The DiskANN library: Graph-Based Indices for Fast, Fresh and Filtered Vector Search

Ravishankar Krishnaswamy Magdalen Dobson Manohar Harsha Vardhan Simhadri

Abstract

Approximate nearest neighbor search has become a core component of AI systems on cloud and edge, spanning extremes of scales and form factors. We overview the DiskANN library of graph-based indices and algorithms that enable the practical construction and deployment of approximate nearest neighbor search indices across a variety of such systems. Specifically, we present indices that are capable of running efficiently out of an SSD, preserving recall over a stream of updates, and incorporating attributes alongside vector data to support predicate filters. They also support performance at least on par with other “in-memory" graph-based indices. Interestingly, all these algorithms arise from a variation of the prune procedure used in most graph-based indexing algorithms.

1 Introduction

Nearest Neighbor Search (NNS) is a classical problem in computer science, aimed at identifying the closest points in a dataset relative to a specified query point, based on a chosen distance metric (e.g., Euclidean or cosine similarity for vector datasets, or Jaccard similarity for sets of words). Formally, the input is a dataset P of points in Euclidean space along with a distance function, and the goal is to design a data structure that, given a query point q and target k, efficiently retrieves the k closest neighbors for q in the dataset P according to the given distance function. Algorithms and bounds for fundamental problem are well studied in the research community [3, 8, 15, 16, 20, 43, 47, 48, 55, 62, 80].

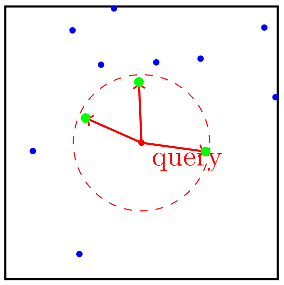


Figure 1: Nearest neighbor search in two dimensions.

1.1 Approximate Nearest Neighbor Search (ANNS)

Since it is impossible to retrieve the exact nearest neighbors without exhaustive search of the dataset in the general case [43, 80] due to a phenomenon known as the *curse of dimensionality* [25], one aims instead to find the *approximate nearest neighbors* (ANN) where the goal is to retrieve k neighbors that are close to being optimal with the help of *ANNS indices*. We also refer to this problem as *vector search*. The quality of an index and the corresponding search algorithm is judged by the trade-off it provides between accuracy and the time and space complexity of a query, or in the case of specific implementations of the algorithm, the hardware resources such as compute, memory and I/O needed for a query.

Theoretical analyses [8, 14, 43] typically provide bounds on the quality of the solution using the *approximation ratio*, namely, the ratio of the distance of the candidate returned by the algorithm and the true closest nearest neighbor for the query vector, for the case when $k = 1$. For larger values of k, a common generalization is the ratio of the distance of the k^{th} candidate returned by the algorithm and the true k^{th} closest nearest neighbor for the query. In this paper, we use a slightly different, but related, notion of *recall* relevant to practical applications of ANNS indices.[1]

Definition 1 (k-recall@k') *For a query vector q over dataset P, suppose that (a) $G \subseteq P$ is the set of actual k nearest neighbors in P, and (b) $X \subseteq P$ is the output of a top-k' ANNS query to an index. Then the k-recall@k', where $k \leq k'$ for the index for query q is $\frac{|X \cap G|}{k}$. Recall for a set of queries refers to the average recall over all queries.*

1.2 Usage: Scenarios and Scale

Indices for ANNS are critical in diverse applications in computer vision [78], data mining [22], information retrieval [56], classification [35], to state of a few. In these applications, objects are *embedded* in a high dimensional vector space by a semantic embedding model that captures the intended notion of semantic similarity as geometric distance in a high-dimensional Euclidean vector space [28, 39, 68, 82]. Typical embeddings have dimensions range from 100 to 1000, and use ℓ_2-distance, cosine similarity, or inner product as distance functions[2]. With improvements in machine learning models and scale of data they are trained on, such *dense vector indices* over embeddings of objects have become pivotal to the quality of search [23, 82] and recommendation systems [27]. Industrial use cases from web and enterprise search to device local search (e.g., Windows Copilot Runtime [64]) now rely on vector search. Further, with the evolution of Large Language Models (LLMs), such indices have also found powerful use in grounding LLMs and providing access to valuable private or proprietary data via *Retrieval Augmented Generation (RAG)* [52]. Agents such as OpenAI DeepResearch, Microsoft Copilots [1] that combine LLMs generative and reasoning abilities with knowledge stores such as web or enterprise document indices primarily use vector search for retrieval.

Due to the pervasive nature of this workload, standalone vector indexing software such as FAISS [31] and nmslib [21] as well as vector indexing extensions for existing systems have been developed. For example, inverted-index based search engines [11, 65, 72, 76], document databases [12, 49, 59, 69], and relational databases [6, 13, 57, 61, 66, 83, 84] now allow vector search to search and retrieve their content. Specialized *vector databases* that primarily support vector search have been developed [7, 26, 34, 67, 77].

Given the range of applications in which vector search is used, their deployments can span extremes of scale in size and throughput:

Index Size. A web index could span hundreds of billions of vectors, each representing a URL and its contents. In addition, each user interaction with the web search interface might trigger requests to dozens of other semantic indices for advertisements, videos, images, entities, etc., each of which might span billions of vectors. Enterprise search system like Microsoft 365 that serve millions of users could be even larger. They might build millions of indices, each representing an enterprise's shared content or personal email inbox. Such systems might index many trillions of vectors overall. On the other extreme, a device local AI runtime might have many small application specific indices each with only thousands of vectors.

Query performance. Web scale indices need to process tens of thousands to hundreds of thousands of queries per second, with each response taking no more than tens of milliseconds. On the other hand, an inbox or an index on a personal device might only be searched a few times a day. These two use cases

[1]An index that provides good k-recall@k can also satisfy other notions of recall such as finding all neighbors within a certain radius in the real-world scenarios.

[2]We will restrict our focus to such datasets, and hence use the terms Vector Search and ANNS interchangeably.

are separated by 10 orders of magnitude in their query throughput. Query throughput for other cases falls at various points in this range.
Update speeds. As the underlying collection of objects these vector indices represent change (e.g. web crawls, new emails, new rows/docs in a database), the vector index must be updated to represent this. The update latency required could vary from near-instant in the case of a database that has committed a transaction, or an index over conversation history with an AI conversation agent, to a few seconds or minutes in case of a real-time web or document index. The update throughput can also be as demanding as tens of thousands of updates/sec for a web based index.

Limitations of previous algorithms: Prior algorithms such as HNSW and FAISS IVF-PQ excelled over static datasets when indices could be stored in memory. In such scenarios, they could process tens of thousands of queries per second on multi-core processors or a GPU. However, the requirement of holding data in memory can be prohibitively expensive. A billion 768 dimensional vectors along with an HNSW index requires about 3.5 TB of main memory. For most scenarios except those requiring thousands of queries per second, it is difficult to justify this cost. It would be strongly preferable to construct an index that can be served from inexpensive SSDs which cost about 25x lesser than main memory. In fact, this is the only path to scaling these indices to industrial scale datasets.

Once built, these indices must be continuously updated but prior work is sparse on details on updating vector indices. The effect of sequences of incremental updates, where they exist in prior work, on the recall of the index are not well understood.

Further, in many scenarios, it is natural to select the closest vectors to the query among those that satisfy a particular clause. For example, one might want to retrieve most relevant documents from a particular domain or relevant to certain geography. In such cases, algorithms that query vector indices and then post-filter for the predicate afterwards can be quite inefficient.

1.3 The DiskANN library

The DiskANN library was developed to address the limitations highlighted above and other requirements. To present a few highlights, the DiskANN library can:

- index about a billion vectors points on a single compute node with a commodity SSD, and serve queries with high recall at upto 10,000 queries per second with single digit millisecond latency [73].
- process thousands of updates per second concurrently with queries. It can also update SSD based indices via a novel graph merge algorithms with limited memory and write amplification [71].
- construct specialized indices for certain predicate patterns that are nearly as efficient to query as plain vector indices [38].
- can process queries at least as efficiently as other graph-based indices such as HNSW when the same index (built for SSD) is hosted in memory, with comparable recall.

These ideas have been deployed in a variety of applications spanning web search, computational advertisements, enterprise search, databases, Copilots and other retrieval-augmented generative AI scenarios across Microsoft and adapted elsewhere in the industry [26, 49, 57, 67]. An open-source implementation of these ideas is available at `https://github.com/Microsoft/DiskANN`.

2 Overview of Different Classes of ANNS Algorithms

Recent surveys and benchmarks [10, 32, 53] provide an overview of the large body of research, and comparison of the state-of-the-art ANN algorithms. Here, we focus on the algorithms relevant for vectors in high-dimensional space with Euclidean metrics, and provides a broad categorization of them. Beyond ANNS for points in Euclidean spaces, there has been work for tailored inputs and other notions of similarity such as those for time series data [2, 22, 51]. See [32] for a comprehensive study of such algorithms.

Trees. Some of the early research on ANNS focused on low-dimensional points (say, $d \leq 20$). For such points, spatial partitioning ideas such as R^*-trees [17], kd-trees [18] and Cover Trees [20] work well, but these typically do not scale well for high-dimensional data owing to the curse of dimensionality. There have been some recent advances in maintaining several trees and combining them with new ideas to develop good algorithms such as FLANN [60] and Annoy [19].

Locality Sensitive Hashing (LSH). In a breakthrough result, Indyk and Motwani [43] developed a class of algorithms, known as *locality sensitive hashing* which resulted in the first, and only-known *provably approximate solutions* to the ANNS problem with a polynomially-sized index and sub-linear query time. Subsequent to this work, there has been a plethora of different LSH-based algorithms [4, 43, 85], including those which depend on the data [5], use spectral methods [81], distributed LSH [75], etc.

Clustering. Similar to LSH, there is another body of work [24, 48] which focuses on a more data-dependent way to solve ANNS in practice. These methods first cluster the dataset points using methods with good empirical performance, such as k-means heuristic. They then store an inverted index mapping each cluster to the list of database points that fall into the cluster. At query time, the search algorithm computes the closest (or closest few) cluster center(s) to the query, retrieves all the database points which belong to these closest clusters and computes the top k vectors from these retrieved points.

Graph-Based Index. One potential drawback of the space partitioning approaches (like the ones discussed above) is that that there are exponentially (in the dimension) many neighboring cells/clusters to a given region of space. Hence, it becomes difficult to select which nearby cells to scan to reduce the query time complexity. To circumvent such boundary issues, there has been an evolution of *graph-based ANNS indexing algorithms* [37, 45, 46, 55, 73, 74]. Several comparative studies [10, 32, 53, 79] of ANNS algorithms have concluded that these graph-based methods significantly out-perform other techniques in terms of search performance on a range of real-world static datasets. These algorithms are also widely used in the industry at scale. This paper shall deal with one such algorithm, called DiskANN [73], which has been used extensively in Microsoft in its core search technologies.

3 The DiskANN Algorithm

The primary data structure in the DiskANN data structure is a directed graph with vertices corresponding to points in P, the dataset that is to be indexed, and edges between them. With slight notation overload, we denote the graph $G = (P, E)$ by letting P also denote the vertex set. Given a node p in this directed graph, we let $N_{\text{out}}(p)$ and $N_{\text{in}}(p)$ denote the set of out- and in-edges of p. We denote the number of points by $n = |P|$. Finally, we let x_p denote the database vector corresponding to p, and let $d(p, q) = ||\mathsf{x}_p - \mathsf{x}_q||$ denote the ℓ_2 distance between two points p and q. We now describe how the DiskANN index is built and searched. We first describe how the search algorithm works assuming that the graph has been built.

Algorithm 1: GreedySearch(s, x_q, k, L)

```
Data: Graph $G$ with start node $s$, query $\mathsf{x}_q$, result size $k$, search list size $L \geq k$
Result: Result set $\mathcal{L}$ containing $k$-approx NNs, and a set $\mathcal{V}$ containing all the visited nodes
begin
    initialize sets $\mathcal{L} \leftarrow \{s\}$, $\mathcal{E} \leftarrow \emptyset$, and $\mathcal{V} \leftarrow \emptyset$
    // $\mathcal{L}$ is the list of best $L$ nodes, $\mathcal{E}$ is the set of all nodes which have
       already been expanded from the list, and $\mathcal{V}$ is the set of all nodes
       which have been visited, i.e., inserted into the list
    initialize $\mathsf{hops} \leftarrow 0$ and $\mathsf{cmps} \leftarrow 0$
    while $\mathcal{L} \setminus \mathcal{E} \neq \emptyset$ do
        let $p* \leftarrow \arg\min_{p \in \mathcal{L} \setminus \mathcal{E}} ||\mathsf{x}_p - \mathsf{x}_q||$
        update $\mathcal{L} \leftarrow \mathcal{L} \cup (N_{\text{out}}(p^*) \setminus \mathcal{V})$ and $\mathcal{E} \leftarrow \mathcal{E} \cup \{p^*\}$
        if $|\mathcal{L}| > L$ then
            update $\mathcal{L}$ to retain closest $L$ points to $\mathsf{x}_q$
        update $\mathsf{hops} \leftarrow \mathsf{hops} + 1$ and $\mathsf{cmps} \leftarrow \mathsf{cmps} + |N_{\text{out}}(p^*) \setminus \mathcal{V}|$
        update $\mathcal{V} \leftarrow \mathcal{V} \cup N_{\text{out}}(p^*)$
    return [closest $k$ points from $\mathcal{V}$; $\mathcal{V}$]
```

3.1 Index Search and Navigable Graphs

In graph-based data structures for vector similarity search like DiskANN, an important aspect of performance is how efficiently the graph index can be traversed to locate nodes closest to a given query. This efficiency depends on the graph's *navigability*—a property that allows a search algorithm to efficiently find nearby nodes to the query using a greedy-like algorithm, without exhaustively examining every node.

Roughly speaking, navigability of a directed graph is the property that ensures that the index can be queried for nearest neighbors using the following *greedy* search algorithm. The greedy search algorithm traverses the graph starting at a designated *start node* $s \in P$. The search iterates by greedily walking from the current node u to a node $v \in N_{\text{out}}(u)$ that minimizes the distance to the query, and terminates when it reaches a *locally-optimal* node, say p^*, that has the property $d(p^*, q) \leq d(p, q) \, \forall p \in N_{\text{out}}(p^*)$. In other words, greedy search terminates when it improve distance to the query point by navigating *out* of p^*, and thus returns it as the candidate nearest neighbor for query q. In practice, we use a generalization of this procedure called beam search, where the algorithm maintains a list of size $L \geq k$, and iterates until the list is locally optimal, and finally outputs the top k from the list. Algorithm 1 formally describes this variant.

3.2 Index Construction and the α-RNG property

We now describe how to build a good navigable graph. Note that the primary goal of the index construction phase is that the greedy search algorithm *quickly converges* to a good local optimal solution (ideally the nearest neighbor(s) of the query) for most queries. We can measure the quickness, i.e., search complexity, in terms of the final number of distance comparisons and number of graph hops (tracked by cmps and hops respectively in Algorithm 1). Roughly speaking, the search complexity can be approximated by $\mathsf{deg} \times \mathsf{hops}$, where deg is the average out-degree of the graph. Furthermore, the space complexity of the index is $O(N\mathsf{deg}) + O(Nd)$ to store the graph data structure and the d-dimensional vectors of the database. In order to keep the data structure implementations simple, we enforce an upper bound of R, a tunable parameter on the maximum out-degree of each node in the graph. Thus the question becomes, how do we choose the (at most) R out-neighbors of any node p, so that the greedy search algorithm converges to a *good local optimum* while minimizing the number of *hops*.

Algorithms like NN-*Descent* [30] use gradient descent techniques to determine G. The more recent algorithms start with a specific initial graph — an empty graph with no edges [55, 73] or an approximate $k-$nearest-neighbor graph [36, 37] — and iterate over all the base points x_p to refine G using the following two-step construction algorithm to improve navigability. Each iteration adds edges to ensure that if the query is close to x_p, the the greedy algorithm Algorithm 1 will converge to x_p. Since we don't know the query points at index construction time, we simulate the query as being x_p itself.

- **Candidate Generation** - For each base point x_p, run Algorithm 1 on G to obtain $\mathcal{E}$, the set of all nodes expanded during the search process. In order to ensure that x_p is reachable after the graph update in this iteration, we add $\mathcal{E}$ to $N_{\text{out}}(p)$ and $N_{\text{in}}(p)$, thereby improving the navigability to p in the updated graph G.
- **Edge Pruning** – When the out-degree of a node p exceeds R, a *pruning algorithm* filters out similar kinds of (or redundant) edges from the adjacency list to ensure $|N_{\text{out}}(p)| \leq R$. Different algorithms differ in how they prune the neighborhoods, and this forms a crucial difference between the different graph-based ANNS algorithms.

3.2.1 The DiskANN Pruning Strategy

One of the crucial differentiators between DiskANN and other graph algorithms is in the way it prunes the out neighbors of a node p, when the degree exceeds the threshold R. Indeed, how do we determine which of p's current neighbors to retain while bringing the degree down? To answer this question, prior algorithms like HNSW and NSG applied a very elegant pruning strategy which will greatly sparsify the graph. Loosely speaking, if p has two neighbors p_1 and p_2 such that $||\mathsf{x}_p - \mathsf{x}_{p_2}|| > ||\mathsf{x}_{p_1} - \mathsf{x}_{p_2}||$, then we can declare p_2 as a redundant neighbor, and remove it from the out-neighborhood of p. Intuitively, if the query is close to p_2 (which is why the greedy search algorithm will find the edge $p \to p_2$ useful), then p_1 is a candidate neighbor which gets the search procedure closer to p_2 (a surrogate for the target region) than p, and hence we can eliminate the edge to p_2. However, as we shall see in subsequent sections, this pruning strategy might end up being be too aggressive, and has some drawbacks.

Definition 2 (α Relative Neighborhood Graph (RNG) Property) *The crucial concept used in the DiskANN graph construction is a more* relaxed *pruning procedure, which removes an edge* (p, p_2) *only if there is an edge* (p, p_1) *and* $||\mathsf{x}_p - \mathsf{x}_{p_2}|| > \alpha||\mathsf{x}_{p_1} - \mathsf{x}_{p_2}||$ *for some constant* $\alpha > 1$.

Intuitively, building such a graph using $\alpha > 1$ helps the distance to the query vector to decrease *geometrically* by a factor of α in Algorithm 1 in each step over the graph. Note that graphs become denser as α increases as this is a more relaxed pruning criterion. We formalize this pruning strategy in Algorithm 3. In practice, α is typically set between 1.0 to 1.4, with 1.2 being the typical choice. It is very unusual for $\alpha < 1.0$ or $\alpha > 1.4$ to yield desirable results. See Figure 2 for an illustration of the pruning algorithm.

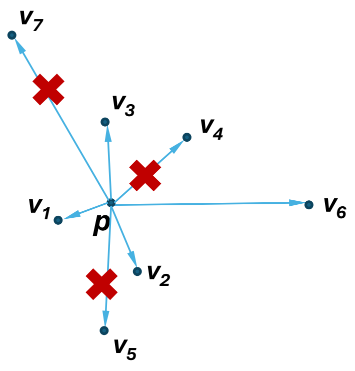


Figure 2: An illustration of Algorithm 3 on an adjacency list of length 7 in two-dimensional space. The seven vertices are numbered in order of distance from p. In application of the algorithm, v_4 and v_7 are pruned out due to proximity to v_3, and v_5 is pruned due to proximity to v_2. Lower α would cause more aggressive pruning and a sparser adjacency list, while higher α would cause fewer vertices to be pruned out for a denser adjacency list. While algorithms such as HNSW implictly set the α value to 1.0, DiskANN's crucial insight was that configuring α to higher values results in a much higher quality graph.

Algorithm 2: Insert($\mathsf{x}_p, s, L, \alpha, R$)

```
Data: Graph $G(P, E)$ with start node $s$, new vector $\mathsf{x}_p$, parameter $\alpha > 1$, out degree bound $R$, list size $L$
Result: Graph $G'(P', E')$ where $P' = P \cup \{p\}$
begin
    initialize expanded nodes $\mathcal{E} \leftarrow \emptyset$
    initialize candidate list $\mathcal{L} \leftarrow \emptyset$
    $[\mathcal{L}, \mathcal{E}] \leftarrow \text{GreedySearch}(s, p, 1, L)$
    set $N_{\text{out}}(p) \leftarrow \text{RobustPrune}(p, \mathcal{E}, \alpha, R)$
    foreach $j \in N_{\text{out}}(p)$ do
        if $|N_{\text{out}}(j) \cup \{p\}| > R$ then
            set $N_{\text{out}}(j) \leftarrow$
              $\text{RobustPrune}(j, N_{\text{out}}(j) \cup \{p\}, \alpha, R)$
        else
            update $N_{\text{out}}(j) \leftarrow N_{\text{out}}(j) \cup \{p\}$
```

Algorithm 3: RobustPrune($p, \mathcal{E}, \alpha, R$)

```
Data: Graph $G$, point $p \in P$, candidate set $\mathcal{E}$, distance threshold $\alpha \geq 1$, degree bound $R$
Result: $G$ is modified by setting at most $R$ new out-neighbors for $p$
begin
    $\mathcal{E} \leftarrow (\mathcal{E} \cup N_{\text{out}}(p)) \setminus \{p\}$
    $N_{\text{out}}(p) \leftarrow \emptyset$
    while $\mathcal{E} \neq \emptyset$ do
        $p^* \leftarrow \arg\min_{p' \in \mathcal{E}} d(p, p')$
        $N_{\text{out}}(p) \leftarrow N_{\text{out}}(p) \cup \{p^*\}$
        if $|N_{\text{out}}(p)| = R$ then
            break
        for $p' \in \mathcal{E}$ do
            if $\alpha \cdot d(p^*, p') \leq d(p, p')$ then
                remove $p'$ from $\mathcal{E}$
```

3.2.2 The Overall Index Construction Algorithm

We now have all the pieces to state our index construction algorithm. The input comprises of a dataset P of n points, degree bound R, list size parameter L, and RNG parameter $\alpha \geq 1$.

Algorithm 4: Index Construction

```
Data: Dataset $P$, degree bound $R$, list size parameter $L$, RNG parameter $\alpha$
Result: Navigable Graph $G = (P, E)$, start node $s$
let $s \leftarrow \arg\min_{p \in P} ||\mathsf{x}_p - \frac{\sum_{p' \in P} \mathsf{x}_{p'}}{n}||$ // s is the medoid of the dataset

for each point p in dataset P do
    run Insert($\mathsf{x}_p, s, L, \alpha, R$)
```

3.3 Theoretical Analysis of DiskANN Graphs

As mentioned in the earlier section, one of the crucial differences between the DiskANN graph construction algorithm and other graph methods like HNSW and NSG, is the α parameter used during the pruning procedure. The other algorithms implicitly use $\alpha = 1$, thereby producing much sparser graphs. It turns out that, even from a theoretical perspective, the α parameter plays a crucial role in ensuring that the DiskANN algorithm constructs graphs with provable guarantees. In a very elegant paper, Indyk and Xu [44] proved the following theorem for a so-called slow-preprocessing variant of DiskANN. The interested reader may refer to the paper for more complete details.

Definition 3: For any point p in dataset P and radius $r \geq 0$, we use $B(p, r)$ to denote a ball of radius r centered at p, i.e., $B(p, r) = \{x \in P : d(x, p) \leq r\}$. We say that a dataset P has the doubling constant C if any ball $B(p, 2r)$ centered at some p can be covered using at most C balls of radius r, and C is the smallest number with this property. The value $\log_2 C$ is called the *doubling dimension* of P.

The doubling dimension is often used as a measure of the "intrinsic dimensionality" of a data set. Moreover, recent empirical studies [9] show that several real-world datasets for vector search reside in large ambient dimensional space, but have significantly smaller intrinsic dimensionality.

Theorem 3.1: Consider a dataset P of doubling dimension r, and suppose Δ is its aspect ratio, i.e., $D_{\max}/D_{\min}$. Then the graph constructed using the DiskANN slow-preprocessing algorithm has maximum degree at most $O(\alpha)^{\mathrm{d}} \log \Delta$. Moreover, the greedy search Algorithm 1 converges to an $\left(\frac{\alpha+1}{\alpha-1} + \epsilon\right)$-approximate solution in $O(\log_\alpha \left(\frac{\Delta}{(\alpha-1)\epsilon}\right)$ hops.

Interestingly, note that the theorem gives good convergence bounds only if $\alpha > 1$, which serves as validation for the more relaxed pruning strategy employed by DiskANN.

4 DiskANN Features

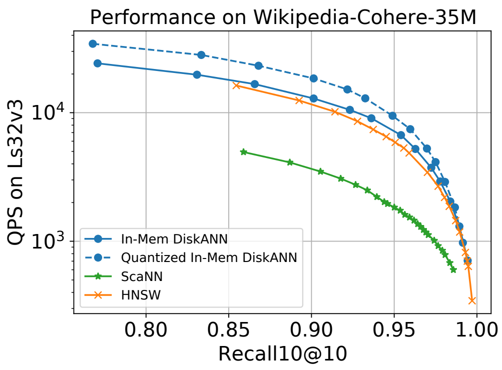
We now illustrate how this simple graph construction procedure can accommodate various important features, and present experimental results supporting our conclusions.

4.1 In-Memory Index

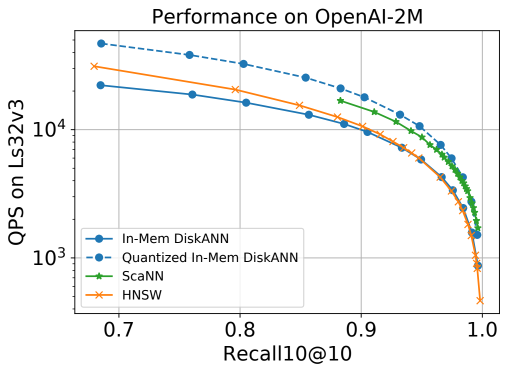
In this section, we showcase the performance of the *in-memory* DiskANN graph on state-of-the-art, modern datasets, as compared to other popular in-memory ANNS algorithms. In these experiments,

Algorithm	Build Time(s) on Wikipedia-Cohere-35M	Build Time (s) on OpenAI-ArXiv-2M
In-Mem DiskANN	4556	1210
In-Mem Quantized DiskANN	2948	851
HNSW	13778	1137
ScaNN	2059	206

Figure 3: Build times of each algorithm



(a) Results on Wikipedia-Cohere dataset. DiskANN was build with parameters $R = 64, L = 128, \alpha = 1.2$, and a 16-bit scalar quantization where indicated. ScaNN was built with an anisotropic quantization threshold of 0.2, 2 dimensions per block, and 8000 leaves. HNSW was built with parameters $M = 32, ef\_search = 400$.



(b) Results on OpenAI-ArXiv dataset. DiskANN was build with parameters $R = 40, L = 400, \alpha = 1.2$, and a 16-bit scalar quantization where indicated. ScaNN was built with an anisotropic quantization threshold of 0.2, 2 dimensions per block, and 1200 leaves. HNSW was built with parameters $M = 32, ef\_search = 400$.

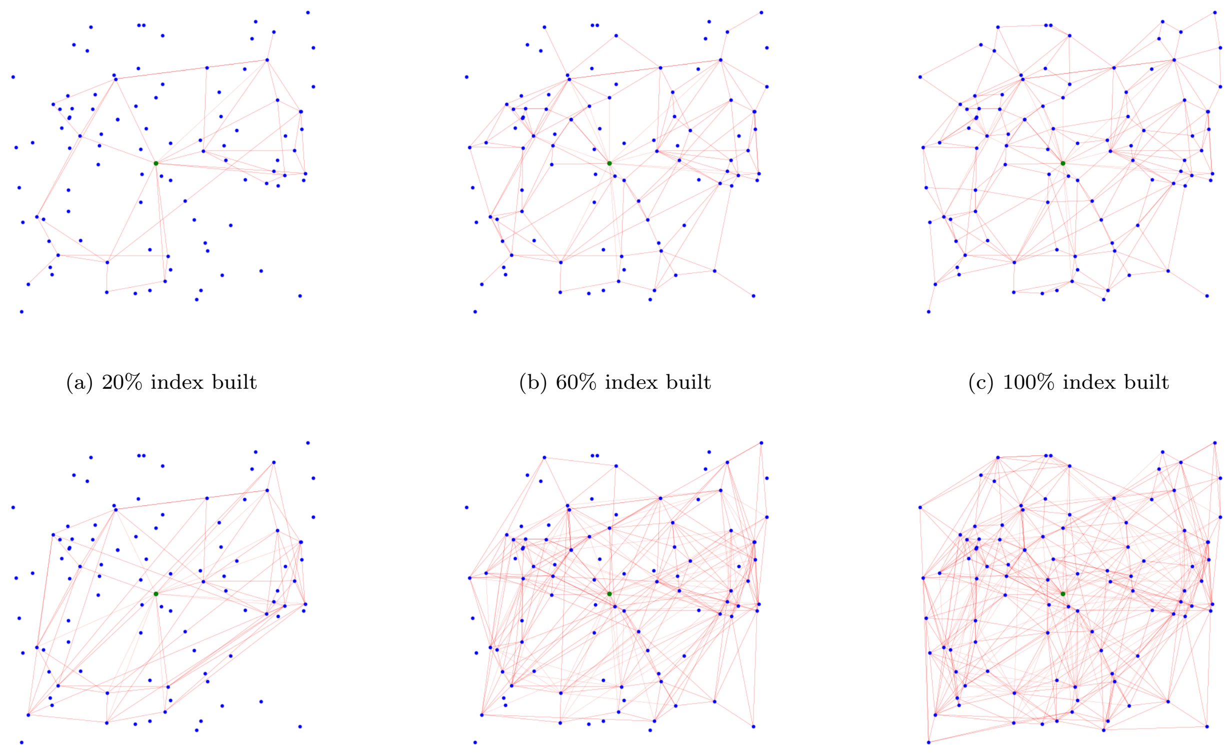
Figure 4: Recall/QPS curves comparing in-memory DiskANN with ScaNN and HNSW.

we compare DiskANN to ScaNN [41], a popular partition-based alternative algorithm, and HNSW [55], a widely used graph-based algorithm. For each algorithm, we used publicly available guidelines [42, 70] as well as our own parameter sweeps to select the best parameters for each dataset. When configuring DiskANN and ScaNN, we used each algorithm's respective quantization options–namely, scalar quantization to 16 bits for DiskANN and anisotropic vector quantization for ScaNN. The most widely used implementation of HNSW did not include a native quantization scheme, so for the sake of a fair comparison with HNSW, we also include the full-precision version of DiskANN in our experiments. Our experiments were run on an Azure Standard_L32s_v3 [58] machine with a third generation Intel processor with 32 vCPUs, using 8 threads for search.

In Figure 4 and Figure 3, we show the results of our experiments on two datasets, Wikipedia-Cohere-35M, with 35 million points, and OpenAI-ArXiV-2M, with 2 million points. Both datasets are publicly available at the Big ANN Benchmarks repository [33]. Overall, our results show strong performance of DiskANN, especially on larger datasets.

4.2 SSD-Resident Index

While graph-based ANN indices offer state of the art search performance in terms of latency/throughput vs recall, they are quite expensive to host due to consuming a significant amount of RAM. Indeed, one needs to store an additional graph data structure of size $O(nd_{\text{avg}})$ over and above the data vectors, where d_{avg} is the average degree of the graph index. One way to mitigate this space issue is to store the graph on cheaper auxiliary storage devices such as SSDs (solid state drives). Doing this in a naive


(a) 20% index built

(b) 60% index built

(c) 100% index built

(d) 20% index built

(e) 60% index built

(f) 100% index built

Figure 5: Stages of Index Construction with $\alpha = 1$ (Figures 5a to 5c) and $\alpha = 2$ (Figures 5d to 5f)

manner would unfortunately make the search latencies increase significantly. Indeed, each time the greedy search Algorithm 1 needs to expand one node to fetch its neighbors, the algorithm needs to make one round-trip to the SSD to fetch this adjacency list information, and the SSD round-trip latencies are an order of magnitude larger than a random RAM access.

This was in fact one of the main reasons [73] introduced a new graph construction algorithm: the graphs constructed based on the α-RNG property (with $\alpha > 1$) would end up considerably reducing the number of hops at search time, resulting in improved query latencies. Figure 5 illustrates the improved connectivity of the graph (which in turn will bring down the number of hops at search time) by using large values of α, in a dataset comprising of 100 random points in the unit square. In the plot in Figure 6, we show how the indices built with $\alpha = 1$ and $\alpha = 1.2$ differ in terms of how many hops it takes for the greedy search Algorithm 1 to achieve a desired recall on a real-world dataset. For this experiment, we used the `arxiv-openai` dataset [33] which comprises of around $2,000,000$ vectors in 1536 dimensions, and the `amazon-cohere` dataset [33] which comprises of around $2,000,000$ vectors in 384 dimensions

4.3 Streaming Index

In this section, we show how DiskANN can be adapted to the *streaming* scenario; that is, maintaining an index under a stream of insertions and deletions rather than a static index built in one shot. Since the build routine is naturally composed of a stream of insertions, the insert procedure is adapted to this scenario without modification. On the other hand, maintaining a high-performance index under a

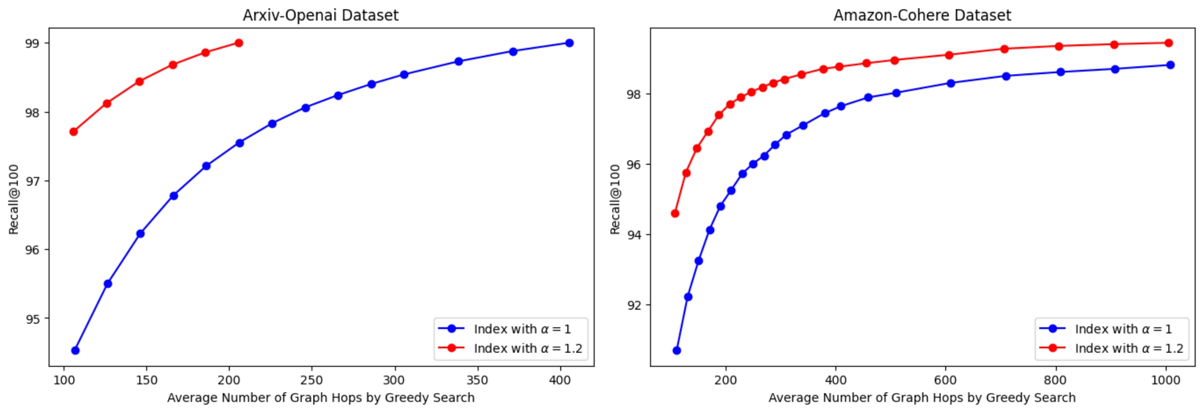


Figure 6: Recall vs Hops for Indices built with different values of α

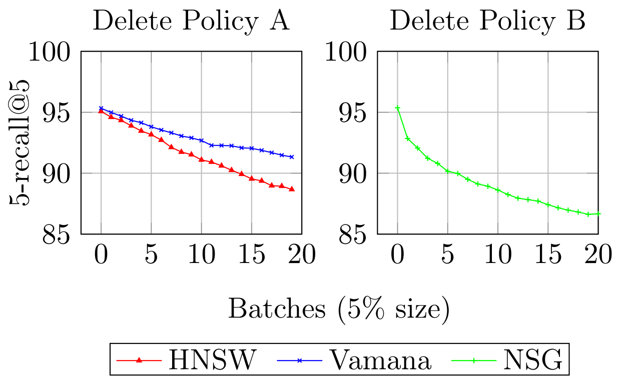


Figure 7: Search recall over 20 cycles of deleting and re-inserting 5% of SIFT1M dataset with statically built HNSW, Vamana, and NSG indices with $L_s = 44, 20, 27$, respectively.

stream of deletions is significantly more difficult. In this section, we explain this difficulty and how to mitigate it, and show the performance of streaming DiskANN under a variety of situations.

The most natural approach to deleting a point p from a DiskANN graph is to simply delete its corresponding vertex, as well as all edges pointing to the vertex representing p. However, this simple policy, which we refer to as the "Drop Policy", fails to maintain consistent recall for search with the same L_s, and rather degrades over time (see Figure 7. The reason for this degradation is that the deleted point may be an important routing node for queries to reach their correct answers, so losing the in- and out-neighbors of the node leads without any attempt at repair results in a lower quality graph.

This need for repair led to Algorithm 5 [71], which loops over vertices which have an edge to a deleted vertex, and replaces that edge with the out-neighbors of the deleted vertex while respecting the degree bound. In Figure 10 we refer to this policy as the "Consolidate Policy." This global algorithm can be called as a background process after a certain percentage of the index has been deleted; in the meantime, deleted nodes can be used as part of the search path but not returned as query results.

Algorithm 5: Consolidate Deletes

```
Data: Dataset $P$, degree bound $R$, list size parameter $L$, RNG parameter $\alpha$, delete set $D \subseteq P$
Result: Navigable Graph $G = (P \setminus D, E)$, start node $s$
for $p \in P$ do
    $E_p \leftarrow \emptyset$
    if $p \notin D$ then
        for $q \in N_{out}(p)$ do
            if $q \in D$ then
                $E_p \leftarrow E_p \cup \{e | e \in N_{out}(q) e \notin D\}$
            else
                $E_p \leftarrow E_p \cup q$
        if $|E_p| > R$ then
            $E_p \leftarrow$ RobustPrune$(p, E_p, \alpha, R)$
        $N_{out}(p) \leftarrow E_p$
    else
        continue
```

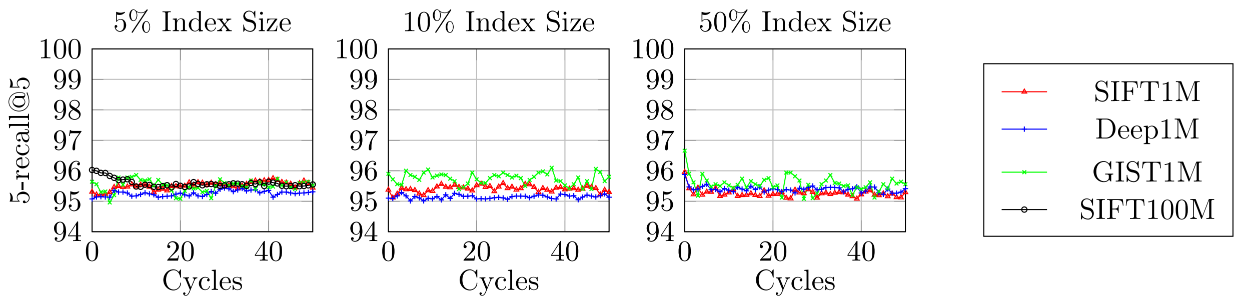


Figure 8: 5-recall@5 for FreshVamana indices for 50 cycles of deletion and re-insertion of 5%, 10%, and 50% of index size on the million-point and 5% of SIFT100M datasets. L_s is chosen to obtain 5-recall@5$\approx$ 95% on Cycle 0 index.

4.4 Recall stability of FreshVamana

We now demonstrate how using our insert and delete consolidation algorithms ensures that the resulting index is stable over a long stream of updates. We start with a statically built Vamana index and subject it to multiple cycles of insertions and deletions. In each cycle, we delete 5%, 10% and 50% of randomly chosen points from the existing index, and re-insert the same points. We then choose appropriate L_s (the candidate list size during search) for 95% 5-recall@5 and plot the search recall as the index is updated. Since both the index contents and L_s are the same after each cycle, a good set of update rules would keep the recall stable over these cycles. Figure 8 confirms that is indeed the case, for the million point datasets and the 100 million point SIFT100M dataset. In all these experiments, we use an identical set of parameters $L, \alpha = 1.2, R$ for the static Vamana index we begin with as well as our FreshVamana updates.

Effect of α on recall stability. To study the effect of α on recall we run the update rules for a stream of deletions and insertions with different α values, and track how the recall changes as we perform our

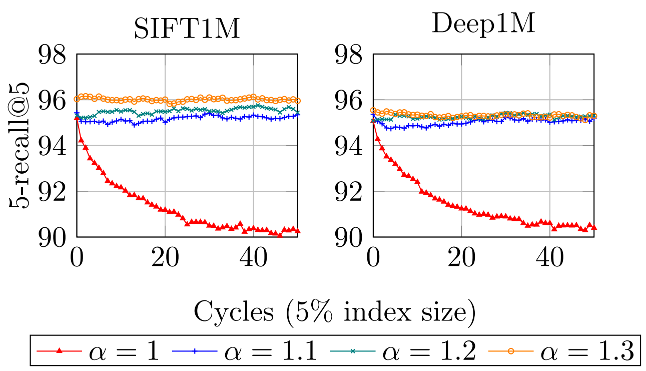
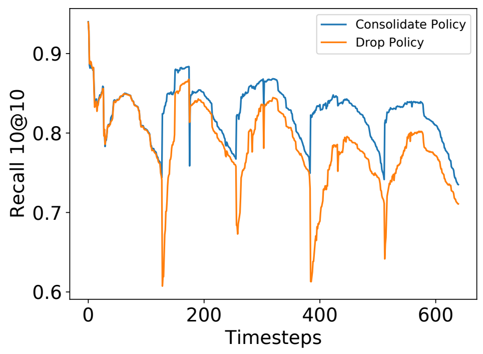
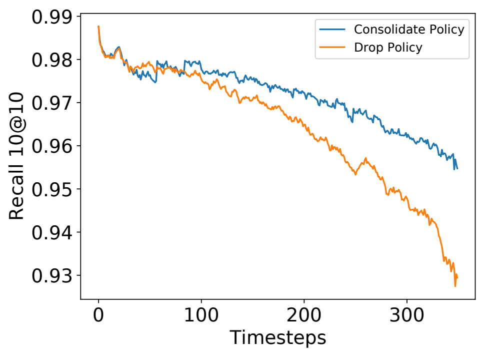


Figure 9: Recall trends for FreshVamana indices on SIFT1M and Deep1M over multiple cycles of inserting and deleting 5% of points using different values of α for building and updating the index. L_s is chosen to obtain $5-recall@5 \approx 95\%$ for Cycle 0 index.



(a) Recall over time for the MSTuring clustered runbook, built using $R = 50, L_{build} = 85, L_{search} = 85$.



(b) Recall over time for the Wikipedia-Cohere expiration time runbook, built using $R = 64, L_{build} = 128, L_{search} = 128$.

Figure 10: Recall over time for two streaming scenarios.

updates in Figure 9. Note that recall is stable for all indices except for the one with $\alpha = 1$, validating the importance of using $\alpha > 1$.

We examine the recall of the algorithms described above with two further streaming *runbooks*. The Drop Policy is called in a global background process at the same rate as the Consolidate Policy. A runbook consists of a sequence of insert, delete, and search operations, and recall is measured at each search operation. The first runbook is based the MSTuring-30M [33] dataset and designed to mimic the real-world trend of distribution shift within a streaming scenario. The dataset is clustered into 64 clusters, and points are inserted and deleted in clustered order over five rounds. The second runbook, based on the Wikipedia-35M [33] dataset, is designed to reflect another real-world scenario: that of an index where data is reliably deleted based on a time window. In this runbook, each point in the dataset is inserted and randomly assigned an expiration date, after which it is deleted. Figure 10 shows the trend in recall over time of both datasets. In Figure 10a, recall drops and then increases each time the delete algorithm is called, in five noticeable peaks corresponding to the five rounds. The recall using the Consolidate Policy is significantly higher than that of the Drop Policy. Similarly, in Figure 10b, recall using the Consoldate Policy remains much more steady over time than recall using the Drop Policy.

4.5 Filtered DiskANN

We now show how the simplicity of the modular graph construction and search approach allows us to solve interesting generalizations of ANNS which also have great real-world significance, by considering the case of *filtered search*. A motivating example is where each database point $p \in P$ corresponds to an advertisement, and the advertiser has an embedding vector x_p capturing the semantic information, and an additional set $L(p)$ of *labels*, corresponding to the *set of countries* where the ad can be displayed. The advertisement retrieval problem now corresponds to *finding the closest vectors from the database which contains the country where the query originated from.*

Formally, each database point $p \in P$, in addition to a vector $\mathsf{x}_p \in \mathbb{R}^d$, also comes with a set of associated labels $L(p) \subseteq U$, where U is a universe of possible labels. The query point again has a vector $\mathsf{x}_q \in \mathbb{R}^d$ and a filter label $\ell(q) \in U$. The goal is to retrieve the closest database points whose label-set contains $\ell(q)$: given a target k, we want to retrieve the top-k set (or a close approximation thereof) $R^* := \arg\min^k_{p:\ell(q)\in L(p)} ||\mathsf{x}_p - \mathsf{x}_q||$, where $\arg\min^k$ notation denotes the set of k elements optimizing the condition.

We now show how to adapt the basic graph construction and search to handle such simple filters. The ideas described are formalized in [38]. More sophisticated ideas to handle more complex filter predicates like conjunctions and disjunctions (as opposed to single filters at query time) using graph-based methods are presented in [63]. The interested reader may refer to these articles for any missing details.

4.5.1 Index Search

There are two main changes we make to the search algorithm, both of which are intuitive. The main change to Algorithm 1 is that we make sure that our candidate list $\mathcal{L}$ of best L nodes only contains candidates p which satisfy the query filter, i.e., $\ell(q) \in L(p)$. Indeed, the step

update $\mathcal{L} \leftarrow \mathcal{L} \cup (N_{\text{out}}(p^*) \setminus \mathcal{V})$ and $\mathcal{E} \leftarrow \mathcal{E} \cup \{p^*\}$

gets replaced with

update $\mathcal{L} \leftarrow \mathcal{L} \cup (\{x : x \in N_{\text{out}}(p^*) \wedge \ell(q) \in L(x)\} \setminus \mathcal{V})$ and $\mathcal{E} \leftarrow \mathcal{E} \cup \{p^*\}$

The second main change is that, instead of the search starting from a global start node s, the start node $s(\ell)$ for a query q with filter label ℓ depends on the filter label ℓ, to guarantee that $\ell \in L(s(\ell))$, i.e., the start node satisfies the query filter. To efficiently enable this, we pre-compute a map from the universe of labels to satisfying start nodes.

4.5.2 Index Build

Given that the search routine for query q only restricts to running a greedy search over the induced sub-graph of points containing the the label $\ell(q)$, we want the index construction to ensure that each of these induced subgraphs (for any label $\ell \in U$) is sufficiently well-connected and resembles a navigable DiskANN index only over these points themselves. To this end, the index construction is once again composed of iterating over all database points. In each iteration, when considering point p with labels $L(p)$, we first obtain some candidates to add edges with p, and then prune these candidates using the

α-RNG property. In the filtered setting, the candidates are precisely the *union* of all the nodes visited during each of the searches FilteredGreedy$(s(\ell), \mathsf{x}_p, 1, L, \ell)$[3] for $\ell \in L(p)$. This ensures that p only has edges to other nodes which share common labels, and can promote the connectivity of these labels. Next, the prune procedure also deviates from Section 3.2.1 in that it now depends both on the geometry of vectors and as well as the label overlap between the possible candidates. We outline this change in Definition 4, and make a corresponding adaption to Algorithm 3 to incorporate the label information. This overall algorithm is known as Filtered DiskANN. In another variant, we build a DiskANN graph for *each label*, namely over the point-set $P(\ell) = \{x : \ell \in L(x)\}$, take a union of all these graphs, and finally perform the filtered prune algorithm on each vertex independently to bring down the degree. Without the prune step, this graph will be very dense but excellent for filtered search; the pruning brings down the degree considerably without compromising on the quality a lot. This algorithm is known as Stitched DiskANN.

Definition 4: (α Relative Neighborhood Graph (RNG) Property for Filtered Index Construction). For any p, p_1 and p_2, the pruning algorithm removes an edge (p, p_2) if there is an edge (p, p_1), $||\mathsf{x}_p - \mathsf{x}_{p_2}|| > \alpha||\mathsf{x}_{p_1} - \mathsf{x}_{p_2}||$ for some constant $\alpha > 1$, and moreover, $L(p_1) \supseteq L(p) \cap L(p_2)$.

We now demonstrate the efficacy of our algorithm on two different use-cases.

Ads Dataset. The Ads datasets represent sponsored advertisements from a large ad corpus relevant across 47 regions (countries). Each ad can be served in one or more geographical regions based on advertiser preference. The vectors are 64-dimensional and derived from the twin-tower encoders [40, 54] applied to advertisements. Each data point p on average has a label set of size ~ 10. We compared our Filtered and Stitched DiskANN algorithms against several baselines: the first is a natural baseline that we refer to as *inline clustering*. In this baseline algorithm, we cluster the dataset into C clusters (whose value depends on the number of data points in the index and is typically $\sim \sqrt{N}$) using k-means algorithm. Then for each label $\ell \in U$, we maintain an inverted index of the set of points which contain the label ℓ. When a query vector x_q arrives with a filter $\ell(q)$, we first look up the inverted index to retrieve the set of base points which contain the label $\ell(q)$; let us call this set $P(\ell(q))$ to denote the valid points. Then we compute the closest m clusters (which is a tunable parameter for search) from the C cluster centroids, and retrieve all the vectors belonging to those closest clusters; let us call the set of this vectors as $\mathcal{N}$, denoting the nearby points. Finally, we intersect the sets $\mathcal{N} \cap P(\ell(q))$ and compute the distances of these points to the query, before outputting the closest k vectors. We can also use a clustering index (like Faiss-IVF) to retrieve the top $k' >> k$ vectors (without any filter requirements), and then post-process to identify the ones matching the query filter – this approach is known as Post-Process (Clustering). Finally, we can employ such a post-processing method on HNSW and (vanilla) DiskANN as well. We group the query based on the specificity of the filters, i.e., what fraction of points satisfy the query filter. A specificity of 100 percent means almost all points satisfy the query filter, and 1 percent means only 1 percent of the points satisfy the query filter. The Filtered and Stitched DiskANN graph was constructed with degree $R = 96$ and list size $L = 150$.

We also run some comparisons on another challenging semi-synthetic dataset called `Wikipedia-Cohere` [33], which embed passages from Wikipedia using the cohere.ai multilingual-22-12 model. The queries comprise of embeddings of sentences from the Wikipedia Simple articles using the same model. The universe U of labels for filtering are the top 4000 highest frequency words in English, excluding the NLTK stop-words. Each base vector has labels corresponding to the subset of words of U occurring in the

[3]This runs the filtered greedy search algorithm for the query being x_p, with candidate list size L, filter predicate being ℓ, starting at the pre-computed start node $s(\ell)$.

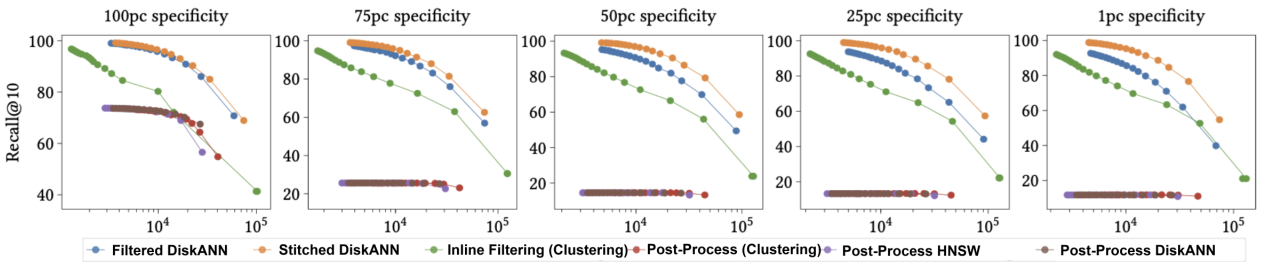


Figure 11: Ads dataset: QPS (x-axis) vs recall@10 for various algorithms with filters of 100, 75, 50, 25 and 1 percentile specificity.

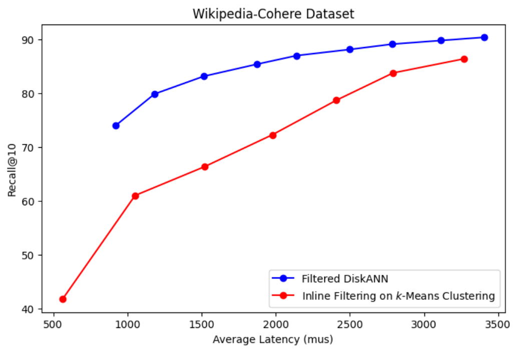


Figure 12: Recall vs Latency for Filtered DiskANN vs Clustering-Based Baseline

corresponding paragraph. Similarly, the query filter corresponds to the most common word from U in the corresponding sentence. This captures a hybrid search scenario, where there is semantic similarity using the vectors, and also hard keyword match requirements given by the filtering constraint. In Figure 12 we compare the performance of our Filtered-DiskANN algorithm against the inline clustering baseline.

5 Conclusion and Open Research Directions

In this article, we surveyed the DiskANN library for Approximate Nearest Neighbor Search, and demonstrated its versatility by adapting it to in-memory vector search (for high throughput scenarios), disk-based vector search (for high-scale scenarios), filtered vector search (for scenarios with query-time predicates), as well as streaming vector search (to handle highly dynamic data corpora such as emails, twitter feeds, etc.). We also described the provable guarantees of the algorithm under specific distributional assumptions on the data.

Going forward, there are many more interesting challenges for vector search algorithms to handle, and it will be important future work to see how the principles underlying the graph construction and search algorithms detailed in this article extend to those scenarios as well. For example, how can we handle more complex query predicates, which involve range filters, conjunctions, and disjunctions, and are crucial to applications such as shopping or targeted advertisements?

Next, many retrieval scenarios lean on the so-called *multi-vector retrieval models*, where documents and queries are represented by multiple vectors, and the similarity score is a more complicated function (compared to a simple euclidean distance we assumed in this article) involving these sets of vectors [50]. Such similarity scores are useful when comparing objects represented by large sets of vectors, such as long texts where each paragraph is represented by an individual vector. Can we extend our approach to handle such scenarios as well? Some examples of such a distance function include Chamfer distance and the Relaxed Earth Mover distance [29].

For the streaming index, we have presented a deletion policy where the graph is *consolidated* at regular intervals to remove the deleted nodes. Can we enable a more immediate, *eager* deletion policy which reflects deletions in real-time, while preserving the update and search latencies and search recalls? In short, there are a lot of exciting and impactful avenues for future work in this space for the interested reader to pursue.

References

[1] URL: `https://www.microsoft.com/en-us/microsoft-copilot/for-individuals/`.

[2] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In D. B. Lomet, editor, *Foundations of Data Organization and Algorithms*, pages 69–84, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[3] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, Jan. 2008. `doi:10.1145/1327452.1327494`.

[4] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, Jan. 2008. URL: `http://doi.acm.org/10.1145/1327452.1327494`, `doi:10.1145/1327452.1327494`.

[5] A. Andoni and I. Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 793–801, New York, NY, USA, 2015. ACM. URL: `http://doi.acm.org/10.1145/2746539.2746553`, `doi:10.1145/2746539.2746553`.

[6] Andrew Kane et al. URL: `https://github.com/pgvector/pgvector`.

[7] Andrey Vasnetsov, Arnaud Gourlay, Tim Visée, et al. URL: `https://github.com/qdrant/qdrant`.

[8] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 271–280, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=313559.313768`.

[9] M. Aumüller and M. Ceccarello. The role of local intrinsic dimensionality in benchmarking nearest neighbor search. In *Similarity Search and Applications: 12th International Conference, SISAP 2019, Newark, NJ, USA, October 2–4, 2019, Proceedings*, page 113–127, Berlin, Heidelberg, 2019. Springer-Verlag. `doi:10.1007/978-3-030-32047-8_11`.

[10] M. Aumüller, E. Bernhardsson, and A. Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87, 2020. URL: `http://www.sciencedirect.com/science/article/pii/S0306437918303685`.

[11] URL: `https://learn.microsoft.com/en-us/azure/search/vector-search-overview`.

[12] Azure Cosmos DB. URL: `https://learn.microsoft.com/en-us/azure/cosmos-db/vector-database`.

[13] AzureSQLServer. URL: `https://learn.microsoft.com/en-us/samples/azure-samples/azure-sql-db-openai/azure-sql-db-openai/`.

[14] A. Babenko and V. Lempitsky. The inverted multi-index. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3069–3076, 2012.

[15] A. Babenko and V. S. Lempitsky. Additive quantization for extreme vector compression. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 931–938. IEEE Computer Society, 2014. `doi:10.1109/CVPR.2014.124`.

[16] D. Baranchuk, A. Babenko, and Y. Malkov. Revisiting the inverted indices for billion-scale approximate nearest neighbors. In *The European Conference on Computer Vision (ECCV)*, September 2018.

[17] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, May 1990. `doi:10.1145/93605.98741`.

[18] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975. `doi:10.1145/361002.361007`.

[19] E. Bernhardsson. *Annoy: Approximate Nearest Neighbors in C++/Python*, 2018. Python package version 1.13.0. URL: `https://pypi.org/project/annoy/`.

[20] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 97–104, New York, NY, USA, 2006. Association for Computing Machinery. `doi:10.1145/1143844.1143857`.

[21] L. Boytsov and B. Naidan. Engineering efficient and effective non-metric space library. In N. R. Brisaboa, O. Pedreira, and P. Zezula, editors, *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*, volume 8199 of *Lecture Notes in Computer Science*, pages 280–293. Springer, 2013. `doi:10.1007/978-3-642-41062-8\_28`.

[22] A. Camerra, E. Keogh, T. Palpanas, and J. Shieh. isax 2.0: Indexing and mining one billion time series. In *2013 IEEE 13th International Conference on Data Mining*, pages 58–67, Los Alamitos, CA, USA, dec 2010. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/ICDM.2010.124`, `doi:10.1109/ICDM.2010.124`.

[23] Q. Chen, X. Geng, C. Rosset, C. Buractaon, J. Lu, T. Shen, K. Zhou, C. Xiong, Y. Gong, P. Bennett, N. Craswell, X. Xie, F. Yang, B. Tower, N. Rao, A. Dong, W. Jiang, Z. Liu, M. Li, C. Liu, Z. Li, R. Majumder, J. Neville, A. Oakley, K. M. Risvik, H. V. Simhadri, M. Varma, Y. Wang, L. Yang, M. Yang, and C. Zhang. Ms marco web search: A large-scale information-rich web dataset with millions of real click labels. In *Companion Proceedings of the ACM Web Conference 2024*, WWW '24, page 292–301, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3589335.3648327`.

[24] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. In M. Ranzato, A. Beygelzimer, Y. Dauphin,

P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 5199–5212. Curran Associates, Inc., 2021. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/299dc35e747eb77177d9cea10a802da2-Paper.pdf.

[25] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the Tenth Annual Symposium on Computational Geometry*, SCG '94, pages 160–164, New York, NY, USA, 1994. ACM. URL: http://doi.acm.org/10.1145/177424.177609, doi:10.1145/177424.177609.

[26] Congqi Xia, Jin Hai, Yihao Dai et al. URL: https://github.com/milvus-io/milvus.

[27] K. Dahiya, D. Saini, A. Mittal, A. Shaw, K. Dave, A. Soni, H. Jain, S. Agarwal, and M. Varma. Deepxml: A deep extreme multi-label learning framework applied to short text documents. In *Proceedings of the 14th International Conference on Web Search and Data Mining*, WSDM '21, New York, NY, USA, 2021. Association for Computing Machinery.

[28] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL: http://arxiv.org/abs/1810.04805, arXiv:1810.04805.

[29] L. Dhulipala, M. Hadian, R. Jayaram, J. Lee, and V. Mirrokni. Muvera: Multi-vector retrieval via fixed dimensional encodings, 2024. URL: https://arxiv.org/abs/2405.19504, arXiv:2405.19504.

[30] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 577–586, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1963405.1963487, doi:10.1145/1963405.1963487.

[31] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. The faiss library. 2024. arXiv:2401.08281.

[32] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Return of the lernaean hydra: Experimental evaluation of data series approximate similarity search. *Proc. VLDB Endow.*, 13(3):403–420, 2019. URL: http://www.vldb.org/pvldb/vol13/p403-echihabi.pdf, doi:10.14778/3368289.3368303.

[33] H. V. S. et al. URL: https://github.com/harsha-simhadri/big-ann-benchmarks/blob/main/benchmark/datasets.py.

[34] Etienna Dilocker, Marcin Antas, Dirk Kulawiak et al. URL: https://github.com/weaviate/weaviate.

[35] E. Fix and J. L. Hodges. Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review / Revue Internationale de Statistique*, 57(3):238–247, 1989. URL: http://www.jstor.org/stable/1403797.

[36] C. Fu and D. Cai. URL: https://github.com/ZJULearning/efanna.

[37] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graphs. *PVLDB*, 12(5):461 – 474, 2019. URL: http://www.vldb.org/pvldb/vol12/p461-fu.pdf, doi:10.14778/3303753.3303754.

[38] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan, A. Singh, and H. V. Simhadri. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*, WWW '23, page 3406–3416, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3543507.3583552`.

[39] J. Guo, Y. Cai, Y. Fan, F. Sun, R. Zhang, and X. Cheng. Semantic models for the first-stage retrieval: A comprehensive review. *ACM Trans. Inf. Syst.*, 40(4), Mar. 2022. `doi:10.1145/3486250`.

[40] J. Guo, Y. Cai, Y. Fan, F. Sun, R. Zhang, and X. Cheng. Semantic models for the first-stage retrieval: A comprehensive review. *ACM Transactions on Information Systems (TOIS)*, 40(4):1–42, 2022.

[41] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 3887–3896. PMLR, 2020. URL: `http://proceedings.mlr.press/v119/guo20h.html`.

[42] HNSW. Hnsw github. URL: `https://github.com/nmslib/hnswlib/blob/master/examples/python/EXAMPLES.md`.

[43] 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*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM. URL: `http://doi.acm.org/10.1145/276698.276876`, `doi:10.1145/276698.276876`.

[44] P. Indyk and H. Xu. Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations, 2023. URL: `https://arxiv.org/abs/2310.19126`, `arXiv:2310.19126`.

[45] M. Iwasaki. URL: `https://github.com/yahoojapan/NGT/wiki`.

[46] M. Iwasaki and D. Miyazaki. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data, 10 2018.

[47] H. Jégou, M. Douze, and C. Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, Jan. 2011. URL: `https://hal.inria.fr/inria-00514462`, `doi:10.1109/TPAMI.2010.57`.

[48] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[49] Jonathan Ellis et al. URL: `https://github.com/jbellis/jvector`.

[50] O. Khattab and M. Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '20, page 39–48, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3397271.3401075`.

[51] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut: A scalable bottom-up approach for building data series indexes. *Proceedings of the VLDB Endowment*, 11, 03 2018. `doi:10.14778/3184470.3184472`.

[52] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL: https://arxiv.org/abs/2005.11401, arXiv:2005.11401.

[53] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2020. doi:10.1109/TKDE.2019.2909204.

[54] W. Lu, J. Jiao, and R. Zhang. Twinbert: Distilling knowledge to twin-structured compressed bert models for large-scale retrieval. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2645–2652, 2020.

[55] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *CoRR*, abs/1603.09320, 2016. URL: http://arxiv.org/abs/1603.09320, arXiv:1603.09320.

[56] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[57] Matvey Arye et al. URL: https://github.com/timescale/pgvectorscale.

[58] M. McInnes. URL: https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/storage-optimized/lsv3-series?tabs=sizebasic.

[59] MongoDB. URL: https://www.mongodb.com/docs/atlas/atlas-vector-search/vector-search-overview/.

[60] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

[61] URL: https://www.oracle.com/database/ai-vector-search/.

[62] R. Panigrahy, K. Talwar, and U. Wieder. Lower bounds on near neighbor search via metric expansion, 2010. URL: https://arxiv.org/abs/1005.0418, arXiv:1005.0418.

[63] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia. Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. *Proc. ACM Manag. Data*, 2(3), May 2024. doi:10.1145/3654923.

[64] Pavan Davuluri. Windows Copilot Runtime. URL: https://blogs.windows.com/windowsdeveloper/2024/05/21/.

[65] N. Pentreath, A. Abdurakhmanov, and R. Royce, 2017. URL: https://github.com/MLnick/elasticsearch-vector-scoring.

[66] URL: https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/how-to-use-pgdiskann.

[67] URL: https://www.pinecone.io/.

[68] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision, 2021. URL: https://arxiv.org/abs/2103.00020, arXiv:2103.00020.

[69] ROCKSET. URL: https://rockset.com/vector-search/.

[70] SCANN. Scann github. URL: https://github.com/google-research/google-research/blob/master/scann/docs/algorithms.md.

[71] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri. Freshdiskann: A fast and accurate graph-based ANN index for streaming similarity search. *CoRR*, abs/2105.09613, 2021. URL: https://arxiv.org/abs/2105.09613, arXiv:2105.09613.

[72] M. Sokolov, 2020. URL: https://issues.apache.org/jira/browse/LUCENE-9004.

[73] S. J. Subramanya, F. Devvrit, R. Kadekodi, R. Krishnawamy, and H. V. Simhadri. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 13748–13758, 2019.

[74] K. Sugawara, H. Kobayashi, and M. Iwasaki. On approximately searching for similar word embeddings. pages 2265–2275, 01 2016. doi:10.18653/v1/P16-1214.

[75] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, Sept. 2013. doi:10.14778/2556549.2556574.

[76] J. Tibshirani, 2019. URL: https://www.elastic.co/blog/text-similarity-search-with-vectors-in-elasticsearch.

[77] Vespa. URL: https://vespa.ai.

[78] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-nn graph construction for visual descriptors. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1106–1113, June 2012. doi:10.1109/CVPR.2012.6247790.

[79] M. Wang, X. Xu, Q. Yue, and Y. Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *CoRR*, abs/2101.12631, 2021. URL: https://arxiv.org/abs/2101.12631, arXiv:2101.12631.

[80] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. URL: http://dl.acm.org/citation.cfm?id=645924.671192.

[81] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Proceedings of the 21st International Conference on Neural Information Processing Systems*, NIPS'08, page 1753–1760, Red Hook, NY, USA, 2008. Curran Associates Inc.

[82] L. Xiong, C. Xiong, Y. Li, K.-F. Tang, J. Liu, P. Bennett, J. Ahmed, and A. Overwijk. Approximate nearest neighbor negative contrastive learning for dense text retrieval, 2020. URL: https://arxiv.org/abs/2007.00808, arXiv:2007.00808.

[83] Yannis Papakonstantinou, Alan Li, Ruiqi Gao, Sanjiv Kumar, Phil Sun. ScaNN for AlloyDB. URL: https://services.google.com/fh/files/misc/scann_for_alloydb_whitepaper.pdf.

[84] Q. Zhang, S. Xu, Q. Chen, G. Sui, J. Xie, Z. Cai, Y. Chen, Y. He, Y. Yang, F. Yang, M. Yang, and L. Zhou. VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 377–395, Boston, MA, July 2023. USENIX Association. URL: `https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi`.

[85] B. Zheng, X. Zhao, L. Weng, N. Q. V. Hung, H. Liu, and C. S. Jensen. Pm-lsh: A fast and accurate lsh framework for high-dimensional approximate nn search. *Proc. VLDB Endow.*, 13(5):643–655, Jan. 2020. `doi:10.14778/3377369.3377374`.