_{s-1}}^{r_s-1} V[i,j] \tag{1}$$

In Algorithm 1, the `SegmentVectors` function (line 2) applies equation 1 to the vectors of the current node, and the `SegmentNode` function (line 6) refines the segmentation of the split node's children per Algorithm 2 that we will now describe in more detail.

Selecting a Segment for Refinement. In the trivial case where only one segment exists, e.g., at the root node, this segment is selected for refinement. Otherwise, KTree first calculates the variance

Algorithm 2: SegmentNode

```
   Input: Node* N, Float** V_N
1  Float* W[α];              ▷ α most variant dimensions
2  Node* L = N.L, R = N.R;
3  W ← MostVariantDimentions(V_N);
4  N.selectedSegment ← VoteSegment(W, N.SG)
5  reorderSegment(V_N, N.selectedSegment);  ▷ L.π and R.π
     will each store a copy of the permutation vector
6  L.SG, R.SG ← RefineSegment(N.selectedSegment)
```

for each dimension j to determine the α most variant dimensions (line 3) within a given node. Considering a node $\mathcal{N}$, the unbiased variance estimator σ_j^2 for dimension j on the m vectors in $\mathbb{V}_\mathcal{N}$ ($m = n$ when $\mathcal{N}$ is the root) is calculated as:

$$\sigma_j^2 = \frac{1}{m-1} \sum_{i=1}^{m} \left(V[i,j] - \frac{1}{m} \sum_{j'=1}^{m} V[i,j'] \right)^2$$

In the non-trivial case, these α dimensions can be spread into multiple segments, KTree adopts a voting strategy where each of the α identified dimensions casts a vote for the segment to which it belongs (line 4). The segment with the highest vote count is selected for refinement. We refer to the number of most variant dimensions within this segment as α'. Note that $\alpha' = \alpha$ when the α dimensions fall within the same segment, e.g., at the root node $\mathcal{R}$.

Refining the Selected Segment. Once the segment to be refined has been identified, it undergoes a permutation that orders its dimensions from the most to the least variant (line 5). This permutation, which is only applied to the selected segment, prioritizes the structural separation of variability: dimensions with high variance, which often correspond to salient features or patterns in the data, are explicitly decoupled from less informative (low-variance) dimensions. This approach not only enhances the interpretability of the segmentation hierarchy, but also ensures that splits capture the most statistically significant boundaries in the data, avoiding fragmentation along dimensions that contribute little to the overall variance. To optimize these computations, particularly on disk-based data, KTree employs the computational formula in a single-pass (i.e, computing the sum of squares and the square of the sum). Let r_s be the right endpoint of the segment s selected for refinement and r_{s-1} the right endpoint of its preceding segment. The segment represented by $\mathcal{SG}_\mathcal{N}[s] = r_s$ is refined by dividing it at the midpoint r_{div} as follows (line 6):

$$r_{\text{div}} = \lfloor \frac{r_{s-1} + r_s}{2} \rfloor \tag{2}$$

This refined segmentation is assigned to the children $\mathcal{L}$ and $\mathcal{R}$ of the split node $\mathcal{N}$, i.e., $\mathcal{L}.\mathcal{SG} = \mathcal{R}.\mathcal{SG} = [1, ..., r_{s-1}, r_{div}, r_s, ..., d]$ (segmentation refinement occurs only during a split).

3.3.2 Kernel-Based Splitting. We now discuss how the node $\mathcal{N}$ is split and its vectors distributed across its children $\mathcal{L}$ and $\mathcal{R}$. KTree exploits approximate Kernel Principal Component Analysis (KPCA)[1] to choose the strategy that will best separate this node's data then redistribute it across its children. As we will formally

[1]Note that kernels are exact and PCA is approximate.

demonstrate in §4, this splitting strategy subsumes those adopted by SoTA similarity search methods, including the R-tree, the k-d-tree, the M-tree, and the DSTree, and covers a new rich class of non-linear space partitioning techniques that, to the best of our knowledge, have not been proposed before in the context of similarity search. Algorithm 3 summarizes KTree's splitting strategy.

Algorithm 3: SplitNode

Input: `Node*` $\mathcal{N}$, `Float**` $\mathbb{V}_{\mathcal{N}}, \mathbb{V}_{\mathcal{L}}, \mathbb{V}_{\mathcal{R}}$

1 $p \leftarrow$ `KPCA`$(\mathbb{V}_{\mathcal{N}}, \mathcal{N}.\mathcal{SG}[\mathcal{N}.selectedSegment])$
2 $Z_{med} \leftarrow$ `Median`$(\mathbb{V}_{\mathcal{N}}, p)$;
3 $\mathbb{V}_{\mathcal{L}}, \mathbb{V}_{\mathcal{R}} \leftarrow$ `PartitionData`$(\mathbb{V}_{\mathcal{N}}, p, Z_{med})$

Finding the Most Informative Component. Once the target segment is selected per §3.3.1, approximate KPCA is applied to augment the α' dimensions of this segment (using the kernel $\mathcal{K}$ per Definition 3,$\beta = \alpha'$) with α'' kernels, and identify the first principal component of the $\alpha' + \alpha'' \ll d$ dimensions, taking into account only $\mathbb{V}_{\mathcal{N}}$ (line 1). The first principal component in PCA, denoted as **p**, is obtained by solving the first optimization problem of SVD, typically using an iterative method to approximate the solution.

Partitioning the Target Node. To redistribute the vectors in $\mathbb{V}_{\mathcal{N}}$ across $\mathcal{N}$'s children, each vector is projected onto the first principal component **p** as follows:

$$Z_i = \mathbf{p}^T V[i,:] \; \forall \; V[i,:] \in \mathbb{V}_{\mathcal{N}} \tag{3}$$

The median of these projected values, denoted as Z_{med}, is computed (line 2) and used to partition the data (line 3) as follows:

$$\begin{aligned} \mathbb{V}_{\mathcal{L}} &= \{V[i,:] \in \mathbb{V}_{\mathcal{N}} \mid Z_i \leq Z_{\text{med}}\}, \\ \mathbb{V}_{\mathcal{R}} &= \{V[i,:] \in \mathbb{V}_{\mathcal{N}} \mid Z_i > Z_{\text{med}}\}. \end{aligned} \tag{4}$$

This ensures that both nodes have approximately the same fill factor, keeping the tree balanced. Note that the only splitting-related information stored at the node is the split equation which consists of the index of the dimensions involved in the split, their coefficients and the threshold Z_{med} (equation 4).

3.3.3 Multi-Resolution Quantization. Once the split completes, Equation 1 is applied on the vectors of each child to represent them according to the new refined segmentation. In Algorithm 1, this occurs (line 2) after calling recursively the `BuildIndex` function on each child (lines 8- 9). However, to optimize storage overhead and computational efficiency, the full segmentations of the vectors are not stored, instead a synopsis for each segment is computed per Equation 5 as follows:

$$\begin{aligned} \psi[s].min &= \underset{\overline{V}[i,s] \in \overline{\mathbb{V}}_{\mathcal{N}}}{\arg\min} \; (\overline{V}[i,s]) \\ \psi[s].max &= \underset{\overline{V}[i,s] \in \overline{\mathbb{V}}_{\mathcal{N}}}{\arg\max} \; (\overline{V}[i,s]) \end{aligned} \tag{5}$$

Then, a multi-resolution quantization scheme is applied on each synopsis (line 3) per Equation 6 :

$$\begin{aligned} \widehat{\psi}[s].min &= min(\overline{V}[:,s]) + \Delta_s \cdot \lfloor \frac{\psi[s].min - min(\overline{V}[:,s])}{\Delta_s} \rfloor, \\ \widehat{\psi}[s].max &= min(\overline{V}[:,s]) + \Delta_s \cdot \lfloor \frac{\psi[s].max - min(\overline{V}[:,s])}{\Delta_s} \rfloor, \end{aligned} \tag{6}$$

$$\Delta_s = \frac{max(\overline{V}[:,s]) - min(\overline{V}[:,s])}{L_s} \tag{7}$$

where $min(\overline{V}[:,s])/max(\overline{V}[:,s])$ are the minimum/maximum values at segment s across the entire dataset (not just the current node), and L_s is the number of quantization levels, i.e., the number of discrete values allowed for quantization, effectively determining how finely the continuous range within the segment is divided.

The synopses of all segments are stored within the node, where the bit budget for each segment depends on its importance. Segments start with a given bit budget, for instance 8 bits at the root node. The bit budget for each refined segment in any given node is double the budget it inherited from its parent.

3.3.4 Dual-Granularity Leaf Representation. Once the KTree index has been built, each leaf $\mathcal{N}$ is represented with a dual-granularity storage scheme: 1) an *Rfile* that stores $\mathbb{V}_{\mathcal{N}}$; and 2) an *SFile* that stores $\widetilde{\mathbb{V}}_{\mathcal{N}}$. The *RFiles* are stored on disk since the raw dataset size may exceed the memory size, while the *SFiles* are pre-loaded into memory with the index tree (as we will see in §5, *SFiles* constitute a small fraction of the total dataset size (#DFT summaries $\times n$). This dual-granularity scheme reduces false positives, and thus minimizes computational overhead both in terms of CPU (distance calculations) and I/O (disk accesses when entire summaries of an SFile are pruned). Recall that the *SFile* and *RFile* for each leaf are populated once the leaf is created in Algorithm 1 (line 12).

3.4 Query Answering

Algorithm 4 outlines KTree's query answering. Note that KTree supports kNN search, but we only include the 1NN algorithm for space considerations since the extension to kNN is straightforward. The algorithm initializes the result structure *Res* with a heuristic answer (line 1) by traversing the tree from the root to the node where the query would have been inserted, then returning the nearest neighbor to the query within this node. The query vector is then summarized according to the root's specific parameters, undergoing permutation, segmentation and quantization (lines 2-3). A priority queue is initialized with the root node, where a node has higher priority if it has a smaller lower-bounding distance to the query (lines 4-5) (per Definition 4).

Definition 4 (Quantization-Aware Lower Bound). *The quantization-aware lower bound d_{lb} is defined as:*

$$d_{lb}(\mathcal{N}, \widehat{Q}) = \sqrt{\sum_{s=1}^{|\mathcal{SG}|} (\delta^s_{lb}(\mathcal{N}, \widehat{Q}) - \min(\Delta_s, \delta^s_{lb}(\mathcal{N}, \widehat{Q})))^2} \tag{8}$$

such that:

$$\delta^s_{lb}(\mathcal{N}, \widehat{Q}) = \begin{cases} \widehat{\psi}[s].min - \widehat{Q}[s] & if \; \widehat{Q}[s] < \widehat{\psi}[s].min \\ \widehat{Q}[s] - \widehat{\psi}[s].max & if \; \widehat{Q}[s] \geq \widehat{\psi}[s].max \\ 0 & otherwise \end{cases}$$

where $\widehat{Q}[s]$ is defined (per Equation 6):

$$\widehat{Q}[s] = min(\overline{V}[:,s]) + \Delta_s \cdot \lfloor \frac{\overline{Q}[s] - min(\overline{V}[:,s])}{\Delta_s} \rfloor.$$

and $\overline{Q}[s]$ *(per Equation 1):*

$$\overline{Q}[s] = \frac{1}{r_s - r_{s-1}} \sum_{j=r_{s-1}}^{r_s-1} Q[j]$$

Note that we could have devised a lower bound between the node $\mathcal{N}$ and $\bar{Q}$, but this would lead to an increased overhead in time and space. The quantization gets finer in lower levels of the tree, thus the information loss introduced by quantization gets smaller. At each iteration (line 6), the algorithm dequeues the node with the smallest lower bound (line 7). If the lower bound exceeds the bsf answer, the node is pruned (line 9). Otherwise, if the non-pruned node is a leaf, its *SFile* is processed to eliminate false positives based on the DFT lower-bounding, then the remaining answers are refined using the Euclidean distance, and the bsf answer is updated if a better match has been found (lines 13-14). On the other hand, if the non-pruned node is not a leaf, the query is segmented and quantized according to each child's summarization (lines 17-18), and the child node is added to the priority queue if it cannot be pruned based on its lower bounding distance to the query (lines 20). Once all nodes are either visited or pruned, the nearest neighbor to the query is returned (line 21).

Algorithm 4: KTreeExactSearch

```
   Input: Float* Q, Node* N
   Output: Result* Res    ▷ Result is a struct with fields
          id and dist
1  Res ← HeuristicSearch(N, Q);
2  Q̄ ← SegmentVector(Q, N);
3  Q̂ ← QuantizeVector(Q̄, N) ;
4  Initialize a priority queue PQ ;
5  Insert(PQ, N, d_lb(N, Q̂));
6  while PQ.IsNotEmpty() do
7      ⟨N, lb_dist⟩ ← DeleteMinPQ(PQ);;
8      if lb_dist > Res.dist then
9          break;
10     if N is a leaf then
11         ⟨id, dist⟩ ← ScanInMemSFile(Q, N);;   ▷ Note that
            this step involves first a pruning based on the
            DFT lb, before a refinement using then ED dist
12         if dist < Res.dist then
13             Res.dist ← dist;
14             Res.id ← id;
15     else
16         foreach Child N′ in N do
17             Q̄ ← SegmentVector(Q, N′); ;
18             Q̂ ← QuantizeVector(Q̄, N′); ;
19             if d_lb(N′, Q̂) < Res.dist then
20                 Insert(PQ, N′, d_lb(N′, Q̂));
21 return Res;
```

4 THEORETICAL ANALYSIS

In this section, we present the theoretical foundations of our approach. First, we establish the theoretical framework to prove that: 1) all summarizations exploited by KTree, i.e., segmentation, quantization and DFT, are lower-bounding, and thus guarantee the exactness of the search; and 2) KTree's kernel-based splitting subsumes the splitting strategies of popular methods (e.g., DSTree, M-tree, k-d tree, R-tree). Then, we analyze KTree's complexity both for indexing and query answering, and compare it to the complexity of the state-of-the-art approaches

4.1 Theoretical Framework

4.1.1 Distance Lower Bounds.

Lemma 5 (Lower Bound via Segmentation). *Let* $V[i,:], V[i',:]$ *two vectors in* $\mathbb{V}_{\mathcal{N}}$*, then:*

$$d_{ed}(V[i,:], V[i',:]) \geq d_{lb}(\overline{V}[i,:], \overline{V}[i',:]), \tag{9}$$

Proof.

Let $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$ be two d-dimensional vectors. We define their respective means as $\bar{\mathbf{a}} = \frac{1}{d}\sum_{l=1}^{d} a_l$ and $\bar{\mathbf{b}} = \frac{1}{d}\sum_{l=1}^{d} b_l$. Let $\epsilon_{\text{mean}} = \bar{\mathbf{a}} - \bar{\mathbf{b}}$ denote the difference between their means. Each component a_l can be expressed as its mean plus a deviation: $a_l = \bar{\mathbf{a}} + \tilde{a}_l$, where $\sum_{l=1}^{d} \tilde{a}_l = 0$. Similarly, $b_l = \bar{\mathbf{b}} + \tilde{b}_l$, with $\sum_{l=1}^{d} \tilde{b}_l = 0$.

The difference between corresponding components is then:

$$a_l - b_l = (\bar{\mathbf{a}} + \tilde{a}_l) - (\bar{\mathbf{b}} + \tilde{b}_l) = (\bar{\mathbf{a}} - \bar{\mathbf{b}}) + (\tilde{a}_l - \tilde{b}_l) = \epsilon_{\text{mean}} + (\tilde{a}_l - \tilde{b}_l).$$

Now, we compute the squared Euclidean distance $d_{ed}(\mathbf{a}, \mathbf{b})^2$:

$$\begin{aligned} d_{ed}(\mathbf{a}, \mathbf{b})^2 &= \sum_{l=1}^{d} (a_l - b_l)^2 \\ &= \sum_{l=1}^{d} \left(\epsilon_{\text{mean}} + (\tilde{a}_l - \tilde{b}_l)\right)^2 \\ &= \sum_{l=1}^{d} \left((\epsilon_{\text{mean}})^2 + 2\epsilon_{\text{mean}}(\tilde{a}_l - \tilde{b}_l) + (\tilde{a}_l - \tilde{b}_l)^2\right) \\ &= d(\epsilon_{\text{mean}})^2 + 2\epsilon_{\text{mean}} \sum_{l=1}^{d} (\tilde{a}_l - \tilde{b}_l) + \sum_{l=1}^{d} (\tilde{a}_l - \tilde{b}_l)^2. \end{aligned}$$

The cross-term vanishes, therefore, the squared Euclidean distance simplifies to:

$$d_{ed}(\mathbf{a}, \mathbf{b})^2 = d(\epsilon_{\text{mean}})^2 + \sum_{l=1}^{d} (\tilde{a}_l - \tilde{b}_l)^2.$$

Since $\sum_{l=1}^{d} (\tilde{a}_l - \tilde{b}_l)^2 \geq 0$, we can establish the following lower bound:

$$d_{ed}(\mathbf{a}, \mathbf{b})^2 \geq d(\epsilon_{\text{mean}})^2.$$

Applying the inequality to each segment in $\mathcal{SG}$ of $\mathcal{N}$, then taking the square root of both sides yields the desired result:

$$d_{ed}(V[i,:], V[i',:]) \geq d_{ed}(\overline{V}[i,:], \overline{V}[i',:]),$$

■

Lemma 6 (Lower Bound Preservation under Both Segmentation and Quantization). *For any* $V[i,:], V[i',:] \in \mathbb{V}_{\mathcal{N}}$*,*

$$d_{lb}(\widehat{V}[i,:], \widehat{V}[i',:]) \leq d_{ed}(V[i,:], V[i',:]).$$

Proof.
Let,

$$d_{lb}(\widehat{V}[i,:],\widehat{V}[i',:]) = \sqrt{\sum_{s=1}^{|\mathcal{SG}|}(\delta^s_{lb}(\widehat{V}[i,:],\widehat{V}[i',:]) - \min(\Delta_s, \delta^s_{lb}(\widehat{V}[i,:],\widehat{V}[i',:]))^2} \quad (10)$$

such that:

$$\delta^s_{lb}(\widehat{V}[i,:],\widehat{V}[i',:]) = \left|\widehat{V}[i,:] - \widehat{V}[i',:]\right|$$

We assume, without loss of generality, that $\widehat{V}[i,s] > \widehat{V}[i',s]$.

Let the s-th term in $d_{lb}(\widehat{V}[i,:],\widehat{V}[i',:])^2$ be $T_s = (\widehat{V}[i,s] - \widehat{V}[i',s] - \min(\Delta_s, \widehat{V}[i,s] - \widehat{V}[i',s]))^2$.
Since the quantization process ensures $|\overline{V}[i,s] - \widehat{V}[i,s]| \leq \Delta_s/2$:

$$T_s \leq (\overline{V}[i,s] - \overline{V}[i',s] + \Delta_s - \min(\Delta_s, \widehat{V}[i,s] - \widehat{V}[i',s]))^2$$

Case 1: $\min(\Delta_s, \widehat{V}[i,s] - \widehat{V}[i',s]) = \Delta_s$

$$T_s \leq (\overline{V}[i,s] - \overline{V}[i',s] + \Delta_s - \Delta_s)^2 \leq (V[i,s] - V[i',s])^2$$

Case 2: $\min(\Delta_s, \widehat{V}[i,s] - \widehat{V}[i',s]) = \widehat{V}[i,s] - \widehat{V}[i',s]$

$$T_s = 0 \leq (\overline{V}[i,s] - V[i',s])^2$$

In all cases we have:

$$T_s \leq (\overline{V}[i,s] - \overline{V}[i',s])^2$$

From applying the square root and Lemma 5:

$$d_{lb}(\widehat{V}[i,:],\widehat{V}[i',:]) \leq d_{lb}(\overline{V}[i,:],\overline{V}[i',:]) \leq d_{ed}(V[i,:],V[i',:]).$$

■

Lemma 7 (Lower Bound Preservation for the distance between a query and a quantized node). *For any query vector Q and any node $\mathcal{N}$, the lower-bounding distance between them, denoted $d_{lb}(Q,\mathcal{N})$, is guaranteed to be less than or equal to the Euclidean distance between Q and any $V[i,:] \in V_{\mathcal{N}}$:*

$$\forall Q, \mathcal{N} \quad d_{lb}(Q,\mathcal{N}) \leq d_{ed}(Q, V[i,:])$$

Proof. Let

$$d_{ed}(\mathcal{N},Q) = \sqrt{\sum_{j=1}^{d}(\delta^j_{ed}(\mathcal{N},Q))^2} \quad (11)$$

such that:

$$\delta^j_{ed}(\mathcal{N},Q) = \begin{cases} (\psi[j].min - Q[j]) & \text{if } Q[j] < \psi[j].min \\ (Q[j] - \psi[j].max) & \text{if } Q[j] \geq \psi[j].max \\ 0 & \text{otherwise} \end{cases}$$

From Lemma 6 if $\widehat{V}[i,s] > \widehat{V}[i',s]$

$(\widehat{V}[i,s] - \widehat{V}[i',s] - \min(\Delta_s, \widehat{V}[i,s] - \widehat{V}[i',s]))^2 \leq (V[i,s] - V[i',s])^2$

the same way we could deduce that if $Q[j] < \psi[j].min$

$(\widehat{\psi}[j].min - \widehat{Q}[j] - \min(\Delta_s, \widehat{\psi}[j].min - \widehat{Q}[j]))^2 \leq (\psi[j].min - Q[j])^2$

and if $Q[j] \geq \psi[j].max$

$(\widehat{Q}[j] - \widehat{\psi[j]}.max - \min(\Delta_s, \widehat{Q}[j] - \widehat{\psi[j]}.max)^2 \leq (Q[j] - \psi[j].max)^2$

From Equation 6, we can deduce that the inequality holds

$$\forall Q, \mathcal{N} \quad d_{lb}(Q,\mathcal{N}) \leq d_{ed}(Q,\mathcal{N})$$

Let V[i,:] be one of the vectors represented by $\mathcal{N}$. Since,

$$\begin{cases} \psi[j].min \leq V[i,j] & \forall j < n \\ -\psi[j].max \leq -V[i,j] & \forall j < n \end{cases}$$

$$\text{Then } \forall j < n : \delta^j_{ed}(\mathcal{N},Q) \leq \delta^j_{ed}(V[i,j],Q)$$

$$\text{Which implies, } d_{ed}(\mathcal{N},Q) \leq d_{ed}(V[i,:],Q)$$

$$\text{Thus, } d_{lb}(Q,\mathcal{N}) \leq d_{ed}(V[i,:],Q)$$

■

Lemma 8 (Lower Bound Preservation under DFT Summarization). *Let $V[i,:], V[j,:] \in \mathbb{V}_{\mathcal{N}}$ be two real-valued vectors. Let $\widetilde{V}[i,:], \widetilde{V}[j,:] \in \widetilde{\mathbb{V}}_{\mathcal{N}}$ denote their Discrete Fourier Transforms (DFTs), where the DFT is defined such that Parseval's theorem holds as $\|[i,:]\|_2 = \frac{1}{\sqrt{d}}\|\widetilde{V}[i,:]\|_2$ (L^2 norm). If we summarize each DFT by retaining only the first k coefficients (i.e., the lowest k frequencies), denoted as $\widetilde{V}_k[i,:]$ and $\widetilde{V}_k[i,:]$, then the Euclidean distance between these truncated DFTs provides a lower bound on the original Euclidean distance:*

$$\frac{1}{\sqrt{d}} d_{ed}(\widetilde{V}_k[i,:],\widetilde{V}_k[j,:]) \leq d_{ed}(V[i,:],V[j,:]).$$

Proof.
By Parseval's theorem, the Euclidean distance in the time domain relates to the frequency domain as

$$d_{ed}(V[i,:],V[j,:]) = \frac{1}{\sqrt{d}} d_{ed}(\widetilde{V}[i,:],\widetilde{V}[j,:]).$$

Truncation to the first $k \leq d$ coefficients yields

$$d_{ed}(\widetilde{V}_k[i,:],\widetilde{V}_k[j,:])^2 \leq \sum_{s=0}^{d-1} |\widetilde{V}[i,s] - \widetilde{V}[j,s]|^2 = d_{ed}(\widetilde{V}[i,:],\widetilde{V}[j,:])^2.$$

Taking square roots and scaling by $1/\sqrt{d}$,

$$\frac{1}{\sqrt{d}} d_{ed}(\widetilde{V}_k[i,:],\widetilde{V}_k[j,:]) \leq d_{ed}(V[i,:],V[j,:]).$$

■

4.1.2 Kernel-based Splitting Functions.

Lemma 9 (Splitting subsumption). *Any continuous split can be approximated by a polynomial kernel based split. Furthermore, any polynomial split can be subsumed by a kernel-based split.*

Proof. We establish that various splitting functions commonly used in tree-based indexing structures can be represented or approximated using kernel methods and Principal Component Analysis (PCA). This follows from the observation that many such splitting criteria are polynomial. By the Weierstrass Approximation Theorem below, any continuous function defined on a compact interval can be uniformly approximated to arbitrary precision by a polynomial function. Consequently, any continuous splitting function can be approximated by a polynomial. Since polynomial kernels and PCA are capable of capturing polynomial decision boundaries, they can thus approximate any continuous splitting function.

Theorem 10 (Weierstrass Approximation Theorem). *Let f be a continuous real-valued function on a closed interval $[a, b]$. For every $\varepsilon > 0$, there exists a polynomial $P(x)$ such that*

$$\sup_{x \in [a,b]} |f(x) - P(x)| < \varepsilon$$

.

Note. If the splitting function f is itself a polynomial, then the approximation by a polynomial kernel or PCA-based method is exact, provided the kernel's degree is sufficient to represent f. Many commonly used splitting criteria in tree-based indexing structures indeed utilize polynomial forms. We briefly illustrate this for several prominent tree types:

- **DSTree:** Splits by average ($x = \bar{x}$) are linear polynomials. Splits by standard deviation ($\sigma_s = \sigma'$, ie $\frac{1}{r_s - r_{s-1}} \sum_{i=r_s}^{r_{s-1}-1} (x_i - \mu_x)^2 = \sigma'^2$) define an irregular quadratic shape.
- **M-tree:** Splits are based on distances defining spheres, e.g., $\|x - c\|^2 = r^2$, which is a polynomial equation.
- **k-d tree:** Splits partition data by thresholding a single coordinate, $x_j = t$, a linear polynomial.
- **R-tree:** Splits use bounding rectangles defined by linear inequalities on coordinates, e.g., $a_j \leq x_j \leq b_j$, which are polynomial constraints.

■

Kernel Representation of Polynomial Splits. Polynomial kernels can exactly reproduce these types of splits because they implicitly map data into a higher-dimensional feature space where polynomial decision boundaries correspond to linear separators. For instance, a polynomial kernel of degree p, $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^\top \mathbf{y} + c)^p$, generates all polynomial combinations of input features up to degree p. This allows it to represent hyperplanes, spherical surfaces, and axis-aligned splits as linear boundaries in the transformed space, thereby reproducing the polynomial splits used by DSTree, M-tree, k-d tree, and R-tree. For example, a DSTree split by average $x \leq \bar{x}$ (a linear polynomial) can be represented using a degree-1 polynomial kernel $K(x, y) = xy + c$. The average $\bar{x}$ can be expressed as a weighted sum of kernel evaluations, demonstrating its linearity in the kernel-induced feature space.

4.2 Complexity Analysis

Table 2 summarizes the complexity analysis for the KTree and its tree-based competitors[2]. Each node in our index structure stores a summary of its data (for an example, refer to node $\mathcal{N}_\infty$ in Figure 2). This summary is represented by two synopsis vectors (of dimensionality $|\mathcal{SG}|$, i.e. $2 \times |\mathcal{SG}| \times 4$ bytes), a permutation vector (of dimensionality d, i.e. $d \times 2$ bytes)), a splitting equation (order of $\alpha' + \alpha''$, i.e. at most $(\alpha + \alpha'') \times 4$ bytes) and a segmentation vector (of dimensionality $|\mathcal{SG}|$, i.e. $|\mathcal{SG}| \times 2$ bytes), thus leading to at most $10|\mathcal{SG}| + 2d + 4(\alpha + \alpha'')S$ bytes. The total storage cost per node is therefore dominated by the storage of these vectors and the splitting equation. Since the dimensionality $\mathcal{SG}$ at each node is much smaller than d, the dimensionality of the data, and the number of nodes in the tree is much smaller than the number of data points n, the overall memory required for storing the index structure is negligible compared to the cost of storing the actual data on disk. Although the number of DFT summaries is n, their dimensionality is much smaller than the dimensionality of the raw data. Consequently, the memory complexity of storing the KTree is $O(nd)$, where n is the number of data points and d is the data dimensionality. For the time complexity of building the index, at each level of the tree, the variance of each dimension is computed, which requires $O(nd)$ operations. Subsequently, KPCA is performed using an approximate PCA method, where only the first optimization problem of the SVD is solved in an approximate manner, also bounded by $O(nd)$ [67]. Since this process is repeated for all nodes in a balanced tree, the total time complexity is multiplied by the tree height, which is $O(\log n)$. Therefore, the overall time complexity for building the index is $O(nd \log n)$. As for query answering, KTree evaluates a query against a node by computing a fixed number of lower-bounding distances at each level of the tree. As a result, the time complexity for answering a query is $O(d \log n)$, corresponding to the height of the balanced tree. The k-d Tree, M-tree and R-tree are all balanced, so they preserve the same optimal worst-case space and time complexity [33, 66, 135]. In the case of DSTree and Hercules, because of their typical unbalanced structure, there are no guarantees on the depth of the tree. Thus, the worst-case scenario would be represented by the construction of a linked list structure $O(nd)$.

Complexity Type	DSTree & Hercules	KTree	k-d Tree & M-tree & R-tree
Space (Indexing)	$O(nd)$	$O(nd)$	$O(nd)$
Time (Indexing)	$O(n^2 d)$	$O(ndlog(n))$	$O(ndlog(n))$
Time (Querying)	$O(d.n)$	$O(d.log(n))$	$O(d.log(n))$

Table 2: Worst-case complexity

Note that worst-case complexity presented in Table 2 is the same as the average-case complexity for the KTree, k-d Tree, M-tree and R-tree. However, there are no guarantees on the average case for DSTree and Hercules because the trees are not balanced.

5 EXPERIMENTAL EVALUATION

5.1 Framework

Setup. All methods were compiled using GCC 6.2.0 under Ubuntu Linux 16.04.2 with default compilation flags, and O2. Experiments were conducted on a server equipped with two Intel(R) Xeon(R) Gold 6234 CPUs running at 3.30GHz, 125GB of RAM, and 135GB of swap space. The RAM was limited to 80GB[3]. Storage was provided by a 3.2TB (2 × 1.6TB) SATA2 SSD array configured in RAID0, delivering a measured throughput of 330 MB/sec. This configuration ensures that all methods are evaluated under consistent and demanding I/O conditions, as some datasets are out-of-core.

Algorithms. We evaluate KTree against the most popular tree-based exact similarity search indexes, i.e., optimized single-thread

[2]For a fair comparison, we assume all the methods have a fan-out of 2.

[3]We used GRUB to limit the amount of RAM, so that all methods are forced to use the disk. Note that GRUB prevents the operating system from using the rest of the RAM as a file cache, which is what we wanted for our experiments.

implementations [50, 119] (using the best-performing parameter configurations as reported in [45, 50]) of the DSTree [148], the M-tree [33], the R-tree [66] and the kd-tree [135]. We also compare the KTree against Hercules on time-independent measures as Hercules core novelty lies in its sophisticated parallel algorithms; therefore a time comparison would be unfair. Applying similar parallelization techniques to the KTree and the other baselines is not within the scope of this paper. Besides, we also consider an optimized serial scan and the skip-sequential scan VA+file. All vectors are represented using single-precision values, and algorithms requiring fixed summarizations use 16 dimensions. For KTree, we use polynomial kernels (using polynomials up to degree 4), with α = 10% of the dimensions in the selected segment and $\alpha'' = 50\%$ of the α' dimensions (α' is deduced by the algorithm per §3.3.1.

Datasets. We use four real datasets: p(i) Deep1B [137], which contains 1 billion 96-dimensional image embeddings produced by the GoogLeNet model pretrained on ImageNet, compressed to 96 dimensions via PCA and l2-normalized, with Euclidean distance as the similarity measure; (ii) Text-to-Image-1B [14], a cross-modal retrieval dataset consisting of 1 billion 200-dimensional image embeddings from the Se-ResNext-101 model and considered a challenging dataset event for approximate search [136] ; (iii) ImageNet [91], which contains 1 million 1024-dimensional images and is widely used for benchmarking in computer vision and deep learning; and (iv) GIST [83], which comprises 1 million 960-dimensional GIST descriptors, representing global image features for scene recognition and large-scale image retrieval. The Deep1B (366 GBs) and Text-to-Image1B (763 GBs) are out-of-core.

Queries. Query workloads consist of 100 query vectors executed one after the other, that is, not in batch mode, which is a common scenario in a real setting where queries unknown in advance [64, 65, 117]. For the Deep, GIST and Text-to-Image datasets, queries were randomly sampled from the public query workloads. Since a real workload is not available for ImageNet, we randomly sampled 100 queries from the dataset and excluded them during index building. In addition to these out-of-dataset (ood) queries, we also generate queries of different difficulty for all the datasets, labeled with a percentage (1%-10%). These queries were generated by randomly selecting vectors from the dataset and perturbing them using different levels of Gaussian noise ($\mu = 0$, $\sigma^2 = 0.01$-0.1) [160]. The percentage indicates the value of σ^2. Our experiments cover k-NN queries, where k $\in [1, 100]$ but for space considerations, we report the 1NN results for all datasets, and the 10NN/100NN results for the Text-to-Image dataset as the overall trends remain the same (detailed results can be found in this archive [1]).

Measures. We measure efficiency using the *wall clock time*, the *pruning* (which is a good indicator of the number of sequential accesses in the case of tree-based indexes), the number of *random accesses*, and the tightness of the lower bound (TLB). We measure the space overhead using *footprint*.

$$Pruning = 1 - \frac{\text{\# of Raw Vectors Examined}}{\text{\# of Vectors in Dataset}}$$

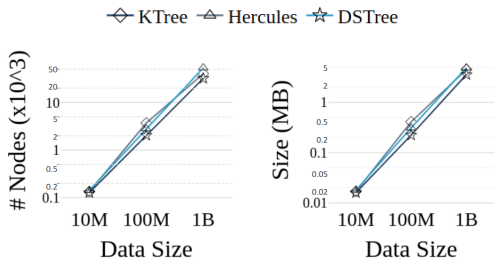
$$TLB = \frac{\text{Lower Bounding Distance}(\widehat{Q}, \mathcal{N})}{\text{Average True Distance}(Q, \mathcal{N})}$$

Procedure. Unless specified otherwise, our experiments are conducted using workloads of 100 exact queries. For experiments simulating workloads of 10,000 queries, we extrapolate the results based on 90 queries (after removing the 5 slowest/fastest queries). The cache is cleared before each query workload and is kept warm otherwise. Some baselines are omitted from some experiments because their index building exceeded 48 hours.

5.2 Results

Parametrization. We fine-tune each index according to best practices established in the literature. For DSTree, M-tree and R-tree, we adopt the parameter settings that have demonstrated optimal performance as reported in [51]. For Hercules, we follow the configuration recommended in [54], which has been shown to yield state-of-the-art results. For KTree, we employ polynomial kernels, focusing on the 10% most variant directions in the data. Across all tree-based indexes, we ensure that whenever summarization techniques are applied, the maximum granularity at the leaf level does not exceed 16 dimensions and a maximum leaf size of 100K.

Footprint. As Figure 3a demonstrates, the Ktree's index has the smallest number of nodes (internal and leaf nodes), among the evaluated tree-based methods. For example, on the 1B dataset, the KTree's index has 32,767 total nodes, 1.23x and 1.63x lower than the closest contenders Hercules and DStree respectively. Notably, KTree achieves at least a 20% reduction in the total number of nodes compared to its competitors (20% compared to Hercules and 25% compared to DSTree). The compactness of the KTree index translates into a smaller memory footprint overall (Fig. 3b). Note that the memory footprint efficiency translates into is also evident in resource usage: Ktree requires 28% less memory. For improved clarity in the plots, results for other methods are omitted. For instance, on the Deep1B dataset, the VA+file incurs a footprint of 301 MB while the M-tree and R-tree require, respectively, 740 MB and 792 MB.



(a) Total Nodes **(b) Memory Size**

Figure 3: Footprint (tree-based indexes)

Time Efficiency. Figure 4 summarizes the combined wall-clock time of building an index and answering 10K 1-NN queries, on the full Deep1B dataset and its 10M and 100M subsets, using the *ood* query workload. KTree wins across the board in query answering, e.g., it is 3.09x and 3.14x faster than the second-best method, which is the VA+file on the 10M dataset and the DSTree on the 1B dataset. KTree also wins over the DSTree in index building despite the use of KPCA since it is applied on a small subset of the dimensions (α'),

requires only solving the first optimization problem to get p_1, and relies on an iterative approximation. However, KTree is slower than the VA+file in index building, but KTree's indexing cost is amortized once the number of queries reaches 1K. Beyond 1K, as in Figure 4, KTree is the fastest method when we consider the combined cost of indexing and query answering. We can observe the same trends on the Text-to-Image dataset (Fig. 5) for the three workloads 1NN, 10NN, and 100NN, with the gap widening between the KTree and its competitors. For instance, the KTree is 5.69xand 3.14x faster than the DSTree, on the 10M and 1B experiments respectively using the 1NN workload. Note that only competitive methods were compared against in the kNN scenarios where $k > 1$.

Random Accesses. To explain why KTree outperforms the other methods in query answering, we measure the number of random accesses performed by each method on the full Deep and Text-to-Image datasets, which are both out-of-core, and ImageNet and GIST, also varying the hardness of the 100-query workload. Before query answering, only the indexes are preloaded into memory, not the raw data. One random access corresponds to a non-pruned leaf in the case of tree-based indexes, and a non-pruned vector summary in the case of the skip-sequential VA+file. Figure 6 summarizes the results. KTree issues 31x fewer random I/Os compared to the VA+file and 10% less compared to the DSTree on the Deep1B (Fig. 6a) dataset. The same trends are observed on the other datasets with the KTree outperforming the competitors by a wider margin. For instance, the KTree incurs 23%, 50% and 60% fewer random I/Os than the DSTree and Hercules (both use the same EAPCA tree) on the Text-to-Image (Fig. 6b), ImageNet (Fig. 6c) and GIST (Fig. 6d) datasets.

Pruning. To further explain the superiority of KTree in query answering, we report (Fig. 7) the pruning on all four datasets and query workloads. Note that pruning is a good indicator of the number of sequential accesses in the case of tree-based indexes, and random accesses in the case of the skip-sequential VA+file. KTree's pruning is up to 2.25x higher than that of the second-best tree-based index Hercules, and up to 2.7x, 2.45x, 4.05x and 3.85x higher than that of the DSTree, M-tree, R-tree and k-d-tree respectively. KTree's pruning is better than that of the VA+file with a small margin, which means they incur a similar number of data accesses. However, most of these accesses are sequential in the case of the KTree, and random in the case of the VA+file, which explains why the KTree was up to an order of magnitude faster than the VA+file in query answering (Figs. 4- 5), e.g., it would take 1 min for the Ktree to perform a 1NN exact query on the Text-to-Image dataset, while the VA+ file would need over 10 mins.

Tightness of the Lower Bound (TLB). The pruning of each method depends heavily on the TLB of its summarization(s) [50, 86]. Figure 8 summarizes the results of the TLB experiment on all four real datasets and query workloads. We can observe that the lower-bounding distance of the KTree is the tightest across the board. On Deep1B (Fig. 8a), it is 3%, 15%, and 41% tighter than that of the VA+file, Hercules, and DSTree respectively. The kd-tree has the lowest TLB across the board, which is 68%-88% lower than the KTree. These results confirm the effectiveness of the KTree's summarization techniques, including its segmentation, quantization and DFT leaf representation. The general trends observed in the TLB are consistent across the evaluated scenarios. Notably, the datasets exhibit the greatest challenge when handling query workloads involving out-of-distribution (OOD) data, whereas performance improves markedly for query workloads characterized by low noise variance (Figs. 8a, 8b, 8c).

Ablation Study. We conduct an ablation study on the KTree to evaluate the individual impact of each key design choice, in particular, we consider the KTree without kernel augmentation (KT-$\mathcal{K}$) where approximate PCA is directly performed on the α' dimensions, the KTree without the approximate PCA (KT-PCA), where the splitting direction is determined by the most variant dimension out of the $\alpha' + \alpha''$ dimensions, and the KTree without the DFT summarization in the leaves (KT-DFT). We evaluate the KTree and its three variants against the best competitor, overall, which is the DSTree. Figure 9a summarizes the combined time of indexing the full Text-to-Image dataset and querying 10K exact 1NN queries. The results indicate that KT-$\mathcal{K}$, KT-PCA, and KT-DFT are respectively 1.45x, 6.22x, and 1.91x times slower than the KTree, which confirms that each design choice contributes to an improvement in query performance, with an acceptable indexing overhead.All KTree variants still outperform the DSTree in query answering, except KT-PCA Justifying that incorporating non-contiguous dimensions and greater flexibility in the split are key factors for enhancing DSTree, while the use of Kernels and DFT proves beneficial only when PCA has already been applied. Figure 9b shows that the dual-leaf summarization contributes significantly to the improvement in pruning but this design combined with kernels and PCA leads to the best pruning. Figure 9c focuses on the leaf-level pruning ratio just before the second filtering with DFT. Again, KT-KT and KT-PCA prune 12.85% and 23.55% less data than the KTree. Finally, Figure 9d evaluates the number of random I/Os issued by each method. KT-K, KT-PCA, and KT-DFT generate 133x, 1.36x, and 1.06x times more random I/Os than the original KTree.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced KTree, a novel tree-based index for exact similarity search, which leverages variance-based dimensionality reduction with kernel approximation techniques to outperform the state-of-the-art baselines. In the future, we plan to develop parallel algorithms for KTree's exact indexing and query answering, and extend it to support δ-ϵ approximate search.

REFERENCES

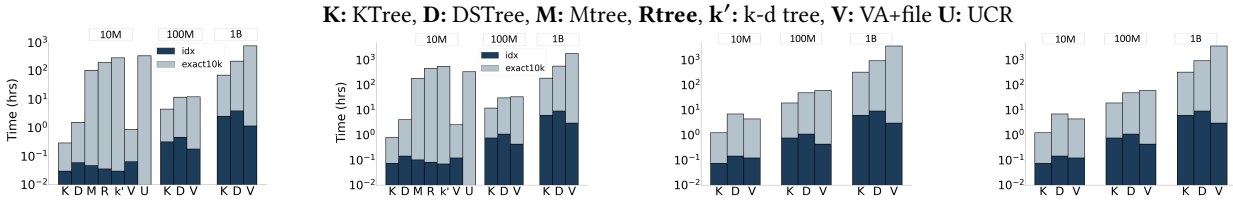
[1] Repository for ktree indexing structures. https://github.com/Ktree0101/Ktree.

[2] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. pages 69–84, 1993.

[3] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 69–84. Springer, 1993.

[4] A. Arora, S. Sinha, P. Kumar, and A. Bhattacharya. HD-index: Pushing the Scalability-accuracy Boundary for Approximate kNN Search in High-dimensional Spaces. *PVLDB*, 11(8), 2018.

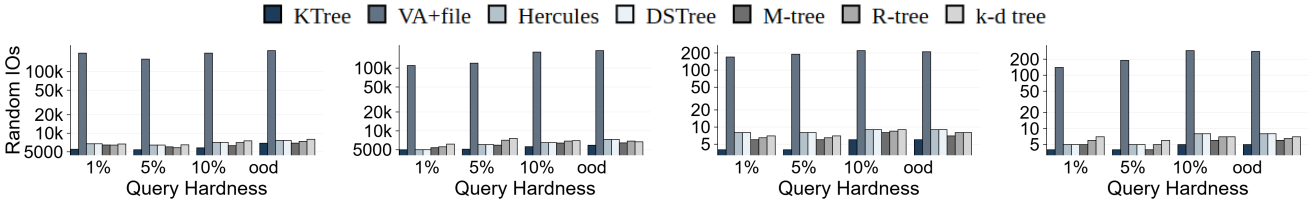
[5] I. Azizi, K. Echihabi, and T. Palpanas. Elpis: Graph-based similarity search for scalable data science. *PVLDB*, 16(6), 2023.



(a) 1NN (b) 10NN (c) 100NN

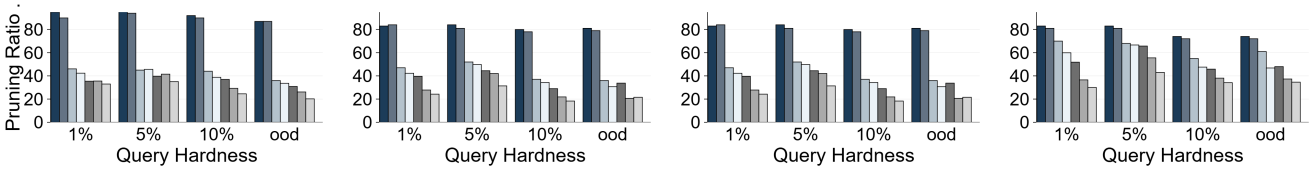
Figure 4: Efficiency (Deep-1B)

Figure 5: Efficiency with increasing k (Text-to-Images-1B)



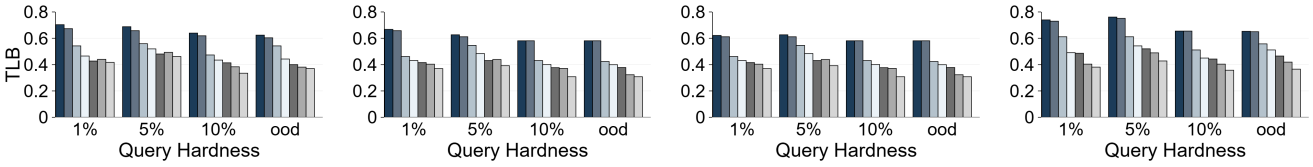
(a) Deep1B (b) Text-to-Image1B (c) ImageNet (d) GIST

Figure 6: Random Accesses



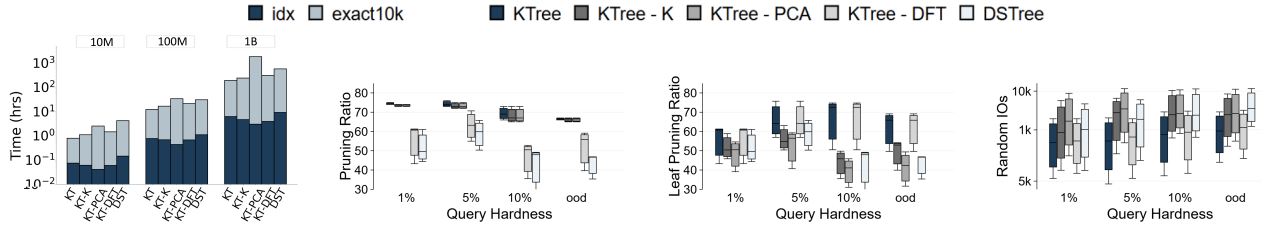
(a) Deep1B (b) Text-to-Image1B (c) ImageNet (d) GIST

Figure 7: Pruning



(a) Deep1B (b) Text-to-Image1B (c) ImageNet (d) GIST

Figure 8: Tightness of the Lower Bound (TLB)



(a) Efficiency (b) Pruning (c) Leaf Pruning (d) Random I/Os

Figure 9: Ablation study (Text-to-Image 1B)

[6] I. Azizi, K. Echihabi, and T. Palpanas. Graph-based vector search: An experimental evaluation of the state-of-the-art. *Proc. ACM Manag. Data*, 3(1), 2025.

[7] J. Aßfalg, H. Kriegel, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Similarity search on time series based on threshold queries. In *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology*, pages 276–294, Munich, Germany, 2006.

[8] A. Babenko and V. Lempitsky. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 931–938. IEEE, 2014.

[9] A. Babenko and V. Lempitsky. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence*, pages 1247–1260, 2014.

[10] A. Babenko and V. Lempitsky. The inverted multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(6):1247–1260, June 2015.

[11] A. Babenko and V. Lempitsky. The Inverted Multi-Index. *TPAMI*, 37(6), 2015.

[12] M. Bach-Andersen, B. Romer-Odgaard, and O. Winther. Flexible non-linear predictive models for large-scale wind turbine diagnostics. *Wind Energy*, 20(5):753–764, 2017.

[13] A. J. Bagnall, R. L. Cole, T. Palpanas, and K. Zoumpatianos. Data series management (dagstuhl seminar 19282). *Dagstuhl Reports*, 9(7):24–39, 2019.

[14] D. Baranchuk and A. Babenko. Benchmarks for billion-scale similarity search. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search, 2021. Webpage. Retrieved March 16, 2023.

[15] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the International Conference on Management of Data (SIGMOD'90)*, pages 322–331. ACM, 1990.

[16] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of IEEE computer society conference on computer vision and pattern recognition*, pages 1000–1006. IEEE, 1997.

[17] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[18] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases (AAAIWS)*, pages 359–370, 1994.

[19] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997 (SEQUENCES '97)*, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society.

[20] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh. iSAX 2.0: Indexing and Mining One Billion Time Series. In *IEEE International Conference on Data Mining (ICDM)*, pages 58–67. IEEE Computer Society, 2010.

[21] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh. iSAX 2.0: Indexing and Mining One Billion Time Series. In *IEEE International Conference on Data Mining (ICDM)*, pages 58–67. IEEE Computer Society, 2010.

[22] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh. Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections With *i*SAX2+. *Knowledge and information systems*, 39(1):123–151, 2014.

[23] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. iSAX2+: Indexing and Mining Very Large Time Series Collections. *Knowl. Inf. Syst.*, 39(1):123–151, 2014.

[24] K. Chakrabarti, E. Keogh, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Transactions on Database Systems (TODS)*, 27(2):188–228, June 2002.

[25] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[26] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing (STOC '02)*, pages 380–388, New York, NY, USA, 2002. ACM.

[27] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, T. Palpanas, S. Athanasiou, and S. Skiadopoulos. Local pair and bundle discovery over co-evolving time series. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases, SSTD*, pages 160–169. ACM, 2019.

[28] Q. Chen, H. Wang, M. Li, G. Ren, S. Li, J. Zhu, J. Li, C. Liu, L. Zhang, and J. Wang. *SPTAG: A library for fast approximate nearest neighbor search*, 2018.

[29] Y. Chen, E. K. Garcia, M. R. Gupta, A. Rahimi, and L. Cazzanti. Similarity-based classification: Concepts and algorithms. *J. Mach. Learn. Res.*, 10:747–776, June 2009.

[30] Y. Chen, M. A. Nascimento, B. C. Ooi, and A. K. H. Tung. Spade: On shape-based pattern detection in streaming time series. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, pages 786–795, Istanbul, Turkey, 2007.

[31] P. Ciaccia and M. Patella. PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In D. B. Lomet and G. Weikum, editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 244–255. IEEE Computer Society, 2000.

[32] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In M. Jarke, M. Carey, K. R. Dittrich, F. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 426–435, Athens, Greece, 1997. Morgan Kaufmann Publishers, Inc.

[33] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In M. Jarke, M. Carey, K. R. Dittrich, F. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*,

pages 426–435, Athens, Greece, Aug. 1997. Morgan Kaufmann Publishers, Inc.

[34] R. Cole, D. E. Shasha, and X. Zhao. Fast window correlations over uncooperative time series. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 743–749, Chicago, Illinois, USA, 2005.

[35] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[36] S. Corlay. A fast nearest neighbor search algorithm based on vector quantization, 2011.

[37] R. R. Curtin, A. G. Gray, and P. Ram. Fast exact max-kernel search, 2012.

[38] R. R. Curtin and P. Ram. Dual-tree fast exact max-kernel search, 2014.

[39] M. Dallachiesa, T. Palpanas, and I. F. Ilyas. Top-k nearest neighbor search in uncertain data series. *PVLDB*, 8(1):13–24, Sept. 2014.

[40] G. Das, D. Gunopulos, and H. Mannila. Finding similar time series. In *Principles of Data Mining and Knowledge Discovery*, pages 88–100, 1997.

[41] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry (SCG '04)*, pages 253–262, New York, NY, USA, 2004. ACM.

[42] M. Dolatshah, A. Hadian, and B. Minaei-Bidgoli. Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces, 2015.

[43] W. Dong. Kgraph, an open source library for k-nn graph construction and nearest neighbor search. http://www.kgraph.org, 2022. Accessed: 2022.

[44] W. Dong, M. Charikar, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, pages 577–586, 2011.

[45] K. Echihabi, P. Fatourou, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Hercules Against Data Series Similarity Search. *PVLDB*, 15(10), 2022.

[46] K. Echihabi, K. Zoumpatianos, and T. Palpanas. Scalable machine learning on high-dimensional vectors: From data series to deep network embeddings. In *WIMS*, 2020.

[47] K. Echihabi, K. Zoumpatianos, and T. Palpanas. High-dimensional similarity search for scalable data science (tutorial). In *ICDE*, 2021.

[48] K. Echihabi, K. Zoumpatianos, and T. Palpanas. New trends in high-d vector similarity search: Ai-driven, progressive, and distributed (tutorial). In *VLDB*, 2021.

[49] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. *PVLDB*, 12(2):112–127, 2018.

[50] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB*, 12(2), 2018.

[51] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB*, 12(2), 2018.

[52] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *PVLDB*, 13(3), 2019.

[53] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Return of the lernaean hydra: Experimental evaluation of data series approximate similarity search. *Proceedings of the VLDB Endowment*, 13(3), 2019.

[54] K. Echihabi, K. Zoumpatianos, T. Palpanas, P. Fatourou, and H. Benbrahim. Hercules: Overcoming the Lernaean Hydra of High-Dimensional Similarity Search. Under Submission.

[55] O. P. Elliott and J. Clark. The impacts of data, ordering, and intrinsic dimensionality on recall in hierarchical navigable small worlds, 2024.

[56] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 419–429, New York, NY, USA, 1994. ACM.

[57] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 202–209, 2000.

[58] C. Fu and D. Cai. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228*, 2016.

[59] C. Fu and D. Cai. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228*, 2016.

[60] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, Jan. 2019.

[61] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, 12(5):461–474, 2019.

[62] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, pages 541–552, New York, NY, USA, 2012. ACM.

[63] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Springer Science & Business Media, 1991.

[64] A. Gogolou, T. Tsandilas, K. Echihabi, T. Palpanas, and A. Bezerianos. Data Series Progressive Similarity Search with Probabilistic Quality Guarantees. In *SIGMOD*, 2020.

[65] A. Gogolou, T. Tsandilas, T. Palpanas, and A. Bezerianos. Progressive Similarity Search on Time Series Data. In *EDBT*, 2019.

[66] A. Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD Record*, 14(2):47–57, 1984.

[67] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[68] J. He, W. Liu, and S.-F. Chang. Scalable similarity search with optimized kernel hashing, 2010.

[69] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware Locality-sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB*, 9(1):1–12, 2015.

[70] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1):1–12, 2015.

[71] P. Huijse, P. A. Estévez, P. Protopapas, J. C. Principe, and P. Zegers. Computational intelligence challenges and applications on large-scale astronomical time series databases. *IEEE Comp. Int. Mag.*, 9(3):27–39, 2014.

[72] G. Hébrail. *Practical Data Mining in a Large Utility Company*. 2000.

[73] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. STOC, 1998.

[74] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)*, pages 604–613, New York, NY, USA, 1998. ACM.

[75] P. Indyk and H. Xu. Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations, 2023.

[76] H. Jegou, M. Douze, and C. Schmid. Product Quantization for Nearest Neighbor Search. *TPAMI*, 33(1), 2011.

[77] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, Jan. 2011.

[78] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 33(1):117–128, 2011.

[79] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, 2011.

[80] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864, May 2011.

[81] K. Jiang, Q. Que, and B. Kulis. Revisiting kernelized locality-sensitive hashing for improved large-scale image retrieval, 2014.

[82] Z. Jin, D. Zhang, Y. Hu, S. Lin, D. Cai, and X. He. Fast and accurate hashing via iterative nearest neighbors expansion. *IEEE transactions on cybernetics*, 44(11):2167–2177, 2014.

[83] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[84] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 500–509, 1994.

[85] S. Kashyap and P. Karras. Scalable kNN search on vertically stored time series. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1334–1342. ACM, 2011.

[86] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 3(3):263–286, 2001.

[87] E. Keogh and M. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *Fourth International Conference on Knowledge Discovery and Data Mining (KDD'98)*, pages 239–241, New York City, NY, 1998. ACM Press.

[88] S. Knieling, J. Niediek, E. Kutter, J. Bostroem, C. Elger, and F. Mormann. An online adaptive screening procedure for selective neuronal responses. *Journal of Neuroscience Methods*, 291:36–42, 2017.

[89] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut: A scalable bottom-up approach for building data series indexes. *Proc. VLDB Endow.*, 11(6):677–690, 2018.

[90] T. Kraska, A. Beutel, E. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[91] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[92] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search, 2009.

[93] P. Laviron, X. Dai, B. Huquet, and T. Palpanas. Electricity demand activation extraction: From known to unknown signatures, using similarity search. In *e-Energy '21: The Twelfth ACM International Conference on Future Energy Systems*, pages 148–159, 2021.

[94] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data: Experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2019.

[95] T. W. Liao. Clustering of time series data: A survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[96] J. Lin, E. J. Keogh, S. Lonardi, and B. Y. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD Workshop*

on Research Issues in Data Mining and Knowledge Discovery (DMKD), pages 2–11, San Diego, California, USA, 2003. ACM.

[97] M. Linardi and T. Palpanas. ULISSE: ULtra compact Index for Variable-Length Similarity SEarch in Data Series. In *ICDE*, 2018.

[98] Y. Linde, A. Buzo, and R. M. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 28(1):84–95, 1980.

[99] T. Liu, A. W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Proceedings of the 17th International Conference on Neural Information Processing Systems (NIPS'04)*, pages 825–832, Cambridge, MA, USA, 2004. MIT Press.

[100] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 950–961. VLDB Endowment, 2007.

[101] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[102] Y. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, pages 824–836, 2018.

[103] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[104] E. P. Marinho and C. M. Andreazza. Anisotropic k-nearest neighbor search using covariance quadtree, 2011.

[105] J. Mcnames. A fast nearest-neighbor algorithm based on a principal axis search tree, 2001.

[106] K. Mirylenka, M. Dallachiesa, and T. Palpanas. Data series similarity using correlation-aware measures. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 11:1–11:12, Chicago, IL, USA, 2017.

[107] R. Motwani, A. Naor, and R. Panigrahy. Lower bounds on locality sensitive hashing. *SIAM Journal on Discrete Mathematics*, 21(4):930–935, 2007.

[108] A. Mueen, Y. Zhu, M. Yeh, K. Kamgar, K. Viswanathan, C. Gupta, and E. Keogh. The fastest similarity search algorithm for time series subsequences under euclidean distance. http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html, 2017. Accessed: August 2017.

[109] A. Mueen, Y. Zhu, M. Yeh, K. Kamgar, K. Viswanathan, C. Gupta, and E. Keogh. The fastest similarity search algorithm for time series subsequences under euclidean distance. http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html, August 2017. Accessed: 2017.

[110] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

[111] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Applications (VISAPP)*, pages 331–340, 2009.

[112] J. V. Munoz, M. A. Gonçalves, Z. Dias, and R. d. S. Torres. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition*, 96:106970, 2019.

[113] J. V. Munoz, M. A. Gonçalves, Z. Dias, and R. d. S. Torres. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition*, 96:106970, 2019.

[114] R. O'Donnell, Y. Wu, and Y. Zhou. Optimal lower bounds for locality-sensitive hashing (except when q is tiny). *ACM Transactions on Computation Theory*, 6(1):5:1–5:13, Mar. 2014.

[115] T. Palpanas. Data series management: The road to big sequence analytics. *ACM SIGMOD Record*, 44(2):47–52, 2015.

[116] T. Palpanas and V. Beckmann. Report on the first and second interdisciplinary time series analysis workshop (itisa). *SIGMOD Rec.*, 48(3), 2019.

[117] T. Palpanas and V. Beckmann. Report on the First and Second Interdisciplinary Time Series Analysis Workshop (ITISA). *ACM SIGMOD Record*, 48(3), 2019.

[118] P. Paraskevopoulos, T.-C. Dinh, Z. Dashdorj, T. Palpanas, and L. Serafini. Identification and characterization of human behavior patterns from mobile phone data. In *D4D Challenge session, NetMob*, 2013.

[119] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12:2825–2830, 2011.

[120] B. Peng, P. Fatourou, and T. Palpanas. Messi: In-memory data series indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 337–348. IEEE, 2020.

[121] B. Peng, P. Fatourou, and T. Palpanas. Paris+: Data series indexing on multi-core architectures. *IEEE Transactions on Knowledge and Data Engineering*, 33(5):2151–2164, 2020.

[122] D. Rafiei. On similarity-based queries for time series data. In *Proceedings of the 15th International Conference on Data Engineering*, pages 410–417, Sydney, Australia, 1999.

[123] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. *SIGMOD Record*, 26(2):13–25, June 1997.

[124] D. Rafiei and A. O. Mendelzon. Efficient retrieval of similar time sequences using dft. *CoRR*, cs.DB/9809033, 1998.

[125] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In Q. Yang, D. Agarwal, and J. Pei, editors, *KDD*, pages 262–270. ACM, 2012.

[126] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM*

SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 262–270. ACM, 2012.

[127] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In Q. Yang, D. Agarwal, and J. Pei, editors, *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 262–270. ACM, 2012.

[128] T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans. Time series epenthesis: Clustering time series streams requires ignoring some data. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 547–556. IEEE, 2011.

[129] P. Ram and A. G. Gray. Maximum inner-product search using cone trees, 2012.

[130] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco. Practical data prediction for real-world wireless sensor networks. *IEEE Trans. Knowl. Data Eng.*, 27(8):2206–2219, 2015.

[131] P. P. Rodrigues, J. Gama, and J. P. Pedroso. Odac: Hierarchical clustering of time series data streams. In J. Ghosh, D. Lambert, D. B. Skillicorn, and J. Srivastava, editors, *SDM*, pages 499–503. SIAM, 2006.

[132] B. Schölkopf, A. Smola, and K.-R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[133] D. Shasha. Tuning time series queries in finance: Case studies and recommendations. *IEEE Data Eng. Bull.*, 22(2):40–46, 1999.

[134] J. Shieh and E. Keogh. iSAX: Indexing and Mining Terabyte Sized Time Series. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 623–631, New York, NY, USA, 2008. ACM.

[135] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[136] H. V. Simhadri, G. Williams, M. Aumuller, M. Douze, A. Babenko, D. Baranchuk, Q. Chen, L. Hosseini, R. Krishnaswamy, G. Srinivasa, S. J. Subramanya, and J. Wang. Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search. *CoRR*, abs/2205.03763, 2022.

[137] Skoltech Computer Vision. Deep billion-scale indexing. http://sites.skoltech.ru/compvision/noimi, 2018.

[138] S. Soldi, V. Beckmann, W. Baumgartner, G. Ponti, C. R. Shrader, P. Lubiński, H. Krimm, F. Mattana, and J. Tueller. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 563:A57, 2014.

[139] S. J. Subramanya, R. Kadekodi, R. Krishaswamy, and H. V. Simhadri. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 13766–13776, 2019.

[140] S. J. Subramanya, R. Kadekodi, R. Krishaswamy, and H. V. Simhadri. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 13766–13776, 2019.

[141] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS: Solving c-approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB*, 8(1), 2014.

[142] K. R. Varshney and H. Alemzadeh. On the safety of machine learning: Cyber-physical systems, decision sciences, and data products. *Big data*, 5(3):246–255, 2017.

[143] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-nn graph construction for visual descriptors. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1106–1113. IEEE, 2012.

[144] M. Wang, X. Xu, Q. Yue, and Y. Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.*, 14(11):1964–1978, jul 2021.

[145] M. Wang, X. Xu, Q. Yue, and Y. Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 14(11):1964–1978, July 2021.

[146] Q. Wang and T. Palpanas. Deep Learning Embeddings for Data Series Similarity Search. In *SIGKDD*, 2021.

[147] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery*, 26(2):275–309, March 2013.

[148] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *Proceedings of the VLDB Endowment*, 6(10):793–804, 2013.

[149] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *PVLDB*, 6(10):793–804, 2013.

[150] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *PVLDB*, 6(10):793–804, 2013.

[151] R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proc. VLDB*, pages 194–205, 1998.

[152] B. M. Williams and L. A. Hoel. Modeling and forecasting vehicular traffic flow as a seasonal arima process: Theoretical basis and empirical results. *Journal of Transportation Engineering*, 129(6):664–672, 2003.

[153] H. Xia, P. Wu, S. Hoi, and R. Jin. Boosting multi-kernel locality-sensitive hashing for scalable image retrieval, 2012.

[154] Y. Xia, K. He, F. Wen, and J. Sun. Joint Inverted Indexing. *ICCV*, 2013.

[155] Y. Xia, K. He, F. Wen, and J. Sun. Joint inverted indexing. In *2013 IEEE International Conference on Computer Vision (ICCV)*, pages 3416–3423, 2013.

[156] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. DPiSAX: Massively Distributed Partitioned iSAX. In *ICDM*, 2017.

[157] D.-E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. DPiSAX: Massively Distributed Partitioned iSAX. *CoRR*, abs/1702.03204, 2017.

[158] H. Yang, X. Bai, J. Zhou, P. Ren, Z. Zhang, and J. Cheng. Adaptive object retrieval with kernel reconstructive hashing, 2014.

[159] K. Zoumpatianos, S. Idreos, and T. Palpanas. ADS: The Adaptive Data Series Index. *VLDBJ*, 25(6):843–866, 2016.

[160] K. Zoumpatianos, Y. Lou, I. Ileana, T. Palpanas, and J. Gehrke. Generating data series query workloads. *VLDBJ*, 27(6), 2018.

[161] K. Zoumpatianos and T. Palpanas. Data series management: Fulfilling the need for big sequence analytics. In *ICDE*, 2018.