DET-LSH: A Locality-Sensitive Hashing Scheme with Dynamic Encoding Tree for Approximate Nearest Neighbor Search

Jiuqi Wei
Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
weijiuqi19z@ict.ac.cn

Botao Peng
Institute of Computing Technology, Chinese Academy of Sciences
pengbotao@ict.ac.cn

Xiaodong Lee
Institute of Computing Technology, Chinese Academy of Sciences
xl@ict.ac.cn

Themis Palpanas
LIPADE, Université Paris Cité
themis@mi.parisdescartes.fr

ABSTRACT

Locality-sensitive hashing (LSH) is a well-known solution for approximate nearest neighbor (ANN) search in high-dimensional spaces due to its robust theoretical guarantee on query accuracy. Traditional LSH-based methods mainly focus on improving the efficiency and accuracy of the query phase by designing different query strategies, but pay little attention to improving the efficiency of the indexing phase. They typically fine-tune existing data-oriented partitioning trees to index data points and support their query strategies. However, their strategy to directly partition the multi-dimensional space is time-consuming, and performance degrades as the space dimensionality increases. In this paper, we design an encoding-based tree called Dynamic Encoding Tree (DE-Tree) to improve the indexing efficiency and support efficient range queries based on Euclidean distance. Based on DE-Tree, we propose a novel LSH scheme called DET-LSH. DET-LSH adopts a novel query strategy, which performs range queries in multiple independent index DE-Trees to reduce the probability of missing exact NN points, thereby improving the query accuracy. Our theoretical studies show that DET-LSH enjoys probabilistic guarantees on query accuracy. Extensive experiments on real-world datasets demonstrate the superiority of DET-LSH over the state-of-the-art LSH-based methods on both efficiency and accuracy. While achieving better query accuracy than competitors, DET-LSH achieves up to 6x speedup in indexing time and 2x speedup in query time over the state-of-the-art LSH-based methods.

PVLDB Reference Format:
Jiuqi Wei, Botao Peng, Xiaodong Lee, and Themis Palpanas. DET-LSH: A Locality-Sensitive Hashing Scheme with Dynamic Encoding Tree for Approximate Nearest Neighbor Search. PVLDB, 17(9): 2241 - 2254, 2024.
doi:10.14778/3665844.3665854

* Botao Peng and Xiaodong Lee are the corresponding authors.

Proceedings of the VLDB Endowment, Vol. 17, No. 9 ISSN 2150-8097.
doi:10.14778/3665844.3665854

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/WeiJiuQi/DET-LSH.

1 INTRODUCTION

Background and Problem. Nearest neighbor (NN) search in high-dimensional Euclidean spaces is a fundamental problem in various fields, such as database [18], information retrieval [28], data mining [52], and machine learning [3]. Given a dataset $\mathcal{D}$ of n data points in d-dimensional space $\mathbb{R}^d$ and a query q, an NN query returns a point $o^* \in \mathcal{D}$ which has the minimum Euclidean distance to q among all points in $\mathcal{D}$. However, NN search in high-dimensional datasets is challenging due to the 'curse of dimensionality' phenomenon [8, 25, 61]. In practice, Approximate Nearest Neighbor (ANN) search is often used as an alternative, sacrificing some query accuracy to achieve a huge improvement in efficiency [20, 32, 53, 54, 59]. Given an approximation ratio c and a query $q \in \mathbb{R}^d$, a c-ANN query returns a point o whose distance to q is at most c times the distance between q and its exact NN o^*, i.e., $\|q, o\| \leq c \cdot \|q, o^*\|$.

Prior Work. Locality-sensitive hashing (LSH)-based methods are known for their robust theoretical guarantees on the accuracy of query results, making them popular in high-dimensional c-ANN search [1, 2, 22, 26, 31, 34, 36, 37, 51, 55, 66, 67]. At the core of LSH-based methods is a family of LSH functions to map points from the original high-dimensional space to low-dimensional projected spaces, and then construct indexes to efficiently support queries, thus reducing the complexity of indexing and querying. Thanks to the properties of LSH, points that are close in the original space are more likely to be close in the projected space than those far away [23]. Therefore, high-quality results can be obtained by only checking the points around the query point in the projected spaces [14]. Based on the query strategies, we classify the mainstream LSH-based methods into three categories: 1) boundary constraint (BC) based methods [1, 35, 53, 55]; 2) collision counting (C2) based methods [22, 26, 31, 36, 37]; and 3) distance metric (DM) based methods [51, 66]. BC methods map all data points to L independent K-dimensional projected spaces, and each projected point is assigned to a hash bucket whose boundary is constrained by a K-dimensional hypercube. Among L hash tables, two points can be considered *colliding* as long as they are assigned to the same

hash bucket at least once. Compared with BC methods, which require simultaneous collisions in K dimensions, C2 methods relax the collision condition. C2 methods select candidate points whose number of collisions with the query point is greater than a predefined threshold. In DM methods, the distance between two points in the projected space can be used to estimate their distance in the original space with theoretical guarantees. Therefore, DM methods select candidate points by conducting range queries based on the Euclidean distance metric in the projected space.

Limitations and Motivation. Efficiency and accuracy are key metrics to evaluate the performance of LSH-based methods in c-ANN search. Nowadays, new data is produced at an ever-increasing rate, and the size of datasets is continuously growing [19, 39, 40, 62]. We need to manage large-scale data more efficiently to support further data analysis [16, 46, 47, 63]. However, existing LSH-based methods mainly focus on reducing query time and improving query accuracy by designing different query strategies, but pay little attention to reducing indexing time [26, 36, 37, 51, 55, 66]. They typically fine-tune existing data-oriented partitioning trees to index data points and support their query strategies, such as R*-Tree [6] for DB-LSH [55], PM-Tree [50] for PM-LSH [66], and R-Tree [24] for SRS [51]. Data-oriented partitioning trees [6, 12, 24, 50] group nearby data points and partition them into their minimum bounding objects (e.g., hyperrectangle, hypersphere) hierarchically. However, partitioning directly in a multi-dimensional space is time-consuming, which limits the efficiency of these methods in the indexing phase. In addition, the performance of data-oriented partitioning trees decreases as the space dimensionality increases [7, 61], which limits the dimensionality of the projected space. Therefore, it is necessary to design a more efficient tree structure to address the limitations. From another perspective, a more efficient tree structure can also help improve query accuracy, since more trees can be constructed in the same indexing time, and query answering based on more trees can be more accurate. For example, the state-of-the-art method among BC methods, DB-LSH [55], constructs five R*-Trees to reduce the probability of missing exact NN points in the query phase.

Our Method. In this paper, we propose a novel tree structure called Dynamic Encoding Tree (DE-Tree) and a novel LSH scheme called DET-LSH to solve the high-dimensional c-ANN search problem more efficiently and accurately. First, we present an encoding-based tree called DE-Tree, which divides and encodes each dimension of the projected space independently (as shown in Figure 1), avoiding to directly partition the multi-dimensional projected space like data-oriented partitioning trees do. This idea leads to improved indexing efficiency. DE-Tree dynamically encodes projected points based on the dataset's distribution, so that nearby points have more similar encoding representations than distant ones, thereby improving query accuracy. DE-Tree supports efficient range queries because the upper and lower bound distances between a query point and any DE-Tree node can be easily calculated. Second, we propose a novel LSH scheme called DET-LSH. DET-LSH dynamically encodes K-dimensional projected points and then constructs L DE-Trees based on the encoded points. We design a two-step query strategy for DET-LSH, which combines the ideas of BC and DM methods. The first step is to perform range queries in DE-Trees and identify in a coarse-grained way a certain proportion of candidate points that are close to the query point. The second step is to calculate the actual distance of each candidate point from the query point in a fine-grained way, then sort the distances and return the final result. Intuitively, the coarse-grained filtering improves the query efficiency, and the fine-grained calculation improves query accuracy. Third, we conduct a rigorous theoretical analysis showing that DET-LSH can correctly answer a c^2-k-ANN query with a constant probability. Furthermore, extensive experiments demonstrate that DET-LSH outperforms existing LSH-based methods in both efficiency and accuracy.

Our main contributions are summarized as follows.

- We present a novel encoding-based tree structure called DE-Tree. Compared with data-oriented partitioning trees used in existing LSH-based methods, DE-Tree has better indexing efficiency and can support more efficient range queries based on the Euclidean distance metric.
- We propose DET-LSH, a novel LSH scheme based on DE-Tree. We design a novel query strategy for DET-LSH, taking into account both efficiency and accuracy. We provide a theoretical analysis showing that DET-LSH answers a c^2-k-ANN query with a constant success probability.
- We conduct extensive experiments, demonstrating that DET-LSH can achieve better efficiency and accuracy than existing LSH-based methods. While achieving better query accuracy than competitors, DET-LSH achieves up to 6x speedup in indexing time and 2x speedup in query time over the state-of-the-art LSH-based methods.

2 RELATED WORK

2.1 Mainstream LSH-based Methods

Boundary Constraint based methods (BC). BC requires $K \cdot L$ hash functions to map all data points to L independent K-dimensional projected spaces. Each projected point is assigned to a hash bucket whose boundary is constrained by a K-dimensional hypercube. Among L hash tables, two points can be considered colliding as long as they are assigned to the same hash bucket at least once. E2LSH [1] is the original BC method that adopts LSH functions following the p-stable distribution [14]. E2LSH needs to continuously generate new hash tables when the search radius r gradually increases, which leads to prohibitively large space consumption in indexing. To alleviate this issue, LSB-Forest [53] adopts B-Tree [5] to index projected points, avoiding building hash tables at different radii. SK-LSH [35] proposes a novel index structure based on B^+-Tree [5], and the search strategy supports it finding better candidates with lower I/O cost. However, neither LSB-Forest nor SK-LSH ensures any LSH-like theoretical guarantees since they are based on heuristics. DB-LSH [55] is the state-of-the-art BC method with strict theoretical guarantees, which presents a dynamic search framework based on R*-Tree [6].

Collision Counting based methods (C2). C2 requires $K' \cdot L'$ hash functions to construct L' independent K'-dimensional hash tables, where $K' < K$ and $L' > L$. C2 selects candidate points whose number of collisions is greater than a threshold t, where $t < L'$. C2LSH [22] proposes the C2 scheme and only maintain K' one-dimensional hash tables ($K' = 1$). C2LSH adopts the *virtual rehashing* technique to count collisions dynamically, reducing index space consumption. QALSH [26] improves C2LSH by using B^+-Trees to locate points projected into the same bucket, avoiding

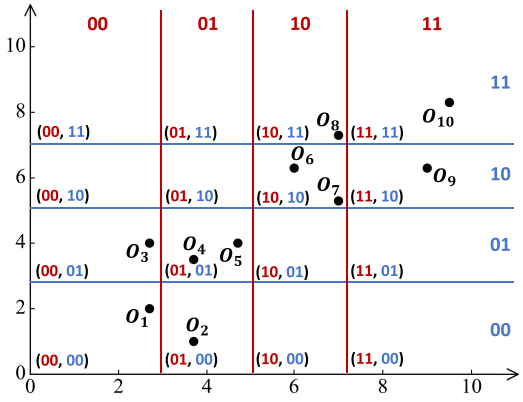
counting the collision numbers among a large number of points dimension by dimension. To further reduce the space consumption of QALSH, R2LSH [36] and VHP [37] are proposed. R2LSH maps data points into multiple two-dimensional projected spaces ($K^{'} = 2$) and VHP considers hash buckets as virtual hypersphere ($K^{'} > 2$). LCCS-LSH [31] proposes a novel search framework, which extends C2's method of counting collisions from the number of discrete points to the length of continuous co-substrings.

Distance Metric based methods (DM). The intuition of DM is that the points close to query q in the original space are also close to query q in the projected space. DM requires K hash functions to map data points into a K-dimensional projected space. SRS [51] utilizes R-tree to index projected points and performs exact NN search in the K-dimensional projected space. PM-LSH [66] designs a range query mechanism based on PM-Tree [50] to improve query efficiency. According to Euclidean distances between queries and points in the projected space, $\beta n + k$ candidates will be selected in PM-LSH, where β is an estimated ratio to guarantee the ANN search performance and n is the dataset cardinality.

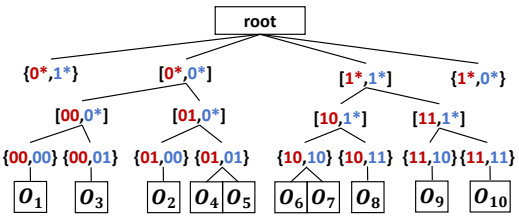
2.2 Tree Structure

Data-oriented Partitioning Tree. As mentioned above, mainstream LSH-based methods adopt data-oriented partitioning trees, such as B-Tree [5], R-Tree [24], M-Tree [12], and their variants [6, 50], to construct indexes and support queries. In these methods, data-oriented partitioning trees group nearby data points and hierarchically partition them into their minimum bounding graphics (e.g., hyperrectangle, hypersphere). For example, R-Tree and M-Tree partition data points into hyperrectangular and hyperspherical partitions, respectively. However, for LSH-based methods, partitioning multi-dimensional projected spaces consumes much time. In addition, with the increase of space dimensionality, the effectiveness of data-oriented partitioning trees decreases [7, 61], which is also the reason why tree-based methods [10, 13, 49, 64] cannot efficiently support ANN search in high-dimensional spaces.

Encoding-based Tree. Encoding-based trees play an important role in data series similarity search [9, 11, 15, 17, 30, 33, 41–45, 57, 58, 60, 69]. Unlike data-oriented partitioning trees, which index a data point directly based on their multi-dimensional coordinates, encoding-based trees independently encode the coordinates of each dimension of the data point into symbolic representations. The indexable Symbolic Aggregate approXimation (iSAX) [48] is a widely used symbolic representation. iSAX divides each dimension into non-uniformly distributed regions and assigns a bit-wise symbol to each region. Figure 1(a) illustrates the encoding process for iSAX representations, and Figure 1(b) illustrates an iSAX index based on the representations. In practice, iSAX requires only 256 symbols for a very good approximation, so the maximum alphabet cardinality can be represented by 8 bits [9]. Based on the iSAX representation, several encoding-based trees with different indexing and query strategies are proposed to support data series similarity search [9, 11, 17, 42, 43, 45, 57, 60, 69]. The advantages of encoding-based trees can be transferred to LSH-based methods for ANN search. Specifically, encoding-based trees divide and encode each dimension of the space independently, avoiding partitioning multi-dimensional projected spaces, improving indexing efficiency. In addition, the



(a) Encode data points into iSAX representations.



(b) An index based on the iSAX representations.

Figure 1: Illustration of an encoding-based tree.

upper and lower bound distances between two points can be calculated easily using their region boundaries, which is suitable for range queries in LSH-based methods, improving query efficiency.

3 PRELIMINARIES

3.1 Problem Definition

Let $\mathcal{D}$ be a dataset of points in d-dimensional space $\mathbb{R}^d$. The dataset cardinality is denoted as $|\mathcal{D}| = n$, and let $\|o_1, o_2\|$ denote the distance between points $o_1, o_2 \in \mathcal{D}$. The query point $q \in \mathbb{R}^d$.

Definition 1 (c-ANN). *Given a query point q and an approximation ratio $c > 1$, let o^* be the exact nearest neighbor of q in $\mathcal{D}$. A c-ANN query returns a point $o \in \mathcal{D}$ satisfying $\|q, o\| \leq c \cdot \|q, o^*\|$.*

The c-ANN query can be generalized to c-k-ANN query that returns k approximate nearest points, where k is a positive integer.

Definition 2 (c-k-ANN). *Given a query point q, an approximation ratio $c > 1$, and an integer k. Let o_i^* be the i-th exact nearest neighbor of q in $\mathcal{D}$. A c-k-ANN query returns k points $o_1, o_2, ..., o_k$. For each $o_i \in D$ satisfying $\|q, o_i\| \leq c \cdot \left\|q, o_i^*\right\|$, where $i \in [1, k]$.*

In fact, LSH-based methods do not solve c-ANN queries directly because o^* and $\|q, o^*\|$ is not known in advance [31, 55, 66]. Instead, they solve the problem of (r,c)-ANN proposed in [27].

Definition 3 ((r,c)-ANN). *Given a query point q, an approximation ratio $c > 1$, and a search radius r. An (r,c)-ANN query returns the following result:*

Table 1: Notations

Notation	Description
$\mathbb{R}^d$	d-dimensional Euclidean space
$\mathcal{D}, n$	Dataset of points in $\mathbb{R}^d$ and its cardinality $\lvert\mathcal{D}\rvert$
o, q	A data point in $\mathcal{D}$ and a query point in $\mathbb{R}^d$
o', q'	o and q in the projected space
o^*, o_i^*	The first and i-th nearest point in $\mathcal{D}$ to q
$\lVert o_1, o_2\rVert$	The Euclidean distance between o_1 and o_2
s, s'	Abbreviation for $\lVert o_1, o_2\rVert$ and $\lVert o_1', o_2'\rVert$
$h(o)$	Hash function
$\mathcal{H}(o)$	$[h_1(o), ..., h_K(o)]$, the coordinates of o'
$\mathcal{H}_i(o)$	Coordinates of o' in the i-th project space
c	Approximation ratio
r, r_{min}	Search radius and the initial search radius
d, K	Dimension of the original and the projected space
L	Number of independent projected spaces

(1) *If there exists a point $o \in \mathcal{D}$ such that $\lVert q, o\rVert \le r$, then return a point $o' \in \mathcal{D}$ such that $\lVert q, o'\rVert \le c \cdot r$;*
(2) *If for all $o \in \mathcal{D}$ we have $\lVert q, o\rVert > c \cdot r$, then return nothing;*
(3) *If for the point o closest to q we have $r < \lVert q, o\rVert \le c \cdot r$, then return o or nothing.*

The c-ANN query can be transformed into a series of (r,c)-ANN queries with increasing radii until a point is returned. The search radius r is continuously enlarged by multiplying c, i.e., $r = r_{min}, r_{min} \cdot c, r_{min} \cdot c^2, ...$, where r_{min} is the initial search radius. In this way, as proven by [27], the ANN query can be answered with an approximation ratio c^2, i.e., c^2-ANN.

3.2 Locality-Sensitive Hashing

The capability of an LSH function h is to project closer data points into the same hash bucket with a higher probability. Formally, the definition of LSH used in Euclidean space is given below [55, 66]:

Definition 4 (LSH). *Given a distance r, an approximation ratio $c > 1$, a family of hash functions $\mathcal{H} = \{h : \mathbb{R}^d \to \mathbb{R}\}$ is called (r,cr,p_1,p_2)-locality-sensitive, if for $\forall o_1, o_2 \in \mathbb{R}^d$, it satisfies both of the following conditions:*

(1) *If $\lVert o_1, o_2\rVert \le r$, $\Pr[h(o_1) = h(o_2)] \ge p_1$;*
(2) *If $\lVert o_1, o_2\rVert > cr$, $\Pr[h(o_1) = h(o_2)] \le p_2$,*

where $h \in \mathcal{H}$ is randomly chosen, and the probability values p_1 and p_2 satisfy $p_1 > p_2$.

A widely adopted LSH family for the Euclidean space is defined as follows [26]:

$$h(o) = \vec{a} \cdot \vec{o}, \tag{1}$$

where $\vec{o}$ is the vector representation of a point $o \in \mathbb{R}^d$ and $\vec{a}$ is a d-dimensional vector where each entry is independently chosen from the standard normal distribution $\mathcal{N}(0, 1)$.

3.3 p-Stable Distribution and χ^2 Distribution

A distribution $\mathcal{T}$ is called p-stable, if for any u real numbers $v_1, ..., v_u$ and identically distributed (i.i.d.) variables $X_1, ..., X_u$ following $\mathcal{T}$ distribution, $\sum_{i=1}^{u} v_i X_i$ has the same distribution as $(\sum_{i=1}^{u} |v_i|^p)^{1/p} \cdot X$, where X is a random variable with distribution $\mathcal{T}$ [14]. p-stable distribution exists for any $p \in (0, 2]$ [68], and $\mathcal{T}$ is the normal distribution when $p = 2$.

Let $o' = \mathcal{H}(o) = [h_1(o), ..., h_K(o)]$ denote the point o in the K-dimensional projected space. For any two points $o_1, o_2 \in \mathcal{D}$, let $s = \lVert o_1, o_2\rVert$ and $s' = \lVert o_1', o_2'\rVert$ denote the Euclidean distances between o_1 and o_2 in the original space and in the projected space.

Lemma 1. *$\frac{s'^2}{s^2}$ follows the $\chi^2(K)$ distribution.*

Proof. Let $h' = h(o_1) - h(o_2) = \vec{a} \cdot (\vec{o_1} - \vec{o_2}) = \sum_{i=1}^{d} (o_1[i] - o_2[i]) \cdot a[i]$, where $a[i]$ follows the $\mathcal{N}(0, 1)$ distribution. Since 2-stable distribution is the normal distribution, h' has the same distribution as $(\sum_{i=1}^{d} (o_1[i] - o_2[i])^2)^{1/2} \cdot X = s \cdot X$, where X is a random variable with distribution $\mathcal{N}(0, 1)$. Therefore $\frac{h'}{s}$ follows the $\mathcal{N}(0, 1)$ distribution. Given K hash functions $h_1(\cdot), ..., h_K(\cdot)$, we have $\frac{h_1'^2 + ... + h_K'^2}{s^2} = \frac{s'^2}{s^2}$, which has the same distribution as $\sum_{i=1}^{K} X_i^2$. Thus, $\frac{s'^2}{s^2}$ follows the $\chi^2(K)$ distribution. □

Lemma 2. *Given s and s' we have:*

$$\Pr[s' > s\sqrt{\chi_\alpha^2(K)}] = \alpha, \tag{2}$$

where $\chi_\alpha^2(K)$ is the upper quantile of a distribution $Y \sim \chi^2(K)$, i.e., $\Pr[Y > \chi_\alpha^2(K)] = \alpha$.

Proof. From Lemma 1, we have $\frac{s'^2}{s^2} \sim \chi^2(K)$. Since $\chi_\alpha^2(K)$ is the α upper quantiles of $\chi^2(K)$ distribution, we have $\Pr[\frac{s'^2}{s^2} > \chi_\alpha^2(K)] = \alpha$. Transform the formulas, we have $\Pr[s' > s\sqrt{\chi_\alpha^2(K)}] = \alpha$. □

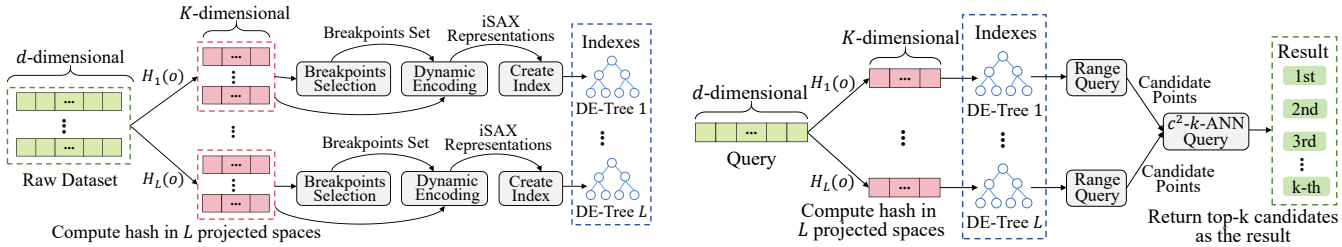
4 THE DET-LSH METHOD

In this section, we present the details of DET-LSH and the design of Dynamic Encoding Tree (DE-Tree). DET-LSH consists of three phases: an encoding phase to encode the LSH-based projected points into iSAX representations; an indexing phase to construct DE-Trees based on the iSAX representations; a query phase to perform range queries in DE-Trees for ANN search. Figure 2 provides a high-level overview of the workflow for DET-LSH.

4.1 Encoding Phase

DET-LSH first encodes projected points into iSAX representations. iSAX uses *breakpoints* to divide each dimension into non-uniform regions, and assigns a bit-wise symbol to each region. For example, Figure 1(a) illustrates an iSAX-based encoding process under a two-dimensional space. In Figure 1(a), we use three breakpoints in each dimension to divide it into four regions, each of which can be represented by a 2-bit symbol: 00/01/10/11. Therefore, the space is divided into 16 regions, and the points in the same region have the same iSAX representations. Figure 1(b) shows an index based on the iSAX representations. In practice, iSAX only requires 256 symbols in each dimension to get a very good approximation [9], which means each dimension can be encoded with an 8-bit alphabet.

Static encoding scheme. In data series similarity search, traditional iSAX-based methods adopt the static encoding scheme [9, 11, 17, 42, 43, 45, 69]. Since normalized data series have highly Gaussian distribution [48], they simply determine the breakpoints $b_1, ..., b_{a-1}$ such that the area under a $\mathcal{N}(0, 1)$ Gaussian curve from



(a) Encoding phase and indexing phase.

(b) Query phase.

Figure 2: Overview of the DET-LSH workflow.

Algorithm 1: Breakpoints Selection

Input: Parameters K, L, n, all points in projected spaces P, sample size n_s, number of regions in each projected space N_r
Output: A set of breakpoints B

1 Initialize B with size $L \cdot K \cdot (N_r + 1)$;
2 **for** $i = 1$ *to* L **do**
3 **for** $j = 1$ *to* K **do**
4 Sample $C_{ij} = [h_{ij}(o_1), ..., h_{ij}(o_{n_s})]$ from P;
5 $round \leftarrow \log_2 N_r$;
6 **for** $z = 1$ *to* $round$ **do**
7 Use *QuickSelect* algorithm and *divide-and-conquer* strategy to find 2^{z-1} breakpoints in round z;
8 Store the found breakpoints in B_{ij};
9 $final\_region\_size \leftarrow \lfloor \frac{n_s}{2^{round}} \rfloor$;
10 $B_{ij}(1) \leftarrow$ the minimum element from $C_{ij}(1)$ to $C_{ij}(final\_region\_size)$;
11 $B_{ij}(N_r + 1) \leftarrow$ the maximum element from $C_{ij}(n_s - final\_region\_size)$ to $C_{ij}(n_s)$;
12 **return** B;

Algorithm 2: Dynamic Encoding

Input: Parameters K, L, n, all points in projected spaces P, sample size n_s, number of regions in each projected space N_r
Output: A set of encoded points EP

1 Initialize EP with size $n \cdot L \cdot K$;
2 $B \leftarrow$ **call** BreakpointsSelection(K,L,n,P,n_s,N_r);
3 **for** $i = 1$ *to* L **do**
4 **for** $j = 1$ *to* K **do**
5 **for** $z = 1$ *to* n **do**
6 Obtain o_z from P;
7 Use *BinarySearch* to find integer $b \in [1, N_r]$ such that $B_{ij}(b) \leq h_{ij}(o_z) \leq B_{ij}(b + 1)$;
8 $EP_{ij}(o_z) \leftarrow b$-th symbol in the 8-bit alphabet;
9 **return** EP;

b_i to b_{i+1} is $\frac{1}{a}$, where b_0 and b_a are defined as $-\infty$ and $+\infty$. Therefore, these breakpoints are static and independent of datasets. Existing methods encode a data series by checking which two breakpoints each of its coordinates falls between in a common statistical table. However, the datasets for ANN search have arbitrary distributions, so the static encoding scheme is no longer suitable.

Dynamic encoding scheme. In DET-LSH, we design a dynamic encoding scheme to dynamically select breakpoints based on the distribution of the dataset, aiming to divide data points into different regions as evenly as possible, i.e., each region contains the same number of points. Specifically, assuming we have a dataset with cardinality n, we first use $K \cdot L$ hash functions to calculate the K-dimensional points in L projected spaces, where $\mathcal{H}_i(o) = [h_{i1}(o), ..., h_{iK}(o)]$ denote a point o in the i-th projected space. Let $C_{ij} = [h_{ij}(o_1), ..., h_{ij}(o_n)]$ denote the set of coordinates of all n points in their i-th projected space and j-th dimension, where $i = 1, ..., L$ and $j = 1, ..., K$. We denote $C_{ij}^{\uparrow}$ as a new set in which the elements of C_{ij} are sorted in ascending order, and use $C_{ij}^{\uparrow}(t)$ to represent the t-th element in $C_{ij}^{\uparrow}$. Intuitively, to make points evenly divided into $N_r = 256$ regions in each dimension, we can select ordered breakpoints B_{ij} from $C_{ij}^{\uparrow}$, where $B_{ij}(z) = C_{ij}^{\uparrow}(\lfloor \frac{n}{N_r} \rfloor \cdot (z-1))$ and $z = 2, ..., N_r$. We set $B_{ij}(1) = C_{ij}^{\uparrow}(1)$ and $B_{ij}(N_r + 1) = C_{ij}^{\uparrow}(n)$. In practice, we dynamically select $N_r + 1$ breakpoints for each dimension. Then, for any point o, each dimension $h_{ij}(o)$ can be independently encoded based on the selected breakpoints B_{ij}.

In terms of algorithm design, the intuitive idea is to completely sort C_{ij} to get the exact $C_{ij}^{\uparrow}$ and then select breakpoints from it. However, we only need $N_r + 1$ discrete elements in $C_{ij}^{\uparrow}$, the complete sorting of C_{ij} is wasteful. Therefore, combining the *QuickSelect* algorithm with the *divide-and-conquer* strategy, we design a dynamic encoding scheme based on the unordered C_{ij}. Algorithm 1 introduces how we dynamically select breakpoints, which is the first step of the encoding scheme. To improve efficiency, we randomly sample n_s points from the dataset and select breakpoints based on these sampled points. In practice, we set $n_s = 0.1n$. For each C_{ij}, Algorithm 1 obtains breakpoints by running multiple rounds of the *QuickSelect* algorithm combined with the *divide-and-conquer* strategy (lines 6-8). For unordered C_{ij}, *QuickSelect*($start$, q, end) can find the q-th smallest element between $C_{ij}(start)$ and $C_{ij}(end)$,

Algorithm 3: Create Index

Input: Parameters K, L, n, encoded points set EP, maximum size of a leaf node max_size
Output: A set of DE-Trees: $DETs = [T_1, ..., T_L]$

1 **for** $i = 1$ *to* L **do**
2 Initialize T_i and generate 2^K first layer nodes as the original leaf nodes;
3 **for** $z = 1$ *to* n **do**
4 $ep_i(o_z) \leftarrow (EP_{i1}(o_z), ..., EP_{iK}(o_z))$;
5 $pos_z \leftarrow$ the position of o_z in the dataset;
6 $target\_leaf \leftarrow$ leaf node of T_i to insert $\langle ep_i(o_z), pos_z \rangle$;
7 **while** $sizeof(target\_leaf) \geq max\_size$ **do**
8 SplitNode($target\_leaf$);
9 $target\_leaf \leftarrow$ the new leaf node to insert $\langle ep_i(o_z), pos_z \rangle$;
10 Insert $\langle ep_i(o_z), pos_z \rangle$ to $target\_leaf$;
11 **return** $DETs$;

and move it to the position of $C_{ij}(start + q)$. Then, $C_{ij}(start + q)$ is greater than all elements from $C_{ij}(start)$ to $C_{ij}(start + q - 1)$ and smaller than all elements from $C_{ij}(start + q + 1)$ to $C_{ij}(end)$. Therefore, we can select a single breakpoint from C_{ij} by running *QuickSelect* once. Since we set $N_r = 256$, the *divide-and-conquer* strategy can be perfectly applied to our algorithm. Specifically, a total of $\log_2 N_r$ rounds need to run, and the z-th round select 2^{z-1} breakpoints by running *QuickSelect* in 2^{z-1} sub-regions generated from the (z-1)-th round, where $z = 1, ..., \log_2 N_r$. For each C_{ij}, we select the minimum element as the first breakpoint $B_{ij}(1)$ and the maximum element as the last breakpoint $B_{ij}(N_r + 1)$ (lines 9-11). In practice, Algorithm 1 achieves 3x speedup in running time over the complete sorting scheme, as shown in Section 6.2. After getting the breakpoint set B, Algorithm 2 will encode all points into iSAX representations and return the set of encoded points EP (lines 3-8).

4.2 Indexing Phase

As mentioned before, DET-LSH requires L DE-Trees to support queries. Algorithm 3 presents how to construct L DE-Trees based on the encoded points set EP. Specifically, for each DE-Tree, the first step is to initialize the first layer nodes, which are the children of the root (line 2). As shown in Figure 1(b), according to the iSAX encoding rules, the initial division of each dimension has two cases: 0^* and 1^*. Therefore, each DE-Tree has 2^K first layer nodes. Then, for each point o_z, we get its encoded representation $ep_i(o_z)$ for the i-th DE-Tree T_i and its position pos_z in the dataset (lines 3-5). Based on $ep_i(o_z)$, we can get the leaf node of T_i to insert $\langle ep_i(o_z), pos_z \rangle$ (line 6). If the leaf node is full, we split it until we get a new leaf node and insert $\langle ep_i(o_z), pos_z \rangle$ (lines 7-10). Note that only leaf nodes contain information about points, such as encoded representations and positions, while internal nodes only contain index information.

In a DE-Tree, except the root node that has 2^K children, other internal nodes have only two children. This is because when an internal node needs to be split, we only select one of its K dimensions for further bit-wise binary division. For example, in Figure 1(b), we choose the first dimension of node [0^*,0^*] to split, and the representations of its two children are [00,0^*] and [01,0^*]. The choice of which dimension to divide is important for splitting nodes. Intuitively, splitting a node works better if the obtained two children contain similar numbers of points. Therefore, when splitting nodes, we choose the dimension that most evenly divides the points.

Algorithm 4: DET Range Query

Input: A projected query point q', the search radius r', the index DE-Tree T, project dimension K
Output: A set of points S

1 Initialize a points set $S \leftarrow \emptyset$;
2 **for** $i = 1$ *to* 2^K **do**
3 $node \leftarrow$ the i-th child of root node in T;
4 **call** TraverseSubtree($node$, q', r', S);
5 **return** S;

Algorithm 5: Traverse Subtree

Input: A node $node$, the projected query point q', the search radius r', the set of points S

1 $lower\_bound\_dist \leftarrow$ calculate the lower bound distance between q' and $node$;
2 **if** $lower\_bound\_dist > r'$ **then**
3 break;
4 **else if** *node is a leaf* **then**
5 $upper\_bound\_dist \leftarrow$ calculate the upper bound distance between q' and $node$;
6 **if** $upper\_bound\_dist \leq r'$ **then**
7 $S \leftarrow S \cup$ all points in $node$;
8 **else**
9 **while** *node has next point* **do**
10 Get next point $o \in node$;
11 $dist \leftarrow$ calculate the distance between q' and the projected o';
12 **if** $dist \leq r'$ **then**
13 $S \leftarrow S \cup o$;
14 **else**
15 **call** TraverseSubtree($node.leftChild$, q', r', S);
16 **call** TraverseSubtree($node.rightChild$, q', r', S);

4.3 Query Phase

Since the query strategy of DET-LSH is based on the Euclidean distance metric, range queries can improve the efficiency of obtaining candidate points. In a DE-Tree, each space is divided into different regions by multiple breakpoints. The breakpoints on all sides of a region can be used to calculate the upper and lower bound distances between two points or between a point and a tree node.

DET Range Query. Algorithm 4 is designed for range queries in DE-Tree. We select all 2^K children of the root node as the entry of the traversal and then traverse their subtrees in order (lines 2-4). Algorithm 5 presents how to obtain points within the search radius

Algorithm 6: (r,c)-ANN Query

Input: A query point q, parameters K, L, n, c, r, ϵ, β, index DE-Trees $DETs = [T_1, ..., T_L]$
Output: A point o or $\emptyset$

1 Initialize a candidate set $S \leftarrow \emptyset$;
2 **for** $i = 1$ *to* L **do**
3 Compute $q'_i = H_i(q) = [h_{i1}(q), ..., h_{iK}(q)]$;
4 $S_i \leftarrow$ **call** DETRangeQuery$(q'_i, \epsilon \cdot r, T_i, K)$;
5 $S \leftarrow S \cup S_i$;
6 **if** $|S| \geq \beta n + 1$ **then**
7 **return** the point o closest to q in S;
8 **if** $|\{o \mid o \in S \wedge \|o, q\| \leq c \cdot r\}| \geq 1$ **then**
9 **return** the point o closest to q in S;
10 **return** $\emptyset$;

Algorithm 7: c^2-k-ANN Query

Input: A query point q, parameters K, L, n, c, r_{min}, ϵ, β, k, index DE-Trees $DETs = [T_1, ..., T_L]$
Output: k nearest points to q in S

1 Initialize a candidate set $S \leftarrow \emptyset$ and set $r \leftarrow r_{min}$;
2 **while** *TRUE* **do**
3 **for** $i = 1$ *to* L **do**
4 Compute $q'_i = H_i(q) = [h_{i1}(q), ..., h_{iK}(q)]$;
5 $S_i \leftarrow$ **call** DETRangeQuery$(q'_i, \epsilon \cdot r, T_i, K)$;
6 $S \leftarrow S \cup S_i$;
7 **if** $|S| \geq \beta n + k$ **then**
8 **return** the *top-k* points closest to q in S;
9 **if** $|\{o \mid o \in S \wedge \|o, q\| \leq c \cdot r\}| \geq k$ **then**
10 **return** the *top-k* points closest to q in S;
11 $r \leftarrow c \cdot r$;

r' by recursively traversing the subtrees. For the node being visited, if its lower bound distance with q' is greater than r', it means that the distance between any point in its subtree and q' is greater than r', so no further traversal is needed (lines 1-3). If the upper bound distance between a leaf node and q' is not greater than r', it means that the distance between any data point in the leaf node and q' is not greater than r', so that all points can be added to S (lines 4-7). If r' falls within the range of the lower bound distance and the upper bound distance between a leaf node and q', we should traverse the data points in the leaf node and add those within the search radius to S (lines 8-13). If the node being visited is not a leaf node and further traversal is required, we need to further traverse its subtrees (lines 14-16).

(r,c)-ANN Query. Algorithm 6 shows that DET-LSH can answer an (r,c)-ANN query with any search radius r. After the indexing phase, DET-LSH obtains L DE-Trees $T_1, ..., T_L$. Given a query q, we consider L projected spaces in order. For the i-th space, we first compute the projected query q'_i (line 3). Then, we call Algorithm 4 to perform a range query in the i-th DE-Tree T_i (line 4). The search radius in the projected space is $\epsilon \cdot r$. The parameter ϵ guarantees that if the distance between a point o and q is not greater than r, then the distance between the projected o' and q' is not greater than $\epsilon \cdot r$ with a constant probability. Detailed analysis and proof will be introduced in Lemma 3 in Section 5. We continuously add the candidate points obtained by range queries to a candidate set S (line 5). If the number of candidate points in S exceeds $\beta n + 1$, the point o closest to q will be returned, where parameter β is the maximum false positive percentage (lines 6-7). After completing range queries in L DE-Trees, if the size of S is still smaller than $\beta n + 1$ and there is at least one point in S whose distance with q is not greater than $c \cdot r$, then return the point o closest to q in S (lines 8-9). Otherwise, the algorithm returns nothing (line 10). According to Theorem 1, to be introduced in Section 5, DET-LSH can correctly answer an (r,c)-ANN query with a constant probability.

c^2-k-ANN Query. Since o^* and $\|q, o^*\|$ are not known in advance, we cannot directly perform an ANN query with a pre-defined r like (r,c)-ANN query does. Instead, we can conduct a series of (r,c)-ANN queries with increasing radii until enough points are returned. Algorithm 7 outlines the query processing. We can see that most of the steps of Algorithm 7 (lines 3-10) are almost the same as Algorithm 6 (lines 2-9), except that Algorithm 7 needs to consider k when judging conditions and returning results. The main difference is that when neither the termination condition at line 8 nor line 10 is satisfied, Algorithm 7 will enlarge the search radius for the next round of queries (line 11). According to Theorem 2, to be introduced in Section 5, DET-LSH can correctly answer a c^2-k-ANN query with a constant probability.

5 THEORETICAL ANALYSIS

5.1 Quality Guarantee

Let $\mathcal{H}_i(o) = [h_{i1}(o), ..., h_{iK}(o)]$ denote a data point o in the i-th projected space, where $i = 1, ..., L$. We define three events as follows:

- **E1:** If there exists a point o satisfying $\|o, q\| \leq r$, then its projected distance to q, i.e., $\|\mathcal{H}_i(o), \mathcal{H}_i(q)\|$, is smaller than ϵr for some $i = 1, ..., L$;
- **E2:** If there exists a point o satisfying $\|o, q\| > cr$, then its projected distance to q, i.e., $\|\mathcal{H}_i(o), \mathcal{H}_i(q)\|$, is smaller than ϵr for some $i = 1, ..., L$;
- **E3:** Fewer than βn points satisfying **E2** in dataset $\mathcal{D}$.

Lemma 3. *Given K and c, setting $L = -\frac{1}{\ln \alpha_1}$ and $\beta = 2 - 2\alpha_2^{-\frac{1}{ln\alpha_1}}$ such that α_1, α_2, and ϵ satisfy Equation 3, the probability that **E1** occurs is at least $1 - \frac{1}{e}$ and the probability that **E3** occurs is at least $\frac{1}{2}$.*

$$\epsilon^2 = \chi^2_{\alpha_1}(K) = c^2 \cdot \chi^2_{\alpha_2}(K). \tag{3}$$

Proof. Given a point o satisfying $\|o, q\| \leq r$, let $s = \|o, q\|$ and $s'_i = \|\mathcal{H}_i(o), \mathcal{H}_i(q)\|$ denote the distances between o and q in the original space and in the i-th projected space, where $i = 1, ..., L$. From Equation 3, we have $\sqrt{\chi^2_{\alpha_1}(K)} = \epsilon$. For each independent projected space, from Lemma 2, we have $\Pr\left[s'_i > s\sqrt{\chi^2_{\alpha_1}(K)}\right] = \Pr\left[s'_i > \epsilon s\right] = \alpha_1$. Since $s \leq r$, $\Pr\left[s'_i > \epsilon r\right] \leq \alpha_1$. Considering L projected spaces, we have $\Pr\left[\mathbf{E1}\right] \geq 1 - \alpha_1^L = 1 - \frac{1}{e}$. Likewise, given a point o satisfying $\|o, q\| > cr$, let $s = \|o, q\|$ and $s'_i = \|\mathcal{H}_i(o), \mathcal{H}_i(q)\|$ denote the distances between o and q in the original space and in the

i-th projected space, where $i = 1, ..., L$. From Equation 3, we have $\sqrt{\chi^2_{\alpha_2}(K)} = \frac{\epsilon}{c}$. For each independent projected space, from Lemma 2, we have $\Pr[s'_i > s\sqrt{\chi^2_{\alpha_2}(K)}] = \Pr[s'_i > \frac{\epsilon s}{c}] = \alpha_2$. Since $s > cr$, i.e., $\frac{s}{c} > r$, $\Pr[s'_i > \epsilon r] > \alpha_2$. Considering L projected spaces, we have $\Pr[\mathbf{E2}] \leq 1 - \alpha_2^L$, thus the expected number of such points in dataset $\mathcal{D}$ is upper bounded by $(1 - \alpha_2^L) \cdot n$. By *Markov's inequality*, we have $\Pr[\mathbf{E3}] > 1 - \frac{(1-\alpha_2^L)\cdot n}{\beta n} = \frac{1}{2}$.

□

Theorem 1. *Algorithm 6 answers an (r,c)-ANN query with at least a constant probability of* $\frac{1}{2} - \frac{1}{e}$.

Proof. We show that when **E1** and **E3** hold at the same time, Algorithm 6 returns an correct (r,c)-ANN result. The probability of **E1** and **E3** occurring at the same time can be calculated as $\Pr[\mathbf{E1E3}] = \Pr[\mathbf{E1}] - \Pr[\mathbf{E1}\overline{\mathbf{E3}}] > \Pr[\mathbf{E1}] - \Pr[\overline{\mathbf{E3}}] = \frac{1}{2} - \frac{1}{e}$. When **E1** and **E3** hold at the same time, if Algorithm 6 terminates after getting at least $\beta n + 1$ candidate points (line 7), due to **E3**, there are at most βn points satisfying $\|o, q\| > cr$. Thus we can get at least one point satisfying $\|o, q\| \leq cr$, and the returned point is obviously a correct result. If the candidate set S has no more than $\beta n + 1$ points, but there exists at least one point in S satisfying $\|o, q\| \leq cr$, Algorithm 6 can also terminate and then return a result correctly (line 9). Otherwise, it indicates that no points satisfying $\|o, q\| \leq cr$. According to the Definition 3 of (r,c)-ANN, nothing will be returned (line 10). Therefore, when **E1** and **E3** hold at the same time, Algorithm 6 can always correctly answer an (r,c)-ANN query. In other words, Algorithm 6 answers an (r,c)-ANN query with at least a constant probability of $\frac{1}{2} - \frac{1}{e}$. □

Theorem 2. *Algorithm 7 answers a* c^2*-k-ANN query with at least a constant probability of* $\frac{1}{2} - \frac{1}{e}$.

Proof. We show that when **E1** and **E3** hold at the same time, Algorithm 7 returns a correct c^2-k-ANN result. Let o_i^* be the i-th exact nearest point to q in $\mathcal{D}$, we assume that $r_i^* = \|o_i^*, q\| > r_{min}$, where r_{min} is the initial search radius and $i = 1, ..., k$. We denote the number of points in the candidate set under search radius r as $|S_r|$. Obviously, when enlarging the search radius $r = r_{min}, r_{min} \cdot c, r_{min} \cdot c^2, ...$, there must exist a radius r_0 satisfying $|S_{r_0}| < \beta n + k$ and $|S_{c \cdot r_0}| \geq \beta n + k$. The distribution of r_i^* has three cases:

(1) **Case 1:** If for all $i = 1, ..., k$ satisfying $r_i^* \leq r_0$, which indicates the range queries in all L index trees have been executed at $r = r_0$ (lines 3-8). Due to **E1**, all r_i^* must in S_{r_0}. Since $S_{r_0} \subsetneq S_{c \cdot r_0}$, all r_i^* also must in $S_{c \cdot r_0}$. Therefore, Algorithm 7 returns the exact k nearest points o_i^* to q.

(2) **Case 2:** If for all $i = 1, ..., k$ satisfying $r_i^* > r_0$, all r_i^* not belong to S_{r_0}. Since Algorithm 7 may terminate after executing range queries in part of L index trees at $r = c \cdot r_0$ (line 8), we cannot guarantee that $r_i^* \leq c \cdot r_0$. However, due to **E3**, there are at least k points o_i in $S_{c \cdot r_0}$ satisfying $\|o_i, q\| \leq c^2 r_0$, $i = 1, ..., k$. Therefore, we have $\|o_i, q\| \leq c^2 r_0 \leq c^2 r_i^*$, i.e., each o_i is a c^2-ANN point for corresponding o_i^*.

(3) **Case 3:** If there exists an integer $m \in (1, k)$ such that for all $i = 1, ..., m$ satisfying $r_i^* \leq r_0$ and for all $i = m + 1, ..., k$ satisfying $r_i^* > r_0$, indicating that **Case 3** is a combination of **Case 1** and **Case 2**. For each $i \in [1, m]$, Algorithm 7 returns the exact nearest point o_i^* to q based on **Case 1**. For each $i \in [m + 1, k]$, Algorithm 7 returns a c^2-ANN point for o_i^* based on **Case 2**.

Therefore, when **E1** and **E3** hold simultaneously, Algorithm 7 can always correctly answer a c^2-k-ANN query, i.e., Algorithm 7 returns a c^2-k-ANN with at least a constant probability of $\frac{1}{2} - \frac{1}{e}$. □

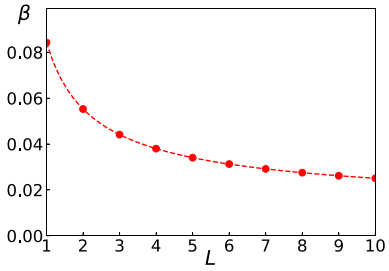


Figure 3: Illustration of the theoretical β when L varies (for $K = 16$ and $c = 1.5$), which is in line with Lemma 3.

5.2 Parameter Settings

The performance of DET-LSH is affected by several parameters: L, K, β, c, and so on. According to Lemma 3, when L and c are set as constants, there is a mathematical relationship between K and β. We set $K = 16$ and $c = 1.5$ by default, and Figure 3 shows the theoretical β as L changes, which is in line with Lemma 3. Figure 3 illustrates that β and L have a negative correlation. Theoretically, a greater β means a higher fault tolerance when querying, so the accuracy of DET-LSH is improved. Meanwhile, a greater L means fewer correct results are missed when querying, so the accuracy of DET-LSH can also be improved. However, both greater β and greater L will reduce query efficiency, so we need to find a balance between β and L. As shown in Figure 3, $L = 4$ is a good choice because as L increases, β drops rapidly until $L = 4$, and then β drops slowly. Therefore, we choose $L = 4$ as the default value.

For the initial search radius r_{min}, we follow the selection scheme proposed in [66]. Specifically, to reduce the number of iterations for different r and terminate the query process faster, we find a 'magic' r_{min} that satisfies the following conditions: 1) when $r = r_{min}$ in Algorithm 7, the number of candidate points in S satisfies $|S| \geq \beta n + k$; 2) when $r = \frac{r_{min}}{c}$ in Algorithm 7, the number of candidate points in S satisfies $|S| < \beta n + k$. Since DET-LSH can implement dynamic incremental queries as r increases, the choice of r_{min} is expected to have a relatively small impact on its performance.

5.3 Complexity Analysis

In the encoding and indexing phases, DET-LSH has time cost $O(n(d + \log N_r))$, and space cost $O(n)$. The time cost comes from four parts: (1) computing hash values for n points, $O(L \cdot K \cdot n \cdot d)$; (2) using Algorithm 1 for breakpoint selection, $O(L \cdot K \cdot n \cdot \log N_r)$; (3) using Algorithm 2 for encoding, $O(L \cdot K \cdot n \cdot \log N_r)$; and (4) using Algorithm 3 for constructing L DE-Trees, $O(L \cdot n \cdot K \cdot \log N_r)$. Since both $K = O(1)$ and $L = O(1)$ are constants, the total time cost is $O(n(d + \log N_r))$. Obviously, the size of encoded points and L DE-Trees are both $O(L \cdot K \cdot n) = O(n)$.

Table 2: Datasets

Dataset	Cardinality	Dimensions	Type
Msong	994,185	420	Audio
Deep1M	1,000,000	256	Image
Sift10M	10,000,000	128	Image
TinyImages80M	79.302,017	384	Image
Sift100M	100,000,000	128	Image
Yandex Deep500M	500,000,000	96	Image
Microsoft SPACEV500M	500,000,000	100	Text
Microsoft Turing-ANNS500M	500,000,000	100	Text

In the query phase, DET-LSH has time cost $O(n(\beta d + \log N_r))$. The time cost comes from four parts: (1) computing hash values for the query point q, $O(L \cdot K \cdot d) = O(d)$; (2) finding candidate points in L DE-Trees, $O(L \cdot K^2 \cdot \log N_r \cdot \frac{n}{max\_size} + L \cdot K \cdot n) = O(n \log N_r)$; (3) computing the real distance of each candidate point to q, $O(\beta n d)$; and (4) finding the top-k points to q, $O(\beta n \log k)$. The total time cost in the query phase is $O(n(\beta d + \log N_r))$.

6 EXPERIMENTAL EVALUATION

In this section, we self-evaluate DET-LSH, conduct comparative experiments with the state-of-the-art LSH-based methods, and compare with graph-based methods. Our method is implemented in C and C++ and compiled using -O3 optimization. All experiments are conducted using a single thread, on a machine with 2 AMD EPYC 9554 CPUs @ 3.10GHz and 756 GB RAM, running on Ubuntu 22.04.

6.1 Experimental Setup

Datasets and Queries. We use eight real-world datasets for ANN search. Table 2 shows the key statistics of the datasets. Note that the points in *Sift10M* and *Sift100M* are randomly chosen from the *Sift1B* dataset[1]. Similarly, the points in *Yandex Deep500M*, *Microsoft SPACEV500M*, and *Microsoft Turing-ANNS500M* are also randomly chosen from their 1B-scale datasets[2]. We randomly select 100 data points as queries and remove them from the original datasets.

Evaluation Measures. We adopt five measures to evaluate the performance of all methods: index size, indexing time, query time, recall, and overall ratio. For a query q, we denote the result set as $R = \{o_1, ..., o_k\}$ and the exact k-NNs as $R^* = \{o_1^*, ..., o_k^*\}$, recall is defined as $\frac{|R \cap R^*|}{k}$ and overall ratio is defined as $\frac{1}{k}\sum_{i=1}^{k} \frac{\|q,o_i\|}{\|q,o_i^*\|}$ [55].

Benchmark Methods. We compare DET-LSH with three state-of-the-art LSH-based in-memory methods mentioned in Section 2, i.e., DB-LSH [55], LCCS-LSH [31], and PM-LSH [66]. Moreover, to study the capability of DE-Tree and the advantages of LSH, we use a single DE-Tree to index points without LSH for ANN searches. We call this method DET-ONLY. Since DET-ONLY is not based on LSH, we adopt the Piecewise Aggregate Approximation (PAA) [29] technique to reduce the dimensionality of points. PAA divides a d-dimensional point into K segments of equal length $\lfloor \frac{d}{K} \rfloor$ and uses the mean value of the coordinates in each segment to summarize the point. DET-ONLY adopts the same query strategy as DET-LSH. To study the characteristics of LSH-based methods and graph-based methods,

[1]http://corpus-texmex.irisa.fr/

[2]https://big-ann-benchmarks.com/neurips21.html

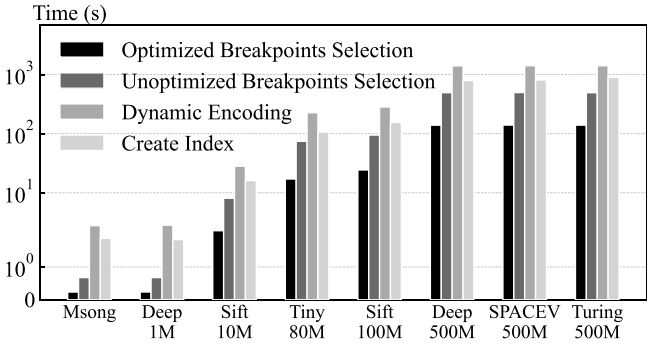


Figure 4: Running time break-down for the DET-LSH encoding and indexing phases.

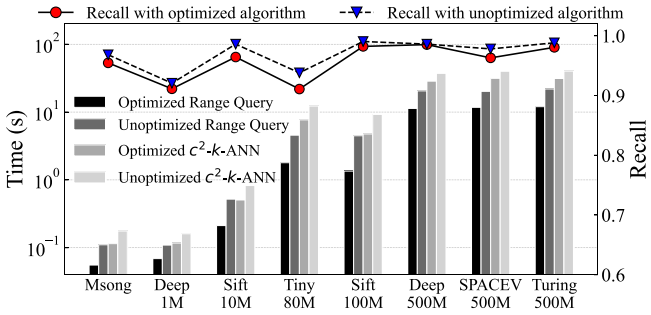


Figure 5: Running time and recall of optimized/non-optimized query-phase algorithms of DET-LSH.

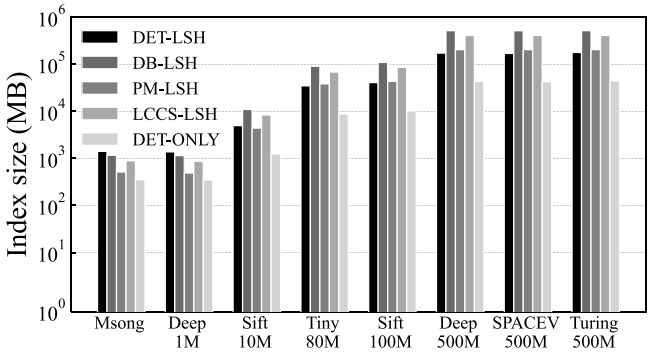


Figure 6: Index size for all datasets.

we also compare DET-LSH with two state-of-the-art graph-based methods, i.e., HNSW [38] and LSH-APG [65].

Parameter Settings. k in k-ANN is set to 50 by default. For DET-LSH, the parameters are set as described in Section 5.2. For competitors, the parameter settings follow their source codes or papers. To make a fair comparison, we set $\beta = 0.1$ and $c = 1.5$ for DET-LSH, DB-LSH, PM-LSH, and DET-ONLY. For DB-LSH, $L = 5$, $K = 12$, $w = 4c^2$. For LCCS-LSH, $m = 64$. For PM-LSH, $s = 5$, $m = 15$. For DET-ONLY, $K = 16$, $L = 1$. For HNSW, $M = 48$, $ef = 100$. For LSH-APG, $K = 16$, $L = 2$, $T = 24$, $T' = 2T$, $p_\tau = 0.95$.

Table 3: Performance comparison with competitors (the best value in each row is highlighted in bold; the number in parentheses indicates how many times slower a method is than the best method).

		DET-LSH	DB-LSH	PM-LSH	LCCS-LSH	DET-ONLY
Msong	Query Time (ms)	112.97 (1.43)	118.10 (1.49)	120.36 (1.53)	170.13 (2.16)	**78.87**
	Recall	**0.9546**	0.9474	0.949	0.849	0.891
	Overall Ratio	**1.0012**	1.0013	1.0013	1.0035	1.0046
	Indexing Time (s)	4.654 (3.90)	4.974 (4.17)	2.950 (2.47)	28.925 (24.2)	**1.194**
Deep1M	Query Time (ms)	109.28 (1.37)	117.79 (1.48)	207.37 (2.61)	136.21 (1.71)	**79.51**
	Recall	**0.9112**	0.8552	0.857	0.848	0.818
	Overall Ratio	**1.0022**	1.0038	1.0042	1.0039	1.0061
	Indexing Time (s)	4.647 (3.97)	4.809 (4.10)	2.991 (2.55)	57.652 (49.2)	**1.172**
Sift10M	Query Time (ms)	506.34 (1.20)	944.23 (2.25)	1482.84 (3.53)	1905.09 (4.53)	**420.43**
	Recall	**0.9644**	0.9438	0.9338	0.8924	0.886
	Overall Ratio	**1.0009**	1.0015	1.0016	1.0021	1.0035
	Indexing Time (s)	44.435 (4.00)	64.861 (5.85)	80.099 (7.22)	509.417 (45.9)	**11.094**
TinyImages80M	Query Time (ms)	7676.23 (1.00)	8164.96 (1.07)	13672.6 (1.79)	11272.8 (1.47)	**7657.08**
	Recall	**0.9108**	0.9056	0.8822	0.87	0.8338
	Overall Ratio	**1.0016**	**1.0016**	1.0023	1.0019	1.0036
	Indexing Time (s)	335.419 (4.08)	641.988 (7.81)	1471.31 (17.89)	12128.1 (147.5)	**82.235**
Sift100M	Query Time (ms)	4757.76 (1.20)	11064.8 (2.78)	15722.8 (3.95)	24221.8 (6.08)	**3983.41**
	Recall	**0.9822**	0.9652	0.944	0.892	0.8848
	Overall Ratio	**1.0005**	1.0007	1.0013	1.0019	1.0034
	Indexing Time (s)	439.434 (4.04)	952.773 (8.76)	1922.7 (17.67)	7519.43 (69.1)	**108.782**
Yandex Deep500M	Query Time (ms)	28546.6 (1.09)	61657.9 (2.35)	91724.2 (3.50)	62411.8 (2.38)	**26200.4**
	Recall	**0.9852**	0.9644	0.9298	0.9506	0.9176
	Overall Ratio	**1.0003**	1.0009	1.0032	1.0009	1.0058
	Indexing Time (s)	2263.87 (4.22)	17182.7 (32.04)	13685.2 (25.52)	85968.3 (160.3)	**536.262**
Microsoft SPACEV500M	Query Time (ms)	31404.3 (1.07)	66632.3 (2.28)	94868.3 (3.25)	70697.5 (2.42)	**29212.6**
	Recall	**0.963**	0.9492	0.9568	0.9198	0.8978
	Overall Ratio	**1.0008**	1.0012	1.0011	1.0026	1.00336
	Indexing Time (s)	2204.94 (4.21)	16114.7 (30.77)	13189.5 (25.19)	87591.1 (167.3)	**523.662**
Microsoft Turing-ANNS500M	Query Time (ms)	31280.1 (1.04)	68636.6 (2.28)	106987 (3.55)	73618.2 (2.44)	**30127.2**
	Recall	**0.9806**	0.9604	0.9636	0.9404	0.9008
	Overall Ratio	**1.0005**	1.0012	1.0009	1.0012	1.0043
	Indexing Time (s)	2301.02 (4.22)	16408.2 (30.11)	12680.2 (23.27)	79162.5 (145.3)	**545.006**

6.2 Self-evaluation of DET-LSH

6.2.1 Encoding and Indexing Phase. Figure 4 shows the specific running time of each algorithm in the encoding and indexing phases. We have the following observations: (1) *Dynamic Encoding* (Algorithm 2) takes longer time than *Create Index* (Algorithm 3). Although we have optimized the process of locating regions when encoding through binary search, it still takes much time to locate a specific region from 256 regions for each dimension of each projected point. (2) Optimized *Breakpoints Selection* (Algorithm 1) achieves 3x speedup in running time over the unoptimized algorithm. As mentioned in Section 4.1, we use *QuickSelect* algorithm with *divide-and-conquer* strategy to avoid complete sorting, thus reducing the time complexity from $O(n \log n)$ to $O(n \log N_r)$.

6.2.2 Query Phase. In practice, in Algorithm 5, if the upper bound distance between a leaf node and q' is greater than the search radius, it will take much time to calculate the distance between each point in the leaf node and q' (lines 8-13). After experiments, we found that if the leaf node size *max_size* is appropriately set in Algorithm 3, most of the points in these 'troublesome' leaf nodes are within the search radius. Therefore, we optimized Algorithm 5 in two aspects: (1) We relax the requirements for candidate points to improve efficiency. In our implementation, as long as the lower bound distance between a leaf node and q' is not greater than r, we will add all its points to S. (2) We maintain a priority queue to hold traversed leaf nodes based on their lower bound distances to q'. A leaf node with a smaller lower bound distance to q' can add all its points to S earlier, guaranteeing the quality of candidate points. As shown in Figure 5, with an acceptable sacrifice of query accuracy, optimized Algorithm 4 and Algorithm 7 improve query efficiency by up to 50% and 30%.

6.3 Comparison with Competitors

6.3.1 Indexing Performance. Figure 6 and Table 3 show the comparison between all methods with default parameter settings on all datasets. To ensure fairness, for DET-LSH and DET-ONLY, the time of the encoding phase is included in the indexing time. We make the following observations: (1) DET-LSH has the best indexing efficiency compared to all LSH-based methods. The reason is that DB-LSH and PM-LSH use data-oriented partitioning trees to construct indexes. It is time-consuming to partition a multi-dimensional projected space. DET-LSH adopts DE-Trees to construct indexes, which divide and encode each dimension of the projected space independently, thereby improving indexing accuracy. LCCS-LSH

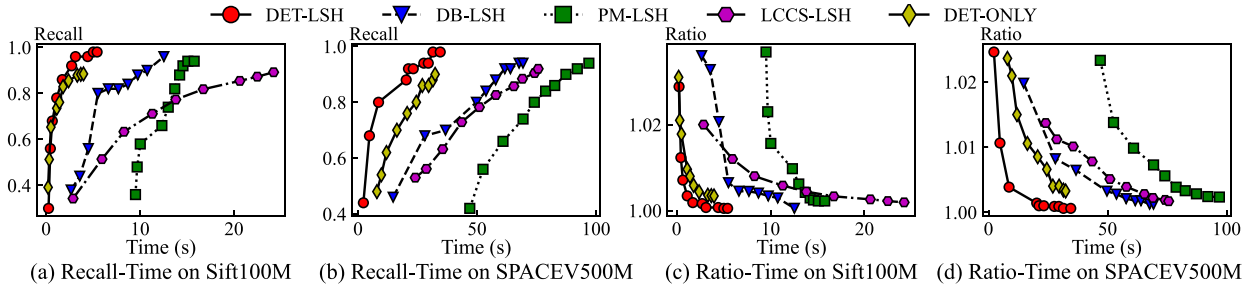


Figure 7: Recall-time and overall ratio-time curves.

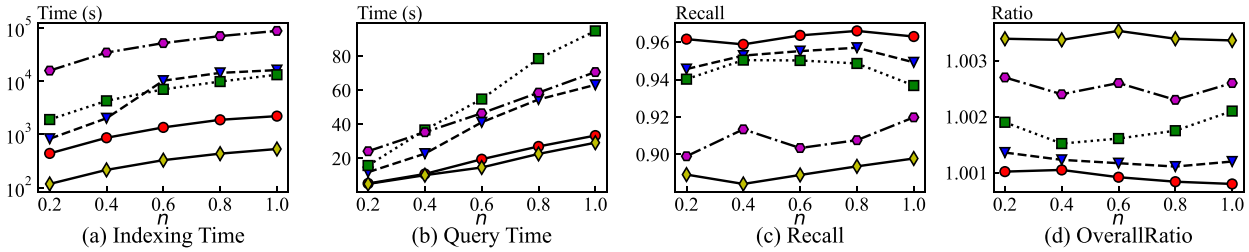


Figure 8: Scalability: performance under different n on Microsoft SPACEV500M.

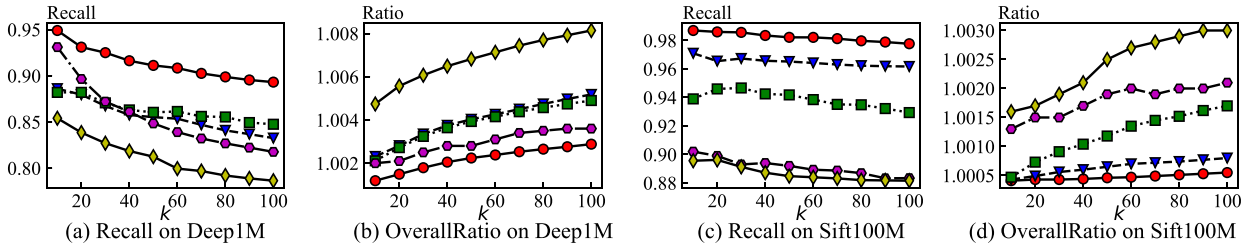


Figure 9: Performance under different k.

has a significantly longer indexing time compared to other methods because of building its proposed data structure Circular Shift Array (CSA). (2) The advantage of DET-LSH's indexing efficiency increases with the dataset cardinality. When n is not greater than 1M, the indexing time of DET-LSH is longer than that of PM-LSH, because DET-LSH constructs 4 DE-Trees, while PM-LSH only constructs one PM-Tree. As n increases from 10M to 500M, DET-LSH achieves from 2x speedup to 6x speedup in indexing time over other methods. The reason is that the construction time of a DE-Tree increases linearly with n. (3) With respect to index size, DET-LSH is not very competitive on small-scale datasets, but its design proves advantageous for large-scale datasets. The reason is that DET-LSH only saves the iSAX representation of each data point in the DE-Tree (not the original data point or the LSH-projected data point). Each iSAX representation is stored as an 'unsigned char', which takes up only one byte. Yet, DET-LSH builds 4 DE-Trees, which restricts its advantage on small-scale datasets. As the dataset size increases, the advantage of DET-LSH becomes more pronounced. (4) The index size and indexing time of DET-ONLY are always about one-quarter of that of DET-LSH. The reason is that DET-ONLY only constructs one DE-Tree, while DET-LSH constructs 4 DE-Trees.

6.3.2 Query Performance. We study query performance based on the query time, recall, and overall ratio shown in Table 3, and the Recall-Time and OverallRatio-Time curves shown in Figure 7. We have the following observations: (1) DET-LSH outperforms all LSH-based methods on both efficiency and accuracy (DET-ONLY is not an LSH-based method). As shown in Table 3, DET-LSH has a shorter query time, higher recall, and smaller ratio on all datasets. As n increases, DET-LSH achieves up to 2x speedup in query time over other LSH-based methods. The reason is that closer points have similar encoding representations in DE-Tree so that range queries can obtain higher-quality candidate points in a shorter time. (2) The query efficiency of DET-ONLY is slightly better than DET-LSH, but the query accuracy is significantly lower than DET-LSH. Since DET-LSH performs queries on 4 DE-Trees, querying is more expensive in terms of time cost, but more accurate than DET-ONLY, which uses a single DE-Tree to answer queries. DET-LSH can control the trade-off between accuracy and efficiency by adjusting the number of DE-Trees, while DET-ONLY cannot do this. The performance of DET-ONLY shows that it is not suitable to support accurate ANN queries, demonstrating the importance of the of the LSH component to guarantee query accuracy. Overall, DET-LSH is

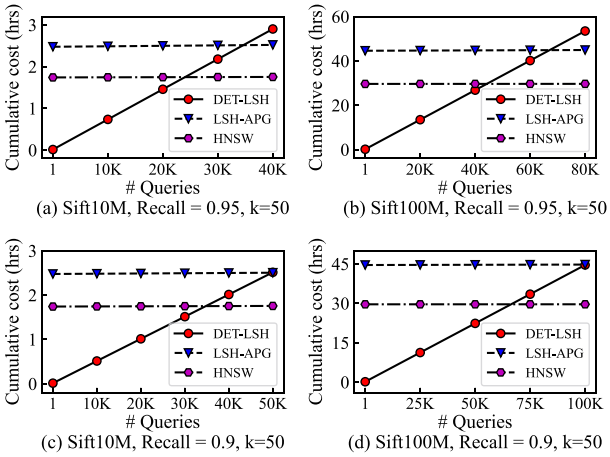


Figure 10: Cumulative query cost (first query includes indexing time).

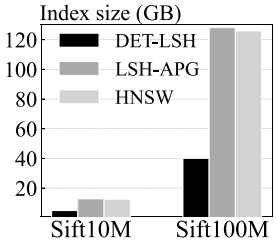


Figure 11: Index size.

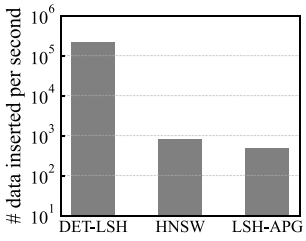


Figure 12: Update efficiency.

more advantageous than DET-ONLY. (3) DET-LSH achieves the best trade-off between efficiency and accuracy. As shown in Figure 7, compared with other LSH-based methods, DET-LSH consumes the least time to achieve the same recall or overall ratio.

6.3.3 Scalability. A method has good scalability if it performs well on datasets of different cardinalities. To investigate the scalability of all methods, we randomly select different number of points from the *Microsoft SPACEV500M* dataset and compare the indexing and query performance of all methods under default parameter settings. Figure 8 shows the results. We have the following observations: (1) Although the indexing and query times increase with the cardinality for all methods, DET-LSH grows much slower than other LSH-based methods due to the efficiency of DE-Tree (Figure 8(a) and Figure 8(b)). DET-ONLY constructs indexes and answers queries faster, but the accuracy is much less than other methods. (2) The recall and overall ratio are relatively stable for all methods. The reason is that the data distribution does not change significantly with the cardinality because we select points randomly. To sum up, DET-LSH has better scalability than other LSH-based methods.

6.3.4 Effect of k. To investigate the effect of k, we evaluate the performance of all methods under different k. Since changing k has little impact on query time, and has no impact on indexing time, we only report the results on recall and overall ratio, shown in Figure 9. We make the following observations: (1) As k increases, the query accuracy of all methods decreases slightly. The reason is that the number of candidate points does not change with k. A larger k means it is more likely to miss the exact NN points. (2) DET-LSH consistently exhibits the best performance among all competitors.

6.4 Comparison with Graph-based Methods

In this section, we compare DET-LSH to graph-based methods [4, 21, 38, 47, 56]. Nevertheless, LSH-based and graph-based methods have different design principles and characteristics [16, 32, 59], making them suitable to different application scenarios. In particular, graph-based methods only support ng-approximate answers [16], that is, they do not provide any quality guarantees on their results. It is important to emphasize that DET-LSH has to pay the cost of providing guarantees for its answers; graph-based methods, that do not provide any guarantees, do not pay this cost.

In our previous experiments, we demonstrated that DET-LSH outperforms other LSH-based methods. In this section, we compare DET-LSH to HNSW [38], the state-of-the-art graph-based method [16]. In addition, we compare to a hybrid method, LSH-APG [65], which uses LSH to retrieve a high-quality entry point for the subsequent search in an Approximate Proximity Graph (APG).

In terms of indexing and query efficiency, Figure 10 shows the cumulative query costs of DET-LSH, HNSW, and LSH-APG, where the cost of the first query also includes the indexing time. We observe that, as expected, DET-LSH has an advantage in indexing efficiency: it creates the index and answers 30K-70K queries before the best competitor (i.e., HNSW) answers its first query. This behavior is partly explained by the more succinct index structure of DET-LSH. Figure 11 shows that the DET-LSH index is almost 3x smaller in size than the index constructed by the competitors. Finally, Figure 12 shows the update efficiency of these methods, by measuring the number of data points per second when inserting the last 10M points of the *Sift100M* dataset into the existing indexes. In this scenario that involves updates, DET-LSH is 2-3 orders of magnitude faster than HNSW and LSH-APG.

In summary, LSH-based methods (DET-LSH) have distinct characteristics and different advantages when compared to graph-based methods, pure (such as HNSW) or hybrid (such as LSH-APG), making each method better suited for different scenarios.

7 CONCLUSIONS

In this paper, we have proposed a novel LSH scheme, called DET-LSH, to efficiently and accurately answer c-ANN queries in high-dimensional spaces with strong theoretical guarantees. DET-LSH combines the ideas of BC and DM methods, constructing multiple index trees to support range queries based on the Euclidean distance metric, which reduces the probability of missing exact NN points and improves query accuracy. To efficiently support range queries in DET-LSH, we designed a dynamic encoding-based tree called DE-Tree, which outperforms data-oriented partitioning trees used in existing LSH-based methods, especially in very large-scale datasets. Extensive experiments demonstrate that DET-LSH outperforms the state-of-the-art LSH-based methods in both efficiency and accuracy.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (NSFC) under the grant number 62202450.

REFERENCES

[1] Alexandr Andoni. 2005. LSH Algorithm and Implementation (E2LSH). https://web.mit.edu/andoni/www/LSH/index.html.

[2] Alexandr Andoni and Ilya Razenshteyn. 2015. Optimal data-dependent hashing for approximate near neighbors. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. 793–801.

[3] Mahendra Awale and Jean-Louis Reymond. 2018. Polypharmacology browser PPB2: target prediction combining nearest neighbors with machine learning. *Journal of chemical information and modeling* 59, 1 (2018), 10–17.

[4] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. ELPIS: Graph-Based Similarity Search for Scalable Data Science. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1548–1559.

[5] Rudolf Bayer and Edward McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 107–141.

[6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[7] Christian Böhm. 2000. A cost model for query processing in high dimensional data spaces. *ACM Transactions on Database Systems (TODS)* 25, 2 (2000), 129–178.

[8] Allan Borodin, Rafail Ostrovsky, and Yuval Rabani. 1999. Lower bounds for high dimensional nearest neighbor search and related problems. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*. 312–321.

[9] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. 2014. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. *Knowledge and information systems* 39, 1 (2014), 123–151.

[10] Lawrence Cayton. 2008. Fast nearest neighbor retrieval for bregman divergences. In *Proceedings of the 25th international conference on Machine learning*. 112–119.

[11] Manos Chatzakis, Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, and Botao Peng. 2023. Odyssey: A Journey in the Land of Distributed Data Series Similarity Search. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1140–1153.

[12] Paolo Ciaccia, Marco Patella, Pavel Zezula, et al. 1997. M-tree: An efficient access method for similarity search in metric spaces. In *Vldb*, Vol. 97. Citeseer, 426–435.

[13] Sanjoy Dasgupta and Yoav Freund. 2008. Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 537–546.

[14] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[15] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules against data series similarity search. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2005–2018.

[16] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *Proc. VLDB Endow.* 13, 3 (2019), 403–420. https://doi.org/10.14778/3368289.3368303

[17] Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, and George Paterakis. 2023. FreSh: A Lock-Free Data Series Index. In *42nd International Symposium on Reliable Distributed Systems, SRDS*. IEEE, 209–220. https://doi.org/10.1109/SRDS60354.2023.00029

[18] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. 2001. Approximate nearest neighbor searching in multimedia databases. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 503–511.

[19] Raul Castro Fernandez, Pranav Subramaniam, and Michael J Franklin. 2020. Data market platforms: trading data assets to solve data problems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1933–1947.

[20] Cong Fu and Deng Cai. 2016. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228* (2016).

[21] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment* 12, 5 (2019), 461–474.

[22] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 541–552.

[23] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[24] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[25] Alexander Hinneburg, Charu C Aggarwal, and Daniel A Keim. 2000. What is the nearest neighbor in high dimensional spaces?. In *26th Internat. Conference on Very Large Databases*. 506–515.

[26] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 9, 1 (2015), 1–12.

[27] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.

[28] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906* (2020).

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

[30] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *Proceedings of the VLDB Endowment* 11, 6 (2018).

[31] Yifan Lei, Qiang Huang, Mohan Kankanhalli, and Anthony KH Tung. 2020. Locality-sensitive hashing scheme based on longest circular co-substring. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2589–2599.

[32] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.

[33] Michele Linardi and Themis Palpanas. 2018. Scalable, variable-length similarity search in data series: The ULISSE approach. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2236–2248.

[34] Wanqi Liu, Hanchen Wang, Ying Zhang, Wei Wang, Lu Qin, and Xuemin Lin. 2021. EI-LSH: An early-termination driven I/O efficient incremental c-approximate nearest neighbor search. *The VLDB Journal* 30 (2021), 215–235.

[35] Yingfan Liu, Jiangtao Cui, Zi Huang, Hui Li, and Heng Tao Shen. 2014. SK-LSH: an efficient index structure for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 7, 9 (2014), 745–756.

[36] Kejing Lu and Mineichi Kudo. 2020. R2LSH: A nearest neighbor search scheme based on two-dimensional projected spaces. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1045–1056.

[37] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: approximate nearest neighbor search via virtual hypersphere partitioning. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1443–1455.

[38] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[39] Themis Palpanas. 2015. Data Series Management: The Road to Big Sequence Analytics. *SIGMOD Record* (2015).

[40] Themis Palpanas and Volker Beckmann. 48(3), 2019. Report on the First and Second Interdisciplinary Time Series Analysis Workshop (ITISA). *SIGREC* (48(3), 2019).

[41] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2018. ParIS: The Next Destination for Fast Data Series Indexing and Query Answering. *IEEE BigData* (2018).

[42] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. Messi: In-memory data series indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 337–348.

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

[44] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. Fast data series indexing for in-memory data. *VLDB J.* 30, 6 (2021).

[45] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. SING: Sequence Indexing Using GPUs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1883–1888.

[46] Yun Peng, Byron Choi, Tsz Nam Chan, and Jianliang Xu. 2022. Lan: Learning-based approximate k-nearest neighbor search in graph databases. In *2022 IEEE 38th international conference on data engineering (ICDE)*. IEEE, 2508–2521.

[47] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[48] Jin Shieh and Eamonn Keogh. 2008. iSAX: indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 623–631.

[49] Chanop Silpa-Anan and Richard Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 1–8.

[50] Tomáš Skopal, Jaroslav Pokorný, and Václav Snášel. 2005. Nearest Neighbours Search using the PM-tree. In *Database Systems for Advanced Applications: 10th International Conference, DASFAA 2005, Beijing, China, April 17-20, 2005. Proceedings 10*. Springer, 803–815.

[51] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment* (2014).

[52] Yukihiro Tagami. 2017. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 455–464.

[53] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 563–576.

[54] Yao Tian, Ziyang Yue, Ruiyuan Zhang, Xi Zhao, Bolong Zheng, and Xiaofang Zhou. 2023. Approximate Nearest Neighbor Search in High Dimensional Vector Databases: Current Research and Future Directions. *IEEE Data Engineering Bulletin* 47, 3 (2023).

[55] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2023. DB-LSH 2.0: Locality-Sensitive Hashing With Query-Based Dynamic Bucketing. *IEEE Transactions on Knowledge and Data Engineering* (2023).

[56] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1964–1978.

[57] Qitong Wang and Themis Palpanas. 2023. SEAnet: A Deep Learning Architecture for Data Series Similarity Search. *IEEE Trans. Knowl. Data Eng.* 35, 12 (2023), 12972–12986.

[58] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A data-adaptive and dynamic segmentation index for whole matching on time series. *VLDB* (2013).

[59] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph- and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *IEEE Data Eng. Bull.* 47, 3 (2023), 3–21.

[60] Zeyu Wang, Qitong Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Dumpy: A Compact and Adaptive Index for Large Data Series Collections. *Proc. ACM Manag. Data* 1, 1 (2023), 111:1–111:27.

[61] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, Vol. 98. 194–205.

[62] Jiuqi Wei, Ying Li, Yufan Fu, Youyi Zhang, and Xiaodong Li. 2023. Data Interoperating Architecture (DIA): Decoupling Data and Applications to Give Back Your Data Ownership. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 438–447.

[63] Kevin Wellenzohn, Michael H Böhlen, Sven Helmer, Antoine Pietri, and Stefano Zacchiroli. 2023. Robust and scalable content-and-structure indexing. *The VLDB Journal* 32, 4 (2023), 689–715.

[64] Peter N Yianilos. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Soda*, Vol. 93. 311–21.

[65] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1979–1991.

[66] Bolong Zheng, Zhao Xi, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S Jensen. 2020. PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search. *Proceedings of the VLDB Endowment* 13, 5 (2020), 643–655.

[67] Yuxin Zheng, Qi Guo, Anthony KH Tung, and Sai Wu. 2016. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *Proceedings of the 2016 International Conference on Management of Data*. 2023–2037.

[68] Vladimir M Zolotarev. 1986. *One-dimensional stable distributions*. Vol. 65. American Mathematical Soc.

[69] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: the adaptive data series index. *The VLDB Journal* 25 (2016), 843–866.