LM-DiskANN: Low Memory Footprint in Disk-Native Dynamic Graph-Based ANN Indexing

Yu Pan[1], Jianxin Sun[2], Hongfeng Yu[2]
[1]Department of Biological System Engineering, University of Nebraska-Lincoln, Lincoln, NE, USA
[2]School of Computing, University of Nebraska-Lincoln, Lincoln, NE, USA

***Abstract*—Approximate Nearest Neighbor (ANN) search has become a fundamental operation in numerous applications, including recommendation systems, computer vision, and natural language processing. The advent of Large Language Models (LLMs) arouses new interest in developing more efficient ANN algorithms, which will be the core functionality of vector databases as long-term memory of LLM. Multiple types of index structures, such as hashing-based, tree-based, and quantization-based, have been developed for ANN, and recently, graph-based algorithms have become the SOTA paradigm with the best trade-off between recall rate and query latency. However, almost all the existing graph-based index structures can only be hosted in memory due to the otherwise frequent I/O operations during searching if the graph-based index is stored on disk. The problem follows that for extremely large datasets, it is infeasible to accommodate the whole graph-based index in memory, and furthermore, it is difficult to build the whole index in memory at once. Thus, it is favorable if the graph-based index can be stored purely on disk and loaded into memory on demand during searching on the graph. There are existing efforts, such as DiskANN, which try to store graph-based index structure on disk while still keeping a compressed version of the dataset in memory to reduce disk I/O and speed up distance calculation. In this paper, we introduce LM-DiskANN, a novel dynamic graph-based ANN index that is designed specifically to be hosted on disk while keeping a low memory footprint by storing complete routing information in each node. By conducting extensive experiments on multiple benchmark datasets, we demonstrate that LM-DiskANN achieves a similar recall-latency curve while consuming much less memory compared with SOTA graph-based ANN indexes. Furthermore, its scalability and adaptability make it a promising solution for future big data applications.**

***Index Terms*—Approximate Nearest Neighbor, Graph Index, Memory Footprint**

I. Introduction

The nearest neighbor search problem is centered around developing an efficient index structure and affiliated algorithms that can swiftly pinpoint the k nearest neighbors (KNN) to any given query point within a specific dataset. This challenge is a cornerstone in the field of algorithm research and finds its utility in various domains, such as computer vision, document retrieval, and recommendation systems. In these contexts, items like images, documents, or user profiles can be mapped into a high-dimensional embedding space, and their similarities can be represented as the distances between their respective embeddings. Recent advancements in Large Language Models (LLMs) demonstrate powerful reasoning capabilities while still presenting problems such as hallucination; that is, LLMs can generate fake answers that may look plausible. To solve this problem, it is natural to connect LLMs with external vector databases [1] storing the embeddings of “facts”. In this setting, LLMs will act as “brain” and vector databases will be long-term memory for LLM to search on. For instance, imagine we need to build a Q&A system for research papers. The system can first store the embedding of each paper on a vector database. To answer questions such as “List recent papers on k nearest neighbor search together with the link of each paper”, instead of generating fake papers or links, the system can map the query into the same vector space of stored papers and search k nearest neighbors of the query among the paper embeddings. The corresponding texts of the k nearest neighbors will be potentially appropriate answers to the question. Recently, application scenarios as above have aroused new interest in developing more efficient KNN algorithms, which will be the core component of vector databases.

However, the rapid growth of dataset sizes and the “curse of dimensionality” often make it infeasible to locate the exact nearest neighbors without performing a comprehensive scan of the data, prompting the adoption of approximate nearest neighbor (ANN) algorithms, which aim to significantly improve efficiency while slightly relaxing accuracy constraints. These algorithms aim to optimize recall k-recall@k (the proportion of actual k nearest neighbors that are successfully retrieved) while reducing the time taken to search, thereby making a trade-off between recall and latency. Existing ANN algorithms can be categorized into four types: hashing-based (such as Locality Sensitive Hashing [2], [3]), tree-based (such as k-d trees [4]), quantization-based [5], [6], and graph-based [7], [8], each with unique index construction techniques and trade-offs. Recently, graph-based algorithms have become a highly effective solution for ANN due to their exceptional ability to express neighboring relationships, which allows them to evaluate fewer points in the dataset to achieve more accurate results.

Graph-based ANN algorithms construct a graph index on the original dataset, where vertices in the graph correspond to points in the dataset, and neighboring vertices are connected by an edge. Given this graph index and a query point, ANN aims to find a set of vertices close to the query. The process involves selecting a (random or predefined) seed vertex as the starting point(s) and keeping a set of points as the search front. Then, the algorithm iteratively updates the search front by replacing closer neighbors until a termination condition is met. The final set of points is the nearest neighbors to the query. Compared to other indexing structures, graph-based algorithms offer a superior trade-off between accuracy and efficiency, which is why they are widely adopted by various vector databases or ANN frameworks.

One problem with current graph-based ANN index structures is their memory footprint is so large that it is prohibitive to build and store an in-memory graph index for large datasets such as those containing more than 1 billion data points. Though there are existing methods that cluster or divide the whole dataset into several chunks, build index separately for each chunk, and host each index structure on a separate node, the scalability of these methods is usually not as expected, not to mention the complicated synchronization issue between memory and disk when the graph index keeps updated due to data insertion or deletion. Thus, it would be favorable if the index can be stored purely on disk. There are existing methods, such as DiskANN [9], which try to store graph-based index on disk at the expense of still keeping a compressed version of the dataset (for example, by using the PQ-based product quantization) in memory to reduce disk I/O and speed up distance calculation. Nevertheless, the compressed version of the dataset still incurs a large memory footprint.

The key problem with existing graph-based ANN algorithms is that each graph node contains incomplete information for making routing decisions and the vectors of all its neighbors need to be loaded into memory, which incurs a huge amount of I/Os during searching. This motivates us to introduce LM-DiskANN, in which each node keeps complete information about all its neighbors. That is, for each node, we store a copy of the PQ-compressed [5] version of all its neighbors, immediately following the original vector of the node itself. During searching, the distance between the query point and all the neighbors can be estimated by calculating the distances between the PQ-compressed query vector and the neighbors. Since the PQ-compressed vectors of neighbors can be loaded into memory at the same time as the vector of the node, we avoid conducting several random disk accesses. The contributions of this paper can be summarized as follows:

- We propose LM-DiskANN: an innovative disk-native graph-based ANN index in which each node stores complete neighboring information for routing. Total I/O times during searching will be similar to DiskANN [9] but with a much lower memory footprint.
- LM-DiskANN incrementally builds graph index and supports insertion and deletion dynamically.
- Extensive experiments are conducted to demonstrate that LM-DiskANN outperforms SOTA graph-based ANN indexes in terms of memory footprint while keeping a similar recall-latency curve.

The rest of the paper is organized as follows: Section II surveys existing work relevant to our research. Section III introduces basic notations and exiting work which our method is based on. Section IV presents the main idea of our methodology. In Section V, we provide the thorough experimental results demonstrating the comparison between our method and other existing methods. Finally, in Section VI, we conclude the paper by discussing the contributions and suggesting future directions.

II. Related Work

A. Approximate Nearest Neighbor Search

The problem of Approximate Nearest Neighbor (ANN) search is fundamental in various domains, including computer vision, machine learning, and computational geometry. The goal of ANN search is to find data points in a dataset that are close to a query point without necessarily finding the exact nearest neighbor. Over the years, several methods and techniques have been proposed to address this problem efficiently.

1) Tree-based Methods: Tree-based structures, such as R-trees [10], KD-trees [11], M-Tree [12] and Ball trees [13], have been among the earliest techniques for ANN search. While KD-trees are efficient for low-dimensional data, their performance degrades in higher dimensions. Ball trees, on the other hand, partition data into hyper-spherical regions and can handle higher dimensions more gracefully.

2) Hashing Techniques: Locality-sensitive hashing (LSH) [14] is a popular hashing technique for ANN search, where LSH involves hashing input items in a way that increases the likelihood of similar items being mapped to the same buckets. Variants of LSH, including multi-probe LSH [15] and cross-polytope LSH [16], have been proposed to improve search efficiency and accuracy. A comprehensive study [17] revisited various hashing algorithms for ANNS. Surprisingly, the study found that the random-projection-based LSH outperformed other state-of-the-art hashing methods, contrary to claims made in many papers.

3) Quantization Methods: Quantization-based methods, such as Product Quantization [5] and Optimized Product Quantization [18], aim to compress the data vectors into compact codes. These methods allow for efficient storage and fast distance computation between the query and the compressed data.

4) Graph-based Methods: Graph-based methods [7]–[9], [19], [20] construct a graph where nodes represent data points and edges connect nearby points. These methods offer state-of-the-art performance in terms of search accuracy and speed. Recently, there have been ML-based approaches that optimize for building or searching on graph-based indexes, for example, by learning node representation to provide better routing [21], building decision tree models to learn and predict when to stop searching [22], and learning vector embeddings in lower dimensional space to preserve local geometry [23]. Wang et al. have conducted a comprehensive survey and comparison of the recent graph-based ANN algorithms [24].

5) Hybrid Methods: Some recent works have combined multiple techniques to achieve better performance. For instance, Douze et al. [25] proposed a method that combines quantization with graph-based search, while Dong et al. [26] integrated LSH with neural network embeddings.

B. Learned Database Index

Several works have explored the use of machine learning techniques to improve the performance of databases, with a particular focus on indexing structures. The Case for Learned Index Structures [27] argues that learned indexes can outperform traditional indexing structures by learning the underlying distribution of the data and adaptively optimizing the index structure. The ML-Index [28] and Learned Index for Spatial Queries [29] both propose learned indexing structures that improve the efficiency of range and spatial queries, respectively. Shift-Table [30] proposes a learned indexing structure for range queries using model correction to reduce the latency of the indexing process. ALEX [31] proposes an updatable adaptive learned index that can be updated in real-time, while Updatable Learned Index with Precise Positions [32] proposes an indexing structure that can maintain precise positions of the indexed elements. AI Meets Database [33] and Learning a Partitioning Advisor for Cloud Databases [34] both explore the use of machine learning to optimize the performance of cloud databases, while An Index Advisor Using Deep Reinforcement Learning [35] proposes an index advisor that can recommend the best index structures based on the workload and data distribution.

To handle complex queries over high-dimensional data, several works have proposed multi-dimensional indexing structures. Learning Multi-dimensional Indexes [36] provides a comprehensive overview of various approaches to designing learned indexes for multi-dimensional data, including partitioning and adaptive indexing.

Finally, several works propose specific indexing structures exploiting correlations between dimensions (columns). Correlation [37] provides a compressed access method for exploiting soft functional dependencies in column correlations, while Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations [38] proposes a machine learning approach to exploiting column correlations to improve secondary indexing. LISA [39] proposes a learned index structure for spatial data that also exploits spatial correlations to optimize the indexing structure.

Overall, the research on machine learning for database indexes has shown great promise in improving the performance and efficiency of database management systems, and the specific approaches proposed in these papers provide useful insights and techniques for designing more effective indexing structures and optimizing database performance.

III. Background

In this section, we will introduce the notations of the problem and existing work on which our work is based.

A. Notations

For the convenience of our discussion in the following sections, we first introduce the notations as in Table I.

TABLE I: Notations of Symbols.

Symbols	**Meaning**
P	Vector dataset
G	Graph index $G = (P, E)$ in which E is the set of connections
R	Degree of the graph G
p	Index node $p \in P$
p^*	Nearest neighbour
$d(,)$	Distance metric on the vector space
$N_{out}(p)$	Outbound neighbours of node p
q	Query target
$\mathcal{V}$	Set of visited node during search
$\mathcal{L}$	Set of candidate set during search
L	Size limit of $\mathcal{L}$
α	Distance threshold in Generalized Sparse Neighborhood Graph

B. Best First Search (BFS)

Almost all graph-based ANNS algorithms adopt certain types of greedy routing strategies, including best first search (BFS) and its variants [24]. During BFS, a starting node s is first seeded, and then the algorithms will iteratively approach the target vector q in which the nearest node to q at each step is visited, and all its neighbors are added to a search front. In Figure 1, we differentiate different sets of nodes by colors. The green node is the starting node s, and the red node is the query target q. The black nodes are the ones that used to be in the search front, and the blue nodes are those visited. Note all the blue nodes also used to be in the search front. The nodes in the red dotted circle are the 4 nearest nodes in the search front when the search stops. Once a blue node is visited, all its neighbors are added to the search front. Since the size of the search front has a pre-defined bound, BFS will converge once all the nodes in the search front have been visited. It turns out that BFS can successfully balance accuracy and efficiency when the graph index is built properly.

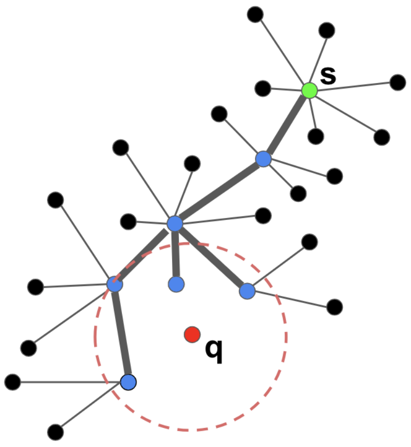


Fig. 1: Visualization of a search path.

C. Generalized Sparse Neighborhood Graph (GSNG)

In the early days of ANN research, researchers wanted to find appropriate sparse graphs while keeping enough connectivity to navigate using greedy search paradigms such as BFS. One of the suitable properties [40], [41] is the Sparse Neighborhood Graph(SNG) in which a connection exists if and only if it is not the longest edge in any triangles. Or more rigorously, for any index node p that has a neighbor p', there will be no edge between p and p'' if $d(p', p^{''}) \leq d(p, p^{''})$ for any other node $p^{''}$. Essentially, this property reduces the number of edges as much as possible, but each node still connects to its nearest neighbors. The conclusion follows that BFS can always converge to the nearest neighbors on SNG. While this property is promising in many cases, the lack of long-range connections makes this type of graph not quite efficient for navigation.

Hence, researchers have proposed the Generalized Sparse Neighborhood Graph (GSNG), in which long-range connections are preserved to some extent, and a trade-off between sparsity and navigability is achieved. GSNG generalizes SNG by stating that for any index node p which has a neighbor p', there will be no edge between p and p'' if $\alpha \times d(p', p^{''}) \leq d(p, p^{''})$, in which $\alpha \geq 1$. If α is larger, more long-range connections will be preserved. It turns out GSNG is a promising property that increases the efficiency of BFS on index graphs.

IV. LM-DiskANN

In this section, we will introduce each of the core algorithms of our method. Our method is based on DiskANN [9] and FreshDiskANN [20]. DiskANN is based on a new graph-based indexing algorithm called Vamana, a static graph index in which the index is built at a time and can not be changed further in the future. FreshDiskANN is built on DiskANN, which dynamically modifies the index as new nodes come in or existing nodes get removed. FreshDiskANN spends much effort to reduce memory footprint and synchronize in-memory data structures and on-disk index. Since our method has a low memory footprint and is mainly on disk, and since our research purpose is to test the impact of removing all the in-memory vectors, we only keep the essential part of FreshDiskANN. Thus, our method can be considered a dynamic version of the DiskANN, and the methods of search, insert, and delete need to be modified accordingly.

A. *Layout of Augmented Index Node on Disk*

For each node, we store the compressed vectors of all the neighbors immediately following the neighbors' IDs. The layout of each node is illustrated in Figure 2. Each node starts with its ID, then its vector, and then all the neighbor IDs. If the number of neighbors is less than the maximum number of neighbors, the remaining space is padded with 0s. Each node ends with the compressed vectors of neighbors, also followed by padding if the number of neighbors is less than the maximum number of neighbors. For instance, we use the dataset SIFT1M for evaluation. SIFT1M has 128 for each data point, and our experiment sets the maximum number of neighbors equal to 70, so the size of each index node for SIFT1M is $(1 + 128 + 70 + 70 \times 8) \times 4 = 3036$ bytes. Thus, each node can be fit into a single 4KB page, and it is not more

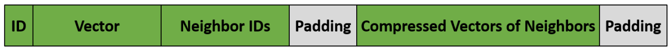


Fig. 2: Layout of our augmented index node on disk.

expensive to load our augmented node than the original node. This is the fundamental reason why our method works.

B. *LM-Search*

Search is the core functionality of an ANN index. Vamana uses the original best-first search algorithm, and since our method is built upon Vamana, we also use the original BFS. The only difference is that during a search, our method retrieves the neighborhood information from the node itself rather than from a separate in-memory array of vectors.

The search function accepts the graph index G, query vector q, maximum candidate set size L, and result size k. To start each search, our method randomly samples a node from the index as the starting node and initializes the candidate list $\mathcal{L}$ as containing the starting node s. The algorithm also keeps a set of visited nodes $\mathcal{V}$. A distance map D is also initialized with $\mathcal{D}(s) = d(s, q)$, in which $d(s, q)$ is the distance from s to q. In each iteration of the search algorithm, the current nearest unvisited node p^* from the query vector q is retrieved from the candidate list, and all its neighbors are inserted into the candidate list. If the size of the resulted neighbor list is larger than L, the candidate set is trimmed to keep only the L closest neighbors to the query vector q.

Since our method uses an augmented node containing the complete neighboring information as illustrated in Figure 2, the distance map $\mathcal{D}$ is calculated based on the compressed vectors unless the node is visited and its distance from the query is updated. The efficiency of our method is not much different from the original implementation. When there is no more unvisited node in the candidate list, the search process converges and returns the closet k nodes in the candidate set, together with all the visited nodes. The latter is returned because it will be used incrementally when building the graph index. It is easy to see that the convergence efficiency of the search depends heavily on the value of L: the larger L is, the more time the search process will take more steps to converge. Also, convergence efficiency depends on the scale of the graph G, if there are more nodes in the graph, it will take longer to converge. We will also prove this in our experiments. Algorithm 1 illustrates the detailed pseudocode.

C. *LM-Insert*

Our index supports dynamic updates by providing LM-Insert. The insert method is also used to build the graph index from scratch. LM-Insert accepts the graph index G and the node to insert p. After the method returns, the graph index will contain a new node p and also satisfy the GSNG property. The method first searches the nearest neighbor of p in the graph index G and gets the set of visited nodes $\mathcal{L}$. Then, all the nodes in $\mathcal{L}$ are inserted into the neighbors of p, followed by a pruning process. Reversely, p is also inserted in the neighbor list of

Algorithm 1: LM-Search(G,q,L,k)

Input: Graph index G, query vector q, maximum candidate list size L, result size k
Output: k nearest neighbours of query data point q

```
1:  begin
2:      $s \leftarrow randomly\ select\ from\ index\ nodes$
3:      $\mathcal{V} \leftarrow \emptyset$
4:      $\mathcal{L} \leftarrow \{s\}$
5:      $\mathcal{D} = \{s : d(s, q)\}$
6:      while $\mathcal{L} \backslash \mathcal{V} \neq \emptyset$ do
7:          $p^* \leftarrow \arg\min_{p \in \mathcal{L} \backslash \mathcal{V}} \mathcal{D}(p)$
8:          $\mathcal{L} \leftarrow \mathcal{L}.insert(N_{out}(p^*))$
9:          foreach neighbor $n$ in $N_{out}(p^*)$ do
                /* Estimate the distance from
                   each neighbour $n$ to $q$ by
                   using its compressed vector
                   from $p^*$                        */
10:             $\mathcal{D}.insert(n : d(p^*.n, q))$
11:         end
12:         $\mathcal{V} \leftarrow \mathcal{V}.insert(p^*)$
            /* Now update the distance from
               $p^*$ to $q$ by its original vector
                                                     */
13:         $\mathcal{D}(p^*) = d(p^*, q)$
14:         if $|\mathcal{L}| > L$ then
15:             $\mathcal{L} \leftarrow L\ cloest\ nodes\ in\ \mathcal{L}$
16:         end
17:     end
18:     return $k\ cloest\ nodes\ in\ \mathcal{L}$ ; $\mathcal{V}$
19: end
```

each of its neighbors, followed by a pruning process. Adding connection in both directions and pruning will guarantee the updated graph G's connectivity and its GSNG property.

If LM-Insert is used to build the graph index from scratch, we can expect the neighbors of exiting nodes to be continuously modified as a new node comes in. Since LM-Insert relies on LM-Search, we can also expect its latency will increase steadily as the graph index grows. Algorithm 2 shows the detailed pseudocode of LM-Insert.

D. LM-Delete

To support dynamic updates, we also need to support deleting nodes. The method LM-Delete accepts graph index G and the node to delete p as augments. The result of running this procedure will be a graph index G without p, which still satisfies GSNG property while keeping connectivity as much as possible. The method first marks node p as deleted and then tries its best to fix the broken paths resulting from the deletion. For each of the original neighbors of p, all the other neighbors will be connected to it, followed by a pruning process. Algorithm 3 details the pseudocode of LM-Delete.

Algorithm 2: LM-Insert(G,p,L)

Input: Graph index G, node to insert p
Output: A new instance of the graph index G containing newly inserted node p while satisfying GSNG property.

```
1:  begin
        /* Search $p$ in $G$, get nearest
           neighbour $p^*$ and all the visited
           nodes during search: $\mathcal{V}$. We also
           assume graph index $G$ has
           defined its candidate set size $L$
           */
2:      $[p^*; \mathcal{V}] \leftarrow$ LM-Search$(G, p, G.L, 1)$
3:      $N_{out}(p) \leftarrow N_{out}(p) \cup \mathcal{V}$
4:      LM-Prune$(G, p, G.\alpha, G.R)$
5:      foreach Node $n$ in $N_{out}(p)$ do
6:          $N_{out}(n) \leftarrow N_{out}(n) \cup p$
7:          LM-Prune$(G, n, G.\alpha, G.R)$
8:      end
9:  end
```

Algorithm 3: LM-Delete(G,p)

Input: Graph index G, node to delete p
Output: Graph index G with p removed while still satisfying GSNG property.

```
1:  begin
2:      Mark $p$ as deleted
        /* Fix broken paths while keeping
           GSNG property.                     */
3:      foreach Neighbour $n$ in $N_{out}(p)$ do
4:          $N_{out}(n) \leftarrow N_{out}(n) \cup N_{out}(p) \backslash n$
5:          LM-Prune$(G, n, G.\alpha, G.R)$
6:      end
7:  end
```

E. LM-Prune

In Section IV-C, the insert method calls the function of neighbor prune. When the number of neighbors exceeds the predefined maximum number of neighbors, a neighbor pruning process will be triggered automatically. This process is designed to guarantee that the index graph satisfies the generalized sparse neighborhood graph(GSNG) property, which keeps a balance between graph sparsity and connectivity.

The method accepts the index graph G, a node p, the neighbors of which we want to prune, a distance threshold α, and the maximum number of neighbors R in the graph index. We first initialize the candidate set $\mathcal{C}$ with the current neighbors of p, that is $N_{out}(p)$, and clear all the current neighbors. The pruning process runs in an iterative fashion, in which each step finds the nearest node p^* in the candidate set $\mathcal{C}$, adds it to the neighbors of p, and then removes all the nodes $p\prime$ in $N_{out}(p)$, which satisfies $\alpha \times d(p^*, p\prime) \leq d(p, p\prime)$. Our method

Algorithm 4: LM-Prune(G,p,α,R)

Input: Graph index G, node p, candidate set $\mathcal{C}$, distance threshold α and maximum node degree R

Output: p has new neighbours which makes G satisfies the GSNG property

```
1:  begin
2:      $\mathcal{C} \leftarrow \mathcal{C} \cup N_{out}(p) \backslash \{p\}$
3:      $N_{out}(p) \leftarrow \emptyset$
4:      while $\mathcal{C} \neq \emptyset$ do
            /* Calculate the distance from $p'$ to $p$ by using the compressed vector in augmented node $p$ */
5:          $p^* \leftarrow \arg\min_{p' \in \mathcal{L} \backslash \mathcal{V}} d(p.p', p)$
6:          $N_{out}(p) \leftarrow N_{out}(p).insert(p^*)$
7:          if $|N_{out}|(p)| \geq R$ then
8:              break
9:          end
10:         foreach $p'$ in $\mathcal{C}$ do
                /* Calculate the distance from $p'$ to $p^*$ by using the compressed vector in augmented node $p$ */
11:             if $\alpha * d(p.p^*, p') \leq d(p, p.p')$ then
12:                 remove $p'$ from $\mathcal{C}$
13:             end
14:         end
15:     end
16: end
```

of neighbor prune is different from the implementation of Vamana in that the neighbors are stored in augmented node p, and we do not need in-memory auxiliary arrays to calculate the distance from the candidate nodes and p and also the distance between each pair of the candidate nodes. Again, the efficiency of our method is not much different from the original implementation. Algorithm 4 provides the detailed pseudocode.

F. Search Efficiency and I/O

As we see in Section IV-B, given a starting node s and a query point q, when we call LM-Search(), a set of nodes is visited, and an extra set of nodes is added into the candidate set $\mathcal{L}$ (or search front). Recall in Figure 1, all its neighbors are added to the search front when a blue node is visited. Also, their distances to the query p are needed for determining the next visited node. Thus, when Vamana implements Best First Search(BFS) on a disk-based index, essentially all the black and blue nodes need to be loaded into memory, which incurs a significant amount of random I/Os. Vamana attempts to mitigate this problem by storing a compressed copy of all the vectors in memory, and then it only needs to load blue nodes. Our method addresses this problem in an alternative way where, instead of keeping in-memory arrays of vectors, we augment each on-disk node by appending the compressed vectors of neighbors for each node right after the information of the node itself. When a blue node is visited and loaded into memory, all its neighboring information is loaded together within one I/O. Therefore, the search efficiency is not sacrificed at the expense of more disk consumption.

TABLE II: Datasets used in Experiments.

Dataset	Dimensions	Train Size	Test Size	Neighbors*	Distance
SIFT1M [5]	128	1,000,000	10,000	100	Euclidean
GIST1M [5]	960	1,000,000	10,000	100	Euclidean
DEEP1M [42]	96	1,000,000	10,000	100	Angular

*Pre-calculated KNN in Ground Truth.

V. Experiments

In this section, we compare our proposed method with the original DiskANN, which is the SOTA disk-native ANN algorithm. We compare the two algorithms in terms of recall-latency curve and memory consumption on different datasets.

A. Experiment Configurations and Dataset

We use Google Cloud E2 virtual instance with 8 vcpu (4 cores), 32GB RAM and 1TB SSD as the machine setups for our experiments.

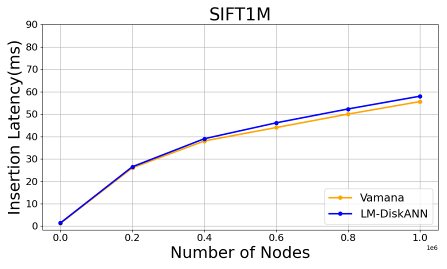
Throughout our evaluations, we use public benchmark datasets as shown in Table II. The attributes for each dataset are also listed in the table. Each dataset is divided into two partitions: a training set and a test set, in which the training set is used to build the graph index and the test set for queries. Each training set contains 1 million data points. For the test set, each dataset contains 10 thousand data points and provides pre-calculated results for the exact K Nearest Neighbor Search up to K=100.

For a fair comparison, we implement our own version of Vamana in Python by strictly following the algorithms as in [9]. Since the original Vamana is essentially a static index that is built once and can not modified afterward, we adapt it to a dynamic index [20]. Thus, in this way, we can compare it with our method in a more convenient way. For both Vamana and our method and for all the datasets, we use the optimal parameter settings: $R = 70$, $L = 75$, $\alpha = 1.2$, and $W = 2$ to build the index. For the Vanama implementation, we build a PQ 32-byte codebook [5] using 10,000 points in the test set, given the data distribution of the test set is similar to that of the train set. Thus, the codebook can be used interchangeably between the two.

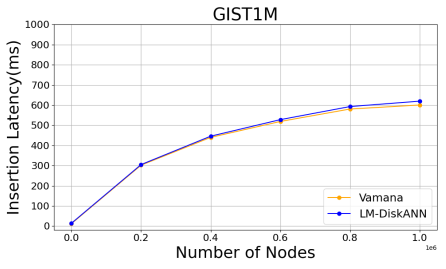
After 1 million data points from the train set are inserted, we query 10 thousand data points from the test set, and then we delete these 10 thousand points. We measure latency, accuracy, and space consumption throughout the whole process. We present the results in the following subsections.

B. Index Building Time

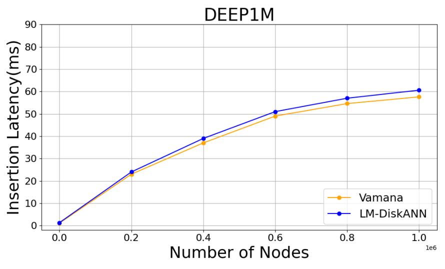
We first measure the index-building time after 1 million data points from the train set are inserted into the index. From Table



(a) Incremental building time for SIFT1M

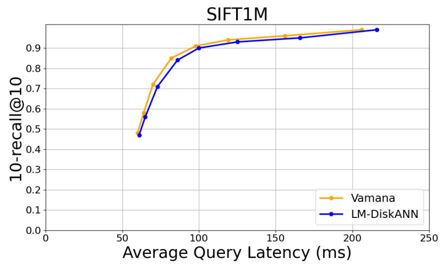


(b) Incremental building time for GIST1M

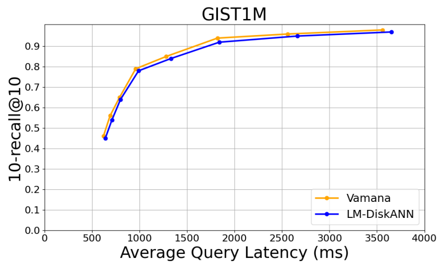


(c) Incremental building time for DEEP1M

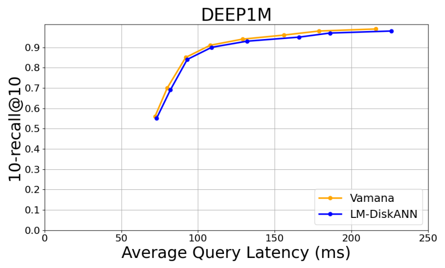
Fig. 3: Building time per node as the function of inserted node numbers.



(a) Latency vs. recall curve for SIFT1M

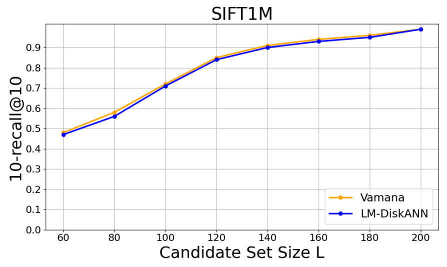


(b) Latency vs. recall curve for GIST1M

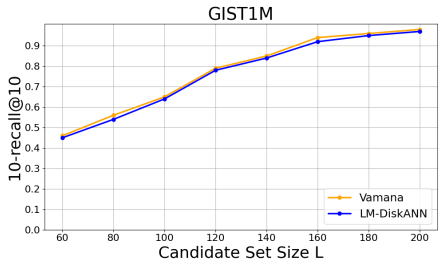


(c) Latency vs. recall curve for DEEP1M

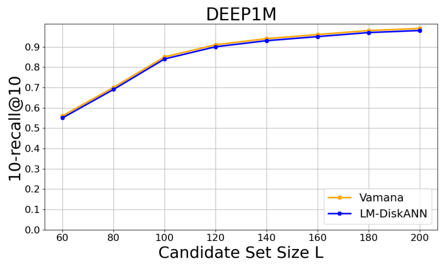
Fig. 4: Latency vs. recall comparison for all three datasets.



(a) Recall vs. L for SIFT1M



(b) Recall vs. L for GIST1M



(c) Recall vs. L for DEEP1M

Fig. 5: Recall for different candidate size L.

TABLE III: Index Building Time.

Algorithm	SIFT1M	GIST1M	DEEP1M
Vamana	2329s	29743s	2078s
LM-DiskANN	2467s	31951s	2182s

*All the statistics correspond to peak index sizes on disk.

TABLE IV: Deletion Latency.

Algorithm	SIFT1M	GIST1M	DEEP1M
Vamana	47ms	56ms	50ms
LM-DiskANN	45ms	58ms	44ms

III, we can see our implementation of Vamana takes more time than reported in the original paper, for we use Python instead of C++ as in the original implementation, and we use incremental building process rather than batch building. We can also see the index building time of our method is slightly longer than that of Vamana because our method needs to store all the neighboring information on disk when a new node is inserted. GIST1M consumes much more construction time than the other two datasets for both Vamana and our method. This is because the data distribution of GIST1M is more complex and takes more steps to converge than the other two datasets.

We sample the node insertion time every 200,000 nodes when building the index. The sampled data is plotted in Figure 3. Our method takes slightly more time to insert each node than Vamana, and the insertion time increases when the number of inserted nodes grows. This is not surprising because when building the index incrementally, it takes more time to search the appropriate neighbors for the newly inserted node as the index grows bigger. GIST1M takes way more time than the other two at all stages, roughly 1 magnitude longer.

TABLE V: Memory Consumption.

Algorithm	SIFT1M			GIST1M			DEEP1M		
	Temporary	Permanent	Total	Temporary	Permanent	Total	Temporary	Permanent	Total
Vamana	59K	16000K	16059K	309K	16000K	16309K	49K	16000K	16049K
LM-DiskANN	227K	4040K	**4267K**	477K	4040K	**4517K**	217K	4040K	**4167K**

*All the statistics are peak memory consumption.

C. Recall-Latency Curve

After all the 1 million vectors from the train set are inserted into the index, we query the 10 thousand vectors from the test set one by one and record the average query latency and 10-recall@10. We repeat this process for different L, that is, adjust the candidate set size for searching and check the impact of searching complexity (represented by L) on the query latency and accuracy. The results for all three datasets are plotted in Figure 4. We can see that, for all three datasets, our method performs almost the same as Vamana. This is also not surprising because we adopt a similar algorithm as Vamana. Our method performs slightly worse because, during the search, we need to load complete routing information for each node, which incurs slightly larger I/O overhead. But our method achieves a similar latency vs. recall curve with much less memory footprint.

Figure 5 illustrates the impact of different choices of candidate set size L on 10-recall@10. Starting from $L = 60$, we sample different values of L at the step of 20 until $L = 200$ is reached. In essence, when L increases, we increase the chances the correct nearest neighbors will be captured and thus increase the time steps of query convergence. We can see when $L = 60$, the 10-recall@10 on all the datasets is around 0.5. Remember that we use $L = 70$ for index building. Thus, this means if we continue to use the relatively small value of L, there will be a quality degeneration when the index continues expanding. Therefore, it is expected we use a larger value of L as the index grows. Further research is required on how L should be increased as the function of the volume of the index.

D. Deletion Latency

As our method supports dynamically updating the index, after 10K test vectors are queried on the index, we further insert 10K vectors from the test set and delete them one by one. The average deletion latency is recorded in Table IV. We can see the deletion latency does not vary much among different datasets. This is because, unlike insertion, which needs to search on the index first, deletion only needs to stitch the hole made from the removed node. The deletion on GIST1M is slightly larger, this is probably due to the more dense connectivity of this dataset. Thus, more nodes have the maximum number of neighbors, and the deletion has more neighbors to fix.

E. Memory Consumption

We measure and compare memory consumption for Vamana and our method on all the datasets. When discussing memory consumption, we often refer to the size of heap usage in the memory. In the context of ANN, heap usage can be further divided into two components: temporary heap usage and permanent heap usage. The former is the heap consumed in the course of searching, inserting, or deletion, the majority of which is determined by the candidate set size in searching a query. The latter is the size of the consistently occupied heap throughout the manipulation of the graph index, which is dominated by the size of auxiliary data structures such as arrays of PQ-compressed vectors for Vamana and the set of IDs of all the nodes for our method. Table V lists the peak usage of the temporary heap and the permanent heap and also calculates the total peak usage of the two parts. The "peak" case usually happens when the index is fully loaded with all the data from the train and test sets. From the table, we can see for all three datasets, Vamana consumes less temporary heap than our method, but in turn, consumes a much more permanent heap. Therefore, our LM-DiskANN method consumes only roughly one-quarter of the memory usage of Vamana in total. This is because our method keeps the complete neighboring information for each node and removes the need to keep a separate compressed version of all the vectors in memory.

F. Index Size on Disk

For a fair comparison, we also calculate the peak index size on disk, which happens when both the train and test sets are stored in the index. Table VI lists the disk usage for both methods and the three datasets. We can see our method requires more disk space than Vamana. This is because we need to store the complete routing information for each node, which brings more redundancy. For datasets with higher dimensions, our method requires less extra disk space than those with lower dimensions. For GIST1M, we only need 50% more disk space than Vamana, but for the other two, we need 300% more disk space. In essence, our method trades disk consumption for memory footprint, which may become a wise choice when the dataset grows large.

TABLE VI: Index Size on Disk.

Algorithm	SIFT1M	GIST1M	DEEP1M
Vamana	799M	4161M	670M
LM-DiskANN	3030M	6423M	2933M

*All the statistics are peak index sizes on disk.

VI. Conclusion

In this paper, we have addressed the challenges associated with Approximate Nearest Neighbor (ANN) search. The primary challenge we identified is the memory-intensive nature of existing graph-based ANN index structures, which becomes prohibitive for large datasets. Our solution, LM-DiskANN, offers a novel approach to this problem by designing a disk-native graph-based ANN index that stores complete neighboring information for routing at each node, thereby significantly reducing the memory footprint.

Our approach diverges from existing methods by eliminating the need for multiple random disk accesses during the search process. Instead, LM-DiskANN ensures that when one data point is loaded into the main memory, its neighbors are likely loaded together in one block I/O. This innovative design not only optimizes the search process but also supports dynamic insertion and deletion, making it adaptable to evolving datasets.

Through rigorous experimentation, we have demonstrated that LM-DiskANN stands out in terms of memory efficiency while maintaining a competitive recall-latency curve, comparable to state-of-the-art graph-based ANN indexes. This balance between efficiency and performance underscores the potential of LM-DiskANN as a promising solution for future big data applications, especially in scenarios that involve integration with LLMs.

As the field of algorithm research continues to evolve, and as datasets grow in size and complexity, the need for efficient and scalable solutions like LM-DiskANN becomes ever more critical. We believe that our contributions in this paper pave the way for further innovations in the domain of on-disk ANN search.

Acknowledgment

This research has been sponsored in part by the National Science Foundation grant IIS-1652846 and the Nebraska Research Initiative grant.

References

[1] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu *et al.*, "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2614–2627.

[2] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of the ACM*, vol. 51, no. 1, pp. 117–122, 2008.

[3] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 1–12, 2015.

[4] C. Silpa-Anan and R. Hartley, "Optimised kd-trees for fast image descriptor matching," in *2008 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2008, pp. 1–8.

[5] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[6] Z. Pan, L. Wang, Y. Wang, and Y. Liu, "Product quantization with dual codebooks for approximate nearest neighbor search," *Neurocomputing*, vol. 401, pp. 59–68, 2020.

[7] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[8] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *arXiv preprint arXiv:1707.00143*, 2017.

[9] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, 1990, pp. 322–331.

[11] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[12] P. Ciaccia, M. Patella, P. Zezula *et al.*, "M-tree: An efficient access method for similarity search in metric spaces," in *Vldb*, vol. 97. Citeseer, 1997, pp. 426–435.

[13] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[14] 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*, 1998, pp. 604–613.

[15] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 950–961.

[16] K. Terasawa and Y. Tanaka, "Spherical lsh for approximate nearest neighbor search on unit hypersphere," in *Algorithms and Data Structures: 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007. Proceedings 10*. Springer, 2007, pp. 27–38.

[17] D. Cai, "A revisit of hashing algorithms for approximate nearest neighbor search," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 6, pp. 2337–2348, 2019.

[18] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013, pp. 2946–2953.

[19] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph," in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[20] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri, "Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search," *arXiv preprint arXiv:2105.09613*, 2021.

[21] D. Baranchuk, D. Persiyanov, A. Sinitsin, and A. Babenko, "Learning to route in similarity graphs," in *International Conference on Machine Learning*. PMLR, 2019, pp. 475–484.

[22] C. Li, M. Zhang, D. G. Andersen, and Y. He, "Improving approximate nearest neighbor search through learned adaptive early termination," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2539–2554.

[23] L. Prokhorenkova and A. Shekhovtsov, "Graph-based nearest neighbor search: From practice to theory," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7803–7813.

[24] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *arXiv preprint arXiv:2101.12631*, 2021.

[25] M. Douze, A. Sablayrolles, and H. Jégou, "Link and code: Fast indexing with graphs and compact regression codes," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 3646–3654.

[26] Y. Dong, P. Indyk, I. Razenshteyn, and T. Wagner, "Learning space partitions for nearest neighbor search," *arXiv preprint arXiv:1901.08544*, 2019.

[27] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 489–504.

[28] A. Davitkova, E. Milchevski, and S. Michel, "The ml-index: A multidimensional, learned index for point, range, and nearest-neighbor queries." in *EDBT*, 2020, pp. 407–410.

[29] H. Wang, X. Fu, J. Xu, and H. Lu, "Learned index for spatial queries," in *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 2019, pp. 569–574.

[30] A. Hadian and T. Heinis, "Shift-table: A low-latency learned index for range queries using model correction," *arXiv preprint arXiv:2101.10457*, 2021.

[31] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, "Alex: an updatable adaptive learned index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 969–984.

[32] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *arXiv preprint arXiv:2104.05520*, 2021.

[33] G. Li, X. Zhou, and L. Cao, "Ai meets database: Ai4db and db4ai," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2859–2866.

[34] B. Hilprecht, C. Binnig, and U. Röhm, "Learning a partitioning advisor for cloud databases," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 143–157.

[35] H. Lan, Z. Bao, and Y. Peng, "An index advisor using deep reinforcement learning," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020, pp. 2105–2108.

[36] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 985–1000.

[37] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik, "Correlation maps: A compressed access method for exploiting soft functional dependencies," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1222–1233, 2009.

[38] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber, "Designing succinct secondary indexing mechanism by exploiting column correlations," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1223–1240.

[39] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "Lisa: A learned index structure for spatial data," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 2119–2133.

[40] S. Arya and D. M. Mount, "Approximate nearest neighbor queries in fixed dimensions." in *SODA*, vol. 93. Citeseer, 1993, pp. 271–280.

[41] J. W. Jaromczyk and G. T. Toussaint, "Relative neighborhood graphs and their relatives," *Proceedings of the IEEE*, vol. 80, no. 9, pp. 1502–1517, 1992.

[42] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2055–2063.