ScaDANN: A Scalable Disk-Based Graph Indexing Method for ANN

Hyein Gu[1], Min-soo Kim[1,*]

[1]Korea Advanced Institute of Science and Technology, Republic of Korea

Abstract

Approximate Nearest Neighbor (ANN) indexing constitutes a fundamental component of modern vector-based databases, facilitating efficient and accurate information retrieval for applications such as retrieval-augmented generation. However, scaling graph-based ANN indices to billion-scale datasets poses substantial challenges, including high memory demands and inefficiencies in handling partitioned graphs. To address these issues, we propose ScaDANN, a scalable disk-based graph indexing method tailored for large-scale datasets under limited memory conditions. ScaDANN introduces two novel techniques: overlapping block-level insertion and grid block merge, which enable the efficient construction of unified graph indices while preserving high search performance. Our approach achieves notable advancements in index construction speed, search accuracy, and memory efficiency, establishing ScaDANN as a robust and effective solution for scalable ANN indexing in resource-limited environments.

Keywords

Graph, ANN, Nearest neighbor search, Vector search

1. Introduction

Large Language Models (LLMs) have revolutionized natural language processing by excelling in tasks such as text generation [1], summarization [2], and question answering [3]. Their ability to leverage pre-trained knowledge has facilitated applications across diverse domains, including healthcare [4], education [5], and customer service [6]. However, despite their strengths, LLMs face critical limitations, such as hallucination [7], where they generate plausible but incorrect outputs, outdated knowledge due to static training data, and privacy concerns when processing sensitive information [8].

Retrieval-Augmented Generation (RAG) [9] has been introduced as a promising framework to mitigate these limitations by integrating LLMs with external knowledge retrieval systems. By leveraging vector databases (VectorDBs) [10], RAG retrieves relevant and up-to-date information to enhance the accuracy and reliability of generated responses. This framework not only reduces the impact of hallucination but also addresses privacy concerns by accessing only the information necessary for a given query.

Approximate Nearest Neighbor (ANN) indexing serves as a foundational component in RAG systems, enabling efficient information retrieval from VectorDBs. ANN indexes can be broadly categorized into clustering-based [11, 12, 13], hash-based [14, 15, 16], and graph-based methods [17, 18, 19, 20, 21]. Clustering-based methods partition datasets into smaller clusters, facilitating memory-efficient index construction but often incurring longer construction times and reduced search accuracy. Hash-based methods employ hash functions to compress datasets and reduce dimensionality, which accelerates construction and search processes but at the cost of reduced accuracy due to compression. Graph-based methods, in contrast, represent vectors as nodes and their relationships as edges, connecting nodes based on proximity [22]. These methods construct neighbor lists through vector operations between nodes (as illustrated in Figure 1) and are particularly effective for large-scale datasets, offering superior search accuracy and performance despite higher memory demands during construction.

The need for scalable ANN indexing becomes increasingly apparent as data is segmented into smaller chunks to meet the input constraints of LLMs. For instance, dividing a 100 MB document into 4 KB chunks suitable for LLM input results in approximately 25,600 chunks—a 25,600-fold increase in data chunks relative to the original document count. This exponential growth in the number of chunks significantly increases the size of the ANN index, necessitating the development of methods capable of efficiently handling billion-scale datasets.

Building large-scale ANN indexes, however, poses significant challenges [23, 24]. First, the substantial size of datasets and indexes requires considerable memory resources. Second, achieving competitive search performance on disk-based systems, which are necessary for handling such datasets, is particularly challenging. To address these issues, we propose ScaDANN, a Scalable Disk-based Graph Indexing Method for ANN, specifically designed to overcome these limitations.

This study pursues two primary objectives: (1) to enable the construction of large-scale ANN indexes in memory-constrained environments and (2) to enhance search performance through the use of merged graph structures that improve efficiency and accuracy during the search process. By focusing on these goals, we provide a scalable and efficient solution for constructing and querying ANN indexes in scenarios where both large-scale datasets and memory limitations are critical challenges.

This work makes the following key contributions:

- We propose a graph-based ANN method that leverages grid blocks to efficiently construct and scale billion-scale indexes in memory-constrained environments.
- Our approach outperforms existing disk-based methods, achieving index construction speeds 1.3 to 1.5 times faster and search speeds 1.2 to 1.4 times faster, while maintaining comparable accuracy.
- We demonstrate that our method enables the construction of single indexes up to 10 times larger than those of existing approaches within the same memory constraints.

Published in the Proceedings of the Workshops of the EDBT/ICDT 2025 Joint Conference (March 25-28, 2025), Barcelona, Spain

*Corresponding author.

hyein99@kaist.ac.kr (H. Gu); minsoo.k@kaist.ac.kr (M. Kim)

0009-0009-8582-5805 (H. Gu); 0000-0002-5065-0226 (M. Kim)

These contributions mark a significant advancement in ANN indexing, particularly in terms of efficiency and scalability for large-scale, memory-constrained applications.

2. Preliminaries

2.1. ANN search

Nearest Neighbor (NN) search identifies the data point(s) in a dataset closest to a given query point based on a defined distance metric, such as Euclidean distance or cosine similarity. While exact NN search computes the distance between the query and every data point to determine the nearest neighbor, this approach becomes computationally infeasible for large datasets. ANN search addresses this challenge by finding points that are approximately close to the true nearest neighbors, significantly reducing computational costs. Although ANN sacrifices some accuracy, it delivers faster query response times, which is particularly advantageous in high-dimensional and large-scale datasets.

Definition 1 (ANN). Given a dataset $D = \{x_1, x_2, \ldots, x_n\}$ and a query point q, the task of nearest neighbor search (whether exact or approximate) is to find the point $x_{NN} \in D$ such that the distance $d(q, x_{NN})$ is minimized, where $d(\cdot, \cdot)$ represents a distance metric. In the case of ANN search, the algorithm guarantees that the returned point x^* satisfies:

$$d(q, x^*) \leq (1 + \epsilon) \cdot d(q, x_{NN}),$$

where ϵ is a small approximation factor, and x_{NN} is the exact nearest neighbor of the query point q.

To evaluate ANN algorithms, two primary metrics are used: Recall and Queries Per Second (QPS). Recall quantifies the effectiveness of an algorithm in retrieving true nearest neighbors, defined as the ratio of true nearest neighbors retrieved to the total number of true nearest neighbors. Higher recall indicates greater accuracy but may increase search time. QPS measures the system's throughput, i.e., the number of queries processed per second. Higher QPS reflects greater efficiency, which is critical for large-scale and real-time applications.

2.2. Existing graph ANN indexing methods

2.2.1. Vamana graph

Vamana [25] is a widely used graph-based ANN indexing algorithm and serves as the foundational graph structure in many recent ANN studies. Its construction is based on two key techniques: Greedy Search and Robust Pruning.

Greedy Search enables efficient exploration of nearest neighbors within the graph. Beginning from an initial node, the algorithm iteratively identifies and prioritizes nodes closest to the query. It maintains a priority queue of visited nodes, extracting the nearest unvisited node at each step, adding its neighbors to the queue, and repeating this process until all relevant nodes are explored.

$$\alpha \cdot d(p^*, p') \leq d(p, p') \quad (\alpha > 1)$$

Robust Pruning optimizes the graph by removing unnecessary edges discovered during Greedy Search. This technique utilizes the Sparse Neighborhood Graph (SNG) property [26] and applies a distance threshold parameter, alpha (α), to determine whether an edge should be retained.

Specifically, an edge between nodes P and P' is removed if the distance between these nodes exceeds the distance between P' and a third node P^*. By retaining longer edges under relaxed conditions, Vamana enhances search efficiency while maintaining graph connectivity.

2.2.2. DiskANN

DiskANN [25] extends the Vamana graph to a disk-based environment, making it well-suited for large-scale datasets. It employs a divide-and-conquer approach, partitioning the dataset into subgraphs and merging them into a unified structure.

Figure 1 illustrates the partitioning and merging process of DiskANN. (1) The dataset is first divided into multiple overlapping partitions using a clustering technique, with each node assigned to at least two partitions. (2) Independent subgraphs are constructed within each partition. (3) These subgraphs are then merged by randomly selecting nodes from the partitioned graphs, followed by merging and pruning operations. This process updates the neighbor lists with merged results, ensuring connectivity between partitions while reducing memory and computational costs. DiskANN's partitioning and merging strategy enables scalable graph construction but introduces challenges in maintaining search performance across partitions.

2.2.3. Batch insertion for graph construction

Efficient parallelization is essential for graph construction in large-scale datasets. Batch insertion, as introduced in ParlayANN [27], addresses the challenge of concurrently adding nodes to a graph while ensuring conflict-free operations. Conflicts are avoided by grouping nodes into independent batches and processing each batch in isolation, ensuring that nodes within a batch do not interfere with each other during insertion.

The batch insertion process occurs in two steps. First, nodes in a batch are independently connected to their nearest neighbors within the existing graph. These connections are established in parallel, as nodes within a batch do not share dependencies, preventing conflicts during edge creation. This step ensures that all newly added nodes are integrated into the graph without disrupting its structure. Second, reversed edges are added to the graph. A reversed

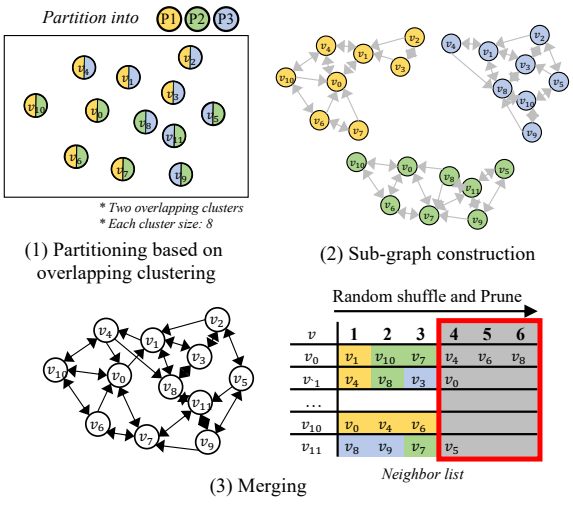


Figure 1: DiskANN partitioning and merging process.

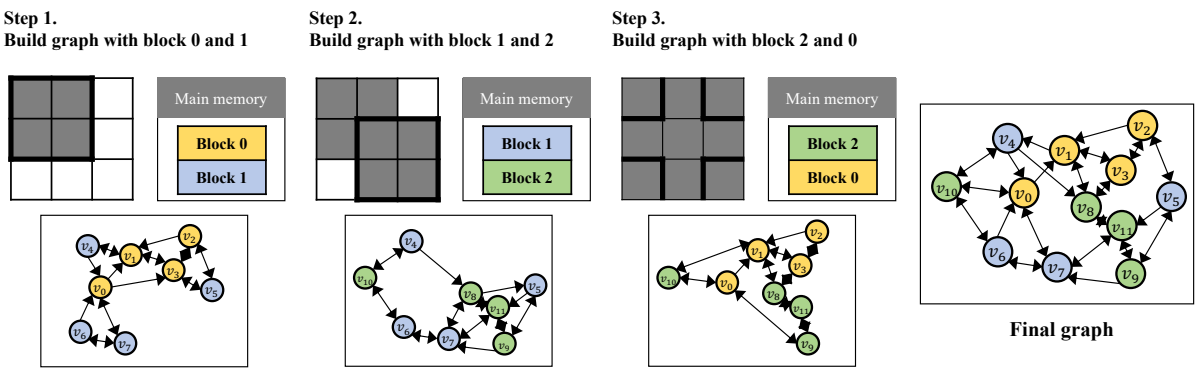


Figure 2: ScaDANN building process with three partitions and corresponding grid blocks at each step. The graph is constructed by loading two partitions into memory at a time, ensuring efficient graph generation. In the next step, the previously used partition is overlapped with a new partition to continue the building process.

edge is defined as an edge where the source vertex and destination vertex are swapped, ensuring bidirectional connectivity between nodes. For example, if an edge connects node v_i (source) to node v_j (destination), the reversed edge connects v_j back to v_i. During this step, the neighbors of each newly added node are updated with reversed edges, completing the bidirectional connections required for robust graph-based indexing.

By separating the insertion of neighbors and the addition of reversed edges into distinct steps, batch insertion guarantees structural consistency while leveraging parallel processing. This approach is particularly effective in large-scale datasets, as it minimizes computational overhead while maintaining the integrity and quality of the graph. Moreover, the absence of interdependencies between batches ensures that the graph remains stable throughout the construction process, enabling efficient and scalable graph-based ANN indexing.

3. ScaDANN Method

3.1. Grid partitioning

Figure 3 illustrates the structure of Graph ANN using a grid format, where each node in the graph consists of a base vector and a neighbor list. Grid partitioning divides large-scale datasets into grid blocks, with rows representing source vertices and columns representing destination vertices. This approach organizes datasets into manageable segments, enabling memory- and disk-efficient graph construction. For example, constructing a Vamana graph for a 1-billion-point dataset with 100 dimensions, requiring 400GB for vector storage, is infeasible under memory constraints of 256GB

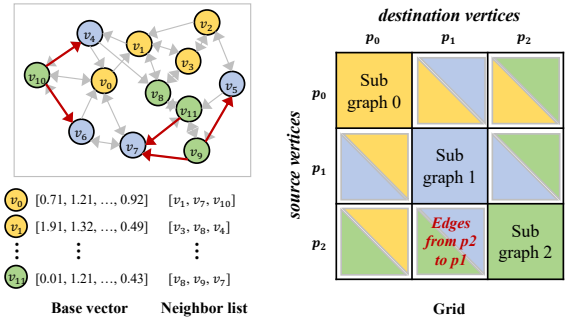


Figure 3: Graph ANN with grid format. The grid format represents the structure of Graph ANN, where each node consists of a base vector and a neighbor list.

or 512GB. However, grid partitioning ensures scalability by allowing the dataset to be partitioned into grid blocks, making it feasible to construct the graph even when the dataset size exceeds the available memory.

Each grid block captures intra- and inter-partition relationships. A grid block where the source and destination partitions are the same forms a subgraph, which follows the Vamana graph structure. Diagonal blocks represent subgraphs within individual partitions, while off-diagonal blocks denote connections between partitions. For example, the red block ($p2$, $p1$) in Figure 3 corresponds to connections from partition $p2$ (green) to partition $p1$ (blue). This structure improves scalability and ensures conflict-free parallel processing during node insertion.

3.2. Overlapping block-level insertion

The overlapping block-level insertion method incrementally constructs the graph by processing data in grid block units. This method employs overlapping blocks, where a block from the previous step is reused in the next step, allowing two blocks to be processed simultaneously. As detailed in Algorithm 1, the process begins with the construction of the graph for the first partition ($B_1 = 0$). In this step(line 3), specifically generates the subgraph for the first partition, ensuring that the intra-partition edges are established within the initial grid block.

In subsequent steps, pairs of overlapping blocks are processed to extend the graph. For each step, the data corresponding to the current blocks is loaded into memory to construct the graph (lines 9–11). Particularly, line 11 involves filling four grid blocks by constructing the subgraphs for two partitions and establishing connections between them. This ensures that intra- and inter-partition edges are created simultaneously, improving efficiency and maintaining the global structure of the graph. Once the graph is constructed, the earlier block is offloaded from memory (lines 12–15), and the next block is loaded (lines 5–10).

By reusing overlapping blocks during the insertion process, ScaDANN integrates graph construction and merging into a single operation, eliminating the need for a separate, computationally expensive merging step. This approach ensures that inter-block connections are captured as the graph is built, thereby avoiding the creation of disjoint subgraphs. Additionally, the overlapping mechanism significantly reduces I/O overhead by combining insertion and merging, which are traditionally distinct steps.

Figure 2 illustrates the construction process. Initially,

Algorithm 1: ScaDANN Construction

Input: block size P and dataset D
Output: Graph

```
/* Initialize first block and interval */
```
first block number $B_1 = 0$, second block number $B_2 = 1$, block interval $itv = 1$, neighbor list $NBR_i = \emptyset$;
```
/* Read dataset for the first block */
```
$D_1 = \text{ReadDataset}(B_1)$;
$NBR_1 = \text{vamana}(B_1, NBR_1)$;
while $itv < P$ **do**
  $B_2 \equiv (B_1 + itv) \pmod P$;
  ```
  /* Increase interval size */
  ```
  if $B_1 = B_2$ **then**
    $itv = itv * 2$;
    continue
  ```
  /* Read the second dataset and graph */
  ```
  $D_2 = \text{ReadDataset}(B_2)$;
  $NBR_2 = \text{ReadGraph}(B_2)$;
  $NBR_1, NBR_2 =$ vamana$((D_1, D_2), (NBR_1, NBR_2))$;
  WriteGraph(NBR_1, B_1);
  $D_1 = D_2$;
  $NBR_1 = NBR_2$;
  $B_1 = B_2$;
WriteGraph(NBR_1, B_1);

blocks 0 and 1 are loaded into memory, and edges between the two blocks are created. In the second step, block 0 is offloaded, block 1 is retained, and block 2 is loaded. The graph is extended based on connections between blocks 1 and 2. Finally, in the third step, block 1 is offloaded, block 2 is retained, and block 0 is reloaded to extend the graph further by connecting blocks 2 and 0. This iterative process efficiently captures all relevant connections between blocks, enabling the construction of a unified graph while minimizing memory usage and I/O costs.

3.3. Grid block merge

The grid block merge process establishes connections between two blocks loaded into memory and ensures bidirectional connectivity through reverse edges. Reverse edges, where the source and destination vertices are swapped, maintain symmetric inter-block connections, ensuring consistent and efficient search performance.

In conventional methods, merging requires all data to be loaded into memory to calculate distances between nodes. ScaDANN overcomes this limitation by storing distance information directly within the neighbor list of each node. This pre-stored distance means that block 1 retains the neighbor list formed in step 1. For example, after step 1, v_5 in block 1 (green) has already stored v_3 and v_2 from block 0 (yellow) as its neighbors. This allows distance comparisons between existing and new neighbors without reloading offloaded data, reducing memory usage and improving scalability.

Figure 4 illustrates the merging process during step 2 in Figure 2, where blocks 1 and 2 are loaded into memory while block 0 remains stored on disk. When updating the neighbor list of v_5, the algorithm uses pre-stored distance data from step 1 to compare block 0 neighbors with block 2 neighbors, avoiding redundant distance calculations. As a result, v_5's neighbors are updated to v_{11}, v_3, and v_2.

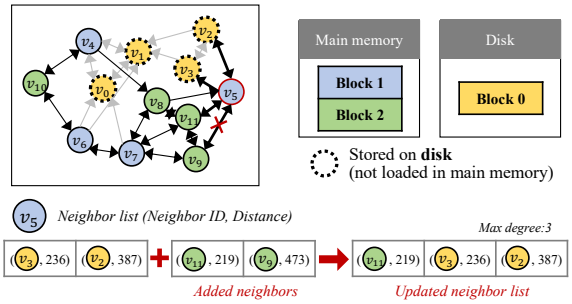


Figure 4: Grid block merge process with distance information.

The merging process follows the Vamana graph construction method, incorporating a greedy search to identify candidate neighbors and a robust pruning step to refine the neighbor list. Inter-block edges are established by adding neighbors from the second block to the neighbor list of the first block. The reverse edges are then added, ensuring bidirectional connectivity, where nodes in the first block are also linked as neighbors of nodes in the second block. This comprehensive approach ensures a robust and efficient merging process.

3.4. Complexity

Table 1 compares the time and space complexities of Vamana, DiskANN, and ScaDANN in the index-building process, focusing on scalability and efficiency. The space complexity considers storage requirements, including neighbor lists.

Vamana constructs a single large-scale graph, resulting in a time complexity of $O(S^{1.16})$ and a space complexity of $O(S \cdot R)$. Since it does not partition the dataset, the entire graph must fit in memory, making it less scalable for large-scale data. Both DiskANN and ScaDANN partition the dataset to construct subgraphs. Partitioning reduces memory usage and enables disk-based processing. The time complexity is inversely proportional to the number of partitions, as increasing P allows for more localized graph construction, reducing the computational cost.

In DiskANN, each node is assigned to multiple overlapping partitions (typically $K = 2$), as shown in Figure 1. Since DiskANN performs clustering to determine partitions, the build time complexity includes the clustering time (C). In contrast, ScaDANN utilizes dynamic partitioning, where the data structure size is determined based on the dataset size. This process does not require separate clustering and is treated as an internal computational step, which is why it is not included in the time complexity. Unlike DiskANN, which incurs overhead due to overlapping partitions, ScaDANN eliminates these redundancies, streamlining subgraph construction and merging. This results in a computationally efficient process that scales well with increasing dataset size.

Methods	Build Time	Space
Vamana	$O(S^{1.16})$	$O(S \cdot R)$
DiskANN	$O(K^{1.16} \cdot P^{-0.84} \cdot S^{1.16} + C)$	$O(S \cdot R \cdot K \cdot P^{-1})$
ScaDANN	$O(P^{-0.84} \cdot S^{1.16})$	$O(S \cdot R \cdot P^{-1})$

Table 1
Comparison of time and space complexities for Vamana, DiskANN, and ScaDANN, where S is the dataset size, R is the maximum degree of the graph, K is the number of overlapping partitions per node, P is the number of partitions, and C is the clustering time.

4. Experiments

4.1. Experimental setup

The experiments were conducted to evaluate the performance of ScaDANN under controlled conditions. The hardware environment included two Gold 6326 16-core processors and 1.9 TB of main memory, while the software environment was based on Ubuntu 20.04.6 LTS. To simulate memory-constrained scenarios, the available memory was deliberately restricted to 256 GB during the experiments. Under these constraints, both Vamana and ParlayANN encountered Out of Memory (OOM) errors, making them unsuitable for comparison. As a result, our evaluation focused on comparing ScaDANN with DiskANN, a state-of-the-art disk-based ANN indexing method.

Table 2 provides a summary of the datasets used in the experiments, including their data types and dimensions. The datasets used were BIGANN, Microsoft SPACEV, Yandex DEEP, and Yandex Text-to-Image, each posing unique challenges for ANN indexing. BIGANN consists of 1 billion 128-dimensional SIFT descriptors from a large-scale image dataset, serving as a benchmark for high-dimensional similarity search. Microsoft SPACEV includes 1 billion 100-dimensional document and query vectors reflecting real-world web search scenarios. Yandex DEEP comprises 1 billion 96-dimensional image descriptors designed to evaluate deep learning-based image representations. Yandex Text-to-Image contains 1 billion multi-modal embeddings, where image embeddings serve as the database and text embeddings form the query set, representing a cross-modal retrieval task with distinct distributions for database and query vectors.

Dataset	Data type	Dimensions
BigANN	uint8	128
Microsoft SPACEV	int8	100
Yandex DEEP	float32	96
Yandex Text-to-Image	float32	200

Table 2
Summary of datasets used in the experiments.

4.2. Results and Analysis

4.2.1. Billion-scale index building in limited memory

Figure 5 presents the results of indexing at a billion-scale under a 256GB memory constraint. The top graph in Figure 5 illustrates the index building time across different datasets. ScaDANN significantly reduced the index-building time compared to DiskANN across all datasets. For example, in the BIGANN dataset, ScaDANN completed the indexing task in 10.7 hours, compared to 16.5 hours for DiskANN, achieving approximately a 35% reduction in build time. This efficiency improvement can be attributed to ScaDANN's optimized data partitioning and overlapping block techniques, which minimize disk I/O and improve memory access patterns.

The bottom graph in Figure 5 shows the memory usage during the index construction process. Both DiskANN and ScaDANN employed dynamic partitioning, which adjusts the number of partitions based on available memory capacity and efficiently divides the data. Since the graph size and memory usage vary depending on the degree of the neighbor list, dynamic partitioning enables optimized partitioning by adapting to these variations. Notably, ScaDANN required a higher number of partitions than DiskANN. This is because ScaDANN stores additional distance information within the graph, which increases memory consumption per block. As a result, ScaDANN divides the data into more partitions than DiskANN to accommodate the additional memory usage. For instance, for the BIGANN dataset, ScaDANN used around 200GB of memory for indexing, while DiskANN remained under the 256GB limit, but required fewer partitions to store its graph.

We conducted a detailed comparison of the index building process with DiskANN, as illustrated in Figure 6. In the subgraph construction phase, our method completed approximately twice as fast as DiskANN, with the BIGANN dataset showing a significant speedup. However, the subgraph merging phase took longer in our approach. This is due to the finer-grained distance comparisons performed during the merging process, which operates at the level of grid blocks. Unlike DiskANN, which uses a more straightforward merging process, ScaDANN's overlapping block approach requires additional computations during the merging phase to ensure accurate distance calculations between partitioned blocks.

In terms of I/O costs, our method demonstrated greater efficiency. This improvement can be attributed to the use of overlapping block techniques, which optimize data loading between memory and disk. Specifically, ScaDANN's approach minimizes unnecessary disk reads and efficiently manages memory, leading to a faster construction process despite the increased memory overhead. This is evident in the I/O time breakdown in Figure 6, where ScaDANN exhibits a more balanced distribution of time between subgraph building, partitioning, and merging, compared to DiskANN, which spends a significant portion of time in I/O operations.

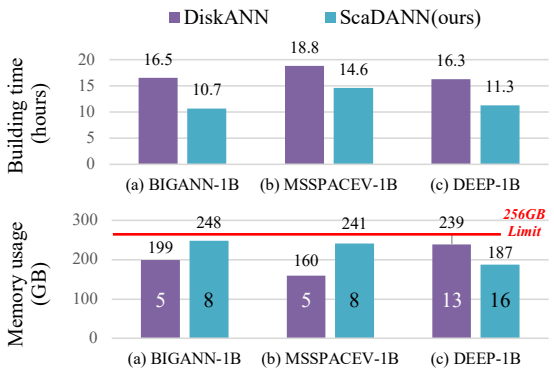


Figure 5: Billion-scale index building experiment results, including building time and memory usage, conducted under a 256GB memory constraint. The memory usage values indicate the number of partitions dynamically determined during the indexing process.

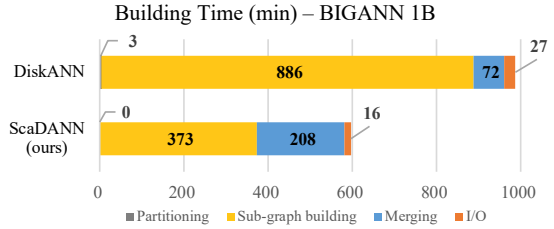
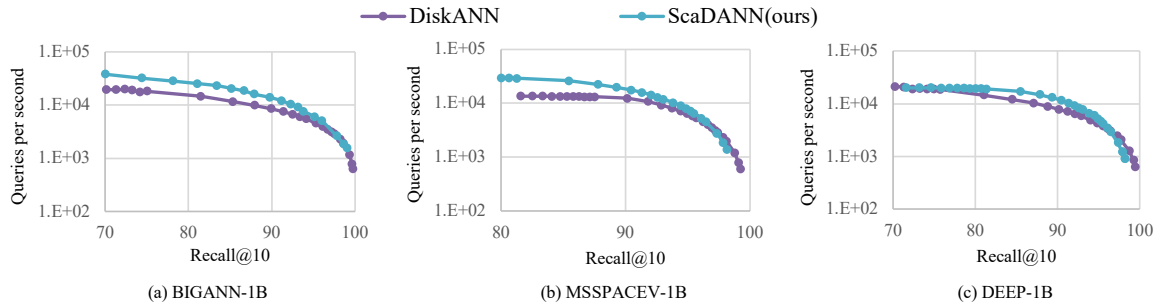


Figure 6: Time taken for each step in the building process, categorized into partitioning, subgraph building, merging, and I/O time for comparison.



(a) BIGANN-1B (b) MSSPACEV-1B (c) DEEP-1B

Figure 7: Disk-based search performance. The graph compares Recall@10 (x-axis) and QPS (y-axis), where higher placement indicates lower latency at a given recall level. A method positioned higher in the graph achieves faster query processing while maintaining similar accuracy.

4.2.2. Disk-based search performance

Figure 7 presents the comparison of the search performance based on disks between ScaDANN and DiskANN. The experiments were carried out using the disk search method for evaluation. Our proposed ScaDANN method demonstrated better search performance than DiskANN for most datasets in terms of QPS, especially for the BIGANN-1B dataset. As shown in the graph, ScaDANN achieved a higher QPS while maintaining competitive recall rates. For example, ScaDANN processed queries at a faster rate than DiskANN for BIGANN and MSSPACEV-1B, achieving a significant speedup while still maintaining similar or slightly better recall at top-10 results. However, for the DEEP-1B dataset, ScaDANN showed a slight performance drop when Recall@10 exceeded 95. This discrepancy can be attributed to the disk search method used, which is specifically optimized for DiskANN.

4.3. Ablation study

4.3.1. Impact of partition size on index construction

Figure 8 illustrates the effects of varying the number of partitions on the BIGANN 100M dataset. Increasing the number of partitions led to a significant increase in building time, with approximately a 70% increase per step. However, RAM usage decreased by around 40% per step, demonstrating the trade-off between memory efficiency and construction time.

Despite the changes in partition numbers, there were no significant differences observed in QPS relative to Recall@10 across the tested configurations. This indicates that increasing the partition number primarily affects the construction phase without adversely impacting search performance.

4.3.2. In-memory graph construction experiments

The in-memory graph construction experiments on BIGANN-100M and Yandex Text-to-Image-100M compare ParlayANN, ScaDANN, and DiskANN in terms of indexing time and query performance. Since ParlayANN encounters out-of-memory (OOM) issues at the billion-scale, the experiments were conducted on 100 million data points without partitioning for ScaDANN and DiskANN. ScaDANN achieves indexing speeds comparable to or faster than ParlayANN by storing distances within the graph, reducing redundant computations. Both methods also benefit from batch insertion, leading to faster construction than DiskANN. ScaDANN not only achieves a faster build time but also maintains strong search performance, striking a balance between efficient indexing and high query throughput.

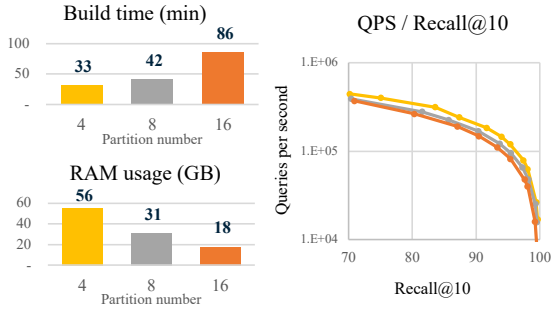
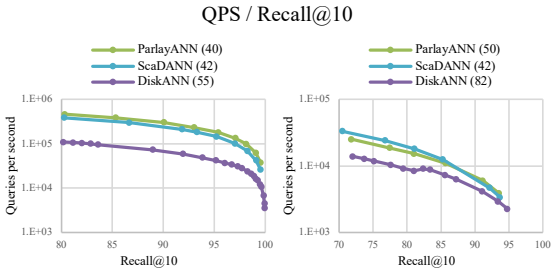


Figure 8: Experimental results of BIGANN 100M index building with varying numbers of partitions.



(a) BIGANN-100M (b) YANDEX Text-to-image-100M

Figure 9: In-memory experimental results of BIGANN 100M and Yandex Text-to-Image 100M. The numbers in parentheses next to each method indicate the index building time in minutes.

5. Conclusion

This study proposed ScaDANN, a scalable disk-based graph indexing method designed for billion-scale ANN datasets in memory-constrained environments. By introducing overlapping block-level insertion and grid block merge, ScaDANN effectively mitigates the high memory demands and inefficiencies of existing methods. These techniques enable seamless integration of blocks and efficient merging of partitioned graphs while minimizing I/O overhead and leveraging stored distance information.

Experimental results demonstrated that ScaDANN achieves up to 1.5× faster index construction and 1.4× higher query throughput compared to DiskANN, while maintaining competitive accuracy across diverse datasets. Furthermore, ScaDANN supports the construction of single indexes 10 times larger than existing approaches within the same memory constraints, making it a robust solution for large-scale ANN indexing. This work establishes ScaDANN as an efficient and scalable approach, with significant potential for real-world applications requiring high-performance ANN indexing.

References

[1] W. Yu, C. Zhu, Z. Li, Z. Hu, Q. Wang, H. Ji, M. Jiang, A survey of knowledge-enhanced text generation, ACM Computing Surveys 54 (11s) (2022) 1–38.

[2] H. Jin, Y. Zhang, D. Meng, J. Wang, J. Tan, A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods, arXiv preprint arXiv:2403.02901 (2024).

[3] Y. Zhuang, Y. Yu, K. Wang, H. Sun, C. Zhang, Toolqa: A dataset for llm question answering with external tools, Advances in Neural Information Processing Systems 36 (2023) 50117–50143.

[4] M. Cascella, J. Montomoli, V. Bellini, E. Bignami, Evaluating the feasibility of chatgpt in healthcare: an analysis of multiple clinical and research scenarios, Journal of medical systems 47 (1) (2023) 33.

[5] M. S. Orenstrakh, O. Karnalim, C. A. Suarez, M. Liut, Detecting llm-generated text in computing education: Comparative study for chatgpt cases, in: 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, 2024, pp. 121–126.

[6] K. Pandya, M. Holia, Automating customer service using langchain: Building custom open-source gpt chatbot for organizations, arXiv preprint arXiv:2310.05421 (2023).

[7] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, et al., A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions, ACM Transactions on Information Systems (2023).

[8] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, Y. Zhang, A survey on large language model (llm) security and privacy: The good, the bad, and the ugly, High-Confidence Computing (2024) 100211.

[9] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al., Retrieval-augmented generation for knowledge-intensive nlp tasks, Advances in Neural Information Processing Systems 33 (2020) 9459–9474.

[10] J. J. Pan, J. Wang, G. Li, Survey of vector database management systems, The VLDB Journal 33 (5) (2024) 1591–1615.

[11] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, J. Wang, Spann: Highly-efficient billion-scale approximate nearest neighborhood search, Advances in Neural Information Processing Systems 34 (2021) 5199–5212.

[12] J. V. Munoz, M. A. Gonçalves, Z. Dias, R. d. S. Torres, Hierarchical clustering-based graphs for large scale approximate nearest neighbor search, Pattern Recognition 96 (2019) 106970.

[13] S. Yu, J. Engels, Y. Huang, J. Shun, Pecann: Parallel efficient clustering with graph-based approximate nearest neighbor search, arXiv preprint arXiv:2312.03940 (2023).

[14] H. Jegou, M. Douze, C. Schmid, Product quantization for nearest neighbor search, IEEE transactions on pattern analysis and machine intelligence 33 (1) (2010) 117–128.

[15] Y. Kalantidis, Y. Avrithis, Locally optimized product quantization for approximate nearest neighbor search, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2014, pp. 2321–2328.

[16] A. Babenko, V. Lempitsky, The inverted multi-index, IEEE transactions on pattern analysis and machine intelligence 37 (6) (2014) 1247–1260.

[17] W. Dong, C. Moses, K. Li, Efficient k-nearest neighbor graph construction for generic similarity measures, in: Proceedings of the 20th international conference on World wide web, 2011, pp. 577–586.

[18] C. Fu, C. Xiang, C. Wang, D. Cai, Fast approximate nearest neighbor search with the navigating spreading-out graph, arXiv preprint arXiv:1707.00143 (2017).

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

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

[21] J. Chen, H.-r. Fang, Y. Saad, Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection., Journal of Machine Learning Research 10 (9) (2009).

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

[23] K. Echihabi, K. Zoumpatianos, T. Palpanas, New trends in high-d vector similarity search: al-driven, progressive, and distributed, Proceedings of the VLDB Endowment 14 (12) (2021) 3198–3201.

[24] H. V. Simhadri, G. Williams, M. Aumüller, M. Douze, A. Babenko, D. Baranchuk, Q. Chen, L. Hosseini, R. Krishnaswamny, G. Srinivasa, et al., Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search, in: NeurIPS 2021 Competitions and Demonstrations Track, PMLR, 2022, pp. 177–189.

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

[26] S. Arya, D. M. Mount, Approximate nearest neighbor queries in fixed dimensions., in: SODA, Vol. 93, Citeseer, 1993, pp. 271–280.

[27] M. D. Manohar, Z. Shen, G. Blelloch, L. Dhulipala, Y. Gu, H. V. Simhadri, Y. Sun, Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms, in: Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 2024, pp. 270–285.