PM-LSH: a fast and accurate in-memory framework for high-dimensional approximate NN and closest pair search

Zheng, Bolong; Zhao, Xi; Weng, Lianggui; Nguyen, Quoc Viet Hung; Liu, Hang; Jensen, Christian S.

Published in:
VLDB Journal

DOI (link to publication from Publisher):
10.1007/s00778-021-00680-7

Publication date:
2022

Document Version
Accepted author manuscript, peer reviewed version

Noname manuscript No.
(will be inserted by the editor)

PM-LSH: a fast and accurate in-memory framework for high-dimensional approximate NN and closest pair search

Bolong Zheng · Xi Zhao · Lianggui Weng · Nguyen Quoc Viet Hung · Hang Liu · Christian S. Jensen

Received: date / Accepted: date

Abstract Nearest neighbor (NN) search is inherently computationally expensive in high-dimensional spaces due to the curse of dimensionality. As a well-known solution, locality-sensitive hashing (LSH) is able to answer c-approximate NN (c-ANN) queries in sublinear time with constant probability. Existing LSH methods focus mainly on building hash bucket-based indexing such that the candidate points can be retrieved quickly. However, existing coarse-grained structures fail to offer accurate distance estimation for candidate points, which translates into additional computational overhead when having to examine unnecessary points. This in turn reduces the performance of query processing. In contrast, we propose a fast and accurate in-memory LSH framework, called PM-LSH, that aims to compute c-ANN queries on large-scale, high-dimensional datasets. First, we adopt a simple yet effective PM-tree to index the data points. Second, we develop a tunable confidence interval to achieve accurate distance estimation and guarantee high result quality. Third, we propose an efficient algorithm on top of the PM-tree to improve the performance of computing c-ANN queries.

In addition, we extend PM-LSH to support closest pair (CP) search in high-dimensional spaces. We again adopt the PM-tree to organize the points in a low-dimensional space, and we propose a branch and bound algorithm together with a radius pruning technique to improve the performance of computing c-approximate closest pair (c-ACP) queries.

Extensive experiments with real-world data offer evidence that PM-LSH is capable of outperforming existing proposals with respect to both efficiency and accuracy for both NN and CP search.

Bolong Zheng, Xi Zhao and Lianggui Weng
Huazhong University of Science and Technology, Wuhan, China
E-mail: {bolongzheng,zhaoxi,liangguiweng}@hust.edu.cn

Nguyen Quoc Viet Hung
Griffith University, Gold Coast, Australia
E-mail: quocviethung.nguyen@griffith.edu.au

Hang Liu
Stevens Institute of Technology, Hoboken, USA
E-mail: hang.liu@stevens.edu

Christian S. Jensen
Aalborg Universigy, Aalborg, Denmark
E-mail: csj@cs.aau.dk

1 Introduction

Nearest neighbor (NN) querying in high-dimensional spaces is classic functionality that is used in a wide variety of important applications, such as sequence matching [1], recommendation [14], similar-item retrieval [30], and de-duplication [38], to name but a few. Let $\mathcal{D}$ be a set of points in d-dimensional space $\mathbb{R}^d$. Given a query point q, an NN query returns a point o^* in $\mathcal{D}$ such that its Euclidean distance to q is the minimum among all points in $\mathcal{D}$.

While the exact NN query in low-dimensional space already has efficient solutions [6, 8], providing an efficient solution for large, high-dimensional datasets remains a challenge, as both the query time and the space cost may increase exponentially with respect to the dimensionality. This phenomenon is called the "curse of dimensionality." Fortunately, it frequently suffices to find an approximate nearest neighbor (ANN). Given an approximation ratio c ($c > 1$) and a query point q, a c-ANN query returns a point o whose distance to q is at most cr^*, where r^* is the distance between q and its exact NN.

A widely adopted locality-sensitive hashing (LSH) method enables computing c-ANN queries in sublinear time with constant probability. Generally, LSH maps the points in the dataset to buckets in hash tables by using a set of predefined hash functions that are designed to be locality-sensitive so that close points are hashed to the same bucket with high probability. A query is answered by examining the points that are hashed to the same bucket as the query point, or to similar buckets. Based on their main ideas, we classify the mainstream LSH methods into three categories: 1) Probing Sequence-based (PS) approaches [33,35,36]; 2) Radius Enlargement-based (RE) approaches [18,27,48]; and 3) Metric Indexing-based (MI) approaches [47]. PS approaches use a carefully derived probing sequence to examine multiple hash buckets that are likely to contain the nearest neighbor of a query. RE approaches process a sequence of range queries by enlarging the query range repeatedly until a qualified point is found. In MI approaches, the points are transformed into a low-dimensional, so-called projected space. The coordinates of a point in the projected space are the point's hash values. MI approaches then use a metric index to organize the points such that the distance between two points in the projected space can be used to approximate the distance between them in the original space.

When evaluating the performance of LSH methods, many pertinent performance metrics for c-ANN search exist, including efficiency, accuracy, memory consumption, and preprocessing overhead. Among these, both *efficiency* and *accuracy* are important metrics since a desirable algorithm should return results as soon as possible with a quality that is as high as possible, while the memory consumption and preprocessing overhead must be tolerable in the setting of a commodity machine. The performance of LSH depends on two aspects: 1) the estimation of distances between the query point and candidate points; and 2) the probing order of buckets/points. It is proved [47] that the ratio of the projected distance to the original distance between any two points follows a χ^2 distribution. Therefore, if we are able to estimate the distance between two points accurately, we are able to find high-quality candidates. In addition, a well-designed index structure is required to quickly locate high-quality candidates.

However, the existing LSH methods suffer from either inaccurate distance estimation or unnecessary point probing overhead. For instance, SRS [47] is the state-of-the-art algorithm that uses an R-tree to index the points in the projected space. By searching the R-tree, SRS is able to iteratively return the next nearest point to q. The problem is that finding the next exact NN in an R-tree generally causes additional computational overhead, while the next NN is not necessarily the best next candidate in the original space. Next, Multi-Probe [35] iteratively identifies the next hash bucket to be examined that has the least distance to q. However, most of the points in the identified buckets have to be probed due to poor estimation of the distance between q and the candidate point. Finally, QALSH [27] shares the same issue as Multi-Probe, and it uses a large number of hash functions that may incur high space consumption.

We propose a fast and accurate in-memory framework, called PM-LSH, for computing c-ANN queries on large-scale, high-dimensional datasets. The framework consists of three components, namely data partitioning, distance estimation, and point probing. First, we adopt the simple yet effective PM-tree [46] to index the points in the projected space. Second, in order to improve the distance estimation accuracy, we exploit the strong relationship between the original and projected distance of any two points, and we develop a tunable confidence interval on the projected distance w.r.t. a given original distance. Third, we propose an efficient algorithm to search the PM-tree with a sequence of range queries with increasingly large radius. PM-LSH is able to achieve both high efficiency and high accuracy when compared with existing LSH methods.

We extend the PM-LSH technique to solve another classical problem, approximate closest pair (CP) search in high dimensional spaces. Like NN search, CP search is used in a wide range of settings, such as unsupervised classification or clustering [42], user pattern similarity search [55], and geographic information systems [22], to name but a few. For a given approximation ratio c ($c > 1$) and a dataset $\mathcal{D}$, a c-approximate closest pair (c-ACP) query returns a point pair (o_1, o_2) with distance at most cr^*, where r^* is the distance of the exact closest pair in $\mathcal{D}$. Early studies mainly adopt space partitioning indexing techniques to solve exact CP queries in two or three dimensions [12, 13, 26, 29, 44, 45]. However, these methods cannot be extended directly to support high-dimensional CP queries efficiently due to the curse of dimensionality. Therefore, improved indexes are proposed to address the effects of dimensionality [17, 19, 31, 41]. Nonetheless, when faced with hundreds or thousands of dimensions, the performance of these methods still degenerates to nearly brute-force performance. Thus, another direction is to use dimension reduction methods to solve c-ACP, such as LSH or random projection. For instance, the LSB-tree [49] uses a compound hash function to project points into a low-dimensional space and adopts the Z-curve to transform projected points into one-dimensional values that are indexed by a B-tree. The candidate point pairs are generated from

points with the same Z-values. To improve the query accuracy, $L = O(\sqrt{n})$ B-trees are built, which yields a large space consumption. Next, ACP-P [7] projects the points directly into a one-dimensional space. The points with close distances in the projected space are considered as candidate point pairs. However, the distance estimation is inaccurate and leads to unnecessary candidate verification.

To compute approximate CP queries, we still employ the PM-tree to index the points in the projected space, which provides an accurate distance estimation for point pairs. Next, we adopt a branch and bound method combined with a radius pruning technique to improve the query efficiency, which enables generation of sufficient candidate pairs with only a small space consumption. We also note that our method is tunable and enables different trade-offs between query accuracy and query efficiency.

The major contributions are summarized as follows:

- We present a unified interpretation of the existing mainstream LSH methods and thoroughly analyze the competitors in relation to our method.
- We propose an accurate and fast method called PM-LSH for c-ANN querying of large-scale, high-dimensional datasets. First, we use the PM-tree to index the points in the projected space. Second, we develop a tunable confidence interval for distance estimation. Third, we propose a c-ANN query algorithm that uses the PM-tree.
- We extend the PM-LSH to support CP queries. First, we still employ the PM-tree to index the points in the projected space. Next, we propose a branch and bound algorithm together with a radius pruning technique for computing c-ACP queries.
- We conduct an extensive performance study using real datasets that covers the state-of-the-art algorithms, which indicates that PM-LSH is efficient as well as accurate in terms of both the overall ratio and recall for both NN and CP search.

The paper extends its conference version [53] in several respects. Key extensions include (1) the extension of PM-LSH to support CP queries, (2) the coverage of related work on high-dimensional CP search, and (3) the paper's report on the experimental evaluations of the corresponding proposals. In addition, other parts of the paper have been revised when compared to the conference version.

The rest of the paper is organized as follows. Section 2 presents the problem setting and preliminaries. Section 3 introduces a unified LSH framework, followed by our PM-LSH framework in Section 4. Sections 5 and 6 introduce the NN and CP query processing based on PM-LSH, respectively. Section 7 covers experimental studies that offer insight into the performance of the proposed PM-LSH and the main competitors for both NN and CP search. Section 8 reviews related work. Finally, Section 9 concludes the paper.

Table 1: Summary of Notations

Notation	Definition
$\mathcal{D}$	Dataset of points in $\mathbb{R}^d$
$n = \lvert\mathcal{D}\rvert$	Dataset cardinality
d	Dataset dimensionality
o	A point in $\mathcal{D}$
o'	A point o in the projected space
c	Approximation ratio
$h(o)$, $h^*(o)$	Hash functions
m	The number of hash functions
T	The number of candidate points or pairs
M	The node capacity of the PM-tree

2 Preliminaries

We proceed to present the problem definitions of approximate nearest neighbor (NN) and closest pair (CP) search, and the basic idea of LSH. Frequently used notation is summarized in Table 1.

2.1 Problem Definition

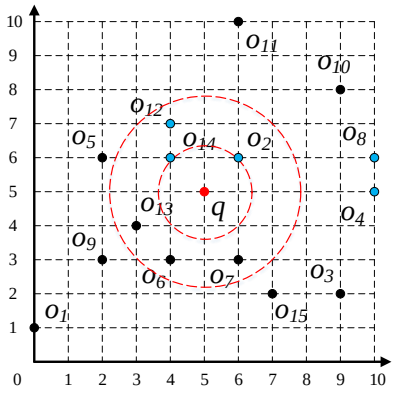
Let $\mathcal{D}$ be a set of points in d-dimensional space $\mathbb{R}^d$ with cardinality $|\mathcal{D}| = n$. Let $\|o_1, o_2\|$ denote the Euclidean distance between points $o_1, o_2 \in \mathcal{D}$. We define approximate nearest neighbor and closest pair queries in turn.

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

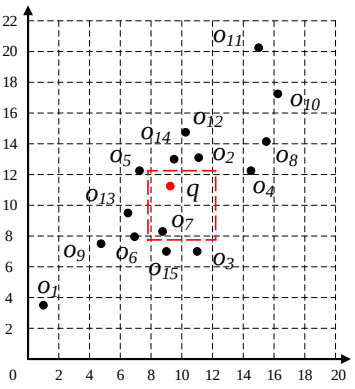
We generalize the c-ANN query to the (c, k)-ANN query that returns k approximate nearest points.

Definition 2 ((c, k)-ANN Query) Assume we have a query point q, an approximation ratio $c > 1$, and a positive integer k. Let o_i^* be the i-th exact nearest neighbor of q in $\mathcal{D}$. A (c, k)-approximate nearest neighbor query returns a sequence of k points $\langle o_1, o_2, \ldots, o_k \rangle$ such that for each o_i, we have $\|q, o_i\| \leq c \cdot \|q, o_i^*\|$, $i \in [1, k]$.

Definition 3 (c-ACP Query) Assume we have an approximate ratio $c > 1$, and let (o_1^*, o_2^*) be the exact closest pair in $\mathcal{D}$. A c-approximate closest pair query returns a point pair $(o_1, o_2) \in \mathcal{D} \times \mathcal{D}$ such that $\|o_1, o_2\| \leq c \cdot \|o_1^*, o_2^*\|$.



(a) Original Space



(b) Projected Space

ID	(x,y)	h*(o)	ID	(x,y)	h*(o)
q	(5,5)	(9.5,11.5)	o_8	(10,6)	(15.4,14.2)
o_1	(0,1)	(0.9,3.7)	o_9	(2,3)	(4.7,7.5)
o_2	(6,6)	(11.4,13.4)	o_{10}	(9,8)	(16.2,17.4)
o_3	(9,2)	(10.8,7.2)	o_{11}	(6,10)	(15.0,20.2)
o_4	(10,5)	(14.5,12.5)	o_{12}	(4,7)	(10.3,14.7)
o_5	(2,6)	(7.4,12.6)	o_{13}	(3,4)	(6.6,9.4)
o_6	(4,3)	(6.7,7.9)	o_{14}	(4,6)	(9.4,13.0)
o_7	(6,3)	(8.7,8.3)	o_{15}	(7,2)	(8.8,6.8)

(c) Points and Hash Values

Fig. 1: Running Example with $h_1(o) = \lfloor \frac{\vec{a_1} \cdot \vec{o}}{4} \rfloor$, $h_2(o) = \lfloor \frac{\vec{a_2} \cdot \vec{o} + 2}{4} \rfloor$ and $\vec{a_1} = [1.0, 0.9]$, $\vec{a_2} = [0.2, 1.7]$

We generalize the c-ACP query to the (c, k)-ACP query that returns k approximate closest pairs.

Definition 4 ((c, k)-ACP Query) Assume we have an approximate ratio $c > 1$, and a positive integer k. Let $(o^*_{i,1}, o^*_{i,2})$ be the i-th exact closest pair in $\mathcal{D}$. A (c, k)-approximate closest pair query returns a sequence of k point pairs $\langle (o_{1,1}, o_{1,2}), (o_{2,1}, o_{2,2}), \ldots, (o_{k,1}, o_{k,2}) \rangle$ such that for each $(o_{i,1}, o_{i,2})$, we have $\|o_{i,1}, o_{i,2}\| \leq c \cdot \|o^*_{i,1}, o^*_{i,2}\|$, $i \in [1, k]$.

Example 1 *As shown in Fig. 1(a), the exact NNs of query q are o_2 and o_{14} with distance $\sqrt{2}$. For a 2-ANN query, any point whose distance to q is within $2\sqrt{2}$ can be considered as a result, i.e., any object in the set $\{o_2, o_{14}, o_{12}, o_{13}, o_6, o_7\}$.*

The exact CPs are (o_4, o_8) and (o_{12}, o_{14}) with distance 1. For a 2-ACP query, any point pair whose distance is within 2 can be considered as a result, i.e., any pair in the set $\{(o_6, o_7), (o_4, o_8), (o_6, o_9), (o_6, o_{13}), (o_9, o_{13}), (o_2, o_{14}), (o_5, o_{14}), (o_{12}, o_{14}), (o_3, o_{15}), (o_7, o_{15})\}$.

2.2 Basic Locality Sensitive Hashing

We first introduce the LSH scheme, and then explain how to answer the (r, c)-ball cover and c-ANN queries using the basic LSH [3, 15].

Hash Family. Given a distance r, an approximation ratio $c > 1$, two probability values p_1 and p_2, where $p_1 > p_2$, a family $\mathcal{H} = \{h : \mathbb{R}^d \to U\}$ is called (r, cr, p_1, p_2)-locality sensitive, if for any $o_1, o_2 \in \mathbb{R}^d$, it satisfies both of the following conditions:

1. If $\|o_1, o_2\| \leq r$ then $Pr[h(o_1) = h(o_2)] \geq p_1$
2. If $\|o_1, o_2\| \geq cr$ then $Pr[h(o_1) = h(o_2)] \leq p_2$

A well-adopted hash function is formally defined as follows:

$$h(o) = \lfloor \frac{\vec{a} \cdot \vec{o} + b}{w} \rfloor, \quad (1)$$

where $\vec{o}$ is the vector representation of a point $o \in \mathbb{R}^d$, $\vec{a}$ is a d-dimensional vector where each dimension is drawn independently from a p-stable distribution [15], b is a real number uniformly and randomly drawn from $[0, w)$, and w is a user-specified constant. The 2-stable distribution is the normal distribution.

Formally, let $\tau = \|o_1, o_2\|$ and let $f(\cdot)$ denote the normal probability distribution function (pdf). We then have:

$$p(\tau) = Pr[h(o_1) = h(o_2)] = \int_0^w \frac{1}{\tau} \cdot f(\frac{t}{\tau}) \cdot (1 - \frac{t}{w})\, dt \quad (2)$$

The intuition behind Eq. 2 is that, given a fixed w, the collision probability of two hash values $h(o_1)$ and $h(o_2)$ grows as the distance $\|o_1, o_2\|$ decreases. Therefore, $h(\cdot)$ in Eq. 1 is (r, cr, p_1, p_2)-sensitive with $p_1 = p(r)$ and $p_2 = p(cr)$.

Before we consider how to answer the c-ANN query, we define an (r, c)-ball cover query that can be answered directly by an (r, cr, p_1, p_2)-sensitive hash family.

Definition 5 ((r, c)-BC Query) Assume a query point q, a distance threshold r, and an approximation ratio $c > 1$. Let $B(q, r)$ denote a ball centered at q with radius r. An (r, c)-ball cover query returns the following result:

1. If $B(q, r)$ covers at least one point in $\mathcal{D}$, it returns a point in $B(q, cr)$;
2. If $B(q, cr)$ covers no points in $\mathcal{D}$, it returns nothing.

E2LSH [3] is a seminal solution that forms L hash tables and randomly chooses m hash functions for each hash table. By concatenating the m hash functions, a

compound hash function $G(o) = (h_1(o), \ldots, h_m(o))$ is formed in each hash table, and each point $o \in \mathcal{D}$ is stored in a hash bucket based on $G(o)$. Given a query point q, E2LSH computes $G(q)$ and enumerates the points in the corresponding hash bucket. In all L hash tables, it examines at most $3L$ points and returns a point o if $\|q, o\| \leq cr$. By setting $m = \log_{1/p_2} n$ and $L = 1/p_1^k$, the (r, c)-BC query can be answered correctly with at least constant probability.

From (r, c)-BC to c-ANN. It is easy to see that the ball cover query can be considered as a decision version of the approximate NN query. Processing a sequence of (r, c)-BC queries with $r = 1, c, c^2, \ldots, x$, once a point is returned, we take it as a result of the ANN query. Interestingly [28], the ANN query can be answered with approximation ratio c^2, i.e., c^2-ANN.

Example 2 *In the example in Fig. 1, we choose $m = 2$ hash functions $h_1(o) = \lfloor \frac{\vec{a_1} \cdot \vec{o}}{4} \rfloor$, $h_2(o) = \lfloor \frac{\vec{a_2} \cdot \vec{o} + 2}{4} \rfloor$ with $\vec{a_1} = [1.0, 0.9]$, $\vec{a_2} = [0.2, 1.7]$, $b_1 = 0$, $b_2 = 2$, and $w = 4$. For simplicity, we only construct $L = 1$ hash table. Figs. 1(b) and 1(c) show the coordinates of the objects in the projected space. To answer a $(1, 2)$-BC query with $r = 1$ and $c = 2$, we first compute $G(q) = (h_1(q), h_2(q)) = (2, 2)$. Then we search the hash bucket $(2, 2)$ that is indicated by a red rectangle; the $(1, 2)$-BC query returns o_7. As o_{14} is the exact NN with $\|q, o_{14}\| = \sqrt{2}$ and $\|q, o_7\| = \sqrt{5} < 4 \times \sqrt{2}$, we have that o_7 is a result of the 4-ANN query for q.*

3 A Unified Interpretation of LSH

We proceed to introduce the main competitors and give a unified interpretation.

3.1 Main Competitors

Probing Sequence (PS). The representative PS methods include Multi-Probe [35,36] and GQR [33] that use a carefully derived probing sequence to examine multiple hash buckets that are likely to contain the nearest neighbors of a query point. Unlike the basic LSH that builds L hash tables and checks only one hash bucket in each hash table, PS probes multiple nearby buckets in order to achieve higher recall with fewer hash tables. Given a query point q, PS adopts a "generate-to-probe" paradigm that iteratively generates the next hash bucket to be examined with the least distance to q in the remaining buckets.

Radius Enlargement (RE). This category mainly includes the LSB-Tree [48], C2LSH [18], and QALSH [27]. These do not build multiple hash tables based on different radii. Generally, RE methods build a hash table like the basic LSH and processes a sequence of (r, c)-BC queries by enlarging $r = 1, c, c^2, \ldots, x$ when a c-ANN query is issued. Suppose $r_i = c^i$ and $r_0 = 1$. It has been shown [18] that $h^{r_i}(\cdot) = \lfloor \frac{h(\cdot)}{r_i} \rfloor$ is (r_i, cr_i, p_1, p_2)-sensitive. Instead of building multiple hash tables with corresponding hash functions $h^{r_i}(\cdot)$ to handle (r_i, cr_i)-BC queries, RE methods adopt the smart idea of "virtual rehashing" to avoid consuming unnecessary space. For the $(1, c)$-BC query, RE probes the hash bucket $h(q)$. For the remaining (r_i, cr_i)-BC queries, RE probes r_i^m hash buckets near $h(q)$ in the original hash table in the i-th iteration. Note that among these r_i^m buckets, r_{i-1}^m buckets were already examined in the previous iteration. Interestingly, it is easy to see that the r_i^m hash buckets in the original hash table correspond to the hash bucket $h^{r_i}(q)$ in the hash table w.r.t. $h^{r_i}(\cdot)$.

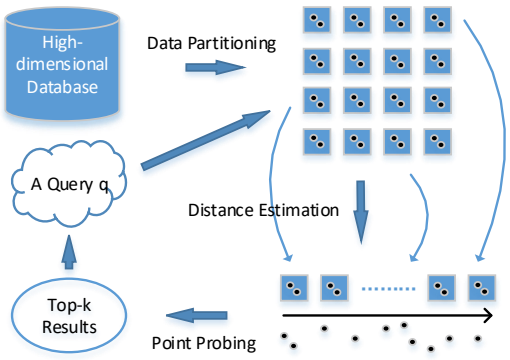


Fig. 2: Unified LSH Framework

Metric Indexing (MI). SRS [47] is the state-of-the-art algorithm that projects the points from the original d-dimensional space into a lower m-dimensional projected space by using m hash functions. It utilizes an R-tree to index the points based on their hash values and uses the Euclidean distance between two points in the projected space to approximate their distance in the original space. The intuition is that the points close to the query point q in the projected space are also likely close to q in the original space. SRS repeatedly calls an incSearch function that utilizes the R-tree to return the next nearest point to q in the projected space.

3.2 A Way of Probing

We proceed to introduce a unified interpretation of existing LSH methods as shown in Fig. 2, which consists of three components, namely data partitioning, distance estimation, and point probing.

Generally, we adopt a random projection $h(o)$ as the locality sensitive hash functions:

$$h^*(o) = \vec{a} \cdot \vec{o} \tag{3}$$

By using $h^*(o)$, the points in the original space are mapped into a projected space, as shown in Figs. 1(a) and 1(b). Let $o' = [h_1^*(o), \ldots, h_m^*(o)]$ denote point o in the projected space. For any two points o_1 and o_2, let $r = \|o_1, o_2\|$ and $r' = \|o_1', o_2'\|$ denote the distance between o_1 and o_2 in the original and in the projected space, respectively. In addition, we let $\rho(o_1, o_2)$ denote an m-dimensional vector, where each dimension is the hash value difference between points o_1 and o_2, i.e., $\rho_i = h_i^*(o_1) - h_i^*(o_2) = o_1'[i] - o_2'[i]$. Therefore, we have $r' = \sqrt{\sum_{i=1}^m \rho_i^2}$.

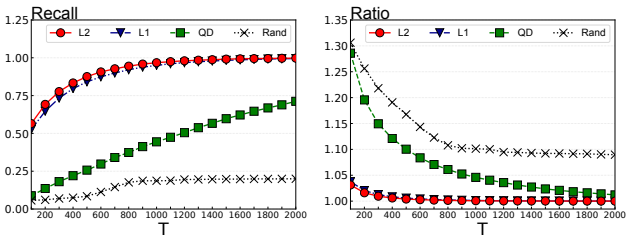
Based on a property of a 2-stable distribution, for any d real numbers $o[1], \ldots, o[d]$, independent and identically distributed (i.i.d.) random variables $X_1, \ldots, X_d$ (corresponding to $\vec{a}$) following the 2-stable distribution, $\sum_i o[i] \cdot X_i$ has the same distribution as the variable $(\sum_{i=1}^d o[i]^2)^{1/2} \cdot X$, where X is a random variable with distribution $N(0, 1)$. For any two points o_1 and o_2, since $\rho = h^*(o_1) - h^*(o_2) = \vec{a} \cdot (\vec{o_1} - \vec{o_2})$, we know that ρ is a random variable with distribution $r \cdot X$. In other words, ρ has distribution $N(0, r^2)$, i.e., $\frac{\rho}{r} \sim N(0, 1)$.

Lemma 1 *r'^2/r^2 follows the distribution $\chi^2(m)$.*

Proof If $Y_1, \ldots, Y_m$ are i.i.d. variables with $N(0, 1)$ then $\sum_{i=1}^m Y_i^2$ follows the χ^2 distribution with m degrees of freedom. Given m hash functions $h_1^*(\cdot), \ldots, h_m^*(\cdot)$, for any o_1 and o_2, we have $\rho_1, \ldots, \rho_m$. Thus, r'^2/r^2 follows the distribution $\chi^2(m)$.

Data Partitioning. After mapping the points into the projected space by using hash functions, the existing LSH methods adopt the "divide-and-conquer" paradigm that partitions the projected space into subspaces. When a query is issued, the regions that are likely to contain the results are probed, and finally the results of these regions are combined and returned. Generally, there are two kinds of data partitioning approaches in the existing LSH methods:

(1) Interval based Partitioning. The basic LSH constructs hash buckets based on $G(o)$, and each bucket can be viewed as an m-dimensional hypercube with equal side lengths w. Most of the LSH methods belong to this class, including Multi-Probe, LSB-Tree, C2LSH, and QALSH. Specifically, an LSB-Tree assigns each hypercube a Z-order value and stores the values in a B-tree. In contrast, QALSH does not physically build hypercubes, but stores the values of $h^*(o)$ in a B$^+$-tree. When a query arrives, the length-w intervals are formed virtually on the B$^+$-tree.



(a) Recall (b) Overall Ratio

Fig. 3: Comparison on Recall and Overall Ratio

(2) Metric Space Partitioning. SRS uses an R-tree to index all the points o' in the projected space such that incremental kNN search is supported. For in-memory processing, it is also able to use a Cover Tree. In our proposed PM-LSH, we partition the projected space using a PM-tree so that efficient range querying can be supported.

Distance Estimation. In order to accurately estimate distances, two aspects are considered, i.e., the distance estimator and the estimation granularity.

(1) Distance Estimator. As ρ follows distribution $N(0, r^2)$, for any o_1 and o_2, $\rho(o_1, o_2) = [\rho_1, \ldots, \rho_m]$. We estimate the value of r^2 by using r'^2 as follows.

Lemma 2 *$\hat{r}^2 = \frac{r'^2}{m}$ is an unbiased estimator of r^2.*

Proof Let $\hat{r}^2$ be the estimated value of r^2. We compute the expectation of r'^2 as follows.

$$E[r'^2] = E[\sum_{i=1}^m \rho_i^2] = \sum_{i=1}^m E[\rho_i^2] = mr^2$$

Therefore, we have $E[\hat{r}^2] = E[r'^2]/m = r^2$.

We also provide an interesting alternative proof that uses maximum likelihood estimation (MLE) [24]. MLE is a procedure for finding the value of one or more parameters for a given statistic that maximizes the known likelihood distribution. As $Pr[\rho = \rho_i] = \frac{1}{\sqrt{2\pi}r} \exp(-\frac{\rho_i^2}{2r^2})$, the probability that the hash value difference $\rho(o_1, o_2)$ between o_1 and o_2 equals $[\rho_1, \ldots, \rho_m]$ is computed as follows.

$$\begin{aligned} Pr[\rho(o_1, o_2) &= [\rho_1, \ldots, \rho_m]] \\ &= f(\rho_1, \ldots, \rho_m | \mu = 0, \sigma = r) \\ &= \prod_{i=1}^m (\frac{1}{\sqrt{2\pi}r})^m \exp(-\frac{\sum_{i=1}^m \rho_i^2}{2r^2}) \end{aligned}$$

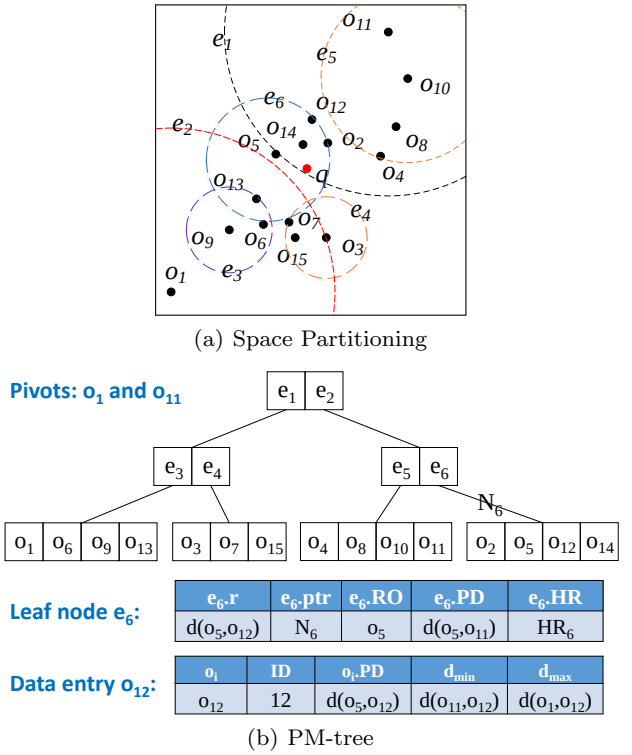
The objective of the maximum likelihood is to find an r such that the above probability is maximized. Given $\ln f = -\frac{1}{2}m\ln(2\pi) - m\ln r - \frac{\sum \rho_i^2}{2r^2}$ and $\frac{\partial(\ln f)}{\partial r} = -\frac{m}{r} + \frac{\sum \rho_i^2}{r^3} = 0$, we obtain $\hat{r}^2 = \frac{\sum_{i=1}^m \rho_i^2}{m} = \frac{r'^2}{m}$.

To evaluate the performance of our estimator in Lemma 2, i.e., $L_2 = r'$ (the same as our estimator when m is fixed), we compare it with three other distance estimators: L_1, QD [33], and Rand (assign a random value). We randomly sample a small dataset that contains 10K points from the *Trevi* dataset [34] and choose 100 points as query points. For each query point q, we first compute its exact 100NNs. With $m = 15$ hash functions, we compute the distances in the projected space between q and all the points based on different estimators. Then, we choose the top-T points with smallest estimated distances (T varies from 100 to 2,000). For each q, we compare its exact 100NNs with the 100NNs from the T points. Finally, we compute the average *recall* and *overall ratio* (discussed in Section 7) of these estimators. As shown in Fig. 3, we can see that our estimator has the best performance in terms of both the recall and overall ratio.

(2) Estimation Granularity. The distance estimation methods may use different granularities:

- Bucket to Bucket. The hash bucket based indexing methods, such as Multi-Probe, LSB-tree, and C2LSH, store points in hash buckets. When a query is issued, we first find its corresponding bucket and then decide which additional buckets to probe. Therefore, the quality of the distance estimation between buckets is affected by the bucket side length w.
- Point to Bucket. QALSH is an improved version of C2LSH that stores points by a B$^+$-tree instead of using a hash table. When a query q arrives, the length-w intervals are conceptually built on the B$^+$-tree with q as the center. So the distance estimation can be considered as between point q and bucket intervals.
- Point to Point. SRS uses the projected Euclidean distance between two points to estimate their original distance. This offers a finer precision than the previous two methods due to the fine granularity. Our PM-LSH also adopts this method.

Point Probing. Suppose we probe T points. In the hash bucket based indexing methods, we directly probe the points in the bucket, where the time cost is $O(T)$. The second approach is QALSH that searches the points in a B$^+$-tree, where the time cost is $O(\log n + T)$. Unlike the previous two approaches, SRS indexes the points with an R-tree and iteratively finds the next NN in the projected space. The time cost is $O(\log n \cdot T)$. Our PM-LSH can be considered as a combination of the second and third approaches in that we build a PM-tree in the projected space and execute range queries to retrieve points.



(a) Space Partitioning

(b) PM-tree

Fig. 4: The Structure of PM-LSH

4 The PM-LSH Framework

We proceed to present the details of the PM-LSH framework. As mentioned previously, the RE methods quickly probe the points stored in the hash buckets by enlarging the search radius, but suffer from inaccurate distance estimation due to a coarse-grained index structure, which translates into computational overhead when having to examine unnecessary points. In contrast, the MI methods index the points with an R-tree and iteratively return the next nearest point to q in the projected space. However, finding the next exact NN in an R-tree is also computationally costly, and the next NN is not necessarily the best next candidate in the original space. To achieve the best of both worlds, PM-LSH combines the ideas of the RE and MI methods. To achieve both efficiency and accuracy, we adopt the PM-tree instead of the R-tree to index the points in the projected space and execute a sequence of range queries with increasingly large radius.

Next, we briefly describe how to construct a PM-tree. Then, we analyze the cost models of the PM-tree and the R-tree to understand how the PM-tree performs better than the R-tree for the relevant range query workload. Finally, we present the details of the algorithms.

4.1 Building a PM-tree in the Projected Space

In the projected space, each o'_i w.r.t. $o_i \in \mathcal{D}$ is an m dimensional vector. For the paper to be self-contained, we briefly explain how to build a PM-tree on the o'_is. Interested readers may refer to [46] for more details on the PM-tree.

Selecting Pivots. The PM-tree combines the M-tree with pivot mapping. Methods for selecting an optimal set of pivots have been studied extensively. For each set of pivots, a PM-tree region is the intersection of the M-tree hyper-spherical region and hyper-rings caused by the pivots. We choose a set of pivots with the aim of making the overall volume of the corresponding PM-tree region the smallest.

PM-tree Structure. The structure of a PM-tree is shown in Fig. 4. Being an extension of the M-tree, it retains all the information of the M-tree. For each node e, it stores the covered radius $\mathsf{e.r}$, a pointer to its covered sub-tree $\mathsf{e.ptr}$, the center of the covered hyper-sphere $\mathsf{e.RO}$, the distance $\mathsf{e.PD}$ between $\mathsf{e.RO}$ and its parent node, and the smallest interval $\mathsf{e.HR}$ covering the distances between the pivots and each of the point stored in leaves. For a data entry o, it stores the point data, the ID of the point o, the distance $\mathsf{o.PD}$ between o and its parent entry, and the minimum and the maximum distances to pivots.

Range Query Processing. A range query, denoted by $\mathbf{range}(q, r)$, returns all points that are located in $B(q, r)$. The nodes in the PM-tree are traversed in a depth-first fashion. When a node is accessed, we verify its pruning condition by using the triangle inequality. When a data entry is accessed, we insert the corresponding point into the result set if it is in $B(q, r)$.

Example 3 *As shown in Fig. 4, we choose o_1 and o_{11} as pivots, and partition the space by using the ball partitioning, as shown in Fig. 4(a). The nodes $e_1, e_2, \cdots, e_6$ contain the points inside a hyper-sphere, whose center and radius are saved as the part of an entry. When a range query $\mathbf{range}(q, 2)$ is issued, we check pruning conditions when accessing the nodes. Only e_4 and e_6 are checked. Finally, we return $\{o_{14}\}$ as the result.*

4.2 Cost Models of the PM-tree Versus the R-tree

To compare the performance of the PM-tree and the R-tree, we adopt a node-based cost model [10] to examine how the PM-tree performs compared to the R-tree from a theoretical point of view.

In this cost model, a concept called distance distribution of a dataset $\mathcal{D}$ is computed as follows.

$$F(x) = Pr[\|o_i, o_j\| \leq x], \tag{4}$$

where $o_i, o_j \in \mathcal{D}$. In addition, for each dataset used in our experiments, we compute its "homogeneity of viewpoints" (HV), which is shown in Table 3. HV evaluates the homogeneity of the distance distributions of the data points. Let $F_o(x)$ denote the distribution of the distances between all points to point o. Given two points o_1 and o_2, a higher HV means that o_1 and o_2 are more likely to have similar distance distributions $F_{o_1}(x)$ and $F_{o_2}(x)$. The HV values of all the datasets are no smaller than 0.9, which enables us to approximate their distance distributions when estimating the cost models of the two trees.

Cost Model of the PM-tree. Consider a range query $\mathbf{range}(q, r_q)$. Assume that a PM-tree has s pivot points $p_1, \cdots, p_s$. A node e is accessed iff the following conditions are satisfied:

$$\begin{cases} \|q, \mathsf{e.RO}\| \leq \mathsf{e.r} + r_q \\ \wedge_{i=1}^{s} \{\|q, p_i\| - r_q \leq \mathsf{e.HR}[i].max\} \\ \wedge_{i=1}^{s} \{\|q, p_i\| + r_q \geq \mathsf{e.HR}[i].min\} \end{cases} \tag{5}$$

Therefore, the probability of e being accessed can be computed as follows.

$$\begin{aligned} Pr[e] =& F(\mathsf{e.r} + r_q) \cdot \prod_{i=1}^{s} [F(\mathsf{e.HR}[i].max + r_q) \\ & - F(\mathsf{e.HR}[i].min - r_q)] \end{aligned} \tag{6}$$

Assume that there are N nodes in the PM-tree. The number of distance computations (computation cost) is estimated by considering the probability that a node is accessed multiplied by its number of entries $N(e)$, thus obtaining the number of distance computations as follows.

$$\mathbf{CC}(\mathbf{range}(q, r_q)) = \sum_{i=1}^{N} N(e_i) \cdot Pr[e_i] \tag{7}$$

Cost Model of the R-tree. For each node e of an m-dimensional R-tree, we denote its minimum bounding rectangle as $\mathbf{MBR}(e) = [l_1, u_1] \times \cdots \times [l_m, u_m]$. Given a range query $\mathbf{range}(q, r_q)$, the condition of e being accessed is that $B(q, r_q)$ intersects with $\mathbf{MBR}(e)$. Since it is hard to quantify the probability that a ball intersects with a high-dimensional rectangle, we substitute an isochoric hyper-cube for the ball. Specifically, an m-dimensional ball with radius r_q is substituted by a hyper-cube with the length of sides $l = \sqrt[m]{\frac{2\pi^{m/2}}{m\Gamma(m/2)}} r_q$ [28]. We also denote the data distribution of dataset $\mathcal{D}$ on the i-th dimension as follows.

$$G_i(x) = Pr[X_i \leq x], \tag{8}$$

Table 2: Computation Cost (CC) of PM-tree and R-tree

Datasets	Audio	Cifar	MNIST	Trevi	NUS	GIST	Deep
PM-tree	38,182	35,210	56,670	34,281	201,448	739,720	964,451
R-tree	40,565	54,869	59,043	63,884	252,187	889,974	1,017,604
Reduction	6%	36%	4%	46%	20%	17%	5%

where X_i is the i-th dimension of a random point in $\mathcal{D}$. Similarly, we let N be the number of nodes in the R-tree and let $N(e_i)$ be the number of entries in node e_i. We obtain the number of distance computations as follows (details are omitted for brevity).

$$\mathbf{CC}(\mathbf{range}(q, r_q)) = \sum_{i=1}^{N} N(e_i) \cdot \prod_{i=1}^{m} [G_i(u_i + l) - G_i(l_i - l)] \tag{9}$$

Comparison of the PM-tree and the R-tree. In order to compare the computation costs for the two trees, we construct PM-trees and R-trees for the points in all the datasets (introduced in Table 3) after transforming them into the projected space. We choose $m = 15$ hash functions and set the maximum number of entries per node to 16. For each dataset, we choose the same range r to estimate the cost of computing a range query. The value of r is chosen to return approximately the nearest 8% of all points, since these points usually suffice to return a c-ANN result. The estimated computation costs are computed based on Eqs. 7 and 9, and the results are presented in Table 2. We can see that using the PM-tree reduces the number of distance computations by about $5\% - 46\%$ for the different datasets. This observation offers evidence that the PM-tree has better performance than the R-tree in our setting.

4.3 Tunable Confidence Interval

Based on Lemma 2, we further estimate the confidence interval of r' between o_1 and o_2 for a given $r = \|o_1, o_2\|$.

Lemma 3 *Given two points o_1 and o_2, we have:*

- ***P1**: The probability that $r' < r\sqrt{\chi^2_{1-\alpha}(m)}$ is α*
- ***P2**: The probability that $r' > r\sqrt{\chi^2_{\alpha}(m)}$ is α*

Here, $\chi^2_\alpha(m)$ is the upper quantile of a χ^2 distribution with m degrees of freedom, where

$$\int_{\chi^2_\alpha(m)}^{+\infty} f(x; m)dx = \alpha,$$

and $f(x; m)$ is the probability density function of a χ^2 distribution with m degrees of freedom.

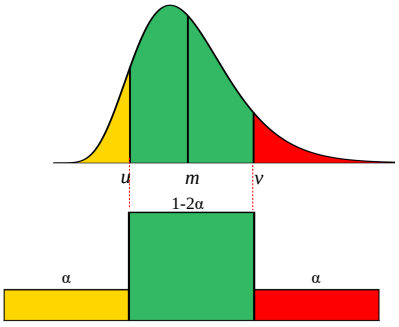


Fig. 5: Confidence Interval

Proof From Lemmas 1 and 2, we know $\frac{r'^2}{r^2} \sim \chi^2(m)$. Constructing a confidence interval $I = [u, v]$ for $\frac{r'^2}{r^2}$ requires that the probability that $\frac{r'^2}{r^2}$ falls into I is $1 - 2\alpha$ for any given α. A standard approach is to select u and v that make $Pr[\frac{r'^2}{r^2} < u] = \alpha$, i.e., $Pr[\frac{r'^2}{r^2} > u] = 1 - \alpha$, and $Pr[\frac{r'^2}{r^2} > v] = \alpha$. Further, $\int_u^{+\infty} f(x; m)dx = 1 - \alpha$ and $\int_v^{+\infty} f(x; m)dx = \alpha$. According to the definition of upper quantile, we have $u = \chi^2_{1-\alpha}(m)$ and $v = \chi^2_\alpha(m)$. The confidence interval and its corresponding probability are shown in Fig. 5.

Lemma 3 establishes a strong relationship between an original distance and the confidence interval of the corresponding projected distance, which we use to answer (r, c)-BC and c-ANN queries.

5 Nearest Neighbor Query Processing

We proceed to cover the nearest neighbor query processing based on PM-LSH. First, we present the details of (r, c)-BC query processing. Then, we extend the coverage to include (c, k)-ANN query processing.

5.1 The (r, c)-BC Query

An (r, c)-BC query can be computed directly by Algorithm 1. Given a query q and m hash functions, we compute the hash value $q' = (h^*_1(q), \ldots, h^*_m(q))$ and use the PM-tree to answer a range query $\mathbf{range}(q', tr)$, where t is a parameter that guarantees that a point inside $B(q, r)$ in the original space will fall into $B(q', tr)$

Algorithm 1: (r, c)-BC Query

Input: A query point q and parameters β, n, t, c, r
Output: A point p in $B(q, cr)$ or nothing
1 Compute $q' = (h_1^*(q), \ldots, h_m^*(q))$;
2 Initialize a candidate set $C \leftarrow$ the results of a range query q' with radius $t \cdot r$ on the PM-tree;
3 **if** $|C| \geq \beta n + 1$ **then**
4 **return** p in C that is closest to q;
5 **else**
6 **if** $|\{p \mid p \in C \wedge \|p, q\| \leq c \cdot r\}| \geq 1$ **then**
7 **return** p in C that is closest to q;
8 **else**
9 **return** $\varnothing$;

Algorithm 2: (c, k)-ANN Query

Input: A query point q, and parameters r_{min}, β, n, t, c, k
Output: k points
1 Initialize a candidate set $C \leftarrow \varnothing$ and $r \leftarrow r_{min}$;
2 Compute $q' = (h_1^*(q), \ldots, h_m^*(q))$;
3 **while** *true* **do**
4 **if** $|\{p \mid p \in C \wedge \|p, q\| \leq c \cdot r\}| \geq k$ **then**
5 **return** *top-k* points closest to q in C;
6 Initialize a range query q' with radius $t \cdot r$ in the PM-tree;
7 **while** $|C| < \beta n + k$ **do**
8 Find a node in $B(q', t \cdot r)$ on the PM-tree;
 $C \leftarrow C \cup \{$the points in the node$\}$;
9 **if** $|C| \geq \beta n + k$ **then**
10 **return** *top-k* points closest to q in C;
11 $r \leftarrow c \cdot r$;

in the projected space with a constant probability. Then we collect the result of the range query into a candidate set C.

According to Lemma 4, introduced in Section 5.3, the correctness of the (r, c)-BC query can be guaranteed. In other words, by properly choosing a parameter β, we examine a sufficient number of βn candidate points such that the following two situations hold with constant probability.

- If the total number of points in C exceeds βn, there must be at least one point from C in $B(q, cr)$.
- If no point from C is in $B(q, cr)$, there exists no point in $\mathcal{D}$ is in $B(q, r)$.

Therefore, we can correctly answer an (r, c)-BC query by processing a range query using the PM-tree. In Section 5.3, we consider how to set parameters t and β.

5.2 The (c, k)-ANN Query

Answering a c-ANN query is more complicated than answering an (r, c)-BC query since we do not know the distance $\|q, o^*\|$ in advance. In order to answer a (c, k)-ANN query with a constant probability, we must ensure that we access enough points, i.e., at least βn points. Therefore, we have to enlarge the search radius in the projected space when fewer than βn points are found until k points inside $B(q, cr)$ have been obtained.

The details of computing a (c, k)-ANN query can be found in Algorithm 2. Most of the steps are similar to ones in Algorithm 1. The difference is that when both termination conditions (Lines 4 and 8) are violated, another range query with a larger radius is required.

Selecting the Radius r of a Range Query. As executing multiple range queries is time consuming, it is attractive to reduce the number of iterations in the while-loop. Intuitively, we hope to find a "magic" r_{min} such that the process terminates quickly. An ideal r_{min} must yield a number of points inside $B(q', tr_{min})$ that exceeds $\beta n + k$ such that Algorithm 2 is able to terminate after processing the range query $B(q', tr_{min})$. In addition, to avoid returning a large number of unnecessary points, which also is costly, the number of points inside $B(q', tr_{min}/c)$ should be below $\beta n + k$. Otherwise, a range query $B(q', tr_{min}/c)$ with a smaller radius is able to return enough points.

As the r_{min} can be selected from a relatively large range, we design a selection scheme as follows. Suppose that we have obtained the distance distribution $F(x)$ of all datasets. Due to a good HV value, the distance distribution of a query point can be estimated for the dataset. Then we can find a suitable r that satisfies $n \cdot F(r) = \beta n + k$, which implies that $\beta n + k$ points locate in $B(q, r)$ on average. However, to avoid the case where the number of points in $B(q, r)$ exceeds $\beta n + k$, we choose an r_{min} that is slightly smaller than r. As the choice of r_{min} is not unique and the selection range is relatively large, and since the performance is not strongly dependent on the specific choice, the effect of the estimation is expected to be small.

Example 4 *Setting $\beta n = 4$, we need to retrieve at least 5 points for a $(2, 1)$-ANN query. Initially, we set $r_{min} = r' = 2$. As explained in Example 3, o_{14} is returned. As the number of returned points is below 5, we set $r' = 4$. In this round, only the subtree of e_5 can be discarded, and we check the points in e_3, e_4, and e_6 and obtain $\{o_2, o_5, o_7, o_{12}, o_{13}, o_{14}\}$. The number of returned points is 6, and the process terminates. Finally, we return the $(2, 1)$-ANN result o_{14}.*

5.3 Analysis

Quality Guarantee. In Algorithms 1 and 2, we execute a range query on the PM-tree with a radius tr in the projected space. Therefore, we have to compare the projected distances of candidate points to q with tr. Specifically, two types of points need to be discussed: true positives (the points inside $B(q,r)$) and false positives (the points outside $B(q,cr)$).

Lemma 4 *Given a query q, we set probabilities α_1 and α_2, and parameter t such that they satisfy Eq. 10:*

$$\begin{cases} t^2 = \chi^2_{\alpha_1}(m) \\ t^2 = c^2\chi^2_{1-\alpha_2}(m) \end{cases} \tag{10}$$

We then have:

- ***E1:** If a point o exists in $B(q,r)$, its projected distance to q is smaller than tr.*
- ***E2:** There are fewer than βn ($\beta > \alpha_2$) points outside $B(q,cr)$ whose projected distances to q are smaller than tr.*

The probability that E1 occurs is at least $1-\alpha_1$, and the probability that E2 occurs is at least $1-\frac{\alpha_2}{\beta}$.

Proof Given a point $o \in B(q,r)$, let $r_o = \|o,q\| \le r$ and $r'_o = \|o',q'\|$ be the original and projected distances to q, respectively. By setting $t = \sqrt{\chi^2_{\alpha_1}(m)}$, according to the Lemma 3, we have $Pr[r'_o > r_o\sqrt{\chi^2_{\alpha_1}(m)}] = Pr[r'_o > tr_o] = \alpha_1$. Since $r_o \le r$, $Pr[r'_o > tr]$ is at most α_1. Therefore, we know that $Pr[E1] = Pr[r'_o \le tr] > 1-\alpha_1$. Likewise, given a point $o \notin B(q,cr)$, let $r_o = \|o,q\| > cr$ and $r'_o = \|o',q'\|$ be the original and projected distances to q, respectively. By setting $t = c\sqrt{\chi^2_{1-\alpha_2}(m)}$, according to the Lemma 3, we have $Pr[r'_o < r_o\sqrt{\chi^2_{1-\alpha_2}(m)}] = Pr[r'_o < t\frac{r_o}{c}] = \alpha_2$. Since $\frac{r_o}{c} > r$, $Pr[r'_o < tr]$ is at most α_2. Therefore, by using Markov's inequality, we have $Pr[E2] > 1-\frac{\alpha_2}{\beta}$.

Note that if $E1$ and $E2$ hold at the same time, then Algorithm 1 computes the (r,c)-BC query correctly.

Lemma 5 *Algorithm 1 answers an (r,c)-BC query with at least a constant probability.*

Proof Let $m = O(1)$. If α_1 is a constant, α_2 is also a constant due to Eq. 10. By setting $\beta = 2\alpha_2$, the lower bound probabilities of $E1$ and $E2$, i.e., $1-\alpha_1$ and $1-\frac{\alpha_2}{\beta}$, will also be constant. Therefore, we can guarantee that $E1$ and $E2$ hold at the same time with at least a constant probability. Thus, if we access at least $\beta n+1$ points with projected distances to q smaller than tR, due to $E2$, there are at most βn points outside $B(q,cr)$, and we thus obtain at least one point inside $B(q,cr)$. On the other hand, if we access no more than $\beta n+1$ points with projected distances to q smaller than tR, the correctness of $E2$ is not guaranteed. Therefore, it is safe to return either no points or the points whose distances to q are at most cr for an (r,c)-BC query.

As a typical setting in the LSH methods, we choose parameters that satisfy $Pr[E1] = 1-1/e$ and $Pr[E2] = 1/2$. Note that we can also choose parameters that achieve more accurate results. In our setting, we have $\alpha_1 = 1/e$ and $t = \sqrt{\chi^2_{\alpha_1}(m)}$. Based on Eq. 10, both α_2 and β can be determined easily.

Theorem 1 *Algorithm 2 returns a c^2-ANN with probability at least $1/2-1/e$.*

Proof Due to Lemma 5, we find that $E1$ and $E2$ can hold at same time with probability at least $1/2-1/e$ in our setting. Now we show that when $E1$ and $E2$ hold, the output of Algorithm 2 is c^2-approximate. We denote the set of points whose projected distances to q are smaller than tr as $C(r)$. When enlarging r according to the sequence $1, c, c^2, \cdots$, there must exist a radius r_{opt} that makes $|C(r_{opt})| \ge 1+\beta n$ and $|C(r_{opt}/c)| < 1+\beta n$ hold. Then, if $r^* = ||o^*,q|| \le r_{opt}/c$, its projected distance to q is smaller than tr_{opt}/c according to $E1$, and we must have found it in $C(r_{opt})$ because $C(r_{opt}) \supset C(r_{opt}/c)$. As a result, Algorithm 2 returns the exact NN. If $r = ||o^*,q|| > r_{opt}/c$, according to $E2$, there is at least one point in $C(r_{opt})$ whose distance to q is at most cr_{opt}. Therefore, we return a point whose distance to q is smaller than c^2r^*.

Algorithm Analysis of PM-LSH. In PM-LSH, if we choose a large m, it will be costly to process a sequence of range queries in the projected space. So we consider m as a constant and fix its value at 15 in all experiments.

Theorem 2 *PM-LSH has space cost $O(n)$ and time cost $O(\log n + \beta n)$, where β is much smaller than 1.*

Proof The space consumption is due mainly to the PM-tree, which has n items. Each item consumes $m+O(1)$ space, so the overall space consumption is $O(n)$ as $m = O(1)$. The query time cost comes from two parts: 1) finding candidate points in the PM-tree; and 2) verifying the real distances of candidate points to q. The former has cost $O(\log n)$, and the latter has cost $O(\beta n)$ when d is considered as a constant. Therefore, the total query time is $O(\log n + \beta n)$.

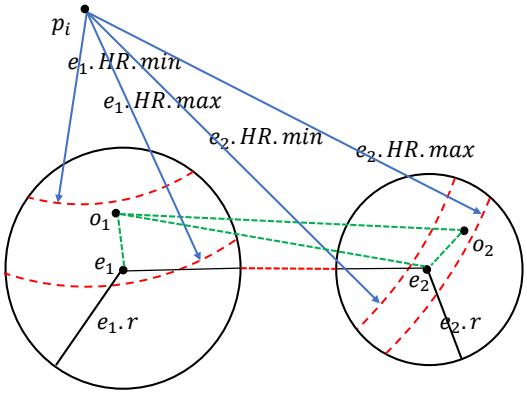


Fig. 6: Illustration of Computing Mindist

6 Closest Pair Query Processing

We proceed to cover closest-pair query processing based on PM-LSH. First, we propose a branch and bound algorithm that processes the nodes in the PM-tree in best-first manner. Due to the low efficiency of the branch and bound algorithm, we develop a radius filtering method to improve the query efficiency while sacrificing only slightly the accuracy of the candidate pairs found in the projected space.

6.1 Branch and Bound Algorithm

A straightforward method is to employ a branch and bound search strategy on the PM-tree. First, we aim to find T point pairs in the PM-tree with the smallest distances in the projected space. Next, we verify their distances in the original space. Finally, we report k closest pairs as the result.

For any two nodes e_1 and e_2, we denote the minimum distance of any point pair $(o_1, o_2) \in e_1 \times e_2$ by $\mathsf{Mindist}(e_1, e_2)$, which is computed as follows.

$$\mathsf{Mindist}(e_1, e_2) = \max \begin{cases} \max_i LB(p_i), \\ \|\mathsf{e_1.RO}, \mathsf{e_2.RO}\| - \mathsf{e_1.r} - \mathsf{e_2.r} \end{cases} \tag{11}$$

For the first term, we define a pivot-based lower bound $LB(p_i)$ of the minimum distance between e_1 and e_2 w.r.t. p_i, where p_i is the i-th global pivot. In Fig. 6, we have two points $o_1 \in e_1$ and $o_2 \in e_2$. According to the property of the PM-tree, we know that $\|o_1, p_i\|$ is in the range I_1:

$$I_1 = [\mathsf{e_1.HR}[i].min, \mathsf{e_1.HR}[i].max]$$

Likewise, $\|o_2, p_i\|$ is in the range I_2:

$$I_2 = [\mathsf{e_2.HR}[i].min, \mathsf{e_2.HR}[i].max]$$

We compute $LB(p_i)$ based on the triangle inequality. Since $\|o_1, o_2\| \geq |\|o_1, p_i\| - \|o_2, p_i\||$, if I_1 overlaps I_2, we have $LB(p_i) = 0$. Otherwise, $LB(p_i)$ is the distance between I_1 and I_2. In Fig. 6, we have $LB(p_i) = \mathsf{e_2.HR}.min - \mathsf{e_1.HR}.max$.

For the second term, we estimate the minimum distance between e_1 and e_2 using their centers. We compute $\|o_1, o_2\|$ with $\mathsf{e_2.RO}$ as follows.

$$\|o_1, o_2\| \geq \|o_1, \mathsf{e_2.RO}\| - \|\mathsf{e_2.RO}, o_2\|$$

We continue to compute $\|o_1, \mathsf{e_2.RO}\|$ with $\mathsf{e_1.RO}$ as follows.

$$\|o_1, \mathsf{e_2.RO}\| \geq \|\mathsf{e_1.RO}, \mathsf{e_2.RO}\| - \|\mathsf{e_1.RO}, o_1\|$$

Combined with the fact that $\|\mathsf{e_1.RO}, o_1\| \leq \mathsf{e_1.r}$ and $\|\mathsf{e_2.RO}, o_2\| \leq \mathsf{e_2.r}$, we obtain the second term.

Let d_T be the current T-th smallest distance in the projected space. We access the node pairs in best-first manner according to the ascending Mindist order. When the Mindist of the next node pair to process exceeds d_T, the search terminates, and T point pairs are returned for verification.

The details of Algorithm 3 are explained as follows.

1. We initialize a point pair candidate set C of size $|C| = T$. We apply a self-join on each leaf node in the PM-tree and update C and d_T accordingly.
2. We maintain a priority queue PQ to store node pairs in ascending Mindist order. We initialize PQ by inserting (e_r, e_r), where e_r is the root of the PM-tree.
3. We pop the top element (e_1, e_2) from PQ. If we have $\mathsf{Mindist}(e_1, e_2) > d_T$, the procedure stops; otherwise, we continue to examine (e_1, e_2). The PM-tree is a balanced tree, and we only consider node pairs at the same level. Therefore, if e_1 and e_2 are leaf nodes, we compute the distance of each point pair in $e_1 \times e_2$ and update C and d_T accordingly. If e_1 and e_2 are non-leaf nodes, for each child node e_1' of e_1 and each child node e_2' of e_2, we insert (e_1', e_2') into PQ. This process terminates when PQ is empty if it did not terminate earlier.
4. We verify the original distance of each point pair in C and return top-k point pairs.

Example 5 *In Fig. 4, for a $(2,2)$-ACP query, we set $T = 3$. First, we apply a self-join to all leaf nodes e_3, e_4, e_5, and e_6, obtaining the top-3 result (o_7, o_{15}), (o_2, o_{14}), and (o_6, o_{13}) with $d_T = 1.70$. Then, we consider pairs of points in different leaf nodes. We initialize PQ with (e_r, e_r). As $\mathsf{Mindist}(e_r, e_r) = 0 < d_T$, we continue to insert (e_1, e_1), (e_2, e_2), and (e_1, e_2) into PQ. Next, (e_1, e_1) and (e_2, e_2) are examined. For e_1's child nodes e_3 and e_4, since (e_3, e_3) and (e_4, e_4) have been*

Algorithm 3: Branch and Bound Algorithm

Input: A dataset $\mathcal{D}$, a PM-tree $\mathcal{T}$ indexing the projected data and parameters T, n, k
Output: k point pairs

```
Apply a self-join on each leaf node in T and store k found pairs with the smallest distance in the projected space;
d_T ← maximum distance of pairs in C;
Initialize a priority queue PQ to store node pairs in ascending Mindist order;
PQ ← (T.root, T.root);
while PQ is not empty do
    (e1, e2) ← PQ.Pop;
    if Mindist(e1, e2) > d_T then
        Break;
    foreach child node e'_1 of e1 do
        if e'_2 is a leaf node then
            foreach point pair (o'_1, o'_2) in e'_1 × e'_2 do
                Compute ||o'_1, o'_2|| and update C and d_T accordingly;
        else
            foreach child node e'_2 of e2 do
                Insert (e'_1, e'_2) into PQ;
Verify the original distance of each point pair in C;
Return Top-k results from the verified pairs;
```

examined, we only need to insert (e_3, e_4) *into* PQ*. After employing a similar operation for* e_2*, the node pairs in* PQ *are* $\langle (e_1, e_2), (e_5, e_6), (e_3, e_4) \rangle$*. This process proceeds until we examine* (e_4, e_6)*, since* $\mathsf{Mindist}(e_4, e_6) = 2.91 > d_T$*. We return the top-3 pairs* (o_7, o_{15})*,* (o_2, o_{14})*, and* (o_6, o_{13}) *in the projected space. We verify their distances in the original space and return* (o_7, o_{15}) *and* (o_6, o_{13}) *as the result.*

6.2 Limitations of the Branch and Bound Algorithm

In the branch and bound algorithm, the search procedure terminates when $\mathsf{Mindist} > d_T$, where $\mathsf{Mindist}$ is a lower bound distance on unexamined pairs. However, this bound is often so loose that the algorithm efficiency suffers. Specifically, due to the property of the PM-tree, the ranges covered by two nodes at the same level overlap with high probability. No matter how small the overlap is, $\mathsf{Mindist}(e_1, e_2) = 0$.

To understand this issue better, we conduct an experiment on dataset *Audio* to count the number of node pairs with $\mathsf{Mindist} = 0$. We employ the branch and bound algorithm to search the PM-tree, and we count the number of node pairs with $\mathsf{Mindist} = 0$ among all verified node pairs. We find that more than 70% of the node pairs have $\mathsf{Mindist} = 0$, which indicates that most node pairs overlap.

This phenomenon may be explained by the fact that PM-trees are built so that structured clusters are achieved for the subtrees of each node. The differences between nodes are not considered during construction, due to the high computational cost. Therefore, when points are located in a dense region, the tree nodes constructed for this region are likely to overlap substantially due to their limited node capacity.

Consequently, we have to examine about 90% of all pairs in the branch and bound algorithm when using a PM-tree with $m = 15$, which makes the algorithm degenerate to nearly a brute-force nested loop algorithm. We observe that we can lower m to reduce the cost of finding exact closest pairs in the projected space. However, a small m may lead to an inaccurate confidence interval when estimating the correlation between original and projected distances. As a result, we have to verify more candidate pairs to achieve a high recall.

6.3 Improvement with Radius Filtering

To fewer pairs, we provide a radius filtering method. The idea is to compute an upper bound distance of the k-th best point pair in the original space. We then estimate a candidate distance in the projected space based on the upper bound and use this distance to prune unnecessary node pairs.

Specifically, we still apply a self-join on each individual leaf node in the PM-tree. Let ub denote the upper bound distance in the original space. We verify the original distances of all self-join pairs and initialize ub to be the current k-th smallest distance. According to Lemma 4, if a point pair exists whose original distance is smaller than ub, its projected distance is smaller than $t \cdot ub$ with a high probability. Therefore, we aim to find point pairs in the PM-tree whose projected distance is within $t \cdot ub$. As we have already examined all point pairs in leaf nodes via self-joins, we only need to check pairs of points from different leaf nodes.

Let (o'_1, o'_2) be the point pair of (o_1, o_2) in the projected space. We observe that there is a strong relationship between the projected distance $\|o'_1, o'_2\|$ and the radius of their lowest common ancestor in the PM-tree. We define the concept of lowest common ancestor as follows.

Definition 6 (Lowest Common Ancestor) The lowest common ancestor (LCA) of two points o'_1 and o'_2 is a node e in the PM-tree such that:

- Points o'_1 and o'_2 are stored in the subtree of e.
- No child node e' of e exists such that o'_1 and o'_2 are also stored in the subtree of e'.

Let $R = \mathsf{e.r}$ denote the radius of the LCA node e of o'_1 and o'_2. We assume that $\gamma \cdot \|o'_1, o'_2\| \leq R$ holds with high probability, where the setting of parameter γ is explained later. Therefore, in order to find point pairs with projected distance smaller than $t \cdot ub$, we only have to examine the points of nodes in the PM-tree whose radius is smaller than $\gamma \cdot t \cdot ub$.

We explain the details of Algorithm 4 as follows.

1. We initialize a point pair candidate set C of size $|C| = k$. We apply a self-join on each leaf node in the PM-tree, and we compute the original distances of all pairs found. We then update C and ub accordingly.
2. Let $R = \gamma \cdot t \cdot ub$ be the radius used for node filtering in the PM-tree.
3. We employ Algorithm FindLCA() that traverses the PM-tree to find the nodes with radius smaller than R. A node e returned by FindLCA() may not be an LCA of the points it covers. But we can find the LCA of any point pair it covers in the subtree of e, and the radius of the LCA is smaller than R. Therefore, it suffices to examine the point pairs covered by e.
4. We consider the nodes returned by FindLCA() in ascending order of their radii. The intuition is that a node with a small radius is likely to cover point pairs with small projected distances.
5. We examine the nodes in turn. For any two points o'_1 and o'_2 in the sub-tree of a node e, we compute $\|o'_1, o'_2\|$, and if $\|o'_1, o'_2\| < t \cdot ub$, we consider (o_1, o_2) as a candidate pair. Then, we compare $\|o_1, o_2\|$ with ub and update both ub and C if necessary. This process stops when we have T candidate pairs.
6. We return C as the result.

Example 6 *In the example in Fig. 4, the PM-tree has 4 leaf nodes e_3, e_4, e_5, and e_6. To compute a $(2,2)$-ACP query, we first apply a self-join to all leaf nodes and obtain the preliminary top-2 pairs (o_4, o_8) and (o_{12}, o_{14}), both with distance 1. We set $ub = 1$. Setting $t = 3$ and $\gamma = 3$, we get $t \cdot ub = 3$ and $R = 9$. We find all inner nodes whose ranges are within 9 and obtain e_2. The unverified pairs in the subtree of e_2 come from $e_5 \times e_6$. As $\|o'_4, o'_2\| = 3.2 > 3$, we skip it and process the remaining pairs. Finally, we obtain $R = \langle (o_4, o_8), (o_{12}, o_{14}) \rangle$.*

Determining the Setting of γ. For any two points o'_1 and o'_2 in the projected space, we observe that $\|o'_1, o'_2\|$ and the radius of their LCA have a strong correlation. Let $\gamma = \frac{R}{\|o'_1, o'_2\|}$ be the ratio of R over $\|o'_1, o'_2\|$. To ensure the quality of the nodes returned by the radius filtering, we need to find an appropriate setting for γ.

Algorithm 4: Radius Filtering Method

Input: A dataset $\mathcal{D}$, a PM-tree $\mathcal{T}$ indexing the projected data and parameters T, n, t, k, γ
Output: k point pairs

```
1  Apply a self-join on each leaf node in T and verify
     all found point pairs;
2  count ← The number of verified pairs;
3  ub ← The k-th smallest real distance in found pairs;
4  R ← γ · t · ub;
5  C ← ∅;
6  Initialize an array A to store the nodes;
7  FindLCA(T.root, R, A);
8  Sort the nodes in A in ascending order of their radii;
9  foreach node e in A do
10     foreach point pair (o1, o2) in e's subtree do
11         if ‖o'1, o'2‖ < t · ub then
12             Verify (o1, o2) and update ub;
13             count++;
14         if count > T then
15             Break;
16     if count > T then
17         Break;
18 Return All pairs in C;
```

Algorithm 5: FindLCA(e, R, A)

Input: A PM-tree node e, a radius R, and an array A
Output: A

```
1  if e is an inner node then
2      if e.r < R then
3          Insert e into A;
4      else
5          foreach child node e_i of e do
6              FindLCA(e_i, R, A);
```

To do so, we study the probability density functions of γ on real datasets.

Let us take dataset *Audio* (Details are provided in Sec. 7) as an example. We use $m = 15$ hash functions. First, we randomly select 10K data points. We then index these points in the projected space using two PM-trees with node capacity $M = 2$ and $M = 16$, respectively. We obtain some 50 million point pairs from 10K points. For each pair, we compute the value of γ. Fig. 7 shows the probability density functions $f_\gamma(x)$ for $M = 2$ and $M = 16$. It is easy to see that the two functions have similar trends. Both peak quickly and then decline quickly. An appropriate value of γ is very likely to be within the neighborhood of the peak, which indicates that γ varies slightly for different pairs. With $\Pr(\gamma)$ being the success probability, we choose γ such that $\Pr(\gamma) = \int_0^\gamma f_\gamma(x)dx = 85\%$ for all datasets. Note that we can enlarge the value of $\Pr(\gamma)$ to examine more nodes. But this represents a tradeoff between accuracy

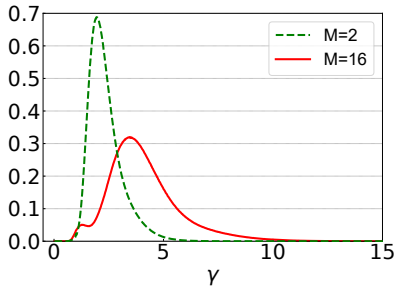


Fig. 7: The Probability Density Function of γ

and efficiency, and $\Pr(\gamma) = 85\%$ already provides good performance. We analyze the cost of computing γ experimentally in Section 7. Specifically, the cost is the time it takes to compute the distances of 50 million point pairs, which is acceptable when compared with the total cost.

Promote Methods for the PM-tree. The PM-tree is built bottom-up by inserting the data points one by one. When a node e overflows after inserting $M+1$ entries, we allocate a new node e' at the same level and distribute the $M+1$ entries among the two nodes. One study [9] contributes the concept of a Promote method that selects two points as the centers of two nodes e and e'. It is easy to see that different centers may lead to different partitioning results, which affects the algorithm performance. We consider two Promote methods as follows.

- m_RAD selects two points from all possible combinations as the centers such that the sum of the two covering radii is the minimum after partitioning. This method incurs many distance computations but also yields high-quality partitioning.
- RANDOM selects two points as node centers at random.

It is obvious that m_RAD provides no worse partitioning than does RANDOM, since m_RAD aims to minimize the sum of the two covering radii, which represents a locally optimal partitioning of the $M+1$ entries. Consequently, the two nodes are covered by a parent node with a small radius. In this case, the radius filtering strategy enables to obtain T candidate pairs with higher quality.

Algorithm Analysis of Radius Filtering. In the radius filtering method, as we have $n(n-1)/2$ pairs, we set $T = \beta n(n-1)/2 + k$, which is similar to the setting for the NN query.

Theorem 3 *PM-LSH answers an ACP query with space cost $O(n)$ and time cost $O(\beta n^2)$, where β is much smaller than 1.*

Table 3: Datasets

Dataset	n ($\times 10^3$)	d	HV	RC	LID
Audio	54	192	0.9273	2.97	5.6
Deep	1,000	256	0.9393	1.96	12.1
NUS	269	500	0.9995	1.67	24.5
MNIST	60	784	0.9531	2.38	6.5
GIST	983	960	0.9670	1.94	18.9
Cifar	50	1,024	0.9457	1.97	9.0
Trevi	100	4,096	0.9432	2.95	9.2

Proof The space consumption is due mainly to the PM-tree with n points. Each point consumes $m+O(1)$ space, so the overall space consumption is $O(n)$ as $m = O(1)$. The query time cost stems from two operations: 1) finding candidate pairs in the PM-tree, and 2) verifying the real distances of candidate pairs. Both operations have cost $O(T)$ when d is considered as a constant. According to the setting of T, the total query time is $O(\beta n^2)$.

7 Experiments

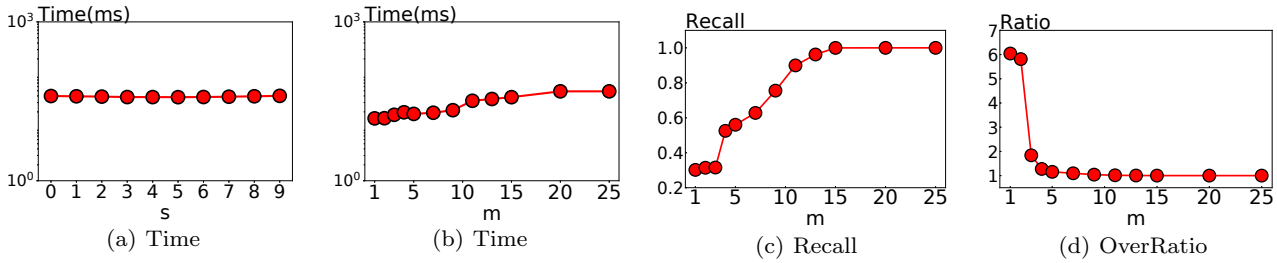
We report on extensive experiments with real datasets that offer insight into the performance of PM-LSH for both NN and CP queries.

7.1 Experimental Settings

All the algorithms are implemented in C++, and compilation is done with the O3 optimization. All experiments are run on a Linux machine with an Intel 3.4GHz CPU and 32GB memory.

Datasets. We use seven real datasets: *Audio*, *Deep*, *NUS*, *MNIST*, *GIST*, *Cifar*, and *Trevi*, which are used widely in existing LSH studies [18, 27, 33, 34, 47]. Table 3 reports key statistics of the datasets: *Homogeneity of Viewpoints* (HV [10]), *Relative Contrast* (RC [25]), and *Local Intrinsic Dimensionality* (LID [2]). RC computes the ratio of the mean distance over the NN distance for the data points. LID computes the local intrinsic dimensionality. A small RC value and a large LID value imply that it is challenging to compute NN results for the dataset. HV evaluates the homogeneity of the distance distributions of the data points. A higher HV means that the points are more likely to have similar distance distributions.

Query Set. For NN queries, we randomly select 200 points from each dataset, and we repeat each experiment 20 times and report the average value. We set the default value of c to 1.5, and vary its value in $\{1.1, 1.2, \ldots, 2.0\}$. We vary the value of k in $\{1, 10, 20, \ldots, 100\}$ and set the default value to 50. For CP queries, we repeat each experiment 20 times and report the average



(a) Time (b) Time (c) Recall (d) OverRatio

Fig. 8: Performance of PM-LSH when Varying s and m

value. We vary the value of k in $\{1, 10, 10^2, \ldots, 10^4\}$ and set the default value to 10^3. The default value of c is 4 in PM-LSH and the LSB-tree.

Competing Algorithms. For NN queries, we compare PM-LSH with the following competitors:

1. **Multi-Probe** [35]: A probing sequence (PS) based algorithm.
2. **QALSH** [27]: A radius enlargement (RE) based algorithm.
3. **SRS** [47]: A metric indexing (MI) based algorithm.
4. **R-LSH**: In order to compare the PM-tree and the R-tree, we index the points in the projected space with an R-tree instead of a PM-tree to see how PM-LSH then performs. We call this method R-LSH.
5. **LScan**: We consider a linear scan algorithm called LScan that randomly selects a portion of points (default 70%) and returns the top-k points with the smallest distances to the query.

For CP queries, we compare PM-LSH with the following competitors:

1. **LSB-tree** [49]: The LSB-tree supports both NN and CP queries.
2. **MkCP** [19]: MkCP supports CP queries with the M-tree. We choose the variant called GMA that uses grouping and N-consider techniques that enables trade-offs between time and accuracy.
3. **ACP-P** [7]: The state-of-the-art solution for CP queries.
4. **NLJ**: Nested loop join (NLJ) is an exact algorithm that computes the distance between any two points with two nested loops and then returns the top-k CPs.

Parameter Settings. For NN queries, we choose $m = 15$ hash functions for all the algorithms except QALSH and Multi-Probe. In our method, we set the number of pivots $s = 5$ and $\alpha_1 = 1/e$, so $\alpha_2 = 0.1405$ and $\beta = 0.2809$ are obtained according to Eq. 10, and r_{min} is determined according to the description in the previous section. For QALSH, the false-positive percentage $\beta = 100/n$, and the error probability $\delta = 1/e$. For SRS, the threshold of its early-termination condition $p'_\tau = 0.8107$, and the maximum percentage of points accessed in the projected space is $T = 0.4010$ when $c = 1.5$.

For CP queries, we choose $m = 15$ hash functions for our algorithm. We set the number of pivots $s = 5$, $\Pr(\gamma) = 0.85$, and $\alpha_1 = 1/e$, so $\alpha_2 = 0.0024$ are obtained according to Eq. 10, and thus $T = \alpha_2 n(n-1) + k$. For ACP-P, we set the hyper parameter $h = 5$ and the range value is set to 5 according to the advice of its authors. For MkCP, we set the number of groupings to $N = 2$. For the LSB-tree, the approximation ratio is set to $c = 4$.

Evaluation Metrics. We adopt three metrics to assess the performance of the algorithms: query time (ms for NN, s for CP), overall ratio, and recall, where the query time quantifies the algorithm efficiency and the overall ratio and recall capture the result quality. For an NN query q, we denote the result of a (c, k)-ANN query by $R = \langle o_1, o_2, \cdots, o_k \rangle$. Let $R^* = \langle o_1^*, o_2^*, \cdots, o_k^* \rangle$ be the exact kNNs. The overall ratio and recall are computed as follows.

$$OverallRatio = \frac{1}{k} \sum_{i=1}^{k} \frac{\|q, o_i\|}{\|q, o_i^*\|} \tag{12}$$

$$Recall = \frac{|R \cap R^*|}{|R^*|} \tag{13}$$

For a CP query, we denote the result of a (c, k)-ACP query by $R = \langle (o_{1,1}, o_{1,2}), (o_{2,1}, o_{2,2}), \ldots, (o_{k,1}, o_{k,2}) \rangle$. Let $R^* = \langle (o_{1,1}^*, o_{1,2}^*), (o_{2,1}^*, o_{2,2}^*), \ldots, (o_{k,1}^*, o_{k,2}^*) \rangle$ be the exact kCPs. The recall is the same as for the NN query, and the overall ratio is computed as follows.

$$OverallRatio = \frac{1}{k} \sum_{i=1}^{k} \frac{\|o_{i,1}, o_{i,2}\|}{\|o_{i,1}^*, o_{i,2}^*\|} \tag{14}$$

Table 4: Performance Overview of NN Queries

		PM-LSH	SRS	QALSH	Multi-Probe	R-LSH	LScan
Audio	Query Time (ms)	**13.5**	15.3	22.5	15.3	14.2	19.6
	Overall Ratio	**1.0014**	1.0025	1.0043	1.0242	1.0019	1.0073
	Recall	**0.9662**	0.9126	0.9003	0.8669	0.9633	0.6839
MNIST	Query Time (ms)	**12.3**	18.4	24.7	19.1	16.2	60.3
	Overall Ratio	**1.0076**	1.0101	1.0085	1.0103	1.0095	1.0276
	Recall	**0.8857**	0.8514	0.8655	0.8502	0.8705	0.7073
NUS	Query Time (ms)	**125.7**	142.1	133.2	125.9	129.6	176.8
	Overall Ratio	**1.0009**	1.0015	1.0027	1.0025	1.0011	1.0053
	Recall	**0.9257**	0.9247	0.8677	0.8782	0.9214	0.7057
Trevi	Query Time (ms)	**37.2**	47.9	145.5	239.3	63.9	57.68
	Overall Ratio	**1.0004**	1.0015	1.0029	1.0057	1.0044	1.0084
	Recall	**0.9961**	0.9342	0.8240	0.8534	0.9568	0.7103
Cifar	Query Time (ms)	**11.6**	16.1	38.3	26.8	35.6	58.2
	Overall Ratio	**1.0009**	1.0025	1.0057	1.0038	1.0056	1.0125
	Recall	**0.9746**	0.9624	0.7917	0.8011	0.9610	0.7081
GIST	Query Time (ms)	**398.7**	452.5	627.7	782.9	425.3	1528.3
	Overall Ratio	**1.0047**	1.0049	1.0037	1.0053	1.0059	1.0076
	Recall	**0.8436**	0.8145	0.8534	0.8122	0.8098	0.7023
Deep	Query Time (ms)	**227.8**	252.9	458.2	401.4	457.5	507.5
	Overall Ratio	**1.0037**	1.0077	1.0124	1.0112	1.0152	1.0145
	Recall	**0.8816**	0.8894	0.646	0.8118	0.8801	0.6938

7.2 Evaluation of NN Query Processing

To evaluate the performance of PM-LSH for NN query processing, we first determine parameter settings. Then, we compare the performance of all algorithms with default parameter settings on all datasets. Finally, we compare the algorithms by studying the changes of the overall ratio and recall under fixed query times.

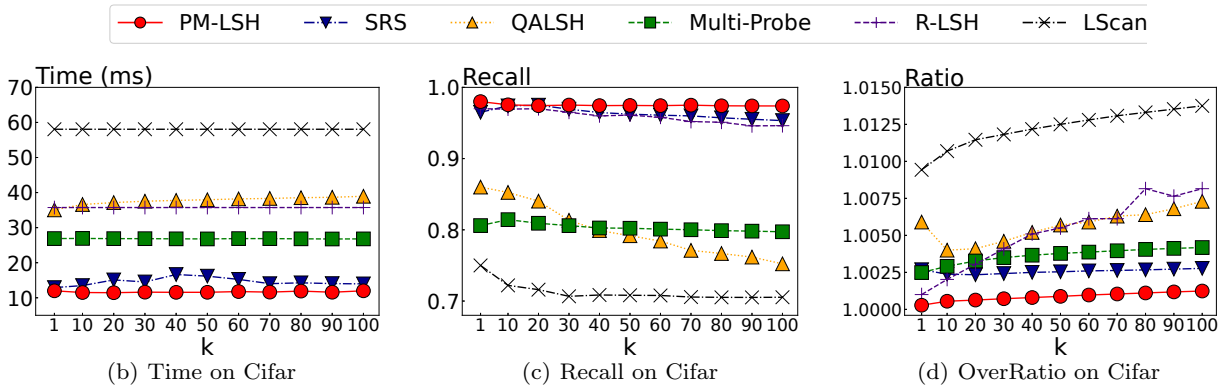
Parameter Study on PM-LSH for NN Query. We consider two parameters that may affect the performance of PM-LSH, i.e., the number of pivots s and the number of hash functions m. Here, we only show results from the *Trevi* dataset. It is easy to see that s only affects the query time. The overall ratio and recall do not change when we vary s. As we can see from the Fig. 8(a), when s changes, the query time remains steady, which indicates that PM-LSH is largely unaffected by different settings for s. When using a larger number of pivots, we have a higher chance to prune subtrees in the PM-tree. However, the cost of checking the pruning condition also increases. In conclusion, we set $s = 5$.

As shown in Fig. 8, when the value of m increases, we obtain a higher overall ratio and recall, but the query time also increases. The higher quality occurs because a larger m leads to more accurate distance estimation. However, the average cost to retrieve a point from the PM-tree also increases. Taking both efficiency and accuracy into consideration, we set $m = 15$.

When comparing PM-LSH with R-LSH, we observe in all the experiments that PM-LSH outperforms R-LSH on all metrics, which confirms the expected superiority of the PM-tree over the R-tree.

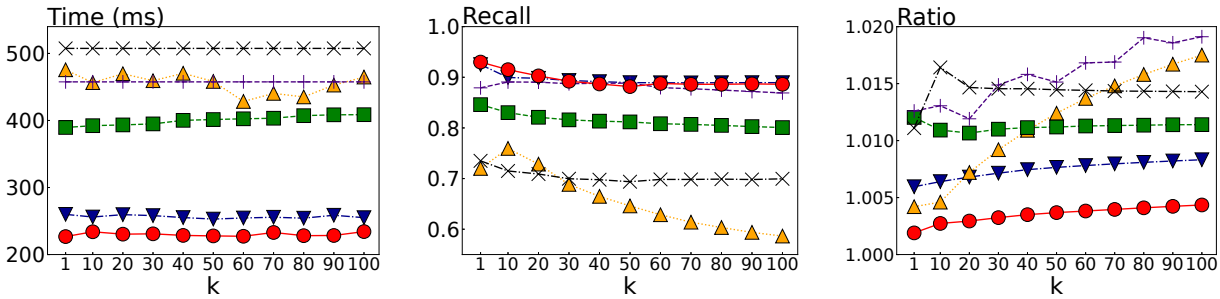
Performance Overview of NN Query. To compare all the algorithms with default parameter settings, we report the query time (ms), overall ratio, and recall on all datasets in Table 4. PM-LSH is more efficient than the competitors on all datasets, and its overall ratio and recall are also better than those of its competitors. Moreover, we find that either query time, overall ratio, or recall depend only slightly on the dataset dimensionality. For instance, *Audio*, *MNIST*, and *Cifar* have nearly the same cardinality, but different dimensionality, i.e., 192, 784, and 1024. However, the query times of PM-LSH on them are different and it is not only affected by data dimensionality. So we explain this by the query time being affected by the data distribution. In Table 3, we can see that dataset *GIST* has large LID value and small RC value, so it is considered as challenging dataset. As shown in Table 4, it has larger query times than the other datasets.

Effect of k. In this set of experiments, we study the performance when varying k in $\{1, 10, 20, \cdots, 100\}$. Due to the space limitation, we only report the performance on *Deep*, *Cifar*, and *Trevi*. The results are shown in Figs. 9–11. In the *Cifar* and *Trevi* datasets, we can see that PM-LSH achieves the best performance on all



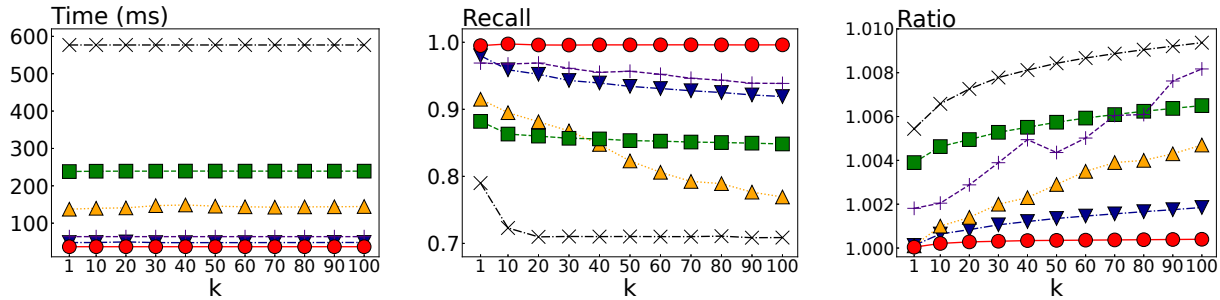
(b) Time on Cifar (c) Recall on Cifar (d) OverRatio on Cifar

Fig. 9: Performance on Cifar when Varying k of NN Queries



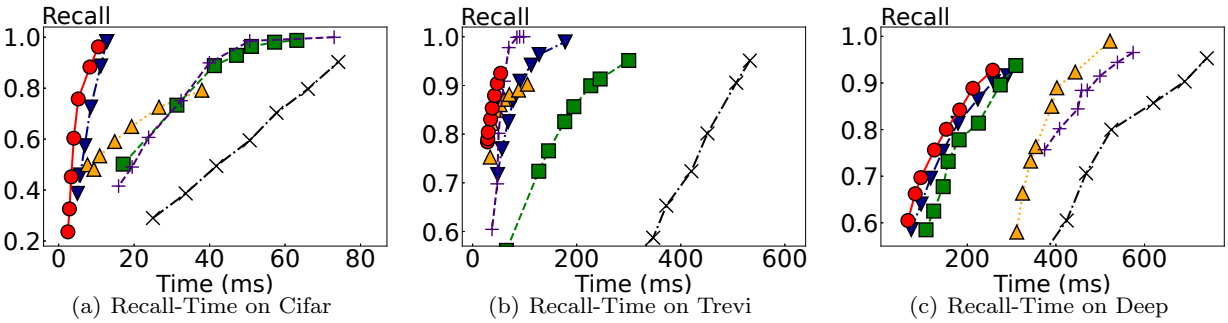
(a) Time on Deep (b) Recall on Deep (c) OverRatio on Deep

Fig. 10: Performance on Deep when Varying k of NN Queries



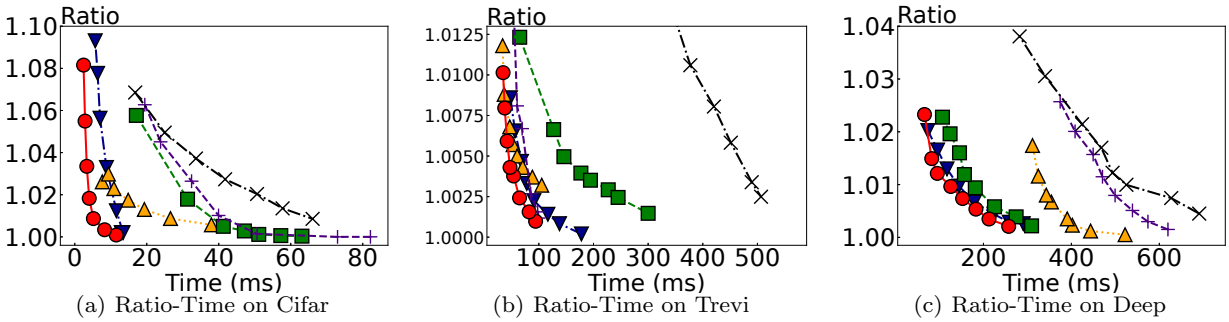
(a) Time on Trevi (b) Recall on Trevi (c) OverRatio on Trevi

Fig. 11: Performance on Trevi when Varying k of NN Queries



(a) Recall-Time on Cifar (b) Recall-Time on Trevi (c) Recall-Time on Deep

Fig. 12: Recall-Time Curve for NN Queries



(a) Ratio-Time on Cifar (b) Ratio-Time on Trevi (c) Ratio-Time on Deep

Fig. 13: Ratio-Time Curve for NN Queries

metrics. SRS is the second-best algorithm. When using the *Deep* dataset, PM-LSH has the smallest query time and overall ratio, and its recall is close to that of SRS.

As k increases, all algorithms achieve a higher overall ratio and a smaller recall, but the query time is relatively steady. In fact, the algorithms return the best k objects from a candidate set whose size exceeds $\beta n + k$. Therefore, a larger k has little effect on the query time but obviously has an adverse effect on the result quality.

When considered across different datasets with different cardinality n and dimension d, PM-LSH exhibits a consistent high accuracy. This is because PM-LSH is unaffected by the dimensionality of the datasets and because its cost is sublinear in the cardinality of the datasets. In contrast, Multi-Probe is affected significantly by the dataset dimensionality. The hash number of QALSH is $O(n \log n)$, so its query time increases super-linearly with the dataset cardinality. Similarly, when the dataset cardinality increases, SRS incurs a higher query cost to find an NN in the projected space.

To sum up, PM-LSH has the smallest query time among all competitors. In addition, the accuracy is high. Only SRS is able to achieve a competitive recall in some cases but exhibits longer query times than PM-LSH.

Recall-Time and OverallRatio-Time Curves. In this set of experiments, we evaluate the relationship between the recall or overall ratio and the query time for (c, k)-ANN queries on all the datasets when varying c to obtain different query times. The results are shown in Figs. 12 and 13. As the tradeoff between the query quality and the query time is the key tradeoff, the LSH methods focus on returning relatively good results with much smaller query time than the exact NN algorithms. The results show that all algorithms return more accurate results when more query time is used. They also show that PM-LSH achieves superior efficiency and accuracy when compared to SRS, QALSH, and Multi-Probe. This can be explained as follows. First, PM-LSH has a better distance estimator than QALSH and Multi-Probe, so PM-LSH outperforms them with the same number of retrieved points. Second, PM-LSH needs lower time to obtain the same number of retrieved points since only one or two range queries are required. In contrast, SRS needs T rounds of incremental NN search.

7.3 Evaluation of CP Query Processing

To evaluate PM-LSH for CP query processing, we first conduct an evaluation to determine the setting of γ and compare two Promote methods. Then, we compare with the competitors by varying the parameter values. Finally, we consider the changes of the overall ratio and recall under different query times.

Determining the Setting of γ. In this set of experiments, we study the effects of the node capacity M and the dataset cardinality on choosing γ in datasets *Audio*, *Trevi*, and *NUS*. We choose $M = 16$ and m_RAD as defaults. We randomly sample $n' = 10K$ points from each dataset. After we build a PM-tree, we compute the value of γ for each pair and use the probability density distribution function $f_\gamma(x)$ to study the effects.

We first consider $f_\gamma(x)$ when varying the value of M in $\{2, 16, 64\}$. As shown in Fig. 14, the tendency of $f_\gamma(x)$ remains nearly unchanged when varying M. However, the peak position, the peak value, and the gradient are affected slightly by M. To make $\Pr(\gamma) = 0.85$, the settings for γ are different. Note that when $M = 2$, $f_\gamma(x)$ has the smallest peak position, the largest peak value, and the largest gradient. This indicates that a small M yields a good partitioning. However, a small γ increases the PM-tree size and leads to additional computational costs. To achieve a good tradeoff, we set $M = 16$.

Next, we study $f_\gamma(x)$ when varying the number of sampled points n' in $\{5000, 10000, 20000\}$. As shown in Fig. 15, $f_\gamma(x)$ changes slightly when varying n, which enables us to determine the setting of γ by using only

Table 5: Construction Time of m_RAD and RANDOM

Dataset	Construction Time (s)	
	RANDOM	m_RAD
Audio	0.82	28.75
NUS	2.84	116.81
Trevi	1.06	45.09

a subset that preserves the information of the whole dataset. The cost of computing γ equals the time needed to compute the distances of 50 million point pairs formed by $10K$ points, which is about 0.3s when we use $m = 15$ hash functions for each dataset.

Effect of Promote methods. We compare the performance of the two Promote methods, m_RAD and RANDOM. Fig. 16 shows that the recall and overall ratio are very similar for the two methods, but that the query time when using m_RAD is smaller than that achieved when using RANDOM. This can be explained by the fact that the PM-tree constructed with m_RAD has a better structure, meaning that fewer candidate pairs need to be verified to achieve a high recall. So we choose m_RAD as the default Promote method. On the other hand, Table 5 shows that the construction of the PM-tree with m_RAD takes more time than with RANDOM, while still being acceptable.

Performance Overview of CP Query. We compare the algorithms with default settings on all datasets and report the query time (s), overall ratio, and recall in Table 6. We observe that PM-LSH has the best performance for all evaluation metrics and datasets. To analyze what affects the query time of PM-LSH on different datasets, we notice that *Cifar* takes more time than *Trevi*. However, the cardinality and dimensionality of *Cifar* are both smaller than those of *Trevi*, indicating that the query time is not only affected by the dataset cardinality and dimensionality. Other factors, including the data distribution, also have an effect. All algorithms exhibit a poor performance on *NUS*. This can be explained by *NUS* having a small RC value and a large LID value, which make it challenging to compute CP queries. MkCP has the worst performance on all datasets. The reason is that MkCP uses the M-Tree to index points directly, causing vulnerability to the curse of dimensionality. For high-dimensional datasets, the MkCP query algorithm nearly degenerates to being a brute-force algorithm. In practice, operations such as computing lower bounds and maintaining priority queues incur additional costs.

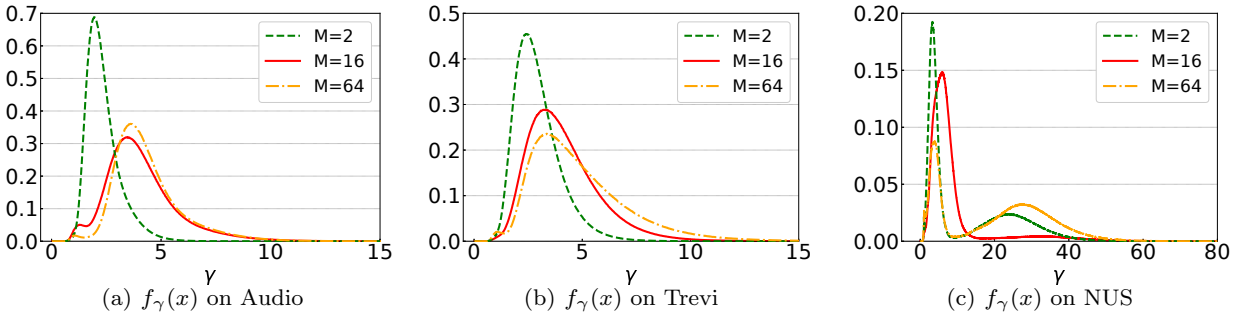
Effect of k. Next, we study the performance when varying k in $\{1, 10, 10^2, 10^3, 10^4\}$. For brevity, we only report the performance on datasets *Audio*, *Trevi*, and *NUS*. We choose *Audio* and *NUS* instead of *Cifar* and *Deep* because MkCP and ACP-P are inefficient for the latter two. The results are shown in Figs. 17–19.

With the increase of k, most algorithms incur longer query times and worse recall and overall ratio. The reason for a larger query time is that k affects the number of candidate pairs. PM-LSH, ACP-P, and MkCP all use the k-th smallest distance for pruning, so a large k means that more candidate pairs must be verified. The LSB-Tree returns the best k objects from a nearly fixed-size candidate sets, so its query time increases only slowly with k. An exceptional case occurs for the LSB-tree on *NUS*. The overall ratio improves with the increase of k. This is because many pairs have almost the same distances. When the result size increases, although the exact results are not found, the ratio of the distance of the i-th returned pair over that of the i-th exact pair decreases.

When considered across datasets, PM-LSH exhibits a consistent high accuracy. However, the query time of each algorithm varies substantially across the different datasets, which can be explained by three observations. (1) The query time is affected significantly by dataset cardinality n. For instance, the query times of PM-LSH, the LSB-tree, and ACP-P are subquadratic to n; the query time of MkCP is $O(n^2)$ in the worst case. (2) The query time is affected by dataset dimensionality d. All algorithms need to verify candidate pairs, and the cost is linear in d. (3) The data distribution also affects the query time, which is a key determining factor for when the algorithms terminate.

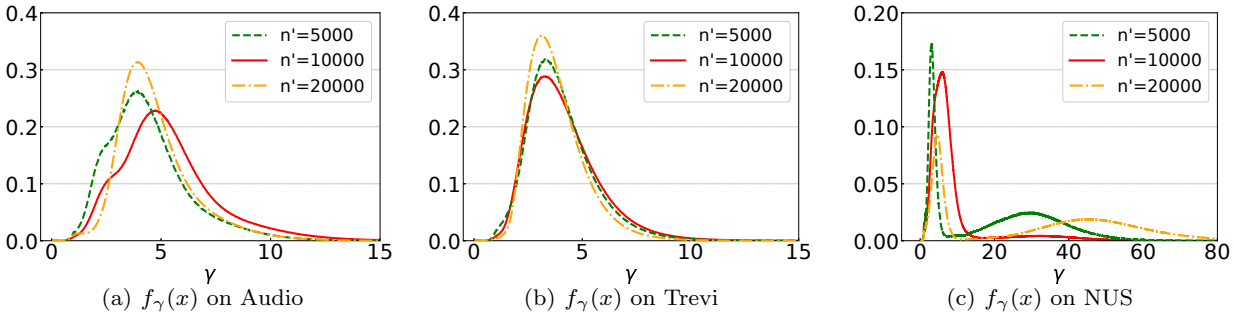
To sum up, PM-LSH has the smallest query time among all competitors. In addition, the accuracy is high. Only the LSB-tree is able to achieve a competitive recall in some cases but incurs longer query time than PM-LSH.

Recall-Time and OverallRatio-Time Curves. We proceed to study the relationship between the recall or overall ratio and the query time for (c, k)-ACP queries on all the datasets when varying their configurations to obtain different query times, such as c for PM-LSH, N for MkCP, L for the LSB-tree, and repeat times for ACP-P. The results are shown in Figs. 20 and 21. As the query quality and the query time represent the key tradeoff, the algorithms focus on returning relatively good results with much smaller query times than those of exact CP algorithms. The results show that all algorithms return more accurate results when more query time is used. They also show that PM-LSH achieves superior efficiency and accuracy when compared to the LSB-tree, ACP-P, and MkCP. This can be explained as follows. First, PM-LSH has a better distance estimator than the LSB-tree and ACP-P, so PM-LSH outperforms them with the same number of retrieved points. Second,



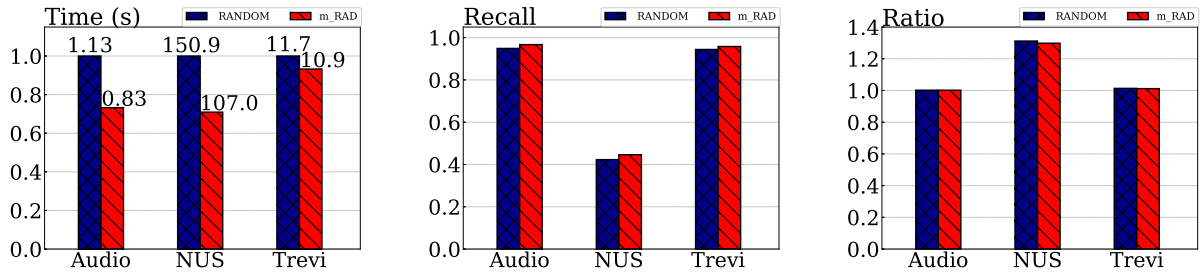
(a) $f_\gamma(x)$ on Audio (b) $f_\gamma(x)$ on Trevi (c) $f_\gamma(x)$ on NUS

Fig. 14: Effect of M for $f_\gamma(x)$



(a) $f_\gamma(x)$ on Audio (b) $f_\gamma(x)$ on Trevi (c) $f_\gamma(x)$ on NUS

Fig. 15: Effect of Dataset Cardinality for $f_\gamma(x)$



(a) Time (s) (b) Recall (c) OverallRatio

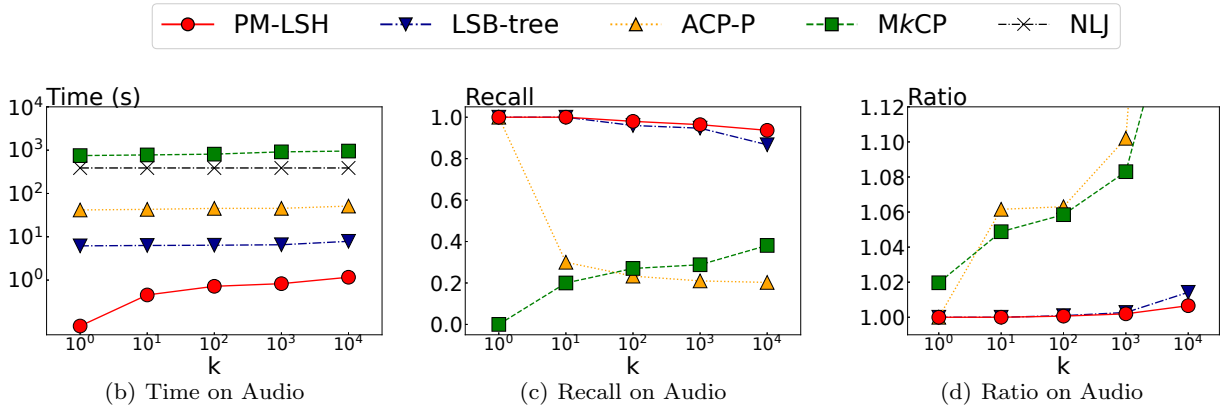
Fig. 16: Effect of Promote methods

PM-LSH uses a radius filtering technique to generate candidate pairs, which reduces substantially the cost of generating candidate pairs and provides a well-designed condition to terminate the process early. Third, the hyper-ball and hyper-ring space partitioning help reduce unnecessary verification overhead. In addition, although MkCP also finds approximate closest pairs in a space partitioning tree, it indexes high-dimensional data directly, which makes pruning difficult. Therefore, its query time is much larger than those of the other methods.

8 Related Work

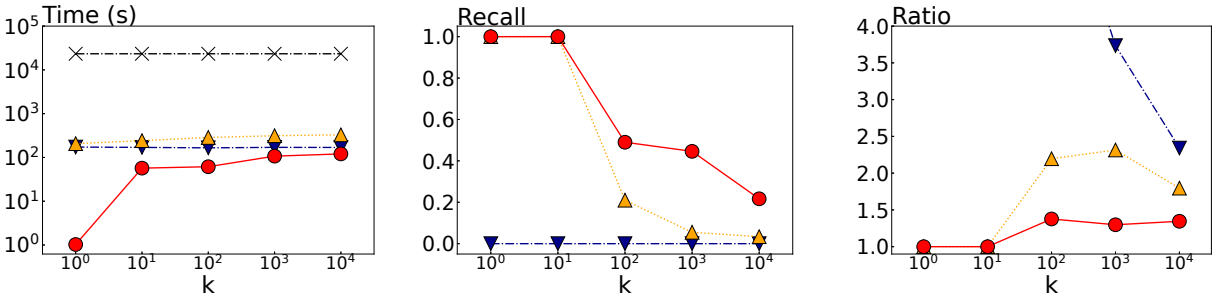
8.1 LSH for Nearest Neighbor Search

Locality-Sensitive Hashing (LSH) is a prominent approach to speeding up the processing of approximate nearest neighbor querying [5, 15, 16, 20, 35]. LSH was originally proposed by Indyk et al. [28] for use in Hamming space, and it has since attracted substantial attention due to its excellent performance. Datar et al. [15] propose an LSH function based on p-stable distributions in Euclidean space, which has become a mainstream method that yields low computation cost, a simple geometric interpretation, and a good quality guar-



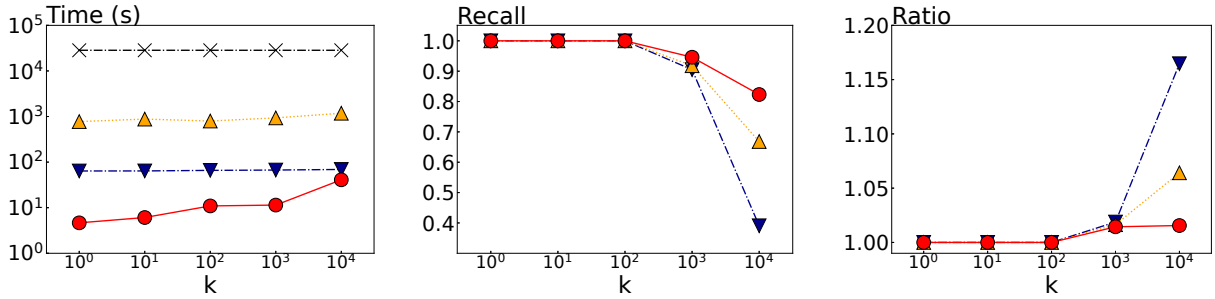
(b) Time on Audio (c) Recall on Audio (d) Ratio on Audio

Fig. 17: Performance on Audio when Varying k of CP Queries



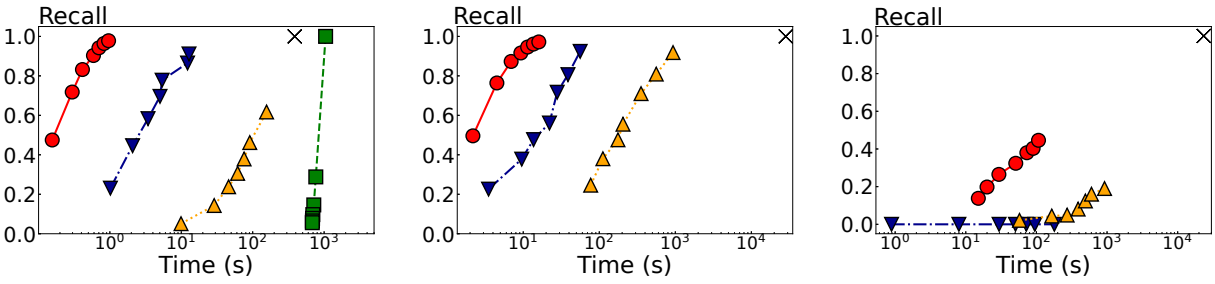
(a) Time on NUS (b) Recall on NUS (c) Ratio on NUS

Fig. 18: Performance on NUS when Varying k of CP Queries



(a) Time on Trevi (b) Recall on Trevi (c) Ratio on Trevi

Fig. 19: Performance on Trevi when Varying k of CP Queries

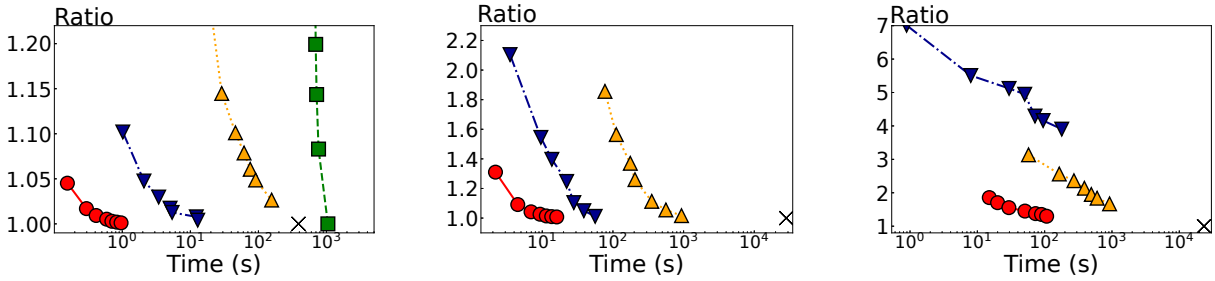


(a) Recall-Time on Audio (b) Recall-Time on Trevi (c) Recall-Time on NUS

Fig. 20: Recall-Time Curve for CP Queries

Table 6: Performance Overview of CP Queries

		PM-LSH	LSB-tree	ACP-P	MkCP	NLJ
Audio	Query Time (s)	**0.83**	12.82	384.60	756.26	388.03
	Overall Ratio	**1.002**	1.004	1.004	1.083	1.000
	Recall	**0.964**	0.911	0.930	0.288	1.000
MNIST	Query Time (s)	**33.59**	38.80	597.53	2946.45	1900.42
	Overall Ratio	**1.004**	1.006	1.005	1.103	1.000
	Recall	**0.937**	0.911	0.928	0.313	1.000
NUS	Query Time (s)	**107.03**	179.43	921.19	/	23322.10
	Overall Ratio	**1.298**	3.904	1.669	/	1.000
	Recall	**0.446**	0.005	0.190	/	1.000
Trevi	Query Time (s)	**10.92**	66.96	933.33	/	28400.6
	Overall Ratio	**1.014**	1.019	1.016	/	1.000
	Recall	**0.946**	0.905	0.918	/	1.000
Cifar	Query Time (s)	**91.83**	106.18	376.17	4140.29	2609.30
	Overall Ratio	**1.034**	1.070	1.047	1.094	1.000
	Recall	**0.721**	0.499	0.619	0.449	1.000
GIST	Query Time (s)	**81.77**	125.45	985.02	/	590321.43
	Overall Ratio	**1.101**	1.998	1.283	/	1.000
	Recall	**0.772**	0.16	0.504	/	1.000
Deep	Query Time (s)	**128.74**	132.73	129.16	/	174900.00
	Overall Ratio	**2.337**	2.420	7.115	/	1.000
	Recall	**0.445**	0.427	0.192	/	1.000



(a) Ratio-Time on Audio (b) Ratio-Time on Trevi (c) Ratio-Time on NUS

Fig. 21: Ratio-Time Curve for CP Queries

antee. Since then, many LSH methods build on this work to choose hash functions [18, 23, 27, 35, 47, 48]. In addition to the competitors introduced in Section 3, other proposals deserve mention. Based on a rigorous theoretical analysis, Panigrahy et al. [39] propose an entropy-based LSH, and Satuluri et al. [43] propose a BayesLSH. The former tries to reduce the number of hash tables by using multiple perturbed queries, and the latter aims to reduce the query time by estimating the similarity between data and query objects based on Bayes rule. However, both yield limited performance improvements as the assumptions made on the underlying dataset are hard to satisfy and verify. Another interesting proposal is LazyLSH [54], which supports queries in multiple l_p spaces by using one index, thus effectively reducing the space overhead. Another line of hashing-based methods is learning to hash (L2H) [50], which is orthogonal to our work. LSH uses predefined hash functions without considering the underlying dataset, while L2H learns tailored, data dependent hash functions. Many learning algorithms have been proposed, such as iterative quantization (ITQ) [21] and generate-to-probe QD ranking (GQR) [33].

8.2 High Dimensional Closest Pair Search

Closest-Pair (CP) search is an important problem in the database domain. Early studies target mainly low-dimensional closest pair search [12, 13, 26, 29, 44, 45]. They adopt spatial index structures, such as the R-tree and Quadtree and their variants, to organize the data.

However, these methods fail to handle high-dimensional closest pair search due to the curse of dimensionality. Corral et al. [11] propose a join method based on the VA-file, which is an array structure rather than a tree structure. Angiulli et al. [4] adopt the Z-curve to reduce the dimensionality and generate candidates in one-dimensional spaces. Tao et al. [49] propose an LSB-tree that uses a compound hash function to project points into a low-dimensional space. Next, they adopt the Z-curve to map the projected points into one-dimensional values that are indexed by a B-tree. Candidate point pairs are generated from the points with the same Z-values. However, $L = O(\sqrt{n})$ B-trees are required, thus causing a large space consumption. Mueen et al. [37] partition the data based on their distances to a pivot and thus map the high-dimensional data to a one-dimensional space. Other studies use LSH [32,52] or random projection [7] to reduce the dimensionality. For instance, Cai et al. [7] project the data directly into a one-dimensional space. Nearby points in the projected space are considered as candidate point pairs. However, the distance estimation is inaccurate and leads to unnecessary verification overhead.

Unlike the previously covered methods that use dimension reduction, yet other studies organize the original data directly by means of novel index structures, such as the LTC index [40], the multi-ball [17, 31], and the eD-Index [41]. Specifically, Gao et al. [19] propose several efficient algorithms using the count M-tree. However, these methods still suffer from the curse of dimensionality.

In addition, distributed indexing based approaches [32, 51] are proposed to accelerate CP search. These enable in-memory processing of large scale datasets.

9 Conclusion

We present a fast and accurate in-memory framework, called PM-LSH, for computing (c, k)-ANN and (c, k)-ACP queries with theoretical result quality guarantees. For NN queries, we first adopt the PM-tree to index the data points to be queried in a projected space. Second, in order to improve the distance estimation accuracy in the projected space, we develop a tunable confidence interval on the projected distance w.r.t. a given original distance. Finally, we propose an efficient algorithm to compute range queries using the PM-tree. The experimental study using seven widely used datasets shows that PM-LSH is capable of outperforming five competitors in terms of both query efficiency and result accuracy. Specifically, PM-LSH improves the query time by an average of 30% when compared to the closest competitor. When all competitors are given approximately the same query time, PM-LSH improves the recall by about 10% when compared to the closest competitor.

For computing CP queries, we also use the PM-tree to index the points in the projected space. We propose a radius filtering technique for finding closest pairs in the PM-tree. The experimental study shows that PM-LSH is capable of outperforming four competitors in terms of both query efficiency and result accuracy. Specifically, PM-LSH improves the query time by an average of 40% when compared to the closest competitor. When all the competitors are given approximately the same query time, PM-LSH improves the recall by about 50% when compared to the closest competitor.

Acknowledgments

This research is supported in part by the NSFC (Grants No. 61902134, 62011530437), the Hubei Natural Science Foundation (Grant No. 2020CFB871), and the Fundamental Research Funds for the Central Universities (HUST: Grants No. 2019kfyXKJC021, 2019kfyXJJS091).

References

1. M. A. Abdulhayoglu and B. Thijs. Use of locality sensitive hashing (LSH) algorithm to match web of science and scopus. *Scientometrics*, 116(2):1229–1245, 2018.
2. L. Amsaleg, O. Chelly, T. Furon, S. Girard, M. E. Houle, K. Kawarabayashi, and M. Nett. Estimating local intrinsic dimensionality. In *KDD*, pages 29–38, 2015.
3. A. Andoni and P. Indyk. LSH algorithm and implementation (E2LSH), 2016.
4. F. Angiulli and C. Pizzuti. An approximate algorithm for top-k closest pairs join query in large high dimensional data. *Data Knowl. Eng.*, 53(3):263–281, 2005.
5. M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *WWW*, pages 651–660, 2005.
6. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.
7. X. Cai, S. Rajasekaran, and F. Zhang. Efficient approximate algorithms for the closest pair problem in high dimensional spaces. In *PAKDD (3)*, volume 10939 of *Lecture Notes in Computer Science*, pages 151–163, 2018.
8. L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen. Efficient metric indexing for similarity search. In *ICDE*, pages 591–602, 2015.
9. P. Ciaccia, M. Patella, F. Rabitti, and P. Zezula. Indexing metric spaces with m-tree. In *SEBD*, pages 67–86, 1997.
10. P. Ciaccia, M. Patella, and P. Zezula. A cost model for similarity queries in metric spaces. In *PODS*, pages 59–68, 1998.
11. A. Corral, A. D'Ermiliis, Y. Manolopoulos, and M. Vassilakopoulos. VA-Files vs. R*-trees in distance join queries. In *ADBIS*, volume 3631 of *Lecture Notes in Computer Science*, pages 153–166, 2005.
12. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.

13. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Algorithms for processing k-closest-pair queries in spatial databases. *Data Knowl. Eng.*, 49(1):67–104, 2004.
14. A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, pages 271–280, 2007.
15. M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.
16. W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling LSH for performance tuning. In *CIKM*, pages 669–678, 2008.
17. K. Fredriksson and B. Braithwaite. Quicker similarity joins in metric spaces. In *SISAP*, volume 8199 of *Lecture Notes in Computer Science*, pages 127–140, 2013.
18. J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552, 2012.
19. Y. Gao, L. Chen, X. Li, B. Yao, and G. Chen. Efficient k-closest pair queries in general metric spaces. *VLDB J.*, 24(3):415–439, 2015.
20. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.
21. Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *TPAMI*, 35(12):2916–2929, 2013.
22. G. Gutierrez and P. Sáez. The k closest pairs in spatial databases - when only one set is indexed. *GeoInformatica*, 17(4):543–565, 2013.
23. P. Haghani, S. Michel, and K. Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *EDBT*, pages 744–755, 2009.
24. J. Harris and H. Stöcker. *Handbook of mathematics and computational science*. 1998.
25. J. He, S. Kumar, and S. Chang. On the difficulty of nearest neighbor search. In *ICML*, 2012.
26. G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, pages 237–248, 1998.
27. 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.
28. P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.
29. Y. J. Kim and J. M. Patel. Performance comparison of the R*-tree and the quadtree for knn and distance join queries. *TKDE*, 22(7):1014–1027, 2010.
30. B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, pages 2130–2137, 2009.
31. H. Kurasawa, A. Takasu, and J. Adachi. Finding the k-closest pairs in metric spaces. In *NTSS*, pages 8–13, 2011.
32. H. Li, S. Nutanong, H. Xu, C. Yu, and F. Ha. C2net: A network-efficient approach to collision counting LSH similarity join. *TKDE*, 31(3):423–436, 2019.
33. J. Li, X. Yan, J. Zhang, A. Xu, J. Cheng, J. Liu, K. K. W. Ng, and T. Cheng. A general and efficient querying method for learning to hash. In *SIGMOD*, pages 1333–1347, 2018.
34. 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. *TKDE*, 32(8):1475–1488, 2020.
35. Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.
36. Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Intelligent probing for locality sensitive hashing: Multi-probe LSH and beyond. *PVLDB*, 10(12):2021–2024, 2017.
37. A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, and M. B. Westover. Exact discovery of time series motifs. In *SDM*, pages 473–484, 2009.
38. A. Narang and S. Bhattacherjee. Real-time approximate range motif discovery & data redundancy removal algorithm. In *EDBT*, pages 485–496, 2011.
39. R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, pages 1186–1195, 2006.
40. R. Paredes and N. Reyes. Solving similarity joins and range queries in metric spaces with the list of twin clusters. *J. Discrete Algorithms*, 7(1):18–35, 2009.
41. S. S. Pearson and Y. N. Silva. Index-based R-S similarity joins. In *SISAP*, volume 8821 of *Lecture Notes in Computer Science*, pages 106–112, 2014.
42. M. Pirbonyeh, V. Rezaie, H. Parvin, S. Nejatian, and M. Mehrabi. A linear unsupervised transfer learning by preservation of cluster-and-neighborhood data organization. *Pattern Anal. Appl.*, 22(3):1149–1160, 2019.
43. V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.
44. J. Shan, D. Zhang, and B. Salzberg. On spatial-range closest-pair query. In *SSTD*, volume 2750 of *Lecture Notes in Computer Science*, pages 252–269, 2003.
45. H. Shin, B. Moon, and S. Lee. Adaptive and incremental processing for distance join queries. *TKDE*, 15(6):1561–1578, 2003.
46. T. Skopal, J. Pokorný, and V. Snásel. Nearest neighbours search using the PM-tree. In *DASFAA*, pages 803–815, 2005.
47. 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):1–12, 2014.
48. Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576, 2009.
49. Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.*, 35(3):20:1–20:46, 2010.
50. J. Wang, T. Zhang, J. Song, N. Sebe, and H. T. Shen. A survey on learning to hash. *TPAMI*, 40(4):769–790, 2018.
51. Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. In *KDD*, pages 829–837, 2013.
52. C. Yu, S. Nutanong, H. Li, C. Wang, and X. Yuan. A generic method for accelerating lsh-based similarity join processing. *TKDE*, 29(4):712–726, 2017.
53. B. Zheng, X. Zhao, L. Weng, N. Q. V. Hung, H. Liu, and C. S. Jensen. PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search. *PVLDB*, 13(5):643–655, 2020.

54. Y. Zheng, Q. Guo, A. K. H. Tung, and S. Wu. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *SIGMOD*, pages 2023–2037, 2016.
55. X. Zhou, B. Wu, and Q. Jin. Analysis of user network and correlation for community discovery based on topic-aware similarity and behavioral influence. *IEEE Trans. Hum. Mach. Syst.*, 48(6):559–571, 2018.