Harmonizing Efficiency and Accuracy in Filtered Vector Search

Zixiang Zhou
Massachusetts Institute of Technology
zpeter@mit.edu

Xuhao Chen
Michigan State University
cxh@msu.edu

ABSTRACT

Approximate nearest neighbor search (ANNS) is increasingly essential for AI-driven applications. In practice, ANNS is often coupled with label filtering to improve accuracy, relevance, and efficiency, a variant known as filtered ANNS or FANNS. Existing FANNS methods fall short in efficiency or accuracy. This is because, first, searching on a label-mixed index often converges to local minima due to label interference. Second, isolating and duplicating labels in the index inflates the index and incurs prohibitive memory overhead. Third, current join-based strategies for multi-filter search perform substantial wasted computation on unpromising candidates.

We present BigFANN, a label-aware FANNS framework that achieves high speed and space efficiency, while retaining high accuracy. Our framework features a hybrid indexing scheme that introduces heterogeneous edge types and flexible IVF-graph indexing. Specifically, we construct graph indices with tunable combinations of *exclusive* and *shared* edges based on label characteristics. This method effectively minimizes label interference for single-filter search under a certain memory budget, ensuring high search accuracy and speed. In addition to heterogeneous edges, we adopt a hybrid of graph and IVF indices to deal with single- and multi-filter queries. Particularly, for multi-filter searches, we propose a join-free search strategy to eliminate wasted computation in the existing join-based strategy. Experimental results on various datasets show that BigFANN significantly outperforms state-of-the-art FANNS frameworks, UNG and ParlayIVF^2, by up to 786× and 4× respectively, while achieving the same or better accuracy.

PVLDB Reference Format:
Zixiang Zhou and Xuhao Chen. Harmonizing Efficiency and Accuracy in Filtered Vector Search. PVLDB, 19(9): 2059 - 2072, 2026.
doi:10.14778/3819518.3819534

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zhouzixiang2004/BigFANN.

1 INTRODUCTION

Recent advancements in machine learning have driven the use of vector databases [10, 21, 28, 48, 52, 53, 59] for managing embedding vectors. For example, given a user query, Retrieval Augmented Generation (RAG) retrieves the most relevant information or documents in the database, so the model can rapidly find relevant pieces of information to add to its context during generation, and thus provide accurate domain-specific or time-sensitive responses.

Approximate nearest neighbor search (ANNS) is the key algorithm in vector databases. In real-world applications, the query context is often domain- or user-specific, and thus retrieving from the entire database wastes bandwidth and harms accuracy. So practical ANNS tasks often involve some label constraints in addition to vector similarity, a problem known as *filtered* ANNS or FANNS [18]. FANNS allows us to retrieve only from the relevant subset, which potentially improves accuracy, relevance, and efficiency.

While the typical ANNS problem has been intensively studied and optimized over the years, we observe that FANNS is under-explored and still extremely inefficient. For example, to answer 10K queries in `arxiv`, a real-world FANNS dataset with only 2.7 million vectors, on a 32-core CPU, it takes FilteredDiskANN [18], an existing FANNS solution, 20 seconds to complete search with a best recall below 19%. Similarly, for a 10M-scale real dataset `yfcc10m` [3], another state-of-the-art solution UNG [4] can serve only 110 queries per second (QPS) on that CPU, with a recall of less than 63%.

We investigate when and why existing FANNS approaches suffer low accuracy and/or poor search efficiency. First, most state-of-the-art methods, including FilteredDiskANN, UNG, and NHQ [60], all build a single, unified proxy graph, in which vertices with various labels are all mixed in the unified index and *interfere* with each other. We observe that search on a unified graph often stops far from the true nearest neighbors with no closer vertices to explore, i.e., *local minima*. Fig. 1 shows a toy example. Each vertex here has a single label shown as its color[1]. Looking for a blue-colored query, many edges are filtered away if the neighboring vertex is not blue. This poor connectivity results in not only an inefficient search path, but also local minima. Label interference, detailed in §3.1, is a unique challenge in FANNS: ANNS indices are generally built based on the spatial 'distance' information, while the 'labels' reflect semantic properties that may conflict with the spatial information.

On the other hand, ParlayIVF^2 [35] adopts an isolation scheme by building a dedicated ANN index for every possible label. While separating labels avoids label interference and achieves high performance, it incurs prohibitive indexing costs, with a memory *inflation* ratio of M, where M is the average number of labels per node. In real datasets, M can easily go beyond 10, as in `yfcc10m` (see §3.2).

Further, in the multi-filter scenario, i.e., when the query applies more than one filter, the search is even more challenging. Existing graph-based approaches are more likely to fall into local minima, given the worse graph connectivity cut by multiple filters (details in §3.3). On the other hand, existing join-based methods are also inherently inefficient. In this method, first, a set of candidates for each of the filters are collected. Then, a join (i.e., set intersection) operation is performed to merge the candidate sets, retaining those

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819534

[1] Edges connecting different colors can exist if their endpoints share some universal label (not shown), since each vertex may carry more than one label in real datasets.

that satisfy all filters. Since filtering in this method is performed implicitly in the join, it has to conservatively collect a large body of candidates for every filter before the join, to meet the target recall. Consequently, it produces significant overhead on intermediate results and unnecessary distance computations (detailed in §3.3).

Based on the above understanding of the tradeoffs, we build a *label-aware* FANNS framework, BigFANN, to improve search efficiency while ensuring high accuracy. Our key idea is to build a hybrid indexing scheme that adaptively balances search speed and space overhead for different scenarios. However, naively combining existing methods does not yield improved efficiency, as they are essentially two extremes and incompatible. In our design, we first introduce heterogeneous edge types: *exclusive* and *shared* edges (§4). While exclusive edges isolate labels and avoid interference, shared edges provide sharing across nodes with many overlapping labels, minimizing space overhead. We introduce parallel indexing construction and search algorithms on top of the heterogeneous graph, which effectively build and search the index with high throughput.

Second, we employ a hybrid of graph-based and clustering-based indices for single-filter and multi-filter queries. This design choice is based on our observation in FANNS workloads: single-filter queries often yield dense top-k results, benefiting from *fine-grained* graph search; multi-filter queries have much fewer data points passing the filter with the top-k results more spread out, and thus, they benefit from *coarse-grained* gathering methods, where a large number of candidates are gathered at once. By separating the query types, we can tailor the index and search strategy to the query structure, resulting in not only better accuracy but also higher search speed.

Specifically, for multi-filter queries, we use clustering-based, i.e., IVF (inverted file), indices to avoid label interference in graph indices, and thus retain high throughput and high accuracy. Nevertheless, naively applying this indexing is inefficient for multi-filter queries, due to wasted distance computation on unpromising candidates introduced by the join operation. To improve search efficiency, we propose a novel join-free strategy that eliminates wasted computation (§5). Instead of gathering candidates for individual filters concurrently, we gather candidates only from the smallest filter, which has fewer matches than the other filters. We then explicitly apply the other filters to prune directly on the candidates gathered in the first step. This way, we avoid the large overhead caused by join, and can potentially reduce distance computations.

We implement BigFANN in C++/OpenMP on multicore CPUs, and evaluate it on various real-world and synthetic datasets. Experimental results show that BigFANN outperforms state-of-the art FANNS frameworks, UNG and ParlayIVF[2], by up to 786× and 4× respectively, while achieving the same or better accuracy.

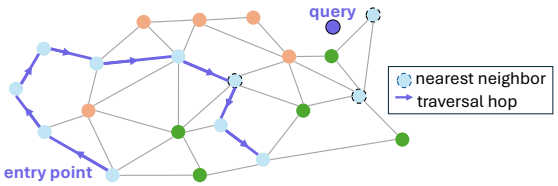


Figure 1: Existing mixed-index FANNS methods run into local minima. Unlike Fig. 2(b) where the search goes straight to the query, this filtered search path is convoluted and inefficient due to label interference. More importantly, it converges at a local minimum with only 1 out of 3 nearest neighbors found, i.e., a recall of 33%.

This paper makes the following contributions:

- We identify the unique challenges in FANNS, namely, (1) label interference, (2) index inflation, and (3) wasted computation in the join-based multi-filter strategy.
- We introduce heterogeneous edge types, i.e., *exclusive* and *shared* edges, to avoid label interference while striking a balance between search speed, accuracy, and space cost in FANNS search.
- We propose a novel join-free strategy for multi-filter FANNS tasks. Our strategy employs an explicit filtering step, which eliminates wasted computation in prior join-based methods.
- We build BigFANN, a label-aware FANNS framework that implements our hybrid indexing with heterogeneous edges, the join-free strategy, and performance optimizations. Experiments show significant improvements in efficiency and accuracy.

2 BACKGROUND

We introduce ANNS in §2.1, existing indexing and search algorithms in §2.2, and the filtered ANNS problem in §2.3.

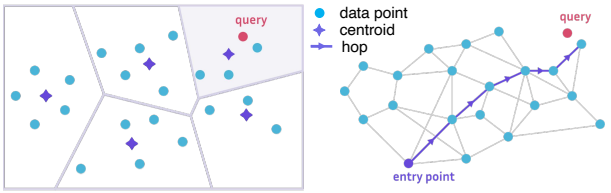
2.1 Vector Database and ANNS

Vector databases [33, 54] have emerged as the computational engines for effective interaction with vector embeddings in applications like knowledge search and recommendation. Industry efforts include Faiss [10], cuVS [41], Milvus [59], Pinecone [53], and more [9, 11, 23, 46, 47, 55, 57, 58, 64]. The core computation in vector databases is the vector similarity search, which finds the top-k "similar" vectors, i.e., nearest neighbors, to a given query. A brute-force search involves computing distances with billions of vectors in the database, which is impractical. Due to the curse of dimensionality [27], there is no good solution to sort high dimensional vectors monotonically for fast search. Instead, Approximate Nearest Neighbor Search (ANNS) is used to speed up the search, at the cost of accuracy loss (e.g., 5% error). At the core of ANNS is an offline-built *index* [14, 25, 28, 38, 42], which is used to navigate the online search, to quickly locate a small region in the database where it is the most promising to find candidates similar to the query. In addition, quantization [1, 6, 15, 16, 19, 29, 56, 61] has been widely used to save memory space and also accelerate the search.

2.2 ANN Indexing and Search Algorithms

The choice of indexing method can significantly impact the search performance, accuracy, and resource consumption. As shown in Fig. 2, commonly used indexing methods fall into two primary categories: clustering-based and graph-based indexing [63].

2.2.1 Clustering-based Indexing. This method is also known as IVF (inverted file). It groups nearby data points into the same clusters [17, 24, 48, 66], and the index contains a mapping from cluster centroids to nodes in the cluster. The index is space-efficient and works well in lower-dimensional spaces. However, its performance tends to decline in higher-dimensional spaces, with decreased recall or increased search work. During the search, a query is first compared to cluster *centroids*, to find the nearest or most relevant clusters. Then, only the relevant clusters are fully searched, and the closest points from these clusters are returned.



(a) clustering-based ANNS, a.k.a IVF (b) graph-based ANNS

Figure 2: Clustering-based vs. graph-based ANNS index and search.

2.2.2 Graph-based Indexing. This method constructs a graph where data points are vertices, and pairs of vertices can be connected if they are close. Beam search is used to traverse this graph, with a priority queue to keep the top-B vertices (B is the beam size) ranked by their distances to the query. The search starts with inserting a few seed vertices into the queue. It then progresses by expanding the highest priority vertex from the queue, examining its neighbors, and evaluating their distances to the query. Close neighbors are then inserted into the queue. Note that B is a key parameter that influences the breadth and depth of the search, and a larger capacity gives higher recall. Hence, B is often varied to meet a required recall. As the geometry of the points are abstracted as edges in the graph, this approach is less sensitive to the dimension of the data.

HNSW [38] and Vamana [28] are popular graph indices. The graph index construction decides which data points, or vertices, should be connected by edges. It involves multiple competing objectives, like space consumption, construction time, search speed and accuracy. In modern ANNS systems, the graph is constructed heuristically. Most of the existing heuristics fall into three categories. A high-quality graph (1) is well-connected, (2) effectively embeds distance information, and (3) is not too dense. The connectivity ensures that every data point can be easily reached from other points, avoiding local minima during the search. The distance information should be well embedded to ensure high search efficiency, i.e., the search is navigated by this information, and hopefully, every step gets closer to the query. The density of the graph is closely related to connectivity and thus accuracy, but also proportional to the computational work in search, as well as the storage consumption. Besides, for simplicity and efficiency, most of the existing solutions choose to build regular graphs, i.e., every vertex has the same number of neighbors (or degrees).

2.3 Filtered ANNS (FANNS)

In plain ANNS, we have N $\mathcal{D}$-dimensional data points, i.e., vectors, $x_i \in \mathbb{R}^{\mathcal{D}}$. Given some query vectors x_q, we want to retrieve the top-k vectors that minimize the distance $\text{dist}(x_i, x_q)$, according to some distance metric (e.g., Euclidean distance). However, this formulation does not support more complex queries required in real-world settings. For instance, a user might wish to search only among objects that have one or more metadata labels (e.g., papers matching the attributes 2026 and VLDB). This is known as filtered ANNS or FANNS [4, 12, 18, 22, 60]. Filtering restricts which vectors are eligible based on metadata or contextual constraints.

FANNS is particularly critical for RAG. Without this, the retrieved passages might be semantically similar but irrelevant to the context, leading to hallucinations or contradictions in the generated answer. FANNS is also well motivated by practical constraints in search and recommendation, where it captures realistic scenarios that combine vector search with keyword filters. For instance, in image search, a user may only be interested in images taken in a certain location. In product recommendation, the user may want to view items with certain specifications of size, make, and model. More broadly, hybrid keyword-semantic search is a core task in many modern applications. Leading vector search platforms such as Weaviate [64] and Pinecone [53] offer "hybrid search" as a service, where retrieval asks for hard keyword matches with vector distance based on dense LLM-generated embeddings.

Formally, in FANNS, each vector $x \in P$ has an associated set of labels f_x. Each query vector x_q is associated with a set of *query filters* f_q. We want to retrieve vectors that pass all the query filters, that is, $f_q \subseteq f_x$. We call the data points that pass the query filter(s) *relevant*. The set of relevant data points $P_{f_q} = \{x \in P : f_q \subseteq f_x\}$ is the *relevant search space* of FANNS. Following the Big-ANN Challenge [3], we focus on single-filter and double-filter queries, i.e., $|f_q| = 1$ or 2, which are most challenging—beyond that, it is feasible to find nearest neighbors simply by a brute-force search, since usually only a few results can pass three or more filters. Following [18], we define the *specificity* of f_q to be $|P_{f_q}|/|P|$: the fraction of indexed data points having the labels f_q associated with them.

There exist two FANNS approaches. One approach, used by FilteredDiskANN, NHQ, and UNG, is to build a single, unified proxy graph, in which nodes are connected to neighbors that are both spatially close and with similar labels. This works well for *trivial* FANNS tasks, i.e., the average number of labels per node M is small, e.g., $M = 1$. However, we show in §3 that this approach suffers poor accuracy when the tasks are non-trivial, i.e., $M > 1$. The other approach, adopted by ParlayIVF[2], is to build a dedicated index for every possible label. While this isolation scheme improves accuracy and performance, it incurs prohibitive indexing costs and blows up memory for large datasets.

# data points	N
# distinct labels	$\lvert\mathcal{L}\rvert$
avg # labels per node	M
label distribution	Φ
# vector dimensions	$\mathcal{D}$
graph degree	Δ

3 FANNS CHALLENGES & TRADEOFF

We identify the following three major obstacles that prevent existing solutions [4, 18, 35, 60] from being adopted in practice.

3.1 Label Interference

A unique challenge in FANNS is that different labels interfere with each other during graph traversal. This is especially true if we construct a single unified graph index with all labels mixed, a strategy used in FilteredDiskANN [18], UNG [4] and NHQ [60]. For example, in Fig. 2(b), since there are no filters or labels, the search is straightforward, takes 5 hops to finish, and successfully finds all the top-3 nearest neighbors. However, in Fig. 1, the same dataset but with labels, the traversal path is tortuous, takes 8 hops to complete, and only 1 of 3 nearest neighbors is found. This is because the search is pruned or 'filtered' on-the-fly by the label matching constraint. We call this effect caused by the mixed-label index *"label interference."*

Theoretically, label interference becomes more severe as the average number of labels M increases. Consider a vertex u with

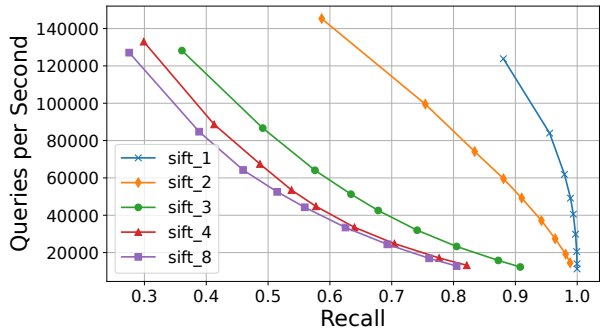


Figure 3: FilteredDiskANN (`StitchedVamana`) performance with $M = 1, 2, 3, 4, 8$ on the `sift1m-s` dataset described in Table 1. We observe a significant drop in performance as M increases, due to label interference. The same trend is observed for UNG and NHQ.

degree Δ in a mixed-label graph. Although the index is constructed with degree Δ, for a given query only a subset of these edges is usable, since many neighbors may not satisfy the query's label constraint. One might even encounter edges (u, v) with $|f_u \cap f_v| = 1$. Such an edge is usable by only a single query filter, namely the one label common to both f_u and f_v. In this worst case, the expected number of *relevant* neighbors (that match the query filter constraint) per query drops to about $\Delta/|f_u|$, which can be drastically smaller than Δ, leading to poor effective graph connectivity.

In practice, the effect is slightly mitigated because edges can be shared across multiple labels ($|f_u \cap f_v| > 1$). Nevertheless, the loss of usable edges is a key limitation of mixed-label indexing. Experimentally, Fig. 3 shows the performance of the StitchedVamana algorithm proposed by FilteredDiskANN [18] on the `sift1m` dataset with synthetic labels. We vary M, the average number of labels per node, from 1 to 8. Each data point has M labels generated uniformly at random from 20 possibilities. All indices are built with the same degree of 32, so the memory usage across the datasets is identical. We observe a large drop in performance as M increases from 1 to 3 due to the label interference problem.

Note that UNG [4] proposes a different indexing strategy from FilteredDiskANN, but it does not address the label interference problem. It only adds edges (u, v) where $f_u \subseteq f_v$, so that certain subgraphs of the index correspond exactly to the subset of data points passing a query filter. Consider a dataset where all label sets have the same size and no two data points have the same set. Then *no* pairs (u, v) satisfy $f_u \subseteq f_v$, so UNG will not put any edges in the index, degenerating into either a brute-force search or achieving very poor recall, depending on how many root nodes the search is initialized at. Furthermore, we note that such a dataset is not overly contrived: it can arise in practice from reformulating attributes, as in the NHQ datasets [60], in terms of labels.

Consequently, FilteredDiskANN, NHQ, and UNG all scale poorly in the *many-label* scenario, where each point has more than one label. This is a common issue as real-world scenarios often involve many possible labels. For instance, the `yfcc10m` dataset used in the NeurIPS Big-ANN Challenge has 200,386 distinct labels and an average of 10.8 labels per data point. As we find in Fig. 6 (in §6), all these mixed-index approaches struggle to achieve any practical throughput-accuracy balance in the many-label scenario with M>1.

3.2 Index Inflation

Other than building a unified index, another approach is to build a fully isolated index, i.e., build a separate graph for every distinct label. This approach, used by ParlayIVF^2 [35], avoids label interference, but at the cost of space overhead. A data point x with label set f_x will need to appear in $|f_x|$ different graphs. Since each copy of the node stores Δ neighbors, the total memory required is $|f_x| \cdot \Delta$ edges per data point. This represents a $|f_x|$-fold increase compared to a mixed index and a Δ-fold increase relative to the size of the raw label information. Such a memory blowup is not scalable: The `yfcc10m` dataset with 10 million data points has an average $|f_x|$ of 10.8, resulting in a 10.8× space blowup compared to mixed indices; for even larger datasets, it will quickly exceed a reasonable memory budget. We refer to this phenomenon as *index inflation*.

One might attempt to mitigate inflation by lowering the graph degree Δ. To some extent, this was done in ParlayIVF^2, which precisely sets a somewhat low value of $\Delta = 8$, 10, or 12 for `yfcc10m` depending on the size of a label group. However, this approach is not scalable either, as a lower degree reduces the navigability of the graph, making searches less effective. Furthermore, reducing Δ uniformly sacrifices connectivity even for dense or overlapping label groups, a source of inefficiency that is not acceptable.

Finally, separate indexing may contain redundancy. In an extreme case where all points share the same label set f_x, $|f_x|$ identical copies of the same graph are stored, wasting memory without improving accuracy. Even in practice, it is common to have correlated labels, where consequently there is overlap between the information stored in the separate graphs.

3.3 Wasted Computation in Multi-Filter Search

As we have seen in Fig. 1, many of the neighbors visited during the graph traversal do not pass the query filter. This is further exacerbated under the multi-filter setting: on `yfcc10m`, a single-filter query matches 304,233 points on average, while a double-filter query has only 3,727 matches on average. Without special designs, a unified graph may suffer even worse connectivity, which in turn results in frequent local minima and poor accuracy. A workaround is to trade speed for accuracy. For example, UNG uses a significantly large beam (e.g., $B = 3000$) and initializes the search at many different entry points to ensure that all candidate nodes are reachable. Consequently, it becomes extremely slow to compute on large datasets (see Fig. 7). Note that FilteredDiskANN and NHQ are tailored toward single-filter queries with $|f_q| = 1$. Directly using their indices for multi-filter queries results in even worse performance.

The other approach, *label-wise isolation*, is efficient only when used for single-filter queries, as each query search is confined exactly within its relevant search space. Its efficiency drops sharply when it comes to the multi-filter scenario. To similarly confine the search in the relevant search space, we have to build a separate index for every possible *combination of labels* in the database, which becomes impractical for large datasets.

If we do not precompute all combinations, it has to be done online. The existing method [35] first concurrently gathers candidates from each of the separate indices (one per label), and then joins the results using set intersection. In this case, a large number of candidates must be conservatively collected, even though most of them do

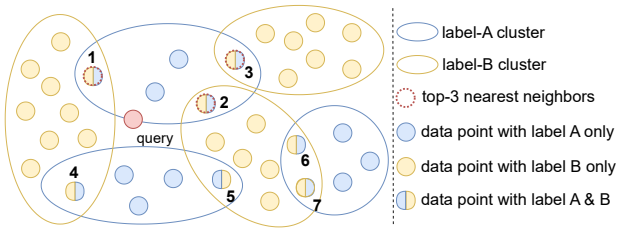


Figure 4: A toy example of double-filter k-NNS using IVF (k=3). In the join-based method, conservatively, nodes in all clusters have to be collected, and 7 nodes pass the join and need distance computations.

not pass the given multiple filters. This conservative gathering creates a large amount of overhead and wasted computation on unpromising candidate data points. As shown in Fig. 4, to find the top-3 nearest neighbors, the join-based approach has to collect nodes in all the clusters. It then performs a join on all the label-A and label-B candidates, and finds 7 nodes that pass both filters. Although the top-3 nodes are all in the same label-A cluster, distance computation is required for all 7 nodes.

To summarize, given multiple filters, graph-based indices are inherently inefficient due to the poor connectivity. Label isolation is also impractical because of combinatorial inflation. Another approach is to build separate indices per each label offline, and gather candidates from each label and join them online. But this results in significant wasted computation, and thus is not efficient either.

4 TO SHARE OR NOT: LABEL-AWARE INDEX

FANNS involves a complex tradeoff among search speed, accuracy, and space cost. Existing methods optimize one aspect at the expense of others. This motivates us to combine and adaptively apply indexing strategies depending on label characteristics and the memory budget, to balance these competing objectives. For label interference specifically, we propose a hybrid indexing and search method that adaptively makes tradeoffs under given memory budgets.

4.1 Generalization of Labeled Indexing Schemes

We present a perspective that unifies existing indexing schemes for FANNS. To abstract away implementation details, we focus only on the information contained in the graph edges, rather than whether nodes are intuitively duplicated or how the data is physically stored.

Fully isolated index. If space were not a concern, the most direct construction would be the *fully isolated index*: for every label ℓ, build an optimal proximity graph on the subset of points containing ℓ, using a graph degree Δ which is as large as is necessary to ensure good connectivity. Each query will have access to a proximity graph tailored to the subset of data points passing the query filter, thus enabling performance similar to classical ANNS without searching any irrelevant points. This approach is used by *ParlayIVF*[2]. However, it suffers from space blowup: A data point with label set f_u needs to store a total of $|f_u| \cdot \Delta$ neighbors across the collection of graphs. On large datasets, this becomes infeasible due to space constraints.

Share-remove framework. The $|f_u| \cdot \Delta$ neighbors required by each node of the fully isolated index typically contain more information than is necessary to achieve good performance. Neighbors chosen for multiple labels can be *shared* across multiple graphs in the collection, and neighbors that do not contribute enough to the graph connectivity can be *removed*, potentially significantly saving space with minimal drop in performance. We propose to view all existing approaches as different ways to remove or share edges from some fully isolated index, as we now describe:

- *FilteredDiskANN* [18] proposes two indexing strategies, FilteredVamana and StitchedVamana. FilteredVamana is a modification of StitchedVamana which speeds up index construction by sacrificing a bit of graph quality, so we only describe StitchedVamana here. This approach first glues together the graphs corresponding to different labels, effectively building the entire fully isolated index, and then applies a pruning rule to decide which edges to keep. Edges that appear in multiple label-specific graphs become shared edges, while pruned edges are removed. A query only traverses nodes and edges consistent with its label, so the search effectively operates on a subgraph of the fully isolated index.
- *NHQ* [60] considers a filtered search formulation using *attributes*, where each vector has the same number of attributes. In our label formulation, each distinct attribute can correspond to a distinct label, so all label sets f_u have the same size. NHQ builds a graph using a fusion distance of the form
$$\Gamma(u, v) = \text{dist}(x_u, x_v) + w \cdot |f_u \oplus f_v|,$$
where f_u and f_v are the label sets of nodes u and v, and $\oplus$ denotes symmetric difference. When all $|f_u|$ are equal, it is equivalent to
$$\text{dist}(x_u, x_v) - w \cdot |f_u \cap f_v|,$$
up to a constant shift. Therefore, NHQ prioritizes edges that are both short (i.e., smaller $\text{dist}(x_u, x_v)$) and shared among many groups (i.e., larger $|f_u \cap f_v|$). In this sense, NHQ explicitly chooses edges that can be shared across multiple label groups.
- *UNG* [4]. The key design choice of UNG is to only include edges (u, v) where $f_u \subseteq f_v$, so that for a label ℓ, the subset of nodes reachable from a root node $\{\ell\}$ corresponds exactly to the subset of data points passing the query filter ℓ. Thus, UNG removes all edges (u, v) where $f_u \not\subseteq f_v$, and shares edges where $f_u \subseteq f_v$ across all label groups in f_u.
- *Baseline: degree reduction.* A simple baseline is to uniformly reduce the graph degree Δ until the $|f_u| \cdot \Delta$ blowup is acceptable. This strategy removes edges without ever sharing them.

Viewed through this perspective, all existing approaches can be understood as particular strategies for this edge removal/sharing tradeoff. However, a fundamental conflict is omitted in prior designs: the edge sharing and removal may significantly affect the graph connectivity, so as to affect the search accuracy. Therefore, the central design choice in FANNS index design is: *How can we remove or share edges between label groups so as to minimize the loss of connectivity?* While each prior method has its own strengths, none achieves the best tradeoff universally. We propose a *hybrid indexing* strategy to address this problem.

4.2 Hybrid Indexing with Heterogeneous Edges

Our hybrid indexing adaptively optimizes index size versus search efficiency for different input data characteristics. For each label ℓ, we gather a list V_ℓ of all vectors that contain label ℓ. Let γ be a cutoff threshold. We call labels with $|V_\ell| < \gamma$ *trivial*, denoted as $\mathcal{L}_t$,

Algorithm 1 Hybrid Graph Construction

Require: Dataset $x_1, \ldots, x_N$ with significant label set $\mathcal{L}_s$, exclusive degrees $\{\Delta_e[\ell]\}_{\ell \in \mathcal{L}_s}$, shared degree Δ_s, beam width B.
Ensure: Hybrid index $G = \left(\{G_e^\ell\}_{\ell \in \mathcal{L}_s}, G_s\right)$.

Initialize empty graphs $\{G_e^\ell\}_{\ell \in \mathcal{L}_s}$ and G_s
Let σ be a random permutation of $1, \ldots, N$
Initialize $init[\ell]$ to first vertex with label ℓ in σ for every $\ell \in \mathcal{L}_s$
$start \leftarrow 1$
while $start \le N$ **do**
  $end \leftarrow \min(2 \cdot start, start + 0.01N, N)$
  BatchInsert($start, end$) ▷insert nodes $\sigma[start, \ldots, end]$
  $start \leftarrow end + 1$ ▷move onto next batch
return $G = \left(\{G_e^\ell\}_{\ell \in \mathcal{L}_s}, G_s\right)$

function BatchInsert($start, end$)
  parallel for $i = start$ **to** end **do** ▷process batch in parallel
    $u \leftarrow \sigma[i]$ ▷add edges from u to previous batches
    for each significant label $\ell \in f_u$ **do**
      $\mathcal{V} \leftarrow$ BeamSearch(G=($\{G_e^\ell\}_{\ell \in \mathcal{L}_s}, G_s$), $u, \ell, init[\ell], B$) ▷BeamSearch towards x_u with filter ℓ starting at $init[\ell]$
      for each $v \in \mathcal{V}$ **do** ▷$\mathcal{V}$ contains visited nodes
        if $|f_u \cap f_v| \ge \omega$ **then** ▷add to shared
          $G_s[u] \leftarrow G_s[u] \cup \{v\}$
        else ▷add to exclusive
          $G_e^\ell[H_\ell(u)] \leftarrow G_e^\ell[H_\ell(u)] \cup \{H_\ell(v)\}$
      PruneShared(u) ▷prune shared edges
    for each significant label $\ell \in f_u$ **do**
      PruneExclusive($\ell, H_\ell(u)$) ▷prune exclusive edges

  for $i = start$ **to** end **do** ▷add reverse edges
    $u \leftarrow \sigma[i]$ ▷add edges from previous batches to u
    for each significant label $\ell \in f_u$ **do**
      for all $v \in G_e^\ell[u]$ **do** ▷exclusive edge
        $G_e^\ell[H_\ell(v)] \leftarrow G_e^\ell[H_\ell(v)] \cup \{H_\ell(u)\}$
    for all $v \in G_s[u]$ **do** ▷shared edge
      $G_s[v] \leftarrow G_s[v] \cup \{u\}$
  parallel for distinct v where $G_s[v]$ was updated **do**
    PruneShared(v) ▷prune shared edges

  parallel for distinct (v, ℓ) where $G_e^\ell[H_\ell(v)]$ updated **do**
    PruneExclusive($\ell, H_\ell(v)$) ▷prune exclusive edges

and otherwise *significant*, denoted as $\mathcal{L}_s$. To handle queries with $f_q = \{\ell\}$, we either enumerate all vectors in V_ℓ for trivial labels, or perform a beam search on the graph for significant ones. Discussion below focuses on $\mathcal{L}_s$. Our index has two tunable parameters Δ_e (exclusive) and Δ_s (shared) and stores three types of edges:

- *Exclusive edges.* An edge $e = (u, v)$ is an exclusive edge if its two endpoints share a common label ℓ, i.e., $\ell \in f_u \cap f_v$. Exclusive edges are used for one particular filter ℓ. We store a graph G_e^ℓ for each label group V_ℓ consisting of exclusive edges, with a maximum degree of Δ_e (possibly depending on $|V_\ell|$).
- *Shared edges.* An edge $e = (u, v)$ is a shared edge if its two endpoints share multiple common labels, i.e., $|f_u \cap f_v| \ge \omega$, where ω is a threshold always set to 2 to minimize replication. This is because a higher threshold would allow the same exclusive edge to be used in multiple exclusive graphs, resulting in redundancy. Shared edges are used by all query filters. We store these edges in one graph G_s with a maximum degree of Δ_s.
- *Inclusive edges.* A shared edge (u, v) with $f_u \subseteq f_v$ is additionally classified as an inclusive edge.

Algorithm 2 Pruning rules for exclusive edges. Shared edges are similar; the key difference is the highlighted part in Line 9.

1: **function** PruneExclusive(ℓ, u) ▷u is a local node ID
2: **if** $|G_e^\ell[u]| \le \Delta_e[\ell]$ **then return**
3: Sort $v \in G_e^\ell[u]$ by increasing distance $\text{dist}(u, v)$
4: $R \leftarrow$ edges of $G_s[V_\ell[u]]$ whose endpoint v satisfies $\ell \in f_v$
5: $S \leftarrow \emptyset$ ▷new neighbor set
6: **for** $a \in \{1, \alpha\}$ **do** ▷α is a Vamana parameter [28]
7: **for** each $v \in G_e^\ell[u]$ in order **do**
8: **if** $|S| = \Delta_e[\ell]$ **then break** ▷neighbor set full
9: $bad \leftarrow \exists w \in (R \cup S)$ s.t. $a \cdot \text{dist}(w, v) \le \text{dist}(u, v)$
10: **if** not bad **then** $S \leftarrow S \cup \{v\}$ ▷keep this edge
11: $G_e^\ell[u] \leftarrow S$

We show how to build this hybrid index in §4.3. Once built, we handle single-filter queries $f_q = \{\ell\}$ with a beam search using the combined edges of G_e^ℓ and G_s. Specifically, when processing a node, we load Δ_e exclusive neighbors from G_e^ℓ and Δ_s shared neighbors from G_s and push them into the priority queue if they match filter ℓ. Thus, during the beam search, we need to efficiently convert between *global node IDs* (used to index into G_s) and *local node IDs* (used to index into G_e^ℓ). To do this, for each significant label ℓ we precompute a hash table H_ℓ mapping elements of V_ℓ to their position. This enables amortized constant-time conversions between ID types with little additional space overhead.

Note that both ParlayIVF^2 ($\Delta_s = 0$) and FilteredDiskANN ($\Delta_e = 0$) are special cases of our hybrid index. Our approach also has the capability to support one of the advantages of UNG, which is that inclusive edges do not need an additional filter check.

Performance Analysis. Note that a node u has $|f_u| \cdot \Delta_e$ exclusive edges and Δ_s shared edges. Despite a total of $|f_u| \cdot \Delta_e + \Delta_s$ neighbors, only $\Delta_e + \Delta_s$ neighbors are relevant to the query, and loaded to memory. For all the exclusive (but not inclusive) edges, we need to check whether the neighbor matches the query filter. However, this overhead is incurred for only (up to) Δ_s out of the $\Delta_e + \Delta_s$ neighbors, thus reducing one of the inefficiencies of prior systems.

By introducing some exclusive edges, the local minima issue in prior systems is alleviated. Before, if many of the nearest neighbors of a node u do not have label ℓ, then u is likely to not have any edges to unvisited points with label ℓ and thus node u becomes a potential local minimum in the beam search (see Fig. 1). Exclusive edges guarantee at least Δ_e connections between u to other nodes with label ℓ (even if they are further away), ensuring that the beam search can continue from u within the relevant search space.

Storage. The storage used is $O(NM\Delta_e + N\Delta_s)$, i.e., the space required to store separate graphs with degree Δ_e plus a single unified graph with degree Δ_s. In practice, if the memory available is sufficient, we set $\Delta_s = 1$ and choose a large Δ_e for optimal per-label

connectivity. Otherwise, we choose a combination (Δ_e, Δ_s) in the middle of all possible pairs that satisfy the memory budget. This is based on our sensitivity study (not shown) varying Δ_e and Δ_s, which shows that this strategy gives the best tradeoff, and performance is not sensitive to the exact parameter choice.

4.3 Hybrid Index Construction

We describe how to construct the hybrid index in Algorithm 1. Following the common practice in many prior index construction algorithms, our algorithm is incremental, i.e., the graph is constructed by starting with an empty one and incrementally inserting nodes and edges into it. We iterate through batches of nodes (Lines 4-9), and in each iteration, we call BatchInsert (Line 7) to insert a batch of nodes in parallel. Whenever a new node u is inserted, we consider every significant label ℓ node u has (Line 14). We run a filtered beam search towards node u (on the current index graph built from previous batches) to gather a list $\mathcal{V}$ of close nodes that also contain label ℓ (Line 15). We then consider adding edges connecting u to nodes $v \in \mathcal{V}$: If they share ω or more labels, we classify it as a shared edge (Lines 17–18), otherwise we classify it as an exclusive edge (Lines 19–20). These steps connect nodes in the new batch to previous batches.

After all the edges are considered, we run PruneShared in Algorithm 2 to select all the shared edges to keep (Line 21). After the shared edges are selected, we then run PruneExclusive (see Algorithm 2) to choose the exclusive edges (Line 23). Then, to connect previous batches to this new batch, we consider adding the reverse of all edges added within this batch (Lines 25–31). Again, we apply PruneShared (Lines 32–33) to decide the shared edges first before running PruneExclusive (Lines 35–36).

Algorithm 2 shows the pruning rules to selectively keep or prune the edges, i.e., neighbors, from the candidates, to meet the maximum degree requirement. Crucially, in PruneExclusive, we apply the Vamana pruning rule using all the shared edges as well as the exclusive edges in order of distance (Line 9). This ensures that the chosen exclusive edges point in different directions from the shared edges and each other, thus maximizing the usefulness of the selected exclusive edges.

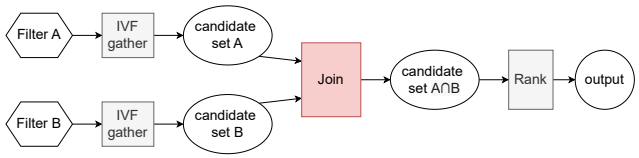
5 REDUCING MULTI-FILTER SEARCH WORK

With heterogeneous edges, we can adaptively balance search speed, accuracy and memory consumption. However, the graph search efficiency is still hampered by the label interference. This issue is exacerbated in the multi-filter scenario as discussed in §3.3.

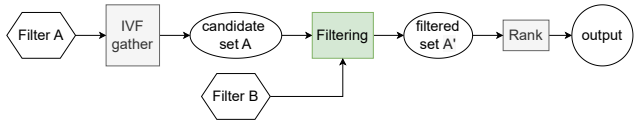
Therefore, our indexing strategy includes another level of heterogeneity. For multi-filter queries, we build a separate IVF index for each significant label. This way, we avoid label interference, but more importantly, unlike the graph index, the IVF index is very lightweight and thus space-friendly. Despite the advantages, directly adopting prior search strategies on the IVF index still leads to poor efficiency. We thus propose a novel search strategy that better leverages IVF to improve search efficiency.

5.1 Join-free Filtered Search

A straightforward search strategy for multi-filter problems is the *gather-join-rank* strategy. Fig. 5 (a) illustrates this strategy under



(a) The gather-join-rank strategy in state-of-the-art FANNS solution.



(b) The explicit-filtering strategy in our BigFANN framework.

Figure 5: Two strategies for double-filter ANNS. The key difference is shown by the highlighted boxes: join vs. filtering. Join is less efficient as it has to conservatively gather large intermediate candidate sets.

the double-filter scenario. First, for each of the two filters A and B, a concurrent search in the corresponding label-specific IVF index is performed. Each search returns a candidate set. It then performs a join, i.e., set intersection, operation to produce the candidates that pass both filters. Finally, as discussed in §3.3, this strategy results in unnecessary work spent on gathering a large volume of candidates that pass only one of the two filters A or B, most of which are subsequently discarded at the intersection stage.

This strategy is fundamentally inefficient due to the dependency between the searches and the join. As the join must be performed after all the searches are done, there are already a large number of distance computations performed unnecessarily. To avoid this huge overhead, we instead propose a novel join-free search strategy, namely *explicit-filtering*. As shown in Fig. 5 (b), given a query x_q, we first search its nearby vectors in the IVF index corresponding to the smaller filter, say A. Then we detect the candidates in set A that also pass filter B, explicitly filtering those without label B. This way, we avoid the overhead introduced by the join operations, and potentially reduce the distance computations, as analyzed in §5.3.

A detailed description is given in Algorithm 3. Given a query x_q with two filters $f_q = \{a, b\}$, we have I_a and I_b as the corresponding cluster sets, assuming $|I_a| < |I_b|$. We first sort the clusters of I_a by their centroids' distance to the query (Line 2), and then loop over the X closest clusters (Line 3) and check whether each data point in the cluster satisfies the filter b, i.e., contains the label specified by b (Line 6). If a vector is in the top-k closest to the query, it is inserted into the priority queue (Lines 8–11).

5.2 Index Construction and Preprocessing

To construct the IVF index, we first group the data points by their labels, so that each label has a set of data points. Note that a data point with multiple labels will appear in multiple sets. Let S_i be the set of the i-th label. Then we build an IVF index for each S_i, i.e., S_i is split into m_i clusters C_i, with an average cluster size of c_i. So $m_i \times c_i = |S_i|$. In addition to the index, we introduce preprocessing and build auxiliary data structures to support fast search.

Intersection Precomputation. When the combination of filters a and b has low specificity, IVF needs to search a large number of clusters to find sufficiently many points that pass both filters. To speed this up, we precompute the sizes of the intersections of every pair of *significant* (size $> \gamma$) label lists V_a and V_b ($|V_a \cap V_b|$), and

Algorithm 3 Our join-free explicit-filtering algorithm

Require: IVF index I_a, filter b, query x_q; num clusters to probe X.
Ensure: Approximate top-k nearest filtered neighbors of x_q.

```
best ← {}                                          ▷priority queue
Compute distances to all cluster centroids in I_a and sort them
for top X clusters c by smallest distance do
    C ← {}
    for vector x_i in cluster c do
        if b ∈ f_i then                            ▷Filtering
            C ← C ∪ {i}
    for i ∈ C do                                   ▷Update the priority queue
        d ← dist(x_i, x_q)
        if d < best[k − 1] then
            Insert d into best, delete tail element if best is full
return best
```

store the intersection result if its size is below a threshold η. For queries where $V_a \cap V_b$ is precomputed and its size is below η, we brute-force search through $V_a \cap V_b$ instead of using the IVF indices. This is a common optimization used in existing solutions [35]. Our sensitivity study on γ shows a reasonable space vs. speed tradeoff.

Fast Bitset for Filter Check. To enable fast checking of whether the set of labels f_i of vector i contains a filter ℓ, we precompute a bitset B_ℓ of N bits to store which vectors among the N dataset vectors contains ℓ (i.e., $\ell \in f_i$). This allows checking whether $\ell \in f_i$ in constant time. However, these bitsets require $N/8$ bytes of memory. Thus, we set a threshold parameter θ and construct bitsets sparingly, only when $|V_\ell| \geq \theta$. To check if $\ell \in f_i$ for query labels ℓ which do not have an associated bitset precomputed, we either perform a binary search on the sorted list f_i, or, in the case when $|f_i|$ is very small, we use SIMD instructions. This fallback was not implemented in ParlayIVF[2] as it was not necessary in their join-based design.

5.3 Performance Analysis

We conduct the analysis for the double-filter case, but it can be easily extended to general multi-filter cases. Given two filters a and b, suppose the two relevant vertex sets are V_a and V_b. Without loss of generality, let $|V_a| < |V_b|$. Let us also assume for simplicity that the elements of $V_a \cap V_b$ are evenly distributed[2] within V_a and V_b, respectively. In our proposed *explicit-filtering* approach, to get the K nearest neighbors, we will need to consider the first $K \times |V_a|/|V_a \cap V_b|$ points of V_a to get K points passing filter b in expectation.

In the prior *gather-join-rank* strategy, however, it will need to gather $K \times |V_b|/|V_a \cap V_b|$ points of V_b to get enough candidates (K points of V_b that also pass filter a). So it needs to set its threshold parameter $\tau \geq K \times |V_b|/|V_a \cap V_b|$, where τ denotes the number of data points to gather from each of V_a and V_b. When $|V_b| \gg |V_a|$, the join-based strategy ends up gathering $2 \times K \times |V_b|/|V_a \cap V_b|$ points, most of which are discarded after taking the join. In contrast, our strategy gathers only $\hat{\tau} = K \times |V_a|/|V_a \cap V_b|$ points.

As the join-based approach has to gather many more candidates, it creates two types of overhead. First, the required number of distance computations (NDC) is much more than ours. The NDC for the join-based approach N_{join} is the size of the set intersection of two size-τ sets; the NDC for ours N_{jfree} is the size of the filtered size-$\hat{\tau}$ sets. As shown in Fig. 4, $N_{join} = 7$, while $N_{jfree} = 3$ because our join-free approach only needs to collect one label-A (blue) cluster closest to the query, and then filter the blue nodes. Second, the join operation requires $O(\tau)$ computation in the worst case, while our filtering needs only $O(\hat{\tau})$. Also, the join-based method needs to sort its list of $K \times |V_b|/|V_a \cap V_b|$ candidates in order to perform the join, which adds another log factor on top of all the unnecessary candidates it gathers. Another overhead is the memory allocation and copy to gather the candidates. Overall, this analysis suggests that our approach outperforms existing approaches when $|V_b| \gg |V_a|$. Otherwise, the NDC required is similar, but the join-based method still has to pay the gathering and join overhead.

Note that if $|V_a \cap V_b|$ is small and precomputed, we can just enumerate $|V_a \cap V_b|$ points, as mentioned in §5.2, instead of gathering $\hat{\tau}$ candidates. When $|V_a \cap V_b|$ is large relative to $|V_a|$, our method is very fast, as $\hat{\tau} = K \times |V_a|/|V_a \cap V_b| \approx K$ is small and so good recall can be achieved by gathering few points.

Storage. Storing an IVF index for each label requires one 32-bit integer per data point, which is the same memory usage as storing the label metadata of all the points. It allows retrieval of many points close to the query with little time overhead per data point, in contrast to the graph-based approaches, which require processing of $O(\hat{\Delta})$ neighbors per retrieved data point plus a distance computation, where $\hat{\Delta}$ is the maximum degree.

IVF- vs. graph-based join-free search. A graph-based join-free search method would search the graph corresponding to the smallest filter a, and after gathering a large number of candidates, filter out the ones that do not pass the remaining filters. We analyze it as follows. Again, consider two filters a and b with $|V_a| < |V_b|$ and assume for simplicity that the elements of $V_a \cap V_b$ are evenly distributed within V_a. To get the K nearest neighbors, we need to consider the first $K_{\text{need}} = K \times |V_a|/|V_a \cap V_b|$ points of V_a to get K points passing filter b in expectation. For $|V_a \cap V_b| \ll |V_a|$, K_{need} can be much larger than K. Even if $V_a \cap V_b$ is not necessarily evenly distributed within V_a, it is common that one must gather $K_{\text{need}} \gg K$ closest points of V_a. During the graph beam search, it is necessary to perform distance computations for *every one* of the top K_{need} points before pushing them into the priority queue (the beam), as well as other points visited before the search converges. Furthermore, each of these points incurs an additional $O(\Delta)$ overhead (Δ is the graph degree) to consider their neighbors, plus priority queue operations. Most of this overhead is wasted because the points do not pass filter b. Using IVF avoids this overhead because it can coarsely gather large numbers of candidates at once. It will require only K_{need} filter checks (which are fast), find only around K points passing the filter, and perform distance computations for only these K relevant points. Thus, using IVF can be much faster, and we choose it over graph.

6 EVALUATION

Implementation. All codes, including baselines, are written in C++. BigFANN uses OpenMP for parallelism. We introduce two performance optimizations in our implementation. First, instead of sorting a list of candidates C by distance as in Parlay [35, 39], which takes $O(|C| \log |C|)$ time, we maintain a list of the k closest vectors and insert into this list whenever a closer point is found. Assuming that vectors are considered in a near-random order, the expected

[2]Our performance improvement does not rely on this assumption.

Table 1: Datasets ordered by M. M is the average number of labels per data point. $|\mathcal{L}|$ is the number of distinct labels. * denotes real-world labels. s, d, and m mean single-filter, double-filter, and mixed queries respectively (if not specified, the dataset has single-filter queries).

Dataset	Dim $\mathcal{D}$	#Base N	#Query	#Labels $\lvert\mathcal{L}\rvert$	Avg. #labels M	Source	Data Type
paper*	200	2,029,997	10,000	12	1.00	Text	float
gist	960	1,000,000	1,000	12	1.00	Image	float
arxiv*	4,096	2,735,264	10,000	38	1.37	Text	float
sift1m-{s,d}	128	1,000,000	10,000	20	8.00	Image	float
sift100m-{s,d}	128	100,000,000	10,000	20	8.00	Image	uint8
yfcc10m-{s,d,m}*	192	10,000,000	61,626 + 38,374	200,386	10.82	Image/Video	uint8

number of insertions is $O(k \log |C|)$, giving a faster complexity of $O(|C| + k^2 \log |C|)$. Second, we improve software prefetching efficiency over prior solutions [35, 39]. Given a list of candidate points C, we need to loop over them, compute their distances to the query, and return the k closest. This operation is memory-bound, so we prefetch the vectors at a distance of $d = 4$, i.e., when processing the i-th vector in C, we prefetch the $(i+d)$-th vector. This optimization was absent in ParlayIVF^2 and yields notable speedups.

Note that our BigFANN index focuses on the static setting, but it can be extended to dynamic settings. We leave it as future work.

We evaluate single-filter §6.2 and double-filter §6.3 query performance, and then present a detailed performance study in §6.4.

6.1 Experimental Setup

All experiments are done on an Intel Xeon Platinum 8380 ("Ice Lake") CPU @2.3 GHz. If not specified, we use 32 threads and 64GB of DRAM to build the indices and answer batches of queries in parallel. We also include a mode with infinite memory (`inf-mem`) to show performance when memory capacity is not a concern.

Datasets. We use representative and diverse datasets listed in Table 1. `sift` is the most widely used ANNS dataset. `gist` and `paper` have been used extensively in prior FANNS systems [4, 18, 60]. `arxiv` is a recently released FANNS benchmark [26], representing modern, high-dimensional transformer embeddings with real-world labels. `yfcc10m` is from the filter track of the 2023 NeurIPS Big-ANN competition [3]. It consists of image and video embeddings, with labels being natural metadata features, such as the camera model and the year. Notably, it has a huge number (200, 386) of distinct labels, and 2747 labels are significant [3]. The original 100, 000 queries are split into 61, 626 single-filter and 38, 374 double-filter queries.

The first three datasets in Table 1, `gist`, `paper` and `arxiv`, have small M and so are easy to handle, as they have little label interference. The next three, `sift1m`, `sift100m` and `yfcc10m`, represent more challenging datasets, with a large M. Note that `paper`, `arxiv`, and `yfcc10m` have real-world labels, while the other three have synthetic labels. Specifically, we use the same synthetic labels for `gist` from NHQ. As `sift1m` and `sift100m` originally have no associated labels, we generate $|\mathcal{L}| = 20$ possible labels, where each vector has each label with 8/20 probability (i.e., the average labels per point $M = 8$), making both datasets hard ones to deal with.

To evaluate on both single- and multi-filter problems, we include double-filter queries for the challenging datasets. For `sift1m` and `sift100m`, we randomly generate both single-filter and double-filter queries with the same label distribution as the base data. `yfcc10m` already includes real-world, mixed single- and double-filter queries.

[3]Label set size $|V_\ell| > \gamma = 5000$. We select a high cutoff of 5000 to minimize storage overhead without compromising performance, based on our detailed sensitivity study.

Note that BigFANN uses graphs (with hetero-edges) for single-filter queries, and IVF (with join-free search) for multi-filter queries. Thus, our experiments separately evaluate the improvements from hetero-edges and join-free search.

Metrics. The search performance is measured in queries per second (QPS), which is the number of queries processed divided by the total search time in seconds. To assess the accuracy of the search results, we use the average recall@$k = \frac{|G \cap S|}{k}$ across all queries, where G is the ground truth top-k closest vectors and S is the set of vectors the system returned. We set $k = 10$ in all our experiments.

Baselines and Parameters. We compare against state-of-the art FANNS solutions FilteredDiskANN [18], NHQ [60], UNG [4], and ParlayIVF^2 [35]. We use the default or recommended parameters if provided. Otherwise, we tune the parameters and use the best-performing ones we found. If not specified, we use a default value of $\alpha = 1.2$ for Vamana graphs [28] and a degree of 32.

- ParlayIVF^2: We use the recommended build beam width of 200. Due to the inflation issue, when memory overflows, we decrease the degree accordingly to fit into memory (e.g. for `sift100m`, we reduce the degree to 10). For `yfcc10m`, we use the recommended varying degrees [3] (8 for when $|V_\ell| < 10^5$, 10 for $10^5 < |V_\ell| < 4 \cdot 10^5$, and 12 for $|V_\ell| > 4 \cdot 10^5$), and $\alpha = 1.175$. For double-filter queries, we use an average cluster size of 1000 for `sift1m`, 5000 for `yfcc10m` (as recommended), and 10000 for `sift100m`.
- UNG: We use the recommended parameters [4]: $\delta = 6$ cross edges, degree $R = 32$, $L_{build} = 100$, and $\sigma = 16$ entry points for queries. For `yfcc10m`, we tuned the parameters and found $R = 48$, $L_{build} = 400$, $\sigma = 24$ to perform better than default.
- FilteredDiskANN: We only evaluate StitchedVamana, as Filtered-Vamana only improves index construction time at the expense of graph quality. For `gist` and `paper` which are used in the original paper [18], we use the recommended parameters $R_{small} = 32, L_{small} = 100, R_{stitched} = 64$. For `arxiv`, we use $R_{small} = 64, L_{small} = 80, R_{stitched} = 64$, which were found to be optimal by [26]. For the remaining (large) datasets, we use $R_{small} = 48, L_{small} = 200, R_{stitched} = 96$ as recommended by UNG [4].
- NHQ: We use the recommended parameters from [60].
- BigFANN: We set the shared edges threshold $\omega = 2$ for all datasets. For fair comparison, we use the same parameters as ParlayIVF^2 with $\Delta_s = 1$, except for `sift100m`, where we use $\Delta_e = 6, \Delta_s = 20$ to fit in 64GB memory.

To record the QPS-recall curve, we vary either the beam width (for graph search) or the number of clusters probed (for IVF search). For ParlayIVF^2 on `yfcc10m`, we vary their `TARGET_POINTS` (which controls how many points to gather in IVF search) and `TINY_CUTOFF`

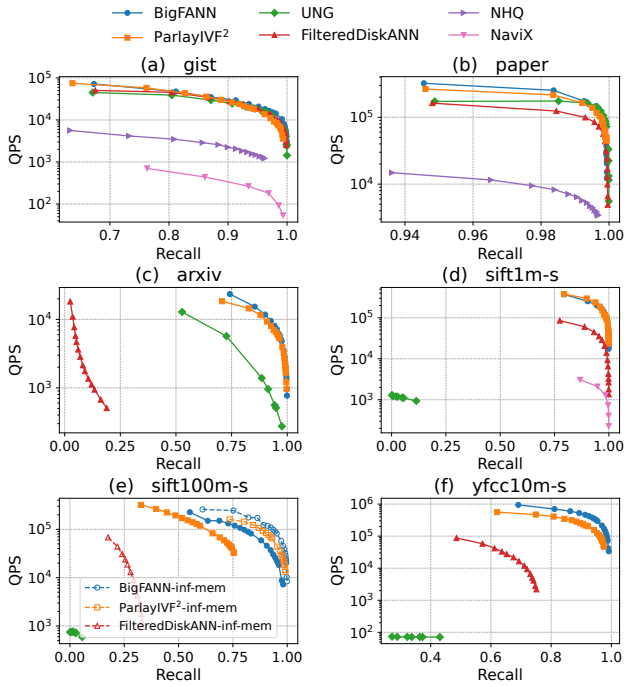


Figure 6: Comparing BigFANN performance with FilteredDiskANN, NHQ, UNG, and ParlayIVF^2 on single-filter datasets. The last two datasets are large ones, which show a significant advantage of our hybrid approach. Note that NHQ only supports `gist` and `paper`.

(which controls the decision between brute-force and IVF) parameters and record the Pareto-optimal curve.

We encountered difficulties running UNG on the large datasets due to two reasons: (1) 32-bit integer overflow bugs and (2) the official UNG code did not implement distance computations on `uint8` vectors. We fixed the overflow bugs and added the same OpenMP SIMD `uint8` distance implementation used in our system for fairness, after which UNG ran successfully on the large datasets. Also, for ParlayIVF^2, we fixed a rounding bug in the clustering algorithm to support floating-point datasets with double-filters.

6.2 Single-filter Query Performance

We evaluate single-filter FANNS on all the datasets in Table 1, while the double-filter case is evaluated on `sift1m-d`, `sift100m-d` and `yfcc10m-d`. We also evaluate `yfcc10m-m`'s overall performance with mixed labels, as required in the Big-ANN competition [3].

Fig. 6 compares the query performance of our BigFANN to prior systems on the single-filter datasets. BigFANN consistently achieves one of the best tradeoffs between query efficiency (QPS) and accuracy (recall), demonstrating the effectiveness of our hybrid index strategy and single-filter related optimizations. We find that ParlayIVF^2 always performs better than all the other baselines. This is because ParlayIVF^2 is the only existing strategy that isolates all the labels and does not suffer label interference. Note that FilteredDiskANN performs extremely poor for `arxiv`, while UNG cannot handle `sift1m` and `yfcc10m`. The differing performance is because the datasets have different label distributions. As NHQ only supports an *attributes* formulation of filtered search, we only compare against it on the two datasets `gist` and `paper` originally from NHQ. For both datasets, we find NHQ performs much worse than the other baselines. This is because the NHQ algorithm searches on the entire graph (not just a subgraph of relevant points) using a fusion distance in order to improve connectivity, and thus wastes time processing many irrelevant points.

For the small datasets, i.e., `gist`, `paper`, `arxiv`, and `sift1m`, ParlayIVF^2 performs similarly to our BigFANN. This is because ParlayIVF^2 manages to avoid label interference, while the small datasets fit in memory even though ParlayIVF^2 introduces significant index inflation. For the two large datasets, `sift100m-s` and `yfcc10m-s`, however, we observe significant speedups over ParlayIVF^2. Specifically, BigFANN achieves speedups of up to 3.6× over ParlayIVF^2 on `sift100m-s` and up to 3.1× over ParlayIVF^2 on `yfcc10m-s`. The performance improvement on `yfcc10m-s` is mainly due to our performance optimizations. For `sift100m-s`, however, our speedup comes from the hybrid indexing that allows sharing edges and saves space for larger degree than ParlayIVF^2.

For `sift100m-s`, ParlayIVF^2's default mode with a degree of 32 runs out of memory on our machine with 64GB memory. Therefore, it is forced to use a small degree of 10 to fit in memory. As shown by the orange solid curve, this low-degree graph results in poor performance, and most importantly, *it is seemingly unable to cross 80% recall.* In BigFANN, however, thanks to our hybrid indexing, we can tune the ratio between exclusive edges and shared edges to fit in memory while preserving good graph connectivity. With $\Delta_e = 6$ exclusive edges and $\Delta_s = 20$ shared edges, BigFANN achieves significantly better performance, reaching high (> 95%) recall, while using similar memory (see Fig. 10). This highlights the benefit of introducing the heterogeneous edges in our hybrid indexing design.

As a reference, we also evaluate on `sift100m-s` with *infinite memory*, i.e., we allocate as much memory as needed to hold the degree-32 graphs for ParlayIVF^2 and BigFANN. As shown by the dashed blue curve (BigFANN-inf-mem) and the dashed orange curve (ParlayIVF^2-inf-mem), we still outperform ParlayIVF^2 with a significant gap, thanks to our prefetching and queue optimizations.

6.3 Double-filter and Mixed Query Performance

Fig. 7 compares the query performance of our BigFANN to prior systems on the double-filter datasets as well as `yfcc10m-m`, a mixed-filter dataset. We do not include FilteredDiskANN and NHQ for double-filter queries as they do not support this mode. We find that BigFANN consistently achieves much better performance than UNG and ParlayIVF^2. Note that UNG performs poorly for all datasets, which confirms our analysis in §3.3: multiple filters exacerbate the label interference issue, as the graph is cut by the filters to be extremely poorly connected. Therefore, this graph index is inherently incapable of handling multi-filter tasks. Specifically, on `yfcc10m-d`, we achieve a 786× speedup at 90% recall and 602× speedup at 95% recall. For other datasets, UNG struggles to reach this high recall.

Notice that for `yfcc10m`, ParlayIVF^2 was originally tuned for the 90% recall specified by the Big-ANN competition [3]. Therefore, its default configuration may perform suboptimally for other recalls. Hence, we tune for different recalls and use the Pareto curve for ParlayIVF^2, so that we get its best performance possible to compare against. Even so, we still observe a less smooth curve (orange) in

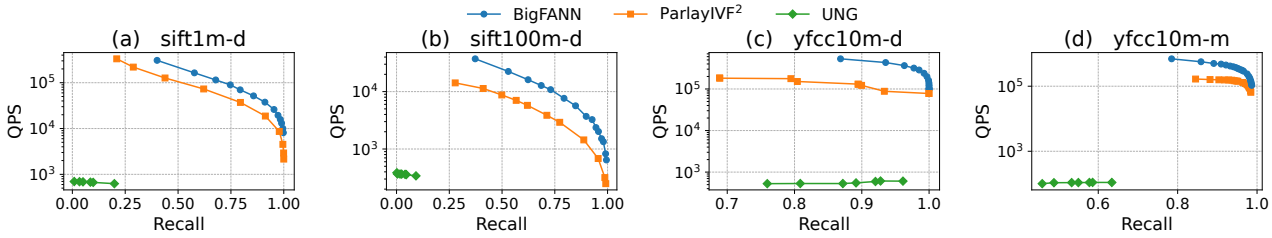


Figure 7: Comparing double-filter (a,b,c) and mixed-filter (d) performance of BigFANN, UNG and ParlayIVF². y-axis is in log-scale.

Fig. 7 (c), with little throughput gain by downgrading the recall requirement. This is because ParlayIVF² elects to use brute-force for most queries since the relevant search space is small.

Across the four datasets in Fig. 7, BigFANN achieves a geomean speedup of 2.7× (up to 3.5×) and 3.1× (up to 4.1×) over ParlayIVF² at recall 90% and 95% respectively. Our improvement over ParlayIVF² comes from our join-free strategy. By getting rid of join operations, we have two performance gains: (1) we can reduce distance computations as illustrated in Fig. 4; (2) we can avoid the sorting overhead required by join, and also the memory allocation and copy overhead caused by gathering. We further demonstrate this in §6.4.

Overall, BigFANN achieves a geomean speedup of 2.7× and 2.9× over ParlayIVF² on single- and double-filter queries on large datasets at recall 90% and 95% respectively. In addition to recall@10, we observe the same trend (not shown) for recall@100 targets.

We also compare against NaviX [49], a filtered vector search engine built in the Kuzu [13] graph database. While our BigFANN, as well as UNG, NHQ, and FilteredDiskANN, is customized for label-filter, NaviX supports arbitrary filter problems, e.g., label-filter and range-filter. This different tradeoff leads to a significant performance gap. As shown in Fig. 6 (a) and (d), BigFANN (blue curve) is up to 150× faster than NaviX (pink curve) on `gist` and 51× faster on `sift1m-s`. NaviX runs into errors on the other datasets, so we exclude them from our experiments.

Besides, we exclude post-filtering as it is always outperformed by FilteredDiskANN, and pre-filtering as it is the same as ParlayIVF². We also exclude ACORN [43] as it is always outperformed by UNG.

6.4 Detailed Performance Analysis

Running Time Breakdown. Fig. 8 compares the time breakdown of BigFANN's explicit-filtering strategy and ParlayIVF²'s gather-join-rank strategy on the large double-filter datasets. We tuned the parameters for 99% recall in both datasets. In `yfcc10m-d`, due to the large difference in label set sizes, our strategy reduced the average number of distance computations needed from 3574 to 2356, thus improving distance computation time as well as other factors. In `sift100m-d`, the label set sizes are uniform, so BigFANN did not get an improvement in distance computation count. Actually, the average number of distance computations increases slightly, so the aggregate time spent on distance computation increases. However, BigFANN still achieves significant speedups because it avoids a massive sorting overhead in computing joins: ParlayIVF² needs to gather and sort 1,000,000 target points to achieve 99% recall, and the log factor overhead becomes overwhelming.

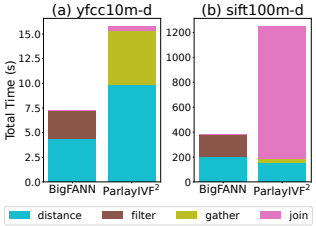


Figure 8: Time breakdown for double-filter queries.

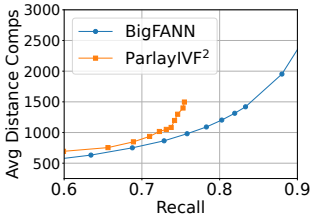


Figure 9: Distance computations on `sift100m-s`.

Distance Computations. Fig. 9 compares the average number of distance computations per query on `sift100m-s`. ParlayIVF² with its low degree of 10 always requires more distance computations than BigFANN to achieve the same level of recall, and has a trend of requiring exponentially more distance computations to improve recall past around 70%. This highlights the advantage of our hybrid indexing strategy, which constructs a better connected graph, allowing it to reach higher recalls without much more computation.

Index Size and Construction Time. Fig. 10 shows the memory needed to store the constructed index and the index build time for each FANNS framework on the two largest datasets. The same index is used for all types of queries (`-s`, `-d`, or `-m`). For BigFANN and ParlayIVF² which build both graph and IVF indices, we split the index size and build time into graph (unhatched) and IVF (hatched) components. The lightweight IVF indices take up a negligible amount of additional space compared to the memory-intensive graphs. For `yfcc10m`, ParlayIVF²'s default parameters fit well within the 64GB memory budget on our machine, so we used the same parameters in BigFANN. On the 100-million-scale `sift100m`, to fit in memory, we reduced the degree in ParlayIVF² to 10, and used $\Delta_e = 6$ exclusive edges and $\Delta_s = 20$ shared edges in BigFANN. Under this parameter choice, BigFANN and ParlayIVF² use about the same amount of memory. Note that FilteredDiskANN would exceed 64GB with its recommended parameters and still obtain much poorer recall on `sift100m-s`.

Varying Specificity of Query Label Sets. Similar to previous work [4, 18], we conduct experiments on `yfcc10m-m` using query

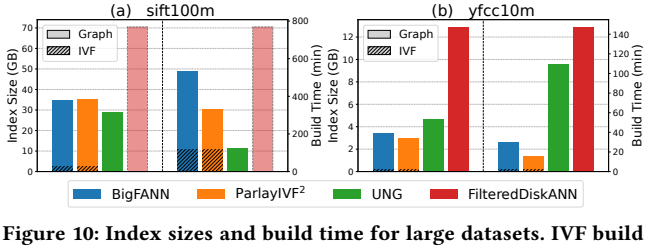


Figure 10: Index sizes and build time for large datasets. IVF build time is hatched. Lighter red bars indicate exceeding 64GB of memory.

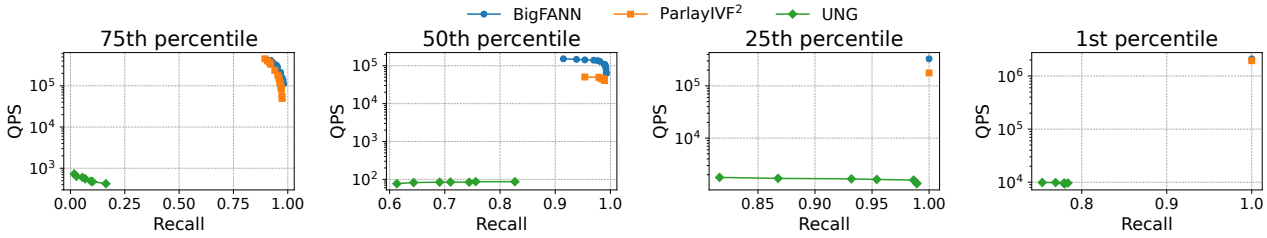


Figure 11: Performance of BigFANN, ParlayIVF^2 and UNG at varying specificity of query label sets in `yfcc10m` with mixed-filter queries.

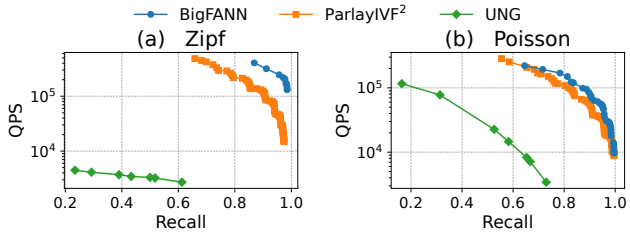


Figure 12: Varying label distributions on `sift1m`.

label sets constructed at various specificity levels, shown in Fig. 11. For the 75th, 50th, 25th and 1st percentiles, the queries of `yfcc10m-m` have specificity (i.e., fractions of points passing the filters) of 0.0154, 0.00141, 1.10×10^{-4}, and 1.03×10^{-5}, respectively. At the 25th and 1st percentile, both BigFANN and ParlayIVF^2 choose to use a brute-force filtered scan, thus achieving 100% recall, while UNG still performs an inefficient graph search. We achieve a speedup over ParlayIVF^2 due to the candidate prefetching optimization. All queries at the 75th percentile are single-filter, and UNG performs much more poorly in this regime (achieving below 20% recall) due to the poor graph connectivity within a large search space.

Varying Label Distribution. Similar to previous work [4], we vary the label distribution. Fig. 12 shows the performance of BigFANN, ParlayIVF^2, and UNG on `sift1m` with Zipf and Poisson distributions. We set the average labels per node to $M = 8$. For Zipf, we increase the number of labels to $|\mathcal{L}| = 2000$ to accommodate $M = 8$ with this distribution. We generate a mix of single- and double-filter queries in the same ratio as in `yfcc10m`, following the same distribution as the labels. Similar to Fig. 5(a) where the uniform distribution is used, we observe a performance improvement over ParlayIVF^2 using Zipf and Poisson distributions. Specifically, BigFANN outperforms significantly on the Zipf distribution because the two label set sizes in double-filter queries usually differ by much more, which is beneficial according to the analysis in §5.3. UNG still performs poorly under both distributions, confirming that this index cannot handle a large average labels per node M.

7 RELATED WORK

Prior FANNS Systems. FilteredDiskANN [18] builds on DiskANN, replacing the `robustPrune` edge-selection procedure in DiskANN's index construction with a filter-aware rule. During search, it only traverses vectors that match the query filter. NHQ [60] proposes a fusion distance that incorporates both the vector distance and the difference between the label sets. The closest data points according to this fusion distance, returned by a search, likely satisfy the query filter. UNG [4] builds the Unified Navigating Graph (UNG) index, which is highly versatile, supporting queries with arbitrary numbers of label filters. UNG groups the data points by their label sets, creating intra-group edges using classical proximity graphs, as well as adding cross-group edges between data points that satisfy label containment and are close. ParlayIVF^2 [35] is the best performing open-source submission to the 2023 Big-ANN Challenge [3]. ParlayIVF^2 uses both graph-based indices for single-filter queries and IVF indices for double-filter queries. While NHQ, UNG and FilteredDiskANN build unified indices for space efficiency, ParlayIVF^2 chooses to build separate indices for high performance. BigFANN, however, is even faster than ParlayIVF^2, and also provides a better tradeoff that balances search speed, accuracy, and space efficiency.

There are optimizations [12, 30, 36, 45, 65, 69, 73, 74] for range-filter ANNS, or RFANNS, where each data point in the database is a pair of a high-dimensional vector and a numeric value, and a query with a vector and a numeric range as parameters returns the data object whose numeric value is in the query range and whose vector is nearest to the query vector. Although they also include filtering, the optimizations, e.g., compression in SeRF [74], cannot be directly applied to the label-filter problems that our work targets.

Performance Engineering for ANNS. There have been efforts to promote high-performance ANNS including parallelizing ANNS using multicore CPUs [31, 39, 44, 67], SIMD [1, 2], GPUs [8, 20, 34, 37, 40, 42, 50, 70–72], FPGAs [32, 51, 68] and ASIC [5, 7, 62]. BigFANN is carefully engineered to include all major optimizations from the state-of-the-art multicore system ParlayANN [39], and add more optimizations in prefetching and queue management.

8 CONCLUSION

We identify the key obstacles that hamper the efficiency of FANNS solutions: label interference, index inflation, and wasted computation. This clarifies the tradeoff among search speed, accuracy, and space consumption. Existing methods optimize one aspect at the expense of others, and thus none is practical to deal with real-world large datasets. This motivates us to combine and adaptively apply indexing strategies that are aware of label characteristics. We introduce heterogeneous edges in the graph indices to deal with single-filter queries, with a tunable ratio between sharing and isolation of edges. This allows us to mitigate label interference and strike a better balance among space consumption, search speed, and accuracy. We also propose a join-free method that does explicit filtering on the gathered candidates from the smallest set. This method saves computation and avoids the overhead introduced by the join operation. Experiments show that our framework, BigFANN, significantly advances the state-of-the-art performance on a variety of datasets, while achieving the same or better accuracy.

REFERENCES

[1] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. Similarity search in the blink of an eye with compressed indices. *Proc. VLDB Endow.*, 16(11):3433–3446, 7 2023.

[2] Cecilia Aguerrebere, Mark Hildebrand, Ishwar Singh Bhati, Theodore Willke, and Mariano Tepper. Locally-adaptive quantization for streaming vector search. *arXiv preprint arXiv:2402.02044*, 2024.

[3] Big-ANN. Neurips'23 competition track: Big-ann, 2023.

[4] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. Navigating labels and vectors: A unified approach to filtered approximate nearest neighbor search. *Proc. ACM Manag. Data*, 2(6), December 2024.

[5] Mingkai Chen, Tianhua Han, Cheng Liu, Shengwen Liang, Kuai Yu, Lei Dai, Ziming Yuan, Ying Wang, Lei Zhang, Huawei Li, et al. Drim-ann: An approximate nearest neighbor search engine based on commercial dram-pims. *arXiv preprint arXiv:2410.15621*, 2024.

[6] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. Finger: Fast inference for graph-based approximate nearest neighbor search. In *Proceedings of the ACM Web Conference 2023*, WWW '23, page 3225–3235, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Sitian Chen, Amelie Chi Zhou, Yucheng Shi, Yusen Li, and Xin Yao. Memanns: Enhancing billion-scale anns efficiency with practical pim hardware. *arXiv preprint arXiv:2410.23805*, 2024.

[8] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, and Wei Zhao. Robustiq: A robust ann search method for billion-scale similarity search on gpus. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, ICMR '19, page 132–140, New York, NY, USA, 2019. Association for Computing Machinery.

[9] chroma core. Chroma - the open-source embedding database, 2024.

[10] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *IEEE Transactions on Big Data*, 2025.

[11] Elastic. Elasticsearch, 2024.

[12] Joshua Engels, Benjamin Landrum, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Approximate nearest neighbor search with window filters. *International Conference on Machine Learning*, 2024.

[13] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system. In *The Conference on Innovative Data Systems Research*, volume 7, pages 25–35, 2023.

[14] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph, 2018.

[15] Jianyang Gao and Cheng Long. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

[16] Jianyang Gao and Cheng Long. Rabitq: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.

[17] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. *Proceeding VLDB '99 Proceedings of the 25th International Conference on Very Large Data Bases*, 99, 05 2000.

[18] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*, pages 3406–3416, 04 2023.

[19] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. Symphonyqg: Towards symphonious integration of quantization and graph for approximate nearest neighbor search. *Proc. ACM Manag. Data*, 3(1), February 2025.

[20] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. Ggnn: Graph-based gpu nearest neighbor search. *IEEE Transactions on Big Data*, 9(1):267–279, 2023.

[21] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, 2020.

[22] Gaurav Gupta, Anup Rao, Tung Mai, Ryan A Rossi, Xiang Chen, Saayan Mitra, and Anshumali Shrivastava. Near neighbor search for constraint queries. In *2023 IEEE International Conference on Big Data (BigData)*, pages 1707–1715. IEEE, 2023.

[23] Sasun Hambardzumyan, Abhinav Tuli, Levon Ghukasyan, Fariz Rahman, Hrant Topchyan, David Isayan, Mark McQuade, Mikayel Harutyunyan, Tatevik Hakobyan, Ivo Stranic, et al. Deep lake: A lakehouse for deep learning. *arXiv preprint arXiv:2209.10785*, 2022.

[24] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[25] Nico Hezel, Kai Uwe Barthel, Konstantin Schall, and Klaus Jung. Fast approximate nearest neighbor search with a dynamic exploration graph using continuous refinement, 2023.

[26] Patrick Iff, Paul Bruegger, Marcin Chrapek, Maciej Besta, and Torsten Hoefler. Benchmarking filtered approximate nearest neighbor search algorithms on transformer-based embedding vectors, 2025.

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

[28] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[29] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[30] Mengxu Jiang, Zhi Yang, Fangyuan Zhang, Guanhao Hou, Jieming Shi, Wenchao Zhou, Feifei Li, and Sibo Wang. Digra: A dynamic graph indexing for approximate nearest neighbor search with range filter. *Proc. ACM Manag. Data*, 3(3), June 2025.

[31] Wenqi Jiang, Hang Hu, Torsten Hoefler, and Gustavo Alonso. Accelerating graph-based vector search via delayed-synchronization traversal. *arXiv preprint arXiv:2406.12385*, 2024.

[32] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes De Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, and Gustavo Alonso. Co-design hardware and algorithm for vector search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[33] James Jie Pan, Jianguo Wang, and Guoliang Li. Survey of vector database management systems. *arXiv preprint arXiv:2310.14021*, 2023.

[34] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[35] Ben Landrum, Magdalen Dobson Manohar, Mazin Karjikar, and Laxman Dhulipala. ParlayIVF, 2025. Accessed: 2025-06-02.

[36] Anqi Liang, Pengcheng Zhang, Bin Yao, Zhongpu Chen, Yitong Song, and Guangxu Cheng. Unify: Unified index for range filtered approximate nearest neighbors search. *Proc. VLDB Endow.*, 18(4):1118–1130, December 2024.

[37] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. Juno: Optimizing high-dimensional approximate nearest neighbour search with sparsity-aware algorithm and ray-tracing core mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 549–565, New York, NY, USA, 2024. Association for Computing Machinery.

[38] Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, 2018.

[39] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan 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*, pages 270–285, 2024.

[40] Bruno Henrique Meyer, Aurora Trinidad Ramirez Pozo, and Wagner M. Nunan Zola. Warp-centric k-nearest neighbor graphs construction on gpu. *50th International Conference on Parallel Processing Workshop*, 2021.

[41] NVIDIA. cuVS, 2025. Accessed: 2025-06-02.

[42] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus, 2023.

[43] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. *Proc. ACM Manag. Data*, 2(3), May 2024.

[44] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 313–328, 2023.

[45] Zhencan Peng, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. Dynamic range-filtering approximate nearest neighbor search. *Proc. VLDB Endow.*, 18(10):3256–3268, June 2025.

[46] pgvector. pgvector, 2024.

[47] Qdrant. Qdrant. efficient, scalable, fast., 2024.

[48] Rapidsai. Rapidsai/raft: Raft contains fundamental widely-used algorithms and primitives for data science, graph and machine learning., 2022.

[49] Gaurav Sehgal and Semih Salihoğlu. Navix: A native vector index design for graph dbmss with robust predicate-agnostic search performance. *Proc. VLDB Endow.*, 18(11):4438–4450, July 2025.

[50] Anil Shanbhag, Holger Pirk, and Samuel Madden. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1557–1570, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Yifeng Song, Chenjie Liu, Rongrong Zhang, Danyang Zhu, and Zhongfeng Wang. An efficient fpga implementation of approximate nearest neighbor search. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2025.

[52] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. Vexless: A serverless vector data management system using cloud functions. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 2024.

[53] Pinecone Systems. Pinecone product overview, 2022.

[54] Toni Taipalus. Vector database management systems: Fundamental concepts, use-cases, and current challenges. *Cognitive Systems Research*, page 101216, 2024.

[55] Vald team. Vald a highly scalable distributed vector search engine, 2024.

[56] Mariano Tepper, Ishwar Singh Bhati, Cecilia Aguerrebere, Mark Hildebrand, and Theodore L Willke. Leanvec: Searching vectors faster by making them fit. *Transactions on Machine Learning Research*, 2023.

[57] Meilisearch Documentation v0.27. What is meilisearch?, 6 2022.

[58] vespa. Big data + ai, online., 2024.

[59] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

[60] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 15738–15751. Curran Associates, Inc., 2023.

[61] Runhui Wang and Dong Deng. Deltapq: lossless product quantization code compression for high dimensional similarity search. *Proceedings of the VLDB Endowment*, 13(13):3603–3616, 2020.

[62] Yitu Wang, Shiyu Li, Qilin Zheng, Linghao Song, Zongwang Li, Andrew Chang, Yiran Chen, et al. Ndsearch: Accelerating graph-traversal-based approximate nearest neighbor search through near data processing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 368–381. IEEE, 2024.

[63] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. Graph-and tree-based indexes for high-dimensional vector similarity search: Analyses, comparisons, and future directions. *Data Engineering*, pages 3–21, 2023.

[64] Weaviate. The ai-native vector database, 2024.

[65] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. irangegraph: Improvising range-dedicated graphs for range-filtering nearest neighbor search. *Proc. ACM Manag. Data*, 2(6), December 2024.

[66] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. Spfresh: Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 545–561, 2023.

[67] Xizhe Yin, Chao Gao, Zhijia Zhao, and Rajiv Gupta. Panns: Enhancing graph-based approximate nearest neighbor search through recency-aware construction and parameterized search. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 369–381, 2025.

[68] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, and Yu Wang. Df-gas: a distributed fpga-as-a-service architecture towards billion-scale graph-based approximate nearest neighbor search. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 283–296, New York, NY, USA, 2023. Association for Computing Machinery.

[69] Fangyuan Zhang, Mengxu Jiang, Guanhao Hou, Jieming Shi, Hua Fan, Wenchao Zhou, Feifei Li, and Sibo Wang. Efficient dynamic indexing for range filtered approximate nearest neighbor search. *Proc. ACM Manag. Data*, 3(3), June 2025.

[70] Jingrong Zhang, Akira Naruse, Xipeng Li, and Yong Wang. Parallel top-k algorithms on gpu: A comprehensive study and new methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[71] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast vector query processing for large datasets beyond {GPU} memory with reordered pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 23–40, 2024.

[72] Weijie Zhao, Shulong Tan, and Ping Li. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1033–1044, 2020.

[73] Chaoji Zuo and Dong Deng. Arkgraph: All-range approximate k-nearest-neighbor graph. *Proceedings of the VLDB Endowment*, 16(10):2645–2658, 2023.

[74] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. Serf: Segment graph for range-filtering approximate nearest neighbor search. *Proc. ACM Manag. Data*, 2(1), March 2024.