“© 2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.”

I/O Efficient Approximate Nearest Neighbour Search based on Learned Functions

Mingjie Li[§†], Ying Zhang[§†*], Yifang Sun[‡], Wei Wang[‡], Ivor W. Tsang[†], Xuemin Lin[‡]
[§] Zhejiang Gongshang University, China
[†] CAI, School of Computer Science, University of Technology Sydney, Australia
[‡] School of Computer Science and Engineering, University of New South Wales, Australia
Mingjie.Li@student.uts.edu.au, {Ying.Zhang, ivor.tsang}@uts.edu.au, {yifangs, weiw, lxue}@cse.unsw.edu.au

***Abstract*—Approximate nearest neighbour search (ANNS) in high dimensional space is a fundamental problem in many applications, such as multimedia database, computer vision and information retrieval. Among many solutions, data-sensitive hashing-based methods are effective to this problem, yet few of them are designed for external storage scenarios and hence do not optimized for I/O efficiency during the query processing. In this paper, we introduce a novel data-sensitive indexing and query processing framework for ANNS with an emphasis on optimizing the I/O efficiency, especially, the sequential I/Os. The proposed index consists of several lists of point IDs, ordered by values that are obtained by learned hashing (i.e., mapping) functions on each corresponding data point. The functions are learned from the data and approximately preserve the order in the high-dimensional space. We consider two instantiations of the functions (linear and non-linear), both learned from the data with novel objective functions. We also develop an I/O efficient ANNS framework based on the index. Comprehensive experiments on six benchmark datasets show that our proposed methods with learned index structure perform much better than the state-of-the-art external memory-based ANNS methods in terms of I/O efficiency and accuracy.**

I. Introduction

The problem of nearest neighbour search in high dimensional space aims at finding an object which has the smallest distance to a query under a specified distance measure in a reference database. It has been a fundamental technique in many applications, such as recommendation [5], feature matching [31], and information retrieval [41]. However, finding the exact nearest neighbor in high dimensional data is computationally expensive due to the curse of dimensionality [18]. To make a tradeoff between the efficiency and accuracy, approximate nearest neighbor search (ANNS) is extensively studied and showed more efficient in many practical problems, thus attracting numerous interests of researchers.

Most of the existing approaches proposed in the literature are *main-memory* algorithms which focus on engineering the best trade-off between the CPU cost and the accuracy. However, increasing amount of applications are being produced and operated on huge volume of the high dimensional data, where external I/O storage and I/O-centric query processing are needed. For instance, billions of objects (users and items) are mapped to 160-dimensional points via a deep learning model at Alibaba for user recommendation [36]. To handle the large scale data, it is desirable to develop highly efficient *external-memory* algorithms for better scalability, which is the focus of this paper.

*Corresponding author

State-of-the-art In the past decades, massive approximate algorithms for ANNS were proposed in the literature[1], which can be classified into three categories: *hash-based*, *partition-based* and *graph-based* approaches. Due to the nature of partition-based and graph-based approaches, it is difficult to develop corresponding I/O efficient external memory-based ANNS algorithms because it is infeasible to cluster nearby objects in high dimensional space in the external memory. Existing external memory-based ANNS algorithms are mainly hash-based approaches, which can be further classified into two categories.

1) *Random hash based approaches*: A number of I/O efficient LSH-based ANNS methods were proposed, such as LSH-forest [35], C2LSH [13], QALSH [19], which aim to obtain a good trade-off between search accuracy and I/O efficiency with theoretical guarantees. I-LSH [24] is the state-of-the-art I/O efficient random hash based method, which has a small I/O cost by using an incremental, rather than exponentially expanding, search strategy. All these approaches rely on the sorted-lists where each list corresponds to the hashed values of the objects; that is, the objects in the high dimensional space are mapped into multiple sorted lists (i.e., a low-dimensional embedding space) and each list is further divided into consecutive buckets. The query processing consists of a set of sequential scans on these lists/buckets. These approaches are I/O efficient in the sense that sequential I/Os are invoked for the search of these lists/buckets. However, the search quality of these methods is far from satisfactory[2] because they employ random projections, which are independent of the actual data distribution.
2) *Learning to hash (L2H) based approaches*: These algorithms significantly outperform the random hash based methods [23], as they can learn data-sensitive hash functions to generate high-quality low dimensional embeddings that preserve the locality in the expected sense rather than in the worst-case sense. However, most of these L2H methods are main-memory based, and do not consider the query processing on external memory. To the best of our knowledge, PQBF [25] and AOSKNN [17] are the only two existing L2H methods that aim to optimize the I/O performance where the objects and their embeddings are located in external memory. However, they still require excessive amount of random I/Os, which are much more expensive than sequential I/Os. Moreover, the hash functions learning in [25], [17], which is the key

[1] e.g., see a recent experiment paper [23].
[2] e.g., See Fig. 7 as well as [23].

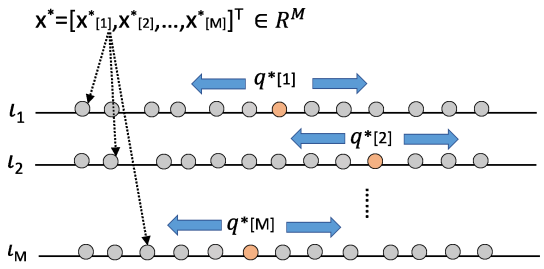


Fig. 1: Illustration of our idea of index and query processing. Grey points are embedding values of data points, orange points are embedding values of the query point (i.e., $\mathbf{q}^*[1], \ldots, \mathbf{q}^*[M]$).

to the performance of the methods, is independent to the index structure, and hence there is no optimization for the I/O efficiency in the learning of their hash functions.

Our Approach Motivated by the above analysis, in this paper, we aim at designing external-memory based indexing and query processing techniques for ANNS such that we can (i) maximally use the inexpensive sequential I/Os during the search; and (ii) conduct end-to-end learning of the hashing functions, which are also aware of the I/O characteristics.

To address the first goal, after the learned hashing (i.e., mapping) function maps every point to its corresponding low dimensional embedding, we build indexes as a set of M sorted lists. Each entry of a list is of the form (ID, value) which records the object ID and its embedding value on a particular dimension; the list is sorted in ascending order of values. By doing this, the objects in high dimensional space are mapped into M sorted lists by learned mapping functions. Our query processing procedure only makes bi-directional sequential access to each list and hence fully exploit the sequential I/Os (See Fig. 1 for an illustration).

To address the second goal, our idea is to leverage machine learning methods to preserve locality of point objects in the embedding space. This is done by designing novel loss function that is aware of the block-based I/O assess characteristics, novel relaxation and optimization techniques. In addition, we consider different models that learn linear hashing functions and non-linear ones, respectively.

Contributions Our contributions are summarized as follows:

- We develop an I/O efficient external-memory ANNS framework, which consists of a set of sorted lists and a querying processing algorithm.
- We propose two hash learning methods for linear and non-linear hash functions. Particularly, linear hashing method learns the sorted lists separately by penalizing the order mismatch. A fast training method based on stochastic gradient descent (SGD) is developed for optimization. Non-linear hashing method learns a neural network for our framework. A training architecture based on a fully-connected neural network is constructed for the optimization of the loss function.
- Comprehensive experiments on six large scale high dimensional datasets demonstrate that our proposed methods outperform the state-of-the-art ANNS techniques in terms of I/O efficiency and search accuracy.

Roadmap The rest of the paper is organized as follows. Section II introduces the problem definition and the related work. Our ANNS framework is presented in Section III. Section IV and Section V present the details of our linear and non-linear hashing methods, respectively. Some discussions are presented in Section VI. Experimental results are reported in Section VII. Finally, we summarize our proposed methods in section VIII.

TABLE I: Summary of Notations

Notation	Definition
$\mathbf{D}$	The dataset represented in a $d \times N$ *design matrix*; each column is a d-dimensional point (object)
$\|\mathbf{x}, \mathbf{y}\|$	The Euclidean distance between $\mathbf{x}$ and $\mathbf{y}$
l	A sorted list with entries of form (ID, value)
M	The dimensionality of the embedding (also the number of sorted lists)
$\mathcal{H}$	The (learned) mapping function to a lower dimensional embedding space
$\mathbf{W}$	The parameters of the linear mapping functions
$\boldsymbol{\theta}$	The parameters of the non-linear mapping functions

II. Background

In this section, we first present the problem definition of approximate nearest neighbour search (ANNS) in Section II-A. Section II-B introduces related work for this problem. Section II-C presents three state-of-the-art external-memory ANNS algorithms.

A. Problem Definition

Notation We use bold lower case letters to denote (column) vectors and bold upper case letters to denote matrices. We use $\mathbf{x}[j]$ to denote the j-th dimension value of a vector $\mathbf{x}$, and $\mathbf{W}[j, i]$ to denote the value at j-th row and i-th column in matrix $\mathbf{W}$. The $\mathbf{w}_i$ denotes a vector identified by an index i. Other important notations in paper are given in Table I.

Approximate Nearest Neighbor Search We denote the dataset of N d-dimensional points as a *design matrix* $\mathbf{D} \in \mathcal{R}^{d \times N}$, where the i-th *column* is a d-dimensional data point $\mathbf{x}_i$. We are particularly interested in the high dimensional case with Euclidean distance, i.e., d is a large number (e.g., $d \geq 100$) and the Euclidean distance is defined as $\|\mathbf{x}, \mathbf{y}\| = \sqrt{\sum_j^d (\mathbf{x}[j] - \mathbf{y}[j])^2}$. Given a query $\mathbf{q} \in \mathcal{R}^d$, *approximate nearest neighbor search* (ANNS) aims at finding the objects in $\mathbf{D}$ that are closest to $\mathbf{q}$. Let the $\mathbf{o}$ be nearest neighbor in $\mathbf{D}$ with respect to a query $\mathbf{q}$, and let $\mathbf{p}$ be the ANNS result returned by an algorithm, we can measure the quality of $\mathbf{p}$ by the ratio of the distance to the query over the nearest neighbor distance, i.e., $\frac{\|\mathbf{p}, \mathbf{q}\|}{\|\mathbf{o}, \mathbf{q}\|}$. It is easy to extend the definition to the top-k version of ANNS.

B. Related Work

Approximate nearest neighbor search (ANNS) in high dimensional space has been extensively investigated in the literature and a large number of algorithms were proposed for this problem. In terms of index storage, these methods can be classified into two categories: *main memory-based* and *external memory-based* approaches.

(1) Main Memory-based ANNS Methods

Algorithms in this type can be mainly classified into three categories: *Hashing-based*, *Partition-based* and *Graph-based*.

Hashing-based methods attract many research efforts due to its good advantages on ANNS problem and the ease of implementation. Representatives are locality sensitive hashing (LSH) [20] and learning to hash (See [38] for a comprehensive survey). The key idea of LSH method is to map data objects from high dimensional space into different buckets by predefined hash functions such that the similar items are projected into the same (or similar) buckets with higher probability than dissimilar ones. LSH-based methods are data-independent, the choice of hash function is crucial. Due to the need of theoretical guarantee, LSH-based methods usually leverage random linear projection as hash functions. By contrast, L2H methods are data-dependent, and have been shown to outperform the LSH-based methods since it can make use of the data distribution to learn specific hash functions. In the past, L2H has been extensively studied and numerous algorithms were proposed such as spectral hashing [40], PCA hashing [15], optimized product quantization [14], and order reserving hashing [37], [39]. Recently, deep learning to hash methods, such as hashNet [3], stochastic generative hashing [4] and unsupervised deep hashing [30], are proposed to apply deep learning techniques to hashing for ANNS, and achieve very promising experimental results with higher training cost. However, their goal is to learn compact hash codes which is efficient for nearest neighbor search in main memory but cannot trivially be extended to support I/O efficient nearest neighbor search in external memory.

Partition-based methods can be deemed as dividing the entire high dimensional space into multiple disjoint regions. The partition process often carries out in a recursive way, so a tree or a forest are ultilized as the index of a partition-based, where the ANNS can be executed. Representatives include Random-Projection Tree [6], Optimized KD-Tree [31] and FLANN [27].

Graph-based methods construct a proximity graph where each point object corresponds to a node and edges connecting some nodes define the neighbor relationship. The main idea of these methods is that a neighbor's neighbor is likely to also be a neighbor. The search could be performed by iteratively expanding neighbors' neighbors in a best-first search strategy following the edges. A number of algorithms in this category were proposed, such as K-Graph [8], Small World Graph [26], DPG [23] and Navigating Spreading-out Graph [12].

(2) External Memory-based ANNS Methods

A set of external memory-based approaches have been proposed in the literature such as MEDRANK [10], LSB-fortest [35], C2LSH [13], QALSH [19], I-LSH [24] and M-tree [29]. The key idea of these hash-based methods is to map the point objects into a set of M lists by M random hash functions. The objects in each list are pre-ordered and hence their indexes are database-friendly in that sequential I/Os are applied during the search. M-tree [29] is a general data structure for indexing data objects in high dimensional metric space. It is constructed using a metric function and relies on the triangle inequality for efficient approximate nearest neighbor queries. Recently, two I/O efficient algorithms are proposed in [25], [17], which leverage existing main memory-based learning to hash methods for I/O efficient index construction. The details of three state-of-the-art external-memory ANNS algorithms are presented in Section II-C.

In addition to high dimensional point data, the problem of nearest neighbor search has also been investigated on more complicated data such as data series (e.g., [9], [2]), uncertain data (e.g., [42]) and networks (e.g., [41]).

C. State-of-the-art.

In this subsection, we introduce three state-of-the-art external memory-based ANNS algorithms.

(1) **I-LSH** [24]. Very recently, Liu et al. [24] proposes I-LSH to dramatically reduce the I/O cost of ANNS with theoretical guarantee. Unlike the previous LSH methods [13], [19], which expand the bucket width in an exponential way, I-LSH adopts a more natural search strategy to incrementally access the hash values of the objects, thus can greatly reduce I/O cost under the same theoretical guarantee.

(2) **PQBF** [25]. Product Quantization (PQ) [21] is a widely used method for ANNS, which decomposes the original vector space into the Cartesian product of L lower dimensional subspaces, and performs vector quantization [16] in each subspace separately. A vector is then represented by a short code composed of its subspace quantization indices (i.e., PQ codes). Liu et al. [25] proposed the first I/O-efficient PQ-based solution for ANNS. They design a linear order on the PQ codes by employing the Z-order [34], where a lower bound for the AQD distance (i.e., an approximation of original Euclidean distance) can be achieved. Then they design an index called PQB+-forest to support efficient similarity search on AQD. Specifically, PQB+-forest first creates a number of partitions of the PQ codes by a coarse quantizer and then builds a B+-tree, called PQB+-tree, for each partition. The search process is expedited by focusing on a few selected partitions that are closest to the query, as well as by the pruning power of PQB+-trees. Note that although the objects in PQB+-tree are indexed by B+-tree, the random I/Os are invoked because it is unlikely to ensure the nearby objects accessed during the search are allocated at the adjacent pages in one order alone.

(3) **AOSKNN** [17]. Gu et al. proposed an external memory-based ANNS algorithm, namely AOSKNN, in [17] based on the "projection-filter-refinement" framework. Specifically, they adopt PCA to embed the high-dimensional point objects into a low-dimensional space. Then, a filter condition is inferred to execute pruning over the projected data. As an R-tree-based index is employed to organize the embedded objects in low dimensional space, random I/Os are invoked during the search.

III. Our ANNS Framework

In this section, we present the details of our proposed indexing and query processing methods for ANNS.

A. Our ANNS Solution

Overview Our idea is to use a mapping (i.e., hashing) function[3] to map d-dimensional points to M-dimensional points, where $M \ll d$. This is known as *embedding* in Machine Learning. Specifically, if we denote the mapping function $\mathcal{H}$:

[3]In this paper, we use *mapping function* and *hashing function* exchangeably when the context is clear.

$\mathcal{R}^d \rightarrow \mathcal{R}^M$, then $\mathbf{x}^* = \mathcal{H}(\mathbf{x})$ is the corresponding embedding for $\mathbf{x}$. We also slightly abuse the notation $\mathbf{X}^* = \mathcal{H}(\mathbf{X})$ to obtain the embeddings as a matrix in $\mathcal{R}^{M \times N}$, where $\mathcal{H}$ is applied to each column of the input design matrix.

We then index the values of the embedded vectors on each dimension individually as a sorted list. We perform the same embedding process for the query $\mathbf{q}$ and then use sequential I/Os to get T candidates. Finally, we perform re-ranking on the T candidates followed by verification to return the top-k nearest points to the query in a progressive manner.

Obviously, our method has several advantages such as leveraging sequential I/Os, and simple and flexible enough to be implemented within a database system where the sorted lists can be easily organized using B^+-trees. The key to make such simple method effective is the quality of the mapping function. We will provide details on how to learn such data-sensitive mapping functions, both linear and non-linear ones, in Sections IV–V.

Indexing After applying the mapping function $\mathcal{H}$, we obtain the collection of embedded vectors $\mathbf{x}_i^*$ ($i \in [1, N]$). We then index each dimension values in a sorted list, which results in M sorted lists. The details of our indexing method can be seen in Algorithm 1. Each entry in the list consists of only a point ID and a dimension value, which is typically 8 bytes.

Nonetheless, we have the option to further reduce the index size by almost 50% by exploiting the external memory access characteristics. Since we consider external memory scenario, the basic unit of access to the index is one page with page size as b bytes. We only need to include the dimension value of the *first* entry on each page, and omit the dimension values of the rest of the entries on the same page. This is similar to the optimization in a clustered index in database systems. In this way, each page consists of a dimension value and $\lfloor \frac{b}{4} - 1 \rfloor$ point IDs. We denote the page as (*IDs*, *value*).

Querying Processing We show the query processing algorithm in Algorithm 2. The searching operation starts by applying the function $\mathcal{H}$ to the query $\mathbf{q}$ to obtain its embedding $\mathbf{q}^*$. Then, it locates the positions where each of the dimension values of $\mathbf{q}^*$ will be on the corresponding sorted lists. And then we insert the pages closest to $\mathbf{q}^*$ on each list into a priority queue. We obtain the page with the highest priority, and remove it from the queue. Then we insert the next page closest to $\mathbf{q}^*$ on the associated list into the priority queue.

We access and bookkeep the entries of the current page in the ascending order of their *rank positions* on the list. We define the *rank position* of a point $\mathbf{x}_i$ with respect to the query $\mathbf{q}$ on list l_m as:

$$r_m(\mathbf{q}, \mathbf{x}_i) = \sum_{j=1}^{N} \mathbf{1}_{|\mathcal{H}(\mathbf{q})[m] - \mathcal{H}(\mathbf{x}_i)[m]| > |\mathcal{H}(\mathbf{q})[m] - \mathcal{H}(\mathbf{x}_j)[m]|} + 1, \tag{1}$$

where the $\mathbf{1}_p$ is the indicator function which returns 1 if the predicate p is true, and returns 0 otherwise. Intuitively, this is 1 plus how many other points (i.e., $\mathbf{x}_j$) has a smaller distance to query on list l_m than that of $\mathbf{x}_i$. The closest point will have a rank position of 1. For simplicity, we abbreviate $r_m(\mathbf{q}, \mathbf{x}_i)$ as $r(\mathbf{x}_i)$ henceforth.

If a point has been seen on all the M list, then we add it to the candidate set $\mathcal{C}$. When the size of the candidate set exceeds a preset value T, the searching stops.

Algorithm 1: ***Indexing***($\mathbf{D}$, $\mathcal{H}$, M)

Input : $\mathbf{D}$: The dataset as a design matrix in $\mathcal{R}^{d \times N}$,
$\mathcal{H}$: The learned mapping function,
M : The dimensionality of the embeddings (also the number of sorted lists)
Output: $\mathcal{L}$: The M sorted lists

```
D* ← H(D);
for m ← 1 to M do
    Y, I ← sort (D*, m) according to the m-th dimension;   /* Y, I ∈ R^{M×N}, where Y contains the dimension values and I contains the associated IDs */
    l_m ← an empty list;
    for i ← 1 to N do
        l_m[i] ← (Y[m, i], I[m, i]);
return L = {l_1, l_2, ..., l_M};
```

Algorithm 2: ***Querying***($\mathcal{L}$, $\mathcal{H}$, $\mathbf{q}$, T)

Input : $\mathbf{q}$: The query point in $\mathcal{R}^d$,
$\mathcal{L}$: The sorted lists $\{l_1, l_2, \ldots, l_M\}$,
$\mathcal{H}$: The mapping function,
T : A parameter to control the size of the candidate set (to be re-ranked and verified)
Output: The approximate nearest neighbour of $\mathbf{q}$

```
C ← ∅;
hits ← an empty hash table;
queue ← an empty priority queue;
isTerminated ← false;
q* ← H(q);
for m ← 1 to M do
    o ← the page closest to q*[m] in list l_m;
    Insert (o, m) into queue with priority as -|o.value − q*[m]|;
while isTerminated = false and queue ≠ ∅ do
    (o, m) ← the node with highest priority in queue;
    o* ← the next page closest to q*[m] in list l_m;
    Remove (o, m) from queue and insert (o*, m) into queue with priority as -|o*.value − q*[m]|;
    for each id in o.IDs do
        hits[id] ← hits[id] + 1;
        if hits[id] = M then
            C ← C ∪ {id};
            if |C| >= T then
                isTerminated ← true;
                break;
Re-rank the candidates in C and verify their distances to the query;
return The point that has the smallest distance to the query;
```

The last step is the re-ranking followed by the verification. Although all the T candidates have all been seen on the M lists, this only indicates that they are not too "faraway" from the query in the original space. We would like to reorder them according to some criteria that favor those who are actually close to the query. Here, we adopt a simple re-ranking method: we sum up the rank position of each candidates on

all the lists, and reorder all the candidates in ascending order of this sum. Finally, we verify each candidate on the re-ranked candidate list by calculating its distance to the query. We also keep the candidate that has the minimum distance during the verification. Note that this can be easily extended to approximate k-nearest neighbour search (k-ANNS) with the help of an extra priority queue.

B. Performance Analysis

In this subsection, we analyze the I/O complexity of the indexing and querying processing of our ANNS framework.

Let b be the page size. The I/O complexity for the indexing is $\mathcal{O}(\frac{Nd}{b} + \frac{NM}{b})$, whereas the two terms correspond to the cost of learning the data-sensitive mapping function $\mathcal{H}$, and the cost of creating the M sorted list. We estimate the mapping function learning cost as $\mathcal{O}(\frac{Nd}{b})$ as most model learning (including the two we will present in the paper) requires fixed number of iterations over the data.

For the query processing algorithm, assume that we access p pages on lists before we collected T candidates, then this phase costs $\mathcal{O}(p)$ *sequential* I/Os, which is equivalent to $\mathcal{O}(\epsilon \cdot p)$ (random) I/Os, where ϵ is typically in the range of $[0.01, 0.1]$. Finally, the verification requires another $\mathcal{O}(\frac{rTd}{b})$ I/Os, where $r \in (0, 1)$.

IV. Learning to Index by Linear Hashing

In this section, we present the first method to learn the mapping function from the data. The idea is to preserve the ordering information in the resulting embedding space. We will formally present the objective function based on the order preservation idea, followed by its optimization, involving the relaxation and stochastic gradient descent (SGD) algorithm.

A. Linear Model and Its Objective Function

We first consider linear mapping functions, which encompass linear projection function considered by learning-to-hash methods [32], [39] and locality sensitive hashing functions [7]:

$$h(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$$

where $\mathbf{w} \in \mathcal{R}^d$ is the parameter of the hash function. Using M such hash functions with different parameters, we can obtain the mapping function $\mathcal{H}$ as:

$$\mathcal{H}(\mathbf{x}) = [h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_M(\mathbf{x})]^\top$$

In order to learn these parameters $\mathbf{w}_m$ ($m \in [1, M]$) from the data, we will prepare a set of training data, which consists of uniform samples from the real query workload, or samples from the data itself if the query workload is not available. Next, we need to define the loss function such that we can learn the $\mathbf{w}_m$ values that achieve the minimum loss value.

Consider a given query, for any $\mathbf{w}_m$ value, we can obtain the order induced by rank position of all N data points based on the function h_m (denoted as l^s_m or simply l^s if there is no ambiguity). We can also easily compute the ground truth ordering of all data points based on the distance to query in the original space (denoted as l^o). We would like to define how much penalty to apply if the rank position ordering does not completely agree with the ground truth ordering. Although there exists measures, such as Kendall's tau coefficient, that defines the distance between two orderings, they are not considering the page-based access characteristics during our query processing, not differentiable and are costly to compute. Instead, we design our own measure based on the idea of block order.

We divide the two ordered lists into L parts, called blocks, with each part containing the same number of objects (assuming N is a multiple of L). After the division, we have $l^o = [l^o_1, l^o_2, \ldots, l^o_L]$, and $l^s = [l^s_1, l^s_2, \ldots, l^s_L]$, where l^o_i and l^s_i are the subset of l^o and l^s, respectively. Then for each block, if the objects in l^o_i are not preserved in the corresponding l^s_i, this incurs a penalty of 1, otherwise, the penalty is 0. Therefore, given any l^s induced by $\mathbf{w}_m$, we define the *loss function* as:

$$J^*(\mathbf{w}_m) = \sum_{i=1}^{L} \sum_{\tilde{x} \in l^o_i} \mathbf{1}_{r(\tilde{x}) \in [t \cdot (i-1), t \cdot i)} \tag{2}$$

$t = \frac{N}{L}$ is the length of the bucket, i.e., the number objects in each partition l^o_i ($1 \le i \le L$). That is, we penalize points that their ranking in the original space and the embedding space are not in the *same* block. If we let the block size be the page size, then this agrees with our page-based sequential access in the query processing, as all point IDs will be accessed as long as they are in the same page.

Therefore, our *final loss function* for all sorted lists can be written as:

$$J^*(\mathbf{W}) = \textstyle\sum_{m=1}^{M} J^*(\mathbf{w}_m) + \lambda \cdot \|\mathbf{W}^\top \mathbf{W} - \mathbf{I}\|_F, \tag{3}$$

where $\|\mathbf{M}\|_F = \sum_{i,j} \mathbf{M}[i,j]^2$ is the Frobenius norm of a matrix and is a common type of quadratic regularizer [28], λ is the hyper-parameter that controls the degree of regularization, and $\mathbf{W}$ is the concatenation of all $\mathbf{w}_m$ ($m \in [1, M]$). Note that $\mathbf{W}^\top \mathbf{W} = \mathbf{I}$ forces all projection vectors to be orthogonal to each other, such that our model is able to learn M different hash functions.

B. Relaxation and Optimization

The loss function (3) is neither convex nor smooth due to the indicator function. This means that it is hard to optimize the function numerically. We adopt a common approach in Machine Learning that relaxes the discrete function into a continuous and differentiable surrogate loss function and optimize this surrogate instead.

Relaxation.

We utilize the fact that the *sigmoid function* (as shown in Fig. 2(left)), $\sigma(z) = \frac{1}{1+\exp(-z)}$, is a continuously differentiable approximation to the indicator function $\mathbf{1}_p$.

We first replace the absolute value function in Eq. (1) by taking the square to both sides of the predicate. Afterwards, we apply the sigmoid relaxation and obtain the *approximate rank position* as:[4]

$$\tilde{r}(\mathbf{x}_i) = \sum_{j=1}^{N} \sigma((h(\mathbf{q}) - h(\mathbf{x}_i))^2 - (h(\mathbf{q}) - h(\mathbf{x}_j))^2) + 1 \tag{4}$$

[4]We omit m and $\mathbf{q}$ in $\tilde{r}_m(\mathbf{q}, \mathbf{x}_i)$ for simplicity.

Similarly, the loss functions in (2) and (3) are also relaxed as

$$J(\mathbf{w}_m) = \sum_{i=1}^{L} \sum_{\tilde{\mathbf{x}} \epsilon l_i^o} (\sigma(t \cdot (i-1) - \tilde{r}(\tilde{\mathbf{x}})) + \sigma(\tilde{r}(\tilde{\mathbf{x}}) - t \cdot i)) \quad (5)$$

and

$$J(\mathbf{W}) = \sum_{m=1}^{M} J(\mathbf{w}_m) + \lambda \cdot \|\mathbf{W}^\top \mathbf{W} - \mathbf{I}\|_F. \quad (6)$$

Progressive Stochastic Gradient Descent.

Stochastic Gradient Descent (SGD) is a fundamental and popular numerical optimization method. To minimize a scalar function with vector parameters, i.e., $f(\mathbf{w})$, the Gradient Descent (GD) algorithm starts with carefully initialized parameter values, and updates it to the next value that maximally minimizes the function value in a small neighborhood by following the negative of the gradient ($\nabla f = [\frac{\partial f}{\partial \mathbf{w}[1]}, \ldots, \frac{\partial f}{\partial \mathbf{w}[M]}]^\top$) direction. SGD is its stochastic version, where the gradient is estimated using a random sample of the training data. SGD drastically reduces the gradient computation cost but may get noisy gradients. In practice, a mini-batched SGD achieves the balance between GD and SGD by estimating the gradient on a mini-batch of B random samples, where B is a hyper-parameter.

Consider our relaxed final loss function (6), it is still non-convex. We cannot guarantee to obtain the optimal solution and hence there are several strategies to learn a sufficiently good solution. One strategy is to learn the entire set of parameters $\mathbf{W}$ simultaneously, and another is to learn each function (i.e., the corresponding $\mathbf{w}_m$) one by one in an incremental manner. We found that the latter approach typically leads to better performance and hence we introduce its details below.

Assum that we have learned the first $m-1$ linear hash functions, i.e., $\{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{m-1}\}$ are learned. We formulate the optimization problem for the next function as

$$J_{\text{inc}}(\mathbf{w}_m) = J(\mathbf{w}_m) + \mu \cdot (\mathbf{w}_m^\top \mathbf{w}_m - 1)^2 + \lambda \cdot \sum_{j=1}^{m-1} (\mathbf{w}_j^\top \mathbf{w}_m)^2 \quad (7)$$

The last two items in (7) are: (i) The first is the additional regularization that encourages $\mathbf{w}_m$ to be an unit vector; (ii) The second is the orthogonality regularization. The square is used to guarantee that the last item (i.e., the sum) is always positive.

Next, we apply the SGD algorithm to minimize the $J_{\text{inc}}(\mathbf{w}_m)$. Firstly, we need to obtain the gradient of the $\tilde{r}(\mathbf{x}_i)$ in (4) with respect to $\mathbf{w}_m$, which is:

$$\nabla_{\mathbf{w}_m} \tilde{r}(\mathbf{x}_i) = \sum_{j=1}^{N} \sigma'(z) \nabla_{\mathbf{w}_m} z \quad (8)$$

where

$$\begin{aligned} \sigma'(z) &= \sigma(z) \cdot (1 - \sigma(z)) \\ z &= (h(\mathbf{q}) - h(\mathbf{x}_i))^2 - (h(\mathbf{q}) - h(\mathbf{x}_j))^2 \\ \nabla_{\mathbf{w}_m} z &= 2((h(\mathbf{q}) - h(\mathbf{x}_i))(\mathbf{q} - \mathbf{x}_i) - (h(\mathbf{q}) - h(\mathbf{x}_j))(\mathbf{q} - \mathbf{x}_j)) \end{aligned}$$

Algorithm 3: ***Optimized Incremental SGD***($\bar{\mathbf{D}}$, $\mathbf{Q}$, M)

Input : $\bar{\mathbf{D}}$: The training dataset,
$\mathbf{Q}$: The training query set,
M: The number of projection vectors
Output: The parameter of the linear mapping functions $\mathbf{W}$

```
Calculate the ground-truth lists l^o for each query in Q with respect to D̄ in R^d ;
for m = 1 to M do
    if m = 1 then
        w_m ← a vector where each component is sampled from the Gaussian distribution N(0,1);
    else
        w_m ← a randomly sampled vector from the null space of {w_1, ..., w_{m-1}};
    w ← w / sqrt(w^T w);                    /* normalize */;
    for i ← 1 to max_iteration do
        Sample a batch of queries from Q and obtain their associated order lists from l^o;
        Calculate the gradient (Equation (9)) for this batch of queries;
        Update the w_m (Equation (10));
return W = {w_1, ..., w_M};
```

Then the gradient of $J_{\text{inc}}(\mathbf{w}_m)$ in (7) is:

$$\begin{aligned} \nabla_{\mathbf{w}_m} J_{\text{inc}}(\mathbf{w}_m) = \sum_{i=1}^{L} \sum_{\tilde{\mathbf{x}} \epsilon l_i^o} \Big(\nabla_{\mathbf{w}_m} \tilde{r}(\tilde{\mathbf{x}}) \cdot (\sigma'(z_1) - \sigma'(z_2)) \Big) \\ + 4\mu(\mathbf{w}_m^\top \mathbf{w}_m - 1)\mathbf{w}_m + 2\lambda \sum_{j=1}^{m-1} \mathbf{w}_j^\top \mathbf{w}_m \mathbf{w}_j \end{aligned} \quad (9)$$

where the $z_1 = \tilde{r}(\tilde{\mathbf{x}}) - t \cdot i$ and $z_2 = t \cdot (i-1) - \tilde{r}(\tilde{\mathbf{x}})$.

Now consider using the SGD with a mini-batch of size B, i.e., the mini-batch consists of B objects randomly sampled from the training query set $\mathbf{Q}$. The gradient descent updating rule for $\mathbf{w}_m$ is:

$$\mathbf{w}_m^{(t+1)} = \mathbf{w}_m^{(t)} - lr \cdot \frac{1}{B} \sum_{j=1}^{B} \nabla_{\mathbf{w}_m} J_{\text{inc}}(\mathbf{w}_m, \mathbf{q}_j) \quad (10)$$

where lr is the learning rate and t is the iteration index, and the$\{\mathbf{q}_1, \ldots, \mathbf{q}_B\}$ is a random subset of $\mathbf{Q}$. We give out the complete mini-batch SGD algorithm for learning the data-sensitive linear functions in Algorithm 3.

There is still a performance issue with the algorithm in that one iteration over the mini-batch needs $\mathcal{O}(B \cdot N^2)$, which is unacceptable if N is large. We take the following measure to mitigate the performance problem by sub-sampling. Specifically, we can interpret the first item in the gradient in (9) as the expectation over all $\tilde{\mathbf{x}}$ in the dataset and then approximate it using a set of random samples $\mathcal{S}$:

$$\begin{aligned} &\mathbf{E}_{\tilde{\mathbf{x}} \epsilon \mathcal{D}} [\![\nabla_{\mathbf{w}_m} \tilde{r}(\tilde{\mathbf{x}}) \cdot (\sigma'(z_1) - \sigma'(z_2))]\!] \\ &\approx \frac{1}{|\mathcal{S}|} \sum_{\tilde{\mathbf{x}} \epsilon \mathcal{S}} \nabla_{\mathbf{w}_m} \tilde{r}(\tilde{\mathbf{x}}) \cdot (\sigma'(z_1) - \sigma'(z_2)) \end{aligned}$$

In addition, instead of using the entire reference dataset, we use a training dataset $\bar{\mathbf{D}}$ of size γN, which is randomly sampled

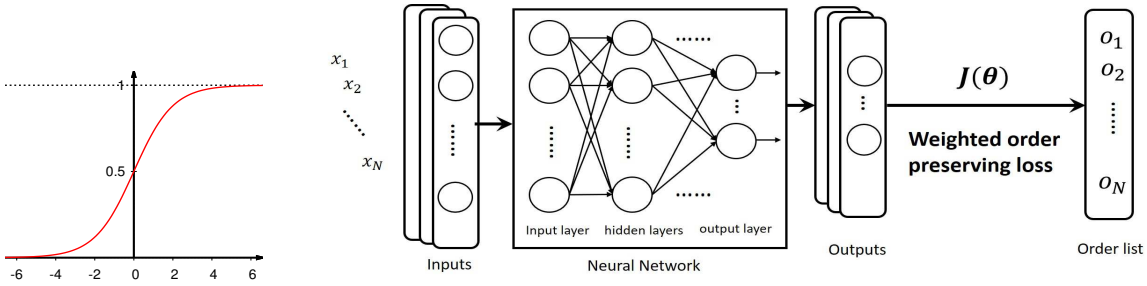


Fig. 2: The Sigmoid Function (Left) and The Architecture of Our Non-Linear Hash Learning (Right)

from the reference dataset, to compute the rank position and subsequently all the losses. So $\bar{\mathbf{D}}$, instead of $\mathbf{D}$, is used to invoke Algorithm 3. Therefore, with the above measures, we can reduce the gradient computation cost to $\mathcal{O}(B \cdot |\mathcal{S}| \cdot \gamma N)$.

V. Learning to Index by Neural Network

In this section, we consider learning non-linear mapping functions via Deep Neural Networks (DNN).

A. DNN Architecture

We choose to model non-linear mapping functions by a DNN, due to its strong modelling power and its success in numerous application areas. However, it is more natural to train the M hash functions collectively, which essentially maps the d-dimensional point $\mathbf{x}$ into a M-dimensional embedding $\mathbf{x}^*$, or $\mathbf{x}^* = \mathcal{H}(\mathbf{x}; \boldsymbol{\theta})$, where $\mathcal{H}$ can deemed as a DNN and $\boldsymbol{\theta}$ represents all the parameters of the DNN.

We design the architecture of our DNN as shown in Figure 2(right). It consists of five fully-connected layers, denoted as: I1-H2-H3-H4-O5. H1 is the input layer which receives the input dataset features, and the number of input units is equal to the dimensionality of the dataset. H2-H4 are three hidden layers, each of which contains 512 units. ReLU is used as the activation function for each hidden layers. O5 is the output layer containing M units.

B. Objective Function

Our DNN-based model requires us to design a new loss function rather than those presented in the previous section. For example, the orthogonality constraints in our previous loss function is inapplicable for DNN.

Nevertheless, we can still apply the same order-preserving idea. We use the following smoothly differentiable surrogate rank position function:

$$\tilde{r}(\mathbf{x}_i; \boldsymbol{\theta}) = \sum_{j=1}^{N} \sigma(\|\mathcal{H}(\mathbf{q}; \boldsymbol{\theta}), \mathcal{H}(\mathbf{x}_i; \boldsymbol{\theta})\| - \|\mathcal{H}(\mathbf{q}; \boldsymbol{\theta}), \mathcal{H}(\mathbf{x}_j; \boldsymbol{\theta})\|) + 1 \tag{11}$$

This function is relaxed from the $\mathbf{1}_p$ function. We penalize how far is the ranking of $\mathbf{x}_i$ away from its groundtruth (in the original space) instead of whether $\mathbf{x}_i$ is preserved in corresponding bucket.

Then, the loss function can be formulated as:

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} \beta_i \log\big((\tilde{r}(\mathbf{x}_i; \boldsymbol{\theta}) - g(\mathbf{x}_i))^2 + 1\big) \tag{12}$$

where the $g(\mathbf{x}_i)$ is the ranking of x_i with respect to query q in the original space R^d, β_i is the weight computed for x_i, defined as

$$\beta_i = \exp\left(-\frac{\|\mathbf{q}, \mathbf{x}_i\|}{\max_{1 \le j \le N} \|\mathbf{q}, \mathbf{x}_j\|}\right) \tag{13}$$

$\log(1+z)$ is used to encourage the model to pay more attention to near-by points rather than faraway points.

Finally, we use the Adam optimizer [22] to train the network in a mini-batch manner.

Note that the sub-sampling strategy introduced in the linear model is also applied into our non-linear model to reduce the training cost. Specifically, a training dataset $\bar{\mathbf{D}}$ of size γN is used to calculate the rank function $\tilde{r}(\mathbf{x}_i; \boldsymbol{\theta})$ (11), and the gradient of $J(\boldsymbol{\theta})$ (12) is computed over a random subset $\mathcal{S}$ of $\bar{\mathbf{D}}$. Therefore, the training cost of our DNN-based model over a mini-batch of size B is $\mathcal{O}(\eta \cdot B \cdot |\mathcal{S}| \cdot \gamma N)$, where the η is due to the computation cost of the neural network.

Remark 1. *We remark that the linear model is easier to be optimized, thus, is faster in training than DNN-based model. However, DNN-based model can learn complex non-linear mapping functions, which makes it have a better performance than linear model on many datasets (see Section VII-C).*

VI. Discussion

In this section, we discuss some issues related to our ANNS framework and the learning to hash models.

For our query processing in Algorithm 2, there are two points need to be discussed. Firstly, one could change the candidate condition such that points that have been seen on more than $\lceil \alpha \cdot M \rceil$ lists are added to the candidate set, reminiscent of the strategy used in MEDRANK [10], C2LSH [13] and QALSH [19]. However, we experimentally found that $\alpha = 1$ always achieves the best performance (see Fig. 4(b)). Secondly, we access each list based on the closeness of the pages to the embedded query instead of other choices such as keeping distance lower or upper bounds as in the threshold algorithm [11]. This is because (i) it is not always possible to keep track of the lower/upper bounds (e.g., when $\mathcal{H}$ is a non-linear mapping function), and (ii) due to the curse of dimensionality, the lower/upper bounds on low dimensional embedding spaces are very loose and do not help much in early stopping the query processing in practice.

Another thing is that can our learning to hash models be applied to other distance metrics and space? In this paper, we focus on the ANNS in metric space $\mathcal{R}^d$, taking the Euclidean distance as distance metric. The key point of our methods is

to map original data points into sorted lists by learned functions, followed by the sequential ANN search. The purpose of our learned functions is to learn the similarity ordering information from the original space. From this perspective, our framework is independent to the actual distance metric. In other words, our models can be easily applied to other distance metrics, such as cosine distance and inner product, by providing the corresponding original distance order (i.e., ground-truth). Nevertheless, it is unclear whether this will lead to good performance for other distance metrics and spaces. We will leave this as a future work.

VII. Evaluation

In this section, we conduct comprehensive experiments to verify the efficiency and effectiveness of our proposed methods, compared with the state-of-the-art I/O efficient ANNS algorithms.

A. Experimental Settings

In this subsection, we introduce the settings of our experiments.

Algorithms We compare the two proposed algorithms with four external memory-based ANNS algorithms. Below are algorithms evaluated in the experiments.

- **I-LSH**. The random hash based incremental LSH algorithm proposed in [24] (Section II-C).
- **PQBF**. The ANNS algorithm proposed in [25] where the product quantization technique is employed (Section II-C).
- **AOSKNN**. The PCA based ANNS algorithm proposed in [17] where the R-tree is employed (Section II-C).
- **M-Tree**. M-tree is a general tree-based data structure that can support ANNS in Euclidean metric space [29] (Section II-B).
- **OPFA** and **NeOPFA**. Our proposed ANNS algorithms where we learn *linear* and *non-linear* mapping functions (Sections IV and V, respectively) using the relaxed, block-based loss functions.

Datasets and query load Six widely-used large scale high-dimensional datasets are used for experiments: **Gist**[5], **Deep**[6], **UQvideo**[7], **Tiny**[8], **Deep1B**[9], **Sift1B**[10]. *Gist* is is an image dataset which contains about 1 million data points with 960 features. *Deep* contains deep neural codes of natural images, which contains about 1 million data points with 256 dimensions. *UQvideo* is a video dataset with each objects being 256 dimensions. *Tiny* is also a image dataset which consists of around 80 million images, each being a 384 feature vector. *Deep1B* contains 1 billion of 96-dimensional DEEP descriptors [1]. *Sift1M* consists of 1 billion 128-dimensional SIFT feature vectors. For each dataset, after the deduplication, we randomly select $1,000$ data points and reserve them as the query points. The details of each dataset are summarized in Table II.

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

[6]https://yadi.sk/d/I_yaFVqchJmoc

[7]http://staff.itee.uq.edu.au/shenht/UQ_VIDEO/

[8]http://horatio.cs.nyu.edu/mit/tiny/data

[9]http://sites.skoltech.ru/compvision/noimi/

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

Training and Implementation For each dataset, we randomly sample two different subsets as training dataset and training query set (i.e., $\bar{\mathbf{D}}$ and $\mathbf{Q}$ in Algorithm 3, respectively). Specifically, for *Gist*, *Deep* and *UQvideo* datasets, we sample 20k data points as training dataset and 10k data points as training query set for each of them, respectively. For *Tiny*, we sample 100k data points and 20k data points as training dataset and training query set. For *Deep1B* and *Sift1B*, we sample 1M data points and 0.2M data pints as training dataset and training query set for each of them, respectively. After sampling, the remaining part of each dataset is regarded test datset for ANNS. The batch size B is set to be 200 and 100 for linear and non-linear learning methods, respectively, for all datasets. The termination of two learning methods depends on the convergence of training procedure, where the $\mathrm{max\_iteration}$ of the linear method is set within $[50, 400]$ for the datasets evaluated. When choosing the sub-sample $\mathcal{S}$ (See the end of Section IV-B), we chose the following strategy: for a given training query q and a training dataset, denote the k-NN points of q as $\mathcal{S}_+$, and the rest of the points as $\mathcal{S}_-$. The final $\mathcal{S}$ consists of the entire $\mathcal{S}_+$ and a random sample in $\mathcal{S}_-$.

Note that, same as [25], we apply the K-means data partition for all datasets in the experiment for better search efficiency. Specifically, the K-means clustering method is leveraged to partition the datasets, and then the learned functions are used to index each partition separately. After that, the partition (i.e., a subset of data points) closest to the query point based on the Euclidean distance is selected for querying processing. The partition number is set to be 10 for *Gist*, *Deep* and *UQvideo*, and 64 for the other three datasets.

Evaluation Metrics We use 6 important metrics to evaluate the performance of the algorithms: ratio, recall, I/O costs, running time, pre-processing time and index size.

- **Ratio**. This is the ratio between distances of the approximate kNN results to the query and those of the actual kNN results, to measure the quality of ANN results, which is widely used in the literatures [13], [33], [19], [25]. Given a query $\mathbf{q}$, let $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k\}$ be the approximate kNN to $\mathbf{q}$ returned by an ANN method, and $\{\mathbf{o}_1, \mathbf{o}_2, \ldots, \mathbf{o}_k\}$ be the true kNN. Then the ratio is defined as:
$$ratio = \frac{1}{k}\sum_{i=1}^{k}\frac{\|\mathbf{q}, \mathbf{p}_i\|}{\|\mathbf{q}, \mathbf{o}_i\|}$$
Clearly, ratio close to 1.0 means the ANNS algorithm returns better results and vice versa. The average ratio of a set of queries is reported in the experiments.
- **Recall**. Recall is the ratio between the number of true kNNs found in the approximate kNN set and the value of k. It measures how many true kNNs can be found by ANNS methods.
- **I/O cost**. The page size b is set to be 4096 for all algorithms in the experiments. We assume the index and dataset reside in the external memory before the query is issued (i.e., cold startup). A unit I/O cost is a random I/O and we set the cost of a sequential I/O as 0.01 for the index accessing according to the profiling of our hardware system running the experiments. During the distance verification, we firstly sort the point IDs and then sequentially access the data points in the external memory. Thus, the cost of a sequential I/O for the verification is

TABLE II: Statistics of Datasets and Index Sizes of All Algorithms (in Megabytes)

Datasets		Statistics			Index Sizes (MB)					
		N	d	Data Type	NeOPFA	OPFA	PQBF	AOSKNN	I-LSH	M-Tree
Million Scale	Gist	982,677	960	Image	102.5	98.4	84.6	144.2	849.7	21.6
	Deep	1,000,000	256	Image	102.8	100.1	70.4	148.7	864.6	20.7
	UQvideo	3,038,478	256	Video	306.9	304.2	210.6	443.9	2662.4	63.6
	Tiny	79,302,017	384	Image	8092.5	8089.6	5836.8	-	39014.4	-
Billion Scale	Deep1B	1,000,000,000	96	Image	102,402.4	102,400	75,673.6	-	491,929.6	-
	Sift1B	1,000,000,000	128	Image	102,402.4	102,400	75,673.6	-	491,929.6	-

TABLE III: Parameter Settings of OPFA

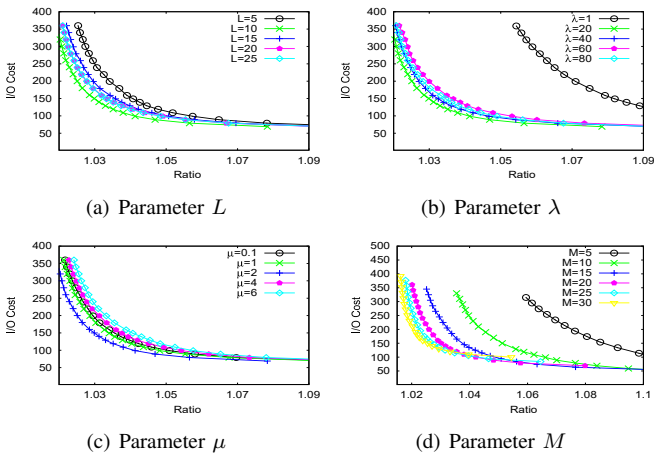
Parameters	Values
The number of sorted lists (M)	$5, 10, 15, 20, \mathbf{25}, 30$
The number of buckets (L)	$5, \mathbf{10}, 15, 20, 25$
Orthogonality regularization factor (λ)	$1, \mathbf{20}, 40, 60, 80$
Additional regularization factor (μ)	$0.1, 1, \mathbf{2}, 4, 6$

set to be 0.1 according to the profiling of our hardware system. The average I/O cost of a set of queries is used in the experiments.

- **Search time**. It is the wall clock time for running a query. We report the average search time among the set of test queries.
- **Preprocessing time**. We report the preprocessing time of the algorithms, including the training time (i.e., the learning of hashing functions) and index construction time (i.e., generate embeddings for every point and build index).
- **Index size**. We also report the size of the indexes used by the algorithms.

Parameter Setting By default, the k value of k-ANNS is set to be 20, which may vary from 10 to 100 in the experiments. The parameters of the algorithms are set to default values as suggested by the original authors unless otherwise specified. Particularly, for PQBF [25], the number of PQB-trees K' is set to be 64, the number of PQB-trees θ selected to perform ANNS is set to be 4 and the ratio ϵ is set to be 0.4. In AOSKNN algorithm [25], the dimensionality of PCA (m) is set to 6. The precision ε and the relaxation factor λ is set to 0.9 and 2, respectively. In I-LSH algorithm [24], the approximate ratio c is set to be 2 for *Tiny*, *Deep1b* and *Sift1B* datasets and 1.7 for other datasets for a good overall performance. The success possibility δ is set to be $1/2 - 1/e$. The setting of the candidate size T in our algorithm depends on the value k and the corresponding dataset, which can be tuned by users for satisfactory performance. Table III shows the possible values of other four parameters in OPFA and their default values (in bold fonts). The impact of these parameters will be evaluated in Section VII-B, where we also evaluate the impact of the number of lists M for NeOPFA.

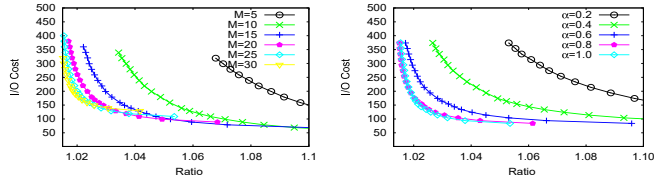
Our two ANNS algorithms are implemented in standard C++ and the source codes of PQBF [25], AOSKNN [17], I-LSH [24] are provided by the original authors. The source code of M-tree [29] is from GitHub (https://github.com/erdavila/M-Tree). Note that we can only successfully setup the main-memory version of M-tree, and we use it in the performance evaluation. We count the number of nodes accessed during the querying as the I/O costs, where we set the node size of M-tree to be the page size (i.e., 4096 bytes). The algorithms are compiled with G++ with -O3 in Linux. All experiments are performed on a machine with Intel Xeon Gold 2.7GHz CPU and Redhat Linux System, with 180G main memory.



(a) Parameter L (b) Parameter λ

(c) Parameter μ (d) Parameter M

Fig. 3: The impact of parameters of OPFA on Deep



(a) Parameter M (b) Parameter α

Fig. 4: The impact of parameters of NeOPFA on Deep

B. Parameter Tuning

In this subsection, we investigate the impact of the parameters for our proposed methods, and decide the default settings by tuning parameters on *Deep* datasets. Note that we do not report the parameter tuning details of neural network in NeOPFA algorithm since it is beyond the focus of this paper.

We first investigate the four parameters of OPFA. The first three parameters are from the linear hash function, i.e., the number of buckets L, the regularization factor λ and factor μ. And the last one, i.e., the number of sorted lists M, is from our ANNS framework. For the NeOPFA, we only need to tune the number of lists M in the experiment given the setting of neural network. We tune the candidate size T to plot the curve of I/O cost $w.r.t$ ratio for each parameter. When tuning one parameter, the other parameters are set to be the default values in Table III.

The impact of L on OPFA is plotted in Fig. 3(a) where the trade-offs between I/O cost and ratio are reported with L varying from 5 to 25. Smaller L leads to a larger mismatch,

but less training time. In addition, larger L makes the loss function harder to be optimized. We observe that OPFA achieves a trade-off for $L = 10$, which is used in the following experiments. Recall that λ is orthogonality regularization and μ is the additional regularization that encourages the learned projection vectors to be unit vectors. The impacts of λ and μ are plotted in Fig. 3(b)-(c). As expected, the larger λ leads to a better ratio but incurs higher I/O cost, since the algorithm needs to access more pages to find out the first candidate that has been seen on all M sorted lists, and vice versa. Compared with λ, μ is smaller such that the optimization focuses more on the loss function and the orthogonality regularization. We observe that OPFA achieves a good overall performance when $\lambda = 20$ and $\mu = 2$.

For the impact of M for OPFA and NeOPFA, the results are plotted in Fig. 3(d) and Fig. 4(a). M has a significant influence on the trade-off between I/O costs and accuracy where a larger M leads to better accuracy, but higher I/O cost. It is reported that both OPFA and NeOPFA reach a good trade-off when $M = 25$, and lager M cannot distinctly achieve a better performance.

We also investigate the situation discussed in Section VI. That is, an object may become a candidate after $\lceil \alpha \cdot M \rceil$ hits, i.e., the object appears $\lceil \alpha \cdot M \rceil$ times during the search with $0 < \alpha \leq 1$. In Fig. 4(b), we report the trade-offs between I/O costs and ratio for different α values given $M = 25$ on *Deep* dataset for NeOPFA. It is shown that NeOPFA achieves the best overall performance when $\alpha = 1$. This confirms the effectiveness of our search strategy in Algorithm 2.

C. Performance Comparison with the State-of-the-art Algorithms

In this section, we present comprehensive experimental results for the six algorithms on all datasets, in terms of the six evaluation metrics. To evaluate the performance of the algorithms on k-ANNS, k is set to be $\{10, 20, \ldots, 100\}$. **Note** that the experimental results of AOSKNN and M-tree on *Tiny*, *Deep1B* and *Sift1B* datasets are not available because they failed to build up the indices under the current system settings.

I/O Cost Fig. 5(a)-(d) and Fig. 6(a)-(b) report the I/O costs of six algorithms on six datasets where k varies from 10 to 100. It is shown that OPFA and NeOPFA outperform the other four ANNS techniques by a large margin. The IO cost of NeOPFA is around 68%-89.3% of OPFA on most of the datasets. Though they share the same ANNS framework, the non-linear hash functions learnt by neural networks are more powerful than the linear hash functions, with higher training cost (see Fig. 9). Due to the use of random I/O, the overall performance of PQBF, AOSKNN and M-tree are not competitive. It is interesting that PQBF outperforms AOSKNN and M-tree, which is because of the good performance of PQ method. M-tree outperforms AOSKNN on most of datasets. Although I-LSH can also take advantage of the sequential I/O, the poor quality of the random hash leads to a larger number of lists to be accessed for a decent search accuracy. Thus, the I/O cost of I-LSH is larger than our proposed algorithms, especially on the three very large datasets.

Ratio We evaluate the average ratio of k-ANN queries for all algorithms on six datasets by varying k. The experimental results are plotted in Fig. 5(e)-(h) and Fig. 6(c)-(d). In addition to the superior performance in terms of I/O costs, We still observe that NeOPFA and OPFA also outperform the other four algorithms on the accuracy of the search results, thanks to the high quality data-sensitive hashing functions and the sequential I/O accesses. The large gap can be observed on *Deep*, *Tiny* and *Deep1B* datasets. As expected, NeOPFA performs better than OPFA, especially on *Gist* and *Tiny*. This is because the new loss function used in NeOPFA can better preserve order information in the high dimensional space and the neural network can learn sophisticated non-linear hash functions. Among six algorithms, I-LSH has the worst performance on most of datasets because of the use of data-independent random hash functions. Same as the I/O cost, PQBF demonstrates better performance compared to AOSKNN and M-tree.

Recall The experimental results of the recall with respect to k for the six algorithms on four datasets are plotted in Fig. 7. Consistent with the observations in *Ratio*, NeOPFA and OPFA have the highest recall when compared with the other ANNS methods. For example, given $k = 100$, the recalls of NeOPFA and OPFA are 0.51 and 0.48, while the recalls of PQBF, I-LSH, AOSKNN and M-tree are 0.40, 0.31, 0.33 and 0.17, respectively, on *Deep*. NeOPFA has a higher recall than OPFA. PQBF performs better than AOSKNN, I-LSH and M-tree on most of datasets.

Search Time Although the focus of this paper is the external-memory algorithms, we also evaluate the running time of the algorithms. Due to the space limitation, we just report the results on two datasets, which can be seen in Fig. 8. It is shown that both NeOPFA and OPFA outperform the state-of-the-art algorithms due to their I/O efficiency. For other four algorithms, PQBF is faster than I-LSH, AOSKNN and M-tree.

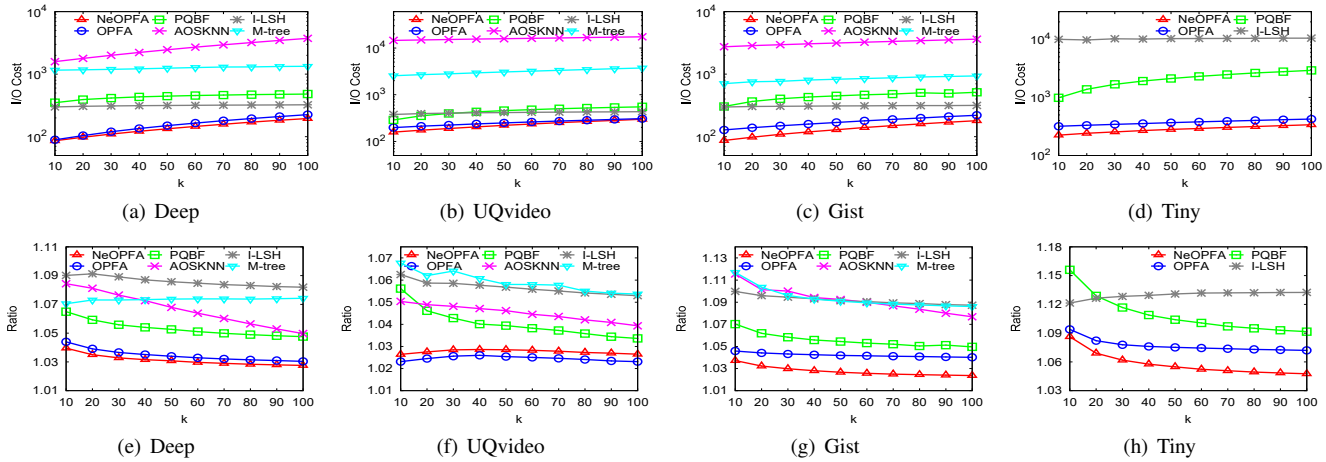
Index Size The index sizes of six methods on six datasets are reported in Table II. We observe that the index size of M-tree is the smallest on the three million-scale datasets, since it just needs to store the object IDs and some distance information for pruning. The index size of PQBF is the second smallest, followed by our methods. The index size of OPFA is 67.3%–68.5% of AOSKNN, and 11.4%–20.8% of I-LSH. The index size of NeOPFA is slightly larger than OPFA because of the parameters of neural network kept.

Preprocessing Time We report the preprocessing time in Fig. 9. Preprocessing time considers learning the mapping function, generating the embeddings, and the index construction for our algorithms. I-LSH does not need to learn the hash functions and the generation of random hash functions is very efficient. Thus, it has the best performance in terms of preprocessing time. PQBF ranks the second due to the efficiency of PQ code generation. As expected, NeOPFA spends more time on pre-processing than OPFA as the the neural network learning takes substantial amount of time.

D. Summary

Based on the experimental results, we have the following observations:

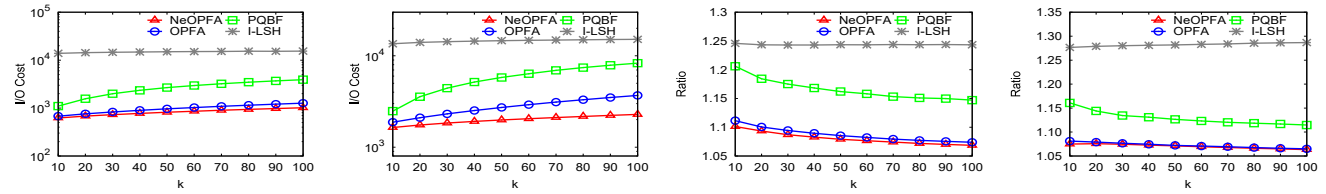
- Among six algorithms, I-LSH is the only algorithm with theoretical guarantee. It also enjoys the efficient sequential I/O accesses and the worst case performance theoretical guarantee. However, the overall performance of I-LSH is far from satisfactory on six real-life datasets because of the use of random hash functions. M-tree is a general



(a) Deep (b) UQvideo (c) Gist (d) Tiny

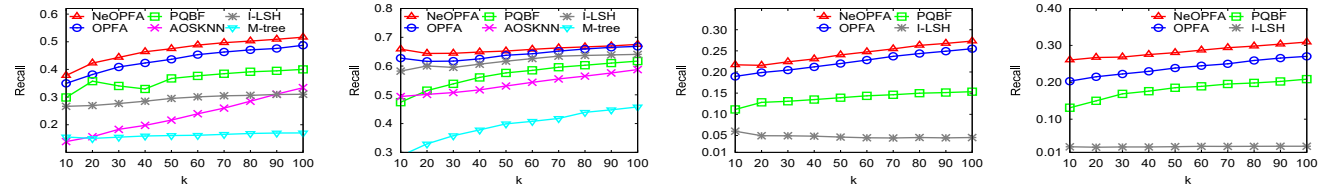
(e) Deep (f) UQvideo (g) Gist (h) Tiny

Fig. 5: I/O Cost and Ratio with respect to k on million-scale datasets



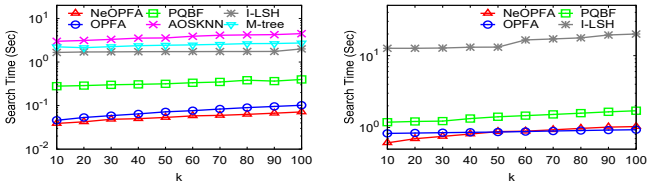
(a) I/O cost, Deep1B (b) I/O cost, Sift1B (c) Ratio, Deep1B (d) Ratio, Sift1B

Fig. 6: I/O cost and Ratio with respect to k on billion-scale datasets



(a) Deep (b) UQvideo (c) Deep1B (d) Sift1B

Fig. 7: Recall with respect to k



(a) Gist (b) Tiny

Fig. 8: Search Time with respect to k

indexing technique for metric space which can support ANN search. But, as demonstrated in experiments, its performance is not competitive due to the random I/Os on the index.

- By utilizing the existing learning to hash approaches (i.e., PQ and PCA), the hash values of PQBF and AOSKNN are data-sensitive. However, the I/O efficiency is not considered in the learning of their hash functions. Moreover, random I/Os are invoked in the search of the index. As reported in the experiments, PQBF and AOSKNN are outperformed by our proposed methods in terms of I/O cost, accuracy and search time.
- Our proposed leaning to hash techniques directly optimize the hash functions against the index (i.e., the lists). Thus the I/O efficiency is considered by the objective functions

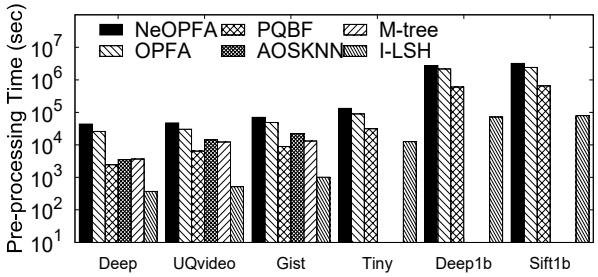


Fig. 9: Pre-processing Time on All Datasets

of the machine learning tasks. Moreover, sequential I/Os are invoked during the search. These make two proposed algorithms OPFA and NeOPFA achieve a good trade-off between I/O cost (search time) and accuracy. As expected, the non-linear hash functions learnt from sophisticated neural network can enhance the performance, at the cost of more training time.

VIII. Conclusion

In this paper, we develop two I/O efficient indexing and query processing methods to achieve highly efficient I/O performance for approximate nearest neighbor search (ANNS)

in high dimensional space. Our methods are based on a framework that uses sequential I/Os for finding candidates for a query based on indexes learned from the data. We consider both linear and non-linear functions to construct the learned index with novel objective functions. Comprehensive experiments on six high-dimensional benchmarking datasets with objects up to 1 billion, show that our proposed methods outperform the state-of-the-art I/O-focused ANNS techniques in terms of I/O efficiency, search accuracy.

ACKNOWLEDGMENT

Ying Zhang is supported by ARC FT170100128 and DP180103096. Yifang Sun and Wei Wang are partially supported by ARC DPs 170103710 and 180103411, and D2DCRC DC25002 and DC25003. Ivor W. Tsang is supported by ARC grant LP150100671 and DP180100106. Xuemin Lin is supported by NSFC 61672235, DP170101628 and DP180103096.

REFERENCES

[1] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *CVPR*, pages 2055–2063, 2016.

[2] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh. Beyond one billion time series: indexing and mining very large time series collections with isax2+. *Knowledge and information systems*, 39(1):123–151, 2014.

[3] Z. Cao, M. Long, J. Wang, and P. S. Yu. Hashnet: Deep learning to hash by continuation. In *ICCV*, pages 5608–5617, 2017.

[4] B. Dai, R. Guo, S. Kumar, N. He, and L. Song. Stochastic generative hashing. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 913–922. JMLR. org, 2017.

[5] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, pages 271–280. ACM, 2007.

[6] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *STOC*, volume 8, pages 537–546. Citeseer, 2008.

[7] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[8] W. Dong, M. Charikar, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, 2011.

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

[10] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD*, pages 301–312, 2003.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

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

[13] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552. ACM, 2012.

[14] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *CVPR*, pages 2946–2953, 2013.

[15] 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.

[16] R. M. Gray. Vector quantization. *Readings in speech recognition*, 1(2):75–100, 1990.

[17] Y. Gu, Y. Guo, Y. Song, X. Zhou, and G. Yu. Approximate order-sensitive k-nn queries over correlated high-dimensional data. *TKDE*, 30(11):2037–2050, 2018.

[18] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of computing*, 8(1):321–350, 2012.

[19] 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.

[20] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[21] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33(1):117–128, 2011.

[22] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] 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*, pages 1–1, 2019.

[24] W. Liu, H. Wang, Y. Zhang, W. Wang, and L. Qin. I-LSH: I/O efficient c-approximate nearest neighbor search in high-dimensional space. In *ICDE*, pages 1670–1673, 2019.

[25] Y. Liu, H. Cheng, and J. Cui. Pqbf: I/o-efficient approximate nearest neighbor search by product quantization. In *Proceedings of CIKM*, pages 667–676. ACM, 2017.

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

[27] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *TPAMI.*, 36(11):2227–2240, 2014.

[28] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[29] C. Paolo, P. Marco, and Z. Pavel. M-tree: An efficient access method for similarity search in metric spaces. *PVLDB*, pages 426–435, 1997.

[30] F. Shen, Y. Xu, L. Liu, Y. Yang, Z. Huang, and H. T. Shen. Unsupervised deep hashing with similarity-adaptive and discrete optimization. *TPAMI*, 40(12):3034–3044, 2018.

[31] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, pages 1–8. IEEE, 2008.

[32] D. Song, W. Liu, R. Ji, D. A. Meyer, and J. R. Smith. Top rank supervised binary coding for visual search. In *ICCV*, pages 1922–1930, 2015.

[33] 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.

[34] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576. ACM, 2009.

[35] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *TODS*, 35(3):20, 2010.

[36] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD*, pages 839–848. ACM, 2018.

[37] J. Wang, J. Wang, N. Yu, and S. Li. Order preserving hashing for approximate nearest neighbor search. In *Proceedings of the 21st ACM international conference on Multimedia*, pages 133–142. ACM, 2013.

[38] J. Wang, T. Zhang, J. Song, N. Sebe, and H. T. Shen. A survey on learning to hash. *TPAMI.*, 40(4):769–790, 2018.

[39] Q. Wang, Z. Zhang, and L. Si. Ranking preserving hashing for fast similarity search. In *IJCAI*, 2015.

[40] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NeurIPS*, pages 1753–1760, 2009.

[41] J. Zhang, J. Tang, C. Ma, H. Tong, Y. Jing, J. Li, W. Luyten, and M.-F. Moens. Fast and flexible top-k similarity search on large networks. *TOIS*, 36(2):13, 2017.

[42] Y. Zhang, X. Lin, G. Zhu, W. Zhang, and Q. Lin. Efficient rank based knn query processing over uncertain data. In *ICDE*, pages 28–39. IEEE, 2010.