The Case for ML-Enhanced High-Dimensional Indexes

(Regular Papers)

Rong Kang
Tsinghua University
Beijing, China
kangrong.cn@gmail.com

Wentao Wu
Microsoft Research
Redmond, USA
wentao.wu@microsoft.com

Chen Wang
Tsinghua University
Beijing, China
wang_chen@tsinghua.edu.cn

Ce Zhang
ETH Zurich
Zurich, Switzerland
ce.zhang@inf.ethz.ch

Jianmin Wang
Tsinghua University
Beijing, China
jimwang@tsinghua.edu.cn

ABSTRACT

The case of learned indexes has recently inspired intensive research in the database community. It has been shown that learned indexes can often outperform traditional indexes such as B-tree on low-dimensional data. However, learned indexes suffer from inherent difficulties due to the "curse of dimensionality" when applied to datasets with very high dimensions (e.g., time series data). In this paper, we explore the "middle ground" between traditional and learned indexes where we can leverage advantages of both, targeting high-dimensional data. Specifically, we study *ML-enhanced indexes* in the context of processing k-nearest-neighbor (kNN) queries over time series data. Our ML-enhanced indexes take traditional tree-based indexes as inputs, and train deep neural networks that capture the distribution of nearest neighbors among index leaf nodes. We then process kNN queries by scanning the leaf nodes with respect to the order suggested by the neural networks (rather than their original order). Experimental evaluation shows that ML-enhanced indexes can significantly outperform their traditional counterparts in terms of the recall of the nearest neighbors with respect to the number of index leaf nodes being scanned.

1 INTRODUCTION

Indexing techniques are fundamental for accelerating query processing in similarity search. Typical applications include k-nearest-neighbor (kNN) queries and range queries. Various index structures have been proposed, such as R-tree [18] and its variants [4], M-tree [8], and DS-Tree [40]. Recently, there has been intensive work on the case of *learned indexes* [28, 33], and it has been shown that learned indexes can often outperform traditional indexes on low-dimensional data given their adaptive capability to workload-specific access patterns. However, in practice, similarity search is often conducted over objects with very high dimensions, such as image, text, and time series. Learned indexes can suffer from the "curse of dimensionality" when applied to such types of data.

In this paper, rather than pursuing pure traditional or learned indexes for high-dimensional data, we instead explore their "middle ground" by studying *ML-enhanced indexes*. Our basic idea is to stay with traditional, tree-based indexes; however, when scanning the leaf nodes to fetch relevant data objects from the disk, rather than following the original order of the leaf nodes suggested by the indexes we instead use a deep neural network (DNN) to reorder the leaf nodes. Our experimental evaluation results show that our ML-enhanced indexes can significantly outperform traditional indexes in terms of the *recall* of kNN queries with respect to the number of index leaf nodes being scanned.

In more detail, we study ML-enhanced indexes in the context of processing kNN queries over time series data, which typically contain hundreds of dimensions. Query answering using tree-based indexes in general involves two steps:

- The *pruning* step (a.k.a. the filtering step), where we traverse the index in a top-down manner to prune irrelevant objects (i.e., objects that cannot be among the k nearest neighbors) and land on a set of candidate leaf nodes in which relevant objects must be included;
- The *post-processing* step (a.k.a. the refinement step [11]), where the objects in the candidate set are compared to the given query object to find the k nearest neighbors.

To ensure that no relevant objects are eliminated by the pruning step, existing tree-based indexes typically employ distance-based bounding techniques. Specifically, one can compute the *lower-bound* of the distances between data objects covered by an index node and the given query object; if the lower-bound is larger than the distance of the current k^{th} nearest neighbor, then that index node can be safely skipped. In the post-processing step, the candidate leaf nodes are further ordered by their distance lower-bounds [27].

To measure the effectiveness of tree-based indexes when processing kNN queries, one common practice is to measure the *recall* (i.e., the number of nearest neighbors that have been found) as we scan the disk to fetch data objects pointed to by the candidate leaf index nodes [22]. Clearly, the goal is to cover as many nearest neighbors within a given budget on the number of disk accesses. If the lower-bounds were tight enough, one would have expected a few disk accesses to reach a good recall. For high-dimensional data, unfortunately, this is rarely the case in practice — the lower-bounds are usually quite loose for the sake of "no false dismissals" (i.e., one cannot miss a true nearest neighbor). Indeed, for the tree-based index structure to achieve this, the region in the high-dimensional space represented by a parent index node must cover all regions represented by its child nodes [4, 27, 36, 40]. Given the high-dimensional nature of the data, the lower-bound of an index node is thus inevitably loose as long as there are "outliers" in its descendants along *any* single dimension. As a result, to reach a certain degree

of recall, the number of disk accesses by following leaf index nodes *ordered by their lower-bounds* is often considerable on real-world high-dimensional datasets.

Our idea in this paper is to use machine learning (ML) technologies to improve convergence on recall, i.e., to reduce the number of disk accesses for reaching a given degree of recall, by *reordering* the leaf index nodes. Specifically, we train a deep neural network (DNN) to *predict* the likelihoods of the leaf index nodes where the nearest neighbors of the given query object may locate; we then visit the leaf index nodes with respect to the decreasing order of their likelihoods. The intuition is that, *once an index is built*, the *distribution* of the nearest neighbors given a query object has been *fixed* and therefore is *learnable*. We have conducted extensive experiments to evaluate the effectiveness of our *ML-enhanced indexes*, and results show that they can significantly improve on recall compared to their corresponding counterparts without ML-based reordering.

The *reordering* idea is, by no means, tied to ML-based approaches. Indeed, one can use other mechanisms for reordering. In this paper, we have further examined such an alternative approach based on *quantization* [22], which typically tries to convert raw high-dimensional data into compact formats using encoding schemes. The compacted data is presumed much smaller and therefore can be resident in main memory. When searching for the nearest neighbors of a given query object, the compacted representation is used to reconstruct an approximate version of the original raw, high-dimensional data object, after which an estimated distance can be computed as usual. We can design a reordering mechanism of leaf index nodes by following the quantization-based, approximate distances. We have further compared our DNN-based reordering mechanism with this quantization-based reordering mechanism, and we find that they are comparable in terms of improving recall on nearest neighbors. However, the construction of the compacted data representations using quantization is both more computation- and resource-intensive.

To summarize, this paper makes the following contributions:

- We conduct a systematic performance study over existing high-dimensional index structures and demonstrate their inefficiency when processing kNN queries.
- We propose ML-enhanced indexes that take advantage of both traditional and learned indexes by reordering leaf index nodes using deep neural networks, with the goal of improving the recall of kNN query processing.
- As an alternative method and a reference point/baseline for comparison, we propose a quantization-based reordering mechanism. We further compare it against the above DNN-based reordering mechanism.
- We evaluate our ML-enhanced indexes, including the DNN-based and quantization-based reordering mechanisms, via extensive experiments. Our experimental results demonstrate the effectiveness of ML-enhanced indexes using the DNN-based reordering mechanism.

The rest of the paper is organized as follows. We summarize related work in Section 2. We then present a systematic study of existing tree-based indexes for high-dimensional data, in terms of their effectiveness when processing kNN queries (Section 3). Following this performance study and analysis, we propose ML-enhanced indexes in Section 4 and report evaluation results in Section 5. We conclude and discuss future work in Section 6.

2 RELATED WORK

The related work on similarity search has been overwhelming in literature. We focus our discussion on indexing techniques that are designed for answering kNN queries. In general, they fall into two categories, depending on whether they support exact or approximate kNN queries. Note that, however, many of the indexes that were proposed for answering exact kNN queries can also be used to support approximate kNN queries [6, 7, 30, 36, 40], which is the focus of our work in this paper.

2.1 Indexes for Exact kNN Queries

Indexes for exact kNN queries need to guarantee that the pruning step should not eliminate any data objects that are true nearest neighbors. To achieve this, such indexes usually use distance (e.g., Euclidean distance) lower-bounds as pruning conditions: If the lower-bound of an index node is larger than the current largest distance between the query object and its nearest neighbors, this index node can be safely ignored.

R-tree [18] is perhaps one of the most classic indexes that follow this pruning strategy based on distance lower-bounds. It divides the multi-dimensional search space into overlapping *hyperrectangles* that are organized as a tree. A leaf node contains pointers to the raw data, and an inner node represents the *minimum bounding rectangle* of its child nodes. There have been a number of variants proposed since the initial introduction of R-tree, such as R* tree [4] that improves space partitioning and balance of R-tree.

Other prominent tree indexes for time series data include M-tree [8] and DS-Tree [40]. M-tree divides data objects based on their relative distances. The lower-bound pruning strategy works correctly in M-tree as long as the distance metric (e.g., Euclidean distance) satisfies the *triangular inequality*. On the other hand, DS-Tree builds the index with respect to the "extended adaptive piecewise constant approximation" (EAPCA) [26], which splits an index node base on the mean or variance of data objects covered.

In the past decade, a series of tree indexes have been proposed for time series data that are based on the so-called "symbolic aggregate approximation" (SAX) [29]. As the first attempt in this line of work, iSAX [36] constructs a multi-level tree based on SAX and the so-called "piecewise aggregate approximation" (PAA) [25]. Follow-up work improves it on various aspects such as splitting policy [6], bulk-loading [7] support, adaptive construction [44], and utilization of new hardware [34].

2.2 Indexes for Approximate kNN queries

Given the overhead of processing exact kNN queries, approximate kNNs are acceptable in lots of applications. As a result, many indexes have been proposed for supporting approximate kNN queries, including hash-based methods [17, 21, 37], graph-based methods (e.g., HSGW [31] and NSG [14]), and tree-based methods [1, 32].

One key idea leveraged by indexes for approximate kNN queries is *quantization*, which aims to convert high-dimensional data into a more compact representation. For instance, PQ [22] partitions the original high-dimensional space into subspaces with lower dimensions, and then learns a quantization encoding in each subspace

independently. The final encoding/representation of the data is formed by *concatenating* the encodings of the subspaces. OPQ [16, 39] improves PQ by rotating the raw data properly to reduce the quantization error. LOPQ [23], on the other hand, proposes a two-level structure. It first uses an inverted multi-index [2] to partition data into two subspaces; it then trains a rotation matrix for each node independently for finer-grained optimization.

2.3 Learned Indexes

Recently, the case of learned indexes has triggered intensive research (e.g., [10, 15, 19, 28, 33, 43]). The basic idea is to view indexes as "models" that predict the "positions" of relevant data objects. It has been shown that learned indexes can outperform traditional tree-based indexes, such as B-tree [28] and R-tree [33], by exploiting workload-specific access patterns. Notable optimizations include fitting-tree [15], which uses piece-wise linear functions with a bounded error specified at index construction time, and ALEX [10], which further supports index updates. The main challenge of applying learned indexes on high-dimensional data is the well-known "curse of dimensionality," which makes it difficult to build good models with limited training data (compared to the number of data dimensions). Notably, the recent work on learned multi-dimensional indexes only tested datasets with less than ten dimensions [33], whereas time series data typically involve tens or hundreds of dimensions (as used in our experiments).

2.4 Other Technologies

In addition to tree-based indexes, which is the focus of this paper, there are other technologies for processing exact kNN queries as well. For example, VA-File [42] converts raw data into quantization-based approximations. VA+File [13] improves VA-File by converting raw data using Karhunen Loeve Transform (KLT) [24].

3 STUDY OF TREE-BASED INDEXES

We start by a systematic study of existing tree-based indexes in terms of their effectiveness when processing kNN queries.

3.1 Experiment Setup

We analyze three indexes that have been covered in Section 2 for processing kNN queries: DS-Tree [40], iSAX [36], and VA+Index [13], which are considered as the state of the art [11, 12] among others. We use C/C++ implementations of these indexes from [11]. For the VA+Index implementation, we further build R-tree on top of the VA+File features [13]. We run experiments using a PC with Intel Xeon E5-2620 CPU that runs Ubuntu 16.04 and GCC 5.4.0.

3.1.1 Datasets. In our experiments, We use one synthetic dataset, *RandomWalk* (*RWalk* for short), that is generated based on a random-walk model. It has been widely used in previous work [11, 40]. In addition, we also use two real datasets, *Deep* and *ECG*. Table 1 presents the details of the datasets.

Deep [3] is an image dataset that contains over 1 billion image vectors. The image vectors were extracted from the last layers of GoogLeNet [38] with parameters trained by ImageNet [35], and were further converted into 96-dimensional vectors using principal component analysis (PCA).

Table 1: Summary of Datasets

Dataset	Type	Dimension	#Rows	Size on Disk
RWalk-{1M,100M} [11]	Synthetic	256	$\{10^6, 10^8\}$	{1GB, 100GB}
Deep-{1M,100M} [3]	Real	96	$\{10^6, 10^8\}$	{384MB, 38.4GB}
ECG-1M [5]	Real	500	10^6	2GB

ECG [5] contains 549 ECG records from 290 patients. Each ECG record contains 15 measurement signals and is further divided into a list of 500-dimensional time series.

3.1.2 Query Processing. We expand the implementation of 1NN search by [11] to support kNN search for all three indexes evaluated. We maintain an in-memory max-heap to maintain the kNN objects found so far. Since for all three indexes we can easily derive a distance lower-bound for each index node given a query, we simply compute the distance between the query and the current k^{th} nearest neighbor (i.e., the root element of the max-heap) and discard any index node with a larger distance lower-bound.

3.1.3 Measurements. In our experiments, raw data objects are resident on disk and leaf nodes contain pointers to them. As we mentioned in the introduction, the goal of kNN query processing is to minimize the number of disk accesses required to reach a certain degree of *recall*. We now formally define this performance metric.

We define *first-N-recall*, i.e., the average recall of the first N index leaf nodes accessed, as follows:

$$\text{first-}N\text{-recall} = \frac{1}{N}\sum_{i=1}^{N}\frac{T(i)}{k}, \tag{1}$$

where $T(i)$ is the number of true nearest neighbors in the first i accessed leafs. Note that, when N=1, *first-1-recall* is equivalent to the *precision* metric used by approximate kNN query processing [7, 40]. Unless otherwise stated, we set N=10 and report *first-10-recall*.

3.2 Analysis of Recall

We next perform an analysis based on the *first-N-recall* metric defined in Equation 1.

3.2.1 Ideal Access Order. Given a set of candidate index leaf nodes, we count the number of *true* nearest neighbors each candidate node contains, and access the candidates in the order of decreasing counts. We call this the *ideal access order*.

Example 3.1. Consider a case when k=5 and N=10. Suppose the first 10 candidate leaf nodes given by the index (ordered by their distance lower-bounds) are L_1, ..., L_{10}. Assume that L_5 contains 3 of the 5 nearest neighbors, L_2 and L_6 each contains 1 of them, whereas the others do not contain any. The **ideal access order** of the leaf nodes is then L_5, L_2, L_6, and others. The *first-10-recall* of the ideal access order is 0.94, whereas it is 0.64 for the original order.

Theorem 3.2. *For any k and N, the ideal access order achieves the highest first-N-recall.*

Proof. Without loss of generality, assume that the ideal access order is $\mathbb{O} = \{L_1, L_2, \cdots, \}$ with first-N-recall R. Suppose that the number of true nearest neighbors in L_i is $c(L_i)$. By definition, we have $c(L_1) \geq c(L_2) \cdots \geq c(L_N)$. Suppose that there exists a different order $\mathbb{O}' = \{L'_1, L'_2, \cdots, L'_N\}$ with the *highest* first-N-recall R' such that $R' > R$. Then there exists

$$1 \leq a < b \leq N \text{ such that } c(L_a) < c(L_b). \tag{2}$$

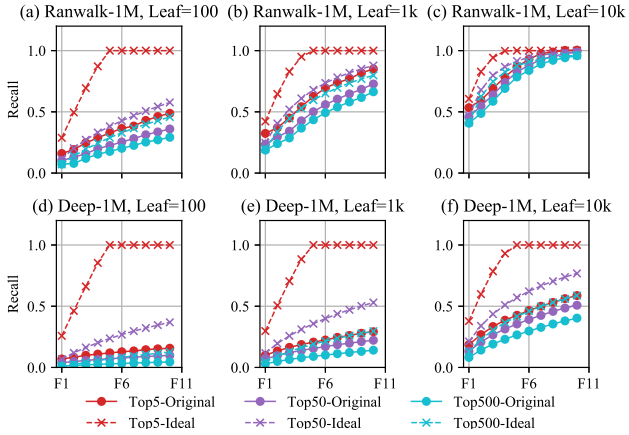


Figure 1: The first-10-recalls of DS-Tree w.r.t. different k's.

By Equation (1), we can write R' as

$$R' = \frac{1}{N}\sum_{i'=1}^{N}\frac{1}{k}\sum_{j'=1}^{i'} c(L_{j'}) = \frac{1}{Nk}\sum_{i'=1}^{N}\sum_{j'=1}^{i'} c(L_{j'})$$
$$= \frac{1}{Nk}\sum_{i'=1}^{N}(N-i'+1)\cdot c(L_{i'}).$$

By (2), we can exchange $L_{i'}$ and $L_{j'}$ in $\mathbb{O}'$, and obtain a new order $\mathbb{O}''$ with first-N-recall R'' such that

$$R'' - R' = \frac{1}{Nk}[(N-a+1)\cdot c(L_b) + (N-b+1)\cdot c(L_a)] -$$
$$\frac{1}{Nk}[(N-a+1)\cdot c(L_a) + (N-b+1)\cdot c(L_b)]$$
$$= \frac{1}{Nk}[(b-a)\cdot c(L_b) + (a-b)\cdot c(L_a)]$$
$$= \frac{1}{Nk}\cdot(b-a)\cdot[c(L_b) - c(L_a)] > 0$$

which contradicts the fact that $\mathbb{O}'$ has the highest first-N-recall. As a result, we must have $R' \le R$, which implies that R is the highest first-N-recall one can achieve. □

3.2.2 Original vs. Ideal Access Orders. Figure 1 depicts the first-10-recall curves of DS-Tree with leaf node size varying among {100, 1k, 10k} and $k \in \{5, 50, 500\}$. In Figure 1, the solid lines are recall curves following the original access orders (i.e., by the distance lower-bounds), whereas the dashed lines are recall curves following the ideal access orders.

We investigate the original access orders first. On *RWalk-1M*, the recall decreases as k increases, because there are more nearest neighbors and we need to look into more index nodes to search for them. For example, when the leaf size is 100, the first-10-recall on *RWalk-1M* drops from 0.292 to 0.143 as k increases from 5 to 500. If we fix k, then the first-10-recall increases when the leaf size increases. For instance, when k=5 and the leaf size increases from 100 to 10k, the first-10-recall increases from 0.334 to 0.837. On the other hand, the first-10-recall drops significantly on *Deep-1M* compared to *RWalk-1M*.

We next study the ideal access orders, which yield the optimal recall curves by Theorem 3.2. we observe that, regardless of the values of k and N, there is a significant gap between the recall curve by the original access order compared to the one by the ideal access order. Moreover, the gap becomes smaller when the leaf size increases. For example, the first-10-recalls on *RWalk-1M* are 0.835, 0.885, and 0.937 following the ideal access orders when the leaf sizes are 100, 1k, and 100k, which are 0.501, 0.267, and 0.1 higher than the first-10-recalls following the original access orders. The observation on *Deep-1M* is similar.

(Summary) The large gap between the original and ideal access orders in terms of the first-N-recall suggests a huge room for potential improvement. Consequently, in the rest of this paper, we will focus on optimizing the access order of index leaf nodes.

Algorithm 1 Index-based kNN query processing

Require: Tree-based index $\mathcal{I}$, kNN query Q
Ensure: The k nearest neighbors of Q

```
1:  H ← new Max-Heap(k); Queue ← new Priority-Queue;
2:  H, bsf ← Initial_kNN(Q, I);
3:  Add I.root to Queue;
4:  while node ← pop next node from Queue do
5:    if node.dist > bsf.dist then
6:      break;
7:    if node is leaf then
8:      Load all raw obj's in node from disk;
9:      for each obj in node do
10:       obj.dist ← computeDistance(Q, obj);
11:       if obj.dist < bsf.dist then
12:         Insert obj into H w.r.t. obj.dist;
13:         bsf.dist ← H.root.dist;
14:   else
15:     for each child in node do
16:       child.lbd ← computeLowerBound(Q, child);
17:       if child.lbd < bsf.dist then
18:         Insert child to Queue according to child.lbd;
19: Return H.toList();
```

4 ML-ENHANCED INDEXES

We propose *ML-enhanced indexes* that can be viewed as a combination of classic tree-based indexes and the ongoing trend of "learned indexes." Specifically, we build a classic index as it is (e.g., DS-Tree, iSAX, R-tree, etc.) and then improve the access order of its leaf nodes when processing a kNN query, using ML techniques. In this section, we present the details of this idea. Moreover, we propose an alternative baseline that reorders leaf nodes using *quantization* techniques from the area of approximate kNN query processing.

In the following, we start by presenting the general framework that is shared by our ML-based and quantization-based reordering techniques. We then present the details of the two reordering techniques, respectively.

4.1 General Framework

Algorithm 1 outlines the general framework of kNN query processing using traditional tree-based indexes. It uses a priority query *Queue* to determine the access order of index that leaf nodes survived after pruning, based on the distance lower-bounds.

In contrast, Algorithm 2 summarizes our enhanced kNN query processing framework by reordering the index leaf nodes being

accessed. Based on the tree-based index and the reordering strategy (details in the following subsections), we obtain the optimized leaf-node access order. We then follow this optimized access order to visit the relevant leaf nodes. For each leaf node we simply load all raw data objects it contains from disk and process them one by one. Algorithm 2 will not miss any true nearest neighbor as it still relies on the node pruning strategy based on distance lower-bounds.

Algorithm 2 Index-based kNN query processing w/ reordering

Require: Tree-based index $\mathcal{I}$, k-NN query Q, Reordering strategy $\mathcal{R}$ for the leaf index nodes of $\mathcal{I}$
Ensure: The k nearest neighbors of Q
1: $\mathcal{H} \leftarrow$ **new** Max-Heap(k); $Queue \leftarrow$ **new** Priority-Queue;
2: $\mathcal{H}$, bsf, $Queue \leftarrow$ optimizedNodeOrder(Q, $\mathcal{I}$, $\mathcal{R}$);
3: **while** $node \leftarrow$ pop next node from $Queue$ **do**
4: **if** $node$.dist > bsf.dist **then**
5: Continue;
6: Load all raw obj's in $node$ from disk;
7: Calculate real distances for all obj's in $node$;
8: Update $\mathcal{H}$ and bsf.dist;
9: Return $\mathcal{H}$.toList();

4.2 Reordering by Deep Neural Networks

We propose an ML-based reordering strategy by leveraging deep neural networks (DNNs). Neural networks can extract features from raw data without specific domain knowledge when performing feature engineering. As a result, they are attractive when modeling high-dimensional data. Meanwhile, the recent advancement in hardware technologies (such as the utilization of GPUs and TPUs) has significantly improved the efficiency of computations (e.g., model training and inference) over DNNs, which makes it feasible when integrating DNNs into *online* index lookup and kNN processing.

Our basic observation is the following: Once the index is built, the *distribution* of the k nearest neighbors of a given query in the index leaf nodes is *fixed* and thus *learnable*. Based on this observation, our goal is to *predict* which index leaf nodes contain the nearest neighbors. Since the leaf nodes are also *fixed* once the index is constructed, we can model this prediction task as a *multi-class* classification problem.

Formally, given a time series dataset $\mathbb{S} = \{S_1, S_2, \cdots, S_n\}$, suppose that the index $\mathcal{I}$ distributes the data objects of $\mathbb{S}$ into M leaf nodes. Given an input query object Q, the output of the ML model is an M-dimensional vector $\mathbf{y}$ such that y_i $(1 \le i \le M)$ represents the *fraction* of nearest neighbors of Q that would land on the index leaf node $\mathbb{L}_i$. In other words, $\mathbf{y}$ represents the *probability densities* of the distribution of nearest neighbors over the index leaf nodes.

As Figure 2 shows, we randomly sample data objects from $\mathbb{S}$ to construct the training set $\mathbb{T} = \{T_1, T_2, \cdots, T_M\}$. We then specify the M-dimensional labels $\mathbf{y}$ for each training data object T, which are the probability densities of its nearest neighbors over the index leaf nodes. Note that this information can be obtained by first finding the ground-truth k-NNs of the training object and then computing their distribution. Specifically, for $T \in \mathbb{T}$ and $1 \le i \le M$,

$$y_i = \frac{1}{k}|\{S \in \mathbb{L}_i | S \in k\text{NN}(T)\}|. \quad (3)$$

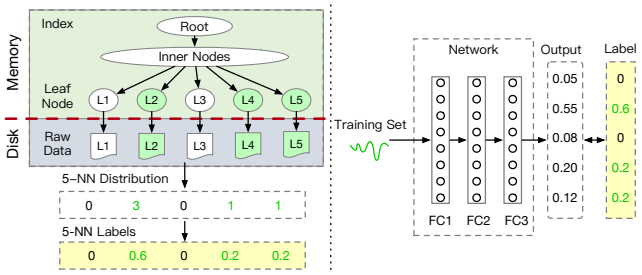


Figure 2: (a) Given a training data object, its labels are the probability densities of its kNNs; (b) Train the neural network with the training labels.

Algorithm 3 DNN-based reordering strategy

Require: Tree index $\mathcal{I}$, k-NN query Q, DNN model $\mathcal{M}$
Ensure: The optimized access order
1: DNN-Order $\mathbb{O}_D \leftarrow \mathcal{M}$.predict($Q$, $\mathcal{I}$);
2: Load the first leaf node in $\mathbb{O}_D$ and initialize $\mathcal{H}$ and bsf;
3: Prune all leaf $node$s in $\mathcal{I}$ with $node$.lbd > bsf.dist and obtain the candidate set C with access order $\mathbb{O}_C$;
4: $Queue \leftarrow$ combineAccessOrders($\mathbb{O}_D$, $\mathbb{O}_C$) with Alg. 4;
5: Return $\mathcal{H}$, bsf, $Queue$;

Algorithm 4 Combine access orders

Require: Original access order $\mathbb{O}_C$ with candidate set C, DNN-order $\mathbb{O}_D$, $\lambda \in [0, 1]$
Ensure: Combined access order of $\mathbb{O}_C$ and $\mathbb{O}_D$
1: Remove leaf nodes from $\mathbb{O}_D$ that are not in C;
2: $W_C^{\text{norm}} \leftarrow$ normalize(W_C);
3: $W_D^{\text{norm}} \leftarrow$ normalize(W_D);
4: $W_F \leftarrow \lambda \cdot W_D^{\text{norm}} + (1 - \lambda) \cdot W_C^{\text{norm}}$
5: Reorder $\mathbb{O}_D$ according to W_F;
6: Return $\mathbb{O}_D$;

We train a deep neural network afterwards, based on $\mathbb{T}$ and the above labeling method.

Algorithm 3 presents our DNN-based reordering strategy. We assume that the index and the neural network have been constructed offline before any query comes.

One problem of purely relying on the output of the DNN model to guide the search of k-NNs is that one has to access all index leaf nodes since the model cannot guarantee that leaf nodes with zero probabilities do not contain true nearest neighbors. As a result, one might end up with visiting all irrelevant index leaf nodes that do not contain any nearest neighbor. We fix this problem by combining the DNN-based access order with the original access order of the candidate leaf nodes given by Algorithm 1. That is, we take only the leaf nodes that appear in the original access order and reorder them based on the output from the DNN model.

As shown in Algorithm 4, we take the two access orders $\mathbb{O}_D$ and $\mathbb{O}_C$ as inputs. $\mathbb{O}_C$ is obtained by sorting C from smallest to largest according to the distance lower-bounds, whereas $\mathbb{O}_D$ is obtained by sorting all leaf nodes from largest to smallest based on their probability densities predicted by the DNN model. We first remove all nodes that are not in C from $\mathbb{O}_D$ (line 1). We then normalize the

Algorithm 5 Quantization-based reordering strategy

Require: Tree index $\mathcal{I}$, k-NN query Q, Quantization $\mathcal{M}$
Ensure: The optimized access order.
1: $\mathcal{H}$, *bsf* ← Initial_kNN(Q, $\mathcal{I}$);
2: Prune all leaf *node*s in $\mathcal{I}$ with *node*.lbd > *bsf*.dist and obtain the candidate set C with access order $\mathbb{O}_C$;
3: **for** each *leaf* in C **do**
4: **for** each *code* in *leaf* by $\mathcal{M}$ **do**
5: *obj*' ← reconstruct *obj* from *code*;
6: Estimate distance d' ← Dist(*obj*', Q);
7: Compute a score for *leaf* w.r.t. estimated distances;
8: *Queue* ← Reorder $\mathbb{O}_C$ according to leaf scores;
9: Return $\mathcal{H}$, *bsf*, *Queue*;

weights of the nodes in $\mathbb{O}_C$ and $\mathbb{O}_D$ (line 2 and line 3). Finally, we compute combined weights W_F for the nodes (line 4) and reorder the nodes according to W_F.

4.3 Reordering by Quantization Methods

Quantization methods have been widely used in approximate kNN query processing. Using various dimension reduction techniques, quantization methods can convert high-dimensional data into compact binary codes (called "quantization codes"). For example, a raw data point represented by a 96-dimensional floating-point vector (96×4×8=3,072 bits) can be represented by a quantization code with 64 bits, with a compression ratio of more than 48 times. We can further estimate the distances between data objects using their quantization formats. However, the pruning strategy based on distance lower-bounds does not guarantee "no false dismissals" when applied to such estimated distances.

We design an alternative reordering strategy based on quantization. The basic idea is to use the estimated distances as the delegates of the corresponding real distances and reorder the objects based on their estimated distances. Algorithm 5 presents the details. Here, each data object in a candidate leaf node has been compressed using quantization method $\mathcal{M}$ and needs to be reconstructed. It also employs a scoring function to rate the index leaf nodes and order them based on their scores. There are various choices for this scoring function, though it should not depend on any specific data object in a leaf node. In our experiments, we have tried both the `Avg` and `TopK` functions. The `Avg` function computes the mean of the distances between *all* reconstructed objects in a leaf node and the query object Q, whereas the `TopK` function only considers the mean of the top k objects with closest distances.

5 EXPERIMENTAL EVALUATION

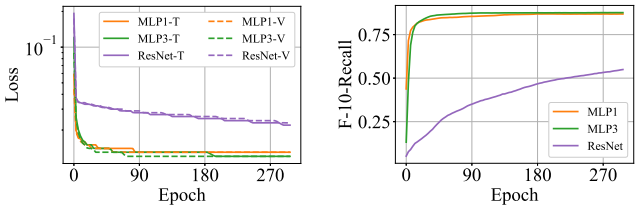
In this section, we evaluate the ML-enhanced indexes, using DNN-based or quantization-based reordering strategies, by comparing them with their traditional counterparts.

5.1 Experiment Settings

5.1.1 DNN-based Reordering Strategy. We implement the DNN model using PyTorch-0.4.1 and a Tesla K80 or GeForce GTX GPU with 12GB memory. In our experiments, we assume that the training and test sets follow the same distribution (by sampling from the base dataset). We used base datasets that contain 1M (10^6) and 100M (10^8) data objects. The training set contains 100K objects and the test set contains 100 (query) objects, respectively. The training set and test set do not overlap. We also experimented with three DNNs [41]: (1) **MLP-1**, which is a one-layer fully-connected network with the following size of the hidden layer [9]

$$\frac{\text{size of training set}}{2 \times (\#\text{ of dimensions} + \#\text{ of categories})};$$

(2) **MLP-3**, which is a three-layer fully-connected network with the size of the hidden layer set to 500 [41]; (3) **ResNet-3**, which contains three ResNet blocks [20, 41].



(a) Training loss and valid loss **(b) The First-10-Recall curve**

Figure 3: Training of three DNNs for DS-Tree on *RWalk-1M*.

5.1.2 Quantization-based Reordering Strategy. We use LOPQ as the quantization method in Algorithm 5, which extends the previous classic quantization methods PQ [22] and OPQ [16]. We obtain the C/C++ implementation of LOPQ from [23]. All our algorithms are single-thread implementations.

5.2 DNN-based Reordering

We now evaluate the DNN-based reordering strategy. We start by examining the training processes of the three aforementioned neural network architectures and choosing appropriate ones. We then test and analyze the performance of the reordering strategy using the DNN models on the three datasets and three tree indexes that have been used in Section 3, in terms of the first-10-recall. We further study the generalizability of the reordering strategy when training and testing on different k's, as well as its scalability when expanding the datasets from 1M to 100M data objects.

5.2.1 DNN Training. Figure 3a shows the training progress of the three neural networks MLP1, MLP3, and ResNet-3, when applied to DS-Tree on *RWalk-1M*. As shown in Figure 3a, the training and validation losses of the three networks decrease rapidly within 20 epochs. After 300 epochs, the training loss of MLP1 and MLP3 decrease to around 0.013, whereas the training and validation losses of ResNet are no less than 0.022. The average time spent on each epoch of the three networks is 24.2s, 26.4s, and 45.5s, respectively.

Figure 3b further presents the first-10-recall on the test (query) set in each epoch. The initial first-10-recall of MLP1 is 0.476, which is much higher than that of MLP3 (0.124) and ResNet (0.078). MLP1 improves the first-10-recall rapidly to around 0.8 and then slows down. MLP3 improves the first-10-recall rapidly and surpasses MLP1 after about 30 epochs. On the other hand, the performance of ResNet rises slowly: The first-10-recall of ResNet after 300 epochs is only 0.537. Considering its training time and low first-10-recall, we will not use ResNet in the rest of our experiments.

5.2.2 First-10-recall. Table 2 presents the first-10-recalls of DNN-based reordering. The average first-10-recalls of MLP1 over three

Table 2: First-10-recalls of DNN-based reordering

Method	RWalk-1M			Deep-1M			ECG-1M		
	DS-Tree	iSAX	VA+Index	DS-Tree	iSAX	VA+Index	DS-Tree	iSAX	VA+Index
NN_MLP3	**0.879**	**0.804**	0.935	0.537	0.408	0.728	0.901	**0.837**	**0.972**
NN_MLP1	0.871	0.788	0.937	0.508	0.387	0.724	0.822	0.795	0.963
QUAN_AVG_V8	0.706	0.51	0.932	0.342	0.097	0.552	0.589	0.505	0.233
QUAN_AVG_V16	0.703	0.509	0.931	0.345	0.105	0.563	0.593	0.509	0.232
QUAN_AVG_V32	0.704	0.505	0.931	0.353	0.108	0.559	0.585	0.508	0.227
QUAN_TopK_V8	0.833	0.795	0.939	0.584	0.471	0.733	0.849	0.797	0.911
QUAN_TopK_V16	0.859	0.786	**0.945**	**0.604**	**0.488**	**0.761**	0.862	0.795	0.918
QUAN_TopK_V32	0.856	0.793	0.941	0.602	0.486	0.749	0.858	0.815	0.917
Original	0.837	0.698	0.833	0.424	0.279	0.095	**0.93**	0.773	0.895
Ideal	0.935	0.907	0.959	0.871	0.855	0.912	0.964	0.932	0.987

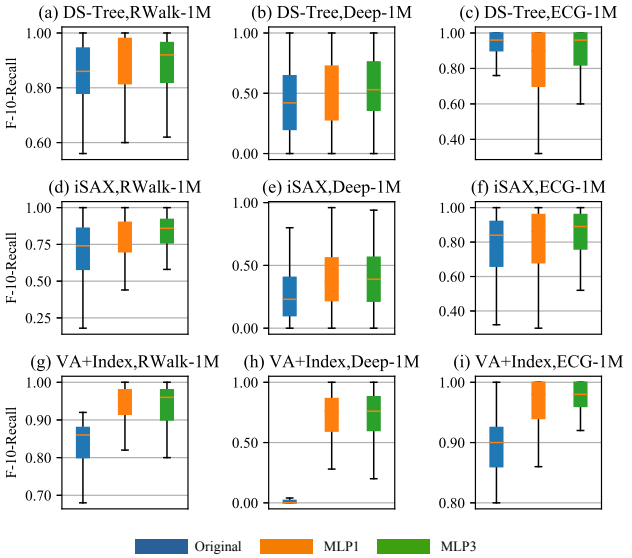


Figure 4: The first-10-recalls of DNN-based reordering.

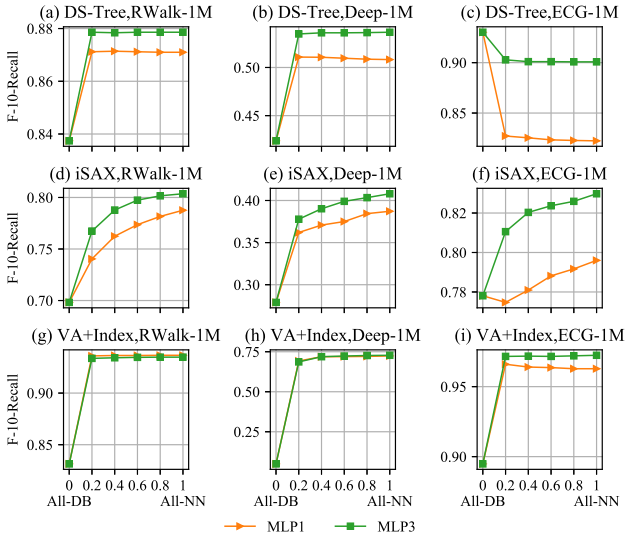


Figure 5: Combining DNN-based with original access order.

indexes are 0.865, 0.540, and 0.860 on *RWalk-1M*, *Deep-1M*, and *ECG-1M*, respectively. Compared with the original access orders, its average first-10-recalls over the three indexes only decrease by 0.6% on *ECG-1M* but increase by 9.72% and 115.29% on *RWalk-1M* and *Deep-1M*. MLP3 achieves better average first-10-recalls on almost all three datasets with 0.873, 0.558, and 0.903, which outperforms the original access orders by 10.65%, 122.47%, and 4.31%.

Figure 4 further presents distributions of the first-10-recalls (w.r.t. the 100 testing query objects) over the three datasets. On *RWalk-1M*, MLP1 and MLP3 exhibit similar variances for DS-Tree and VA+Index; however, the lowest first-10-recall of MLP3 for iSAX is significantly higher than that of both MLP1 and the original access order. On *Deep-1M*, the variances are large for both MLP1 and MLP3. On *ECG-1M*, all approaches have relatively high recalls.

We now evaluate the benefits of combining DNN-based access order with original access order using the weighted approach in Algorithm 4. Figure 5 presents the results. We vary the ratio λ in $\{0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. If λ=0, the combined access order reduces to the original access order, whereas if λ=1 it reduces to the DNN-based access order. Table 3 illustrates the effect of λ with some cases where the combined orders can outperform the uncombined ones.

5.2.3 Time Overhead. Next, we investigate the time overhead of DNN-based reordering. In our experiments, the average prediction time of MLP1 and MLP3 for each query is only 0.21 ms and 0.29 ms, which is negligible. Figure 6 shows the first-10-recalls w.r.t. the elapsed time. On all three datasets, MLP3 outperforms or performs no worse than MLP1. Except for DS-Tree on *ECG-1M*, DNN-based access orders are superior to the original ones in all the other cases.

Table 3: Combined access orders: Case studies

Dataset	Method	Original	DNN-based	Combined	Best λ
RWalk-1M	DS-Tree + MLP1	0.837	0.871	0.872	0.4
RWalk-1M	DS-Tree + MLP3	0.837	0.879	0.881	0.2
Deep-1M	DS-Tree + MLP1	0.424	0.508	0.511	0.2
ECG-1M	VA+Index + MLP1	0.895	0.963	0.966	0.2

5.2.4 Generalizability. Users may issue queries with different k's from the k used when training the DNN models, and it is perhaps not feasible to train a DNN model for each individual k. As a result, it is interesting to investigate the generalization capacity of the trained model when it faces queries with different k's.

We use DS-Tree to train DNN models on the three datasets with $k \in \{5, 10, 20, 50\}$, and then test k-NN queries by varying k from 1 to 100. Figure 7 illustrates the generalization behavior of DNN-based reordering. As the k in the testing queries increases, the number of true nearest neighbors increases as well, which may be distributed into more leaf nodes. As a result, we observe that the first-10-recalls decrease for all access orders evaluated.

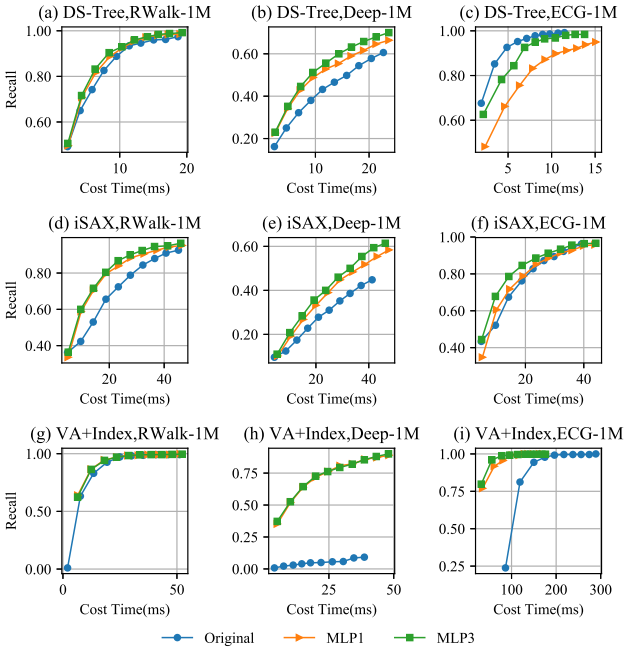


Figure 6: First-10-recalls w.r.t. execution time.

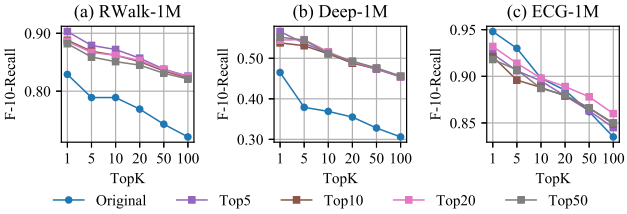


Figure 7: Training and testing on different k's.

On *RWalk-1M* and *Deep-1M*, the DNN-based access orders remain superior to the original access orders. On *ECG-1M*, when k >10, the first-10-recall of the original access order drops rapidly and is surpassed by the DNN-based access order. Overall, when we increase k from 1 to 100, the first-10-recall of DNN-based access order decreases less than that of the original access order, which demonstrates the robustness and generalizability of DNN-based reordering. The fact that our approach is not so sensitive to the k used in model training is intriguing, which implies that we can stay with using relatively small k's to reduce the preprocessing overhead of generating training/testing labels by finding the ground-truth kNNs. Meanwhile, we should also note that, when the discrepancy between the k's used in training and testing becomes large, the trained model will inevitably become less effective.

5.2.5 Scalability. To test the scalability of ML-enhanced indexes, we further conduct experiments on expanded versions of *RWalk* and *Deep* that contain 100M data objects. The corresponding disk file sizes are 100GB and 38.4GB, respectively. We set the leaf-node size to 1M so that the indexes can keep similar number of leaf nodes as in the previous experiments.

Table 4 presents the first-10-recalls on the 100M datasets. MLP3 outperforms other approaches on all combinations of indexes and datasets. This demonstrates the scalability of DNN-based reordering.

Table 4: First-10-recall on 100M datasets

Method	RWalk-100M			Deep-100M		
	DS-Tree	iSAX	VA+Index	DS-Tree	iSAX	VA+Index
Original	0.881	0.815	0.865	0.522	0.463	0.249
MLP1	0.866	0.789	0.960	0.557	0.489	0.774
MLP3	**0.885**	**0.837**	**0.962**	**0.567**	**0.506**	**0.789**

Table 5: Execution time on 100M datasets

Method	RWalk-100M			Deep-100M		
	DS-Tree	iSAX	VA+Index	DS-Tree	iSAX	VA+Index
Index time	242m	74m	78m	102m	7m	42m
DNN-based	572s	321s	32s	437s	357s	28s

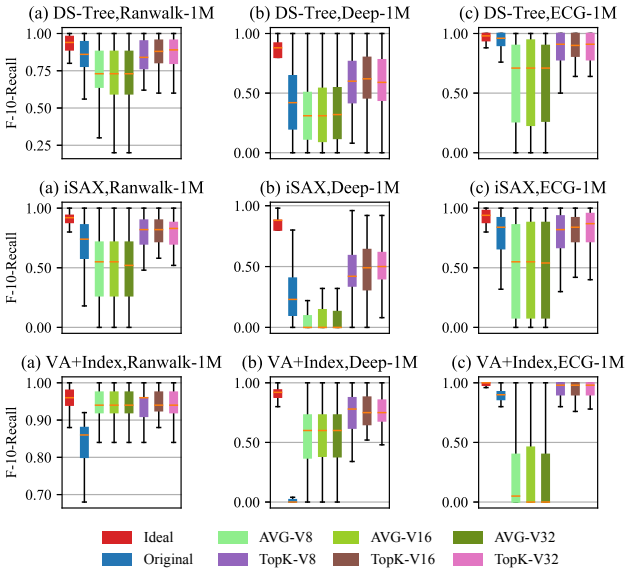


Figure 8: First-10-recalls by quantization-based reordering.

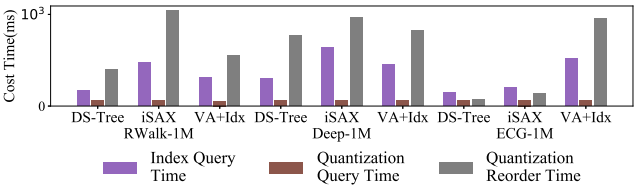


Figure 9: Execution time of quantization-based reordering.

Notably, when the number of data objects increases to 100M, the index construction time and query processing time increase, while the training time and prediction time of the DNN models remain stable. As shown in Table 5, in most cases index construction takes more than one hour to finish. Meanwhile, the first-10-recall of DNN-based access order starts to outperform the original access order after less than 10 minutes of training.

5.3 Quantization-based Reordering

Figure 8 presents the first-10-recalls of LOPQ when `Avg` and `TopK` are used as scoring functions. We vary the cluster number used by LOPQ within {8, 16, 32}. For comparison, Figure 8 also shows the recalls of the ideal access order and the original order.

As we can see, for most combinations of indexes and datasets, the recall of using `Avg` is lower than that of the original access order, whereas the recall of using `TopK` is higher than that of the original access order. There is an observable outlier, though, for VA+Index on *Deep-1M*, where the recall of the original access order is only 0.095 and both `AVG` and `TopK` lead to significant improvements. Meanwhile, the variance of the recall is small when using `TopK`, while it is relatively large when using `Avg`. This suggests that, the performance of `TopK` is more stable than `Avg` across the queries tested. Nonetheless, as shown in Figure 9, the use of `TopK` introduces additional time overhead.

Finally, compared to DNN-based reordering, the extra time overhead introduced by quantization methods has a noticeable impact on query processing efficiency. Although the reconstruction of the data objects and computation of the estimated distances can be both done in memory, there are still a large number of index leaf nodes and raw data objects to be examined in the post-processing step of query processing, usually in the order of $O(M)$ where M is the total number of leaf nodes in the index.

6 CONCLUSION

In this work, we propose *ML-enhanced* indexes for kNN query processing over high-dimensional data, which combine the merits of traditional tree-based indexes and the ongoing trend of "learned indexes." ML-enhanced indexes take as input the leaf nodes of a traditional index, and improve their access order based on predictions from ML models (notably, deep neural networks). Experimental evaluation shows the advantage of ML-enhanced indexes over their original counterparts, in terms of the recall of the nearest neighbors. Future work includes exploring the utilization of other types of ML methods, making ML-enhanced indexes more scalable (to indexes with more leaf nodes) and more adaptable (to data updates).

7 ACKNOWLEDGMENTS

This work was supported by NSFC Grant (No. 62021002).

REFERENCES

[1] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-Index: Pushing the Scalability-Accuracy Boundary for Approximate kNN Search in High-Dimensional Spaces. *VLDB Endow.* 11, 8 (2018), 906–919.

[2] A. Babenko and V. Lempitsky. 2015. The Inverted Multi-Index. *TPAMI* 37, 6 (2015), 1247–1260.

[3] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *CVPR*. IEEE Computer Society, 2055–2063.

[4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*. ACM, 322–331.

[5] R Bousseljot et al. 1995. Nutzung der EKG-Signaldatenbank CARDIODAT der PTB über das Internet. *Biomedizinische Technik/Biomedical Engineering* 40, s1 (1995), 317–318.

[6] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn J. Keogh. 2010. iSAX 2.0: Indexing and Mining One Billion Time Series. In *ICDM*. 58–67.

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

[8] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB*. 426–435.

[9] Howard B Demuth, Mark H Beale, Orlando De Jess, and Martin T Hagan. 2014. *Neural network design*. Martin Hagan.

[10] Jialin Ding et al. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*. ACM, 969–984.

[11] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2018. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB* 12 (2018), 112–127.

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

[13] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. 2000. Vector Approximation Based Indexing for Non-uniform High Dimensional Data Sets. In *CIKM*. ACM, 202–209.

[14] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

[15] Alex Galakatos et al. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD*. ACM, 1189–1206.

[16] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *TPAMI* 36, 4 (2014), 744–755.

[17] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.

[18] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, Beatrice Yormark (Ed.). ACM Press, 47–57.

[19] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. In *AIDB@VLDB*.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. IEEE Computer Society, 770–778.

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

[22] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE TPAMI* 33, 1 (2011), 117–128.

[23] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. IEEE Computer Society, 2329–2336.

[24] Kari Karhunen. 1947. *Uber lineare Methoden in der Wahrscheinlich-keitsrechnung*. Vol. 37. Sana.

[25] Eamonn Keogh et al. 2001. Dimensionality reduction for fast similarity search in large time series databases. *KAIS* 3, 3 (2001), 263–286.

[26] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM SIGMOD Record* 30, 2 (2001), 151–162.

[27] Eamonn J. Keogh and Chotirat (Ann) Ratanamahatana. 2005. Exact indexing of dynamic time warping. *KAIS* 7, 3 (2005), 358–386.

[28] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. ACM, 489–504.

[29] Jessica Lin, Eamonn J. Keogh, Li Wei, and Stefano Lonardi. 2007. Experiencing SAX: a novel symbolic representation of time series. *DMKD* 15, 2 (2007), 107–144.

[30] Michele Linardi and Themis Palpanas. 2018. ULISSE: ULtra Compact Index for Variable-Length Similarity Search in Data Series. In *ICDE*. 1356–1359.

[31] Yury A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *TPAMI* 42, 4 (2020), 824–836.

[32] Marius Muja and David G. Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *VISAPP 2009*. INSTICC Press, 331–340.

[33] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.

[34] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. MESSI: In-Memory Data Series Indexing. In *ICDE*. IEEE, 337–348.

[35] Olga Russakovsky et al. 2015. ImageNet Large Scale Visual Recognition Challenge. *IJCV* 115, 3 (2015), 211–252.

[36] Jin Shieh and Eamonn Keogh. 2009. iSAX: disk-aware mining and indexing of massive time series datasets. *DMKD* 19, 1 (2009), 24–57.

[37] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *VLDB* 8, 1 (2014), 1–12.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *CVPR*. 1–9.

[39] J. Wang, J. Wang, J. Song, X. S. Xu, H. T. Shen, and S. Li. 2015. Optimized Cartesian K-Means. *IEEE TKDE* 27, 1 (2015), 180–192.

[40] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A Data-adaptive and Dynamic Segmentation Index for Whole Matching on Time Series. *PVLDB* 6 (2013), 793–804.

[41] Zhiguang Wang et al. 2017. Time series classification from scratch with deep neural networks: A strong baseline. In *IJCNN*. IEEE, 1578–1585.

[42] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB*. 194–205.

[43] Yingjun Wu et al. 2019. Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations. In *SIGMOD*. 1223–1240.

[44] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: The Adaptive Data Series Index. *The VLDB Journal* 25, 6 (2016), 843–866.