# Filtered − DiskANN: **Graph Algorithms for Approximate Nearest Neighbor Search with Filters**

Siddharth Gollapudi
Microsoft Research
India
sgollapu@berkeley.edu

Neel Karia[*,1]
Columbia University
USA
neel2karia@gmail.com

Varun Sivashankar[*]
Microsoft Research
India
varunsiva@ucla.edu

Ravishankar Krishnaswamy
Microsoft Research
India
rakri@microsoft.com

Nikit Begwani
Microsoft
India
nikit.begwani@microsoft.com

Swapnil Raz
Microsoft
India
swraz@microsoft.com

Yiyong Lin
Microsoft
USA
yiyolin@microsoft.com

Yin Zhang
Microsoft
USA
yinzhang@microsoft.com

Neelam Mahapatro
Microsoft
India
nmahapatro@microsoft.com

Premukumar Srinivasan
Microsoft
USA
prsriniv@microsoft.com

Amit Singh
Microsoft
India
siamit@microsoft.com

Harsha Vardhan Simhadri
Microsoft Research
USA
harshasi@microsoft.com

## ABSTRACT

As Approximate Nearest Neighbor Search (ANNS)-based dense retrieval becomes ubiquitous for search and recommendation scenarios, efficiently answering *filtered ANNS queries* has become a critical requirement. Filtered ANNS queries ask for the nearest neighbors of a query's embedding from the points in the index that match the query's *labels* such as date, price range, language. There has been little prior work on algorithms that use label metadata associated with vector data to build efficient indices for filtered ANNS queries. Consequently, current indices have high search latency or low recall which is not practical in interactive web-scenarios. We present two algorithms with native support for faster and more accurate filtered ANNS queries: one with streaming support, and another based on batch construction. Central to our algorithms is the construction of a graph-structured index which forms connections not only based on the geometry of the vector data, but also the associated label set. On real-world data with natural labels, both algorithms are an order of magnitude or more efficient for filtered queries than the current state of the art algorithms. The generated indices also be queried from an SSD and support thousands of queries per second at over 90% recall@10.

## KEYWORDS

Approximate nearest neighbor search, Filtered Search, Graph algorithms, Dense retrieval, Vector Search


*WWW '23, April 30–May 04, 2023, Austin, TX*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/3543507.3583552

**ACM Reference Format:**
Siddharth Gollapudi, Neel Karia[*,1], Varun Sivashankar[*], Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premukumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered − DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *Proceedings of the ACM Web Conference 2023 (WWW '23), April 30-May 4, 2023, Austin, TX, USA.* ACM, New York, NY, USA, page 11 pages. https://doi.org/10.1145/3543507.3583552

## 1 INTRODUCTION

In the nearest neighbor search problem, we are given a dataset $P$ with $n$ points in some metric space, which we assume to be $\mathbb{R}^d$ with the Euclidean distance in this paper. Given a query $x_q \in \mathbb{R}^d$ and $k \in \mathbb{N}$, we would like to return the $k$ nearest neighbors of $x_q$ from $P$. To exactly find the $k$ nearest neighbors, we cannot do better than a linear scan of the dataset [35] due to the curse of dimensionality [20]. Therefore, in practice, algorithms are designed for *approximate* nearest neighbor search (ANNS), i.e., to efficiently retrieve a set $\mathcal{L}$ of $k$ candidates to maximize $recall@k = \frac{|G \cap \mathcal{L}|}{k}$, where $G$ is the ground truth set of $x_q$'s $k$ nearest neighbors in $P$.

### 1.1 Filtered ANNS

In this setting, for every data point (a.k.a. vector) $x \in P$, we have an associated set of labels $F_x \subseteq \mathcal{F}$, where $\mathcal{F}$ is a finite universe of labels. A query to the index now comprises of the vector $x_q$, the target number of nearest neighbors $k$, *and a label filter* $f \in \mathcal{F}$. The ANNS index is required to find the closest neighbors of $x_q$ from $P_f = \{x \in P : f \in F_x\}$, i.e., points in the index that have the label $f$ associated with them. Once again, the index should maximize recall@k, but relative to ground-truth computed against the set $P_f$, instead of $P$. We also define the *specificity* of $f$ to be $|P_f|/|P|$: the fraction of indexed data points which have the label $f$ associated

[*]Equal Contribution
[1]Work done while at Microsoft Research India

with them. While a natural generalization is to consider ANNS queries with complex predicates, in this paper we focus on the case with single filter associated with the query.

Many important real-world scenarios can be expressed in this simple framework. For example, web search engines offer filtering results by keyword, publishing date, or domain/sub-domain of a website. Image search engines allow filtering by color, resolution, etc. E-commerce search allows filtering by categories such as brand, price range and product size. Enterprise search applications might want to limit documents displayed based on user's privileges. Another scenario where filtering is not exposed to the user but is implicit is that of advertisement display. Here, advertisers instruct the search engine to display only those ads that are relevant to the user's region. Given the ubiquity of filtering requirements for ANNS, a recent wave of start-ups such as Milvus [2], Pinecone [19], Vearch [5], Vespa [6] and Weaviate [7] offer ANNS-as-a-service with various degrees of support for filtering. Moreover, some industrial systems such as AliBaba's AnalyticDB-V [43] and the academic community [42, 45] have considered variants of this problem.

## 1.2 Drawback of Existing Methods

A common thread across most current methods for Filtered ANNS is that they only modify the search step, and not the index build step, which we argue is sub-optimal. We overview previous methods here and their drawbacks.

One straightforward method for answering hybrid queries is the *post-processing* approach: build a standard ANNS index, query as usual, and post-process the results by selecting only those results returned by the index that match the query filter. While this method is easy to implement, it performs poorly in practice. Indeed, for a label filter $f$ with low specificity, we may have to retrieve a very large number of candidates before coming across a single result matching the filter. We have observed this to be the case on real-world datasets (see Figure 1, Figure 2, and Figure 3).

Conversely, one might consider a trivial *pre-processing* method that builds a separate index for each possible filterable label $f \in \mathcal{F}$ so that a query can be routed to the index associated with the query's filter. However, such an approach would quickly become prohibitively expensive in scenarios with either a large number of filters, or where each point could have many associated filters.

Weaviate and Milvus both use a more efficient *pre-processing* step before passing a query through a graph index. Weaviate first passes the query's labels through an inverted index in order to generate an *approved list* of points [8]. Milvus maintains a distribution of attributes over points, and uses a hash-table to map the commonly used attributes from a query to the "approved list" of points [41]. At search time, only the approved list of points are considered. While both these data structures are created at indexing time, they do not change the main vector similarity search index: their approaches just affect how the search procedure traverses the main index.

There are some algorithms where the post-processing approach can be applied "on the fly," which we will refer to as *inline-processing*. For example, FAISS-IVF [29] partitions the data into clusters, and the ANN data structure is an inverted index consisting of all the points belonging to each cluster. By including filter metadata with each entry in the inverted index, an inline-processing search can skip points in the clusters that do not match the filters of the query. Pinecone's hybrid search feature utilises this approach [19]. This technique can also be applied to an LSH [10] index.

However, graph-based ANNS indices such as HNSW [32] and Vamana [39] are an order of magnitude more efficient than IVF/LSH indices in terms of the number of points in the index that a query is compared to for a target recall, and this gap only increases with data size. As a result, interactive web services like search, advertising, and enterprise document recommendation requiring high performance deploy graph-based ANNS indices for achieving high throughput and recall with query latency of a few milliseconds required for these services. Current approaches to supporting filtered queries do not leverage these enormous query efficiencies provided by graph based indices and focus instead on optimizations of more inefficient indexing techniques.

## 1.3 Our Results and Techniques

Our main contributions are two simple-yet-effective graph-based algorithms for filtered ANNS – FilteredVamana and StitchedVamana – that build upon the Vamana graph [39]. Graph-based indices like Vamana work by constructing *navigable graphs*, which are effective at guiding a locally "greedy" search towards the query's nearest neighbor candidates. To the best of our knowledge, existing algorithms with the exception of NHQ only consider the positions (vector co-ordinates) of the points in the data set and not the filter metadata. **Both the algorithms presented here go further by making use of not only the geometric relation between points but also the labels that each point has in constructing the navigational structure of the graph.**

The FilteredVamana algorithm starts with an empty graph and incrementally adds points to the index, along with edges, as follow. For the $i^{\text{th}}$ point $x_i$ with associated labels $F_i$, we find a suitable set of diverse candidate neighbors and add bi-directional edges. Then, whenever any vertex degree exceeds a given threshold $R$, we run a RobustPrune procedure to prune redundant edges by looking at the geometry and the filter information.

The StitchedVamana algorithm takes a bulk-indexing approach. It builds a separate Vamana index over each point set $P_f$ of all points associated with each label filter $f$, and overlays them into a graph whose edges are the union of the edges in filter-specific graphs. This results in an index that could be as large as building a separate index for each filter. To reduce the index size, we run the RobustPrune algorithm for every node whose degree exceeds $R$.

Intuitively, one might expect StitchedVamana to fare better than FilteredVamana, since each node accumulates a large number of useful candidates (when taking the union) before the pruning step, while FilteredVamana prunes on-the-fly whenever any degree exceeds $R$. Our experimental results indeed confirm this. However, FilteredVamana index builds faster, and is more readily amenable to incremental updates. We evaluate the merits of these algorithms compared to each other and to the rest of literature in section 5.

We now summarize our contributions.

(1) Our algorithms generate indices which can support thousands of filters ($|\mathcal{F}| \sim 1000$) with each point in $P$ associated tens or hundreds of these filters. Notably, these indices have near identical resource consumption (e.g. index size) to prior graph-based indices for unfiltered ANNS.

(2) We compare our algorithms with many existing public baselines, including IVF, HNSW, NHQ and Milvus, and show that they outperform baselines by an order-of-magnitude or more. Our algorithms can be tuned to provide recall as close to 100% as possible even for filters with specificity as low as $10^{-4}$ to $10^{-6}$, while other algorithms saturate at lower recall.
(3) We show significant improvement on key metrics in a real-world sponsored advertisement scenario involving filters. The algorithms improve recall significantly within a strict serving latency budget, resulting in revenue gains ranging from 30 − 80% depending on the specificity of the query.
(4) The indices can also be stored in inexpensive SSDs as in the DiskANN system [39], and enable filtered search at low latency and 90+% recall and thousands of queries per second.

## 2 RELATED WORK

There has been a significant amount of research devoted to ANNS algorithms [9, 11, 12, 14–18, 23–25, 27–30, 32–34, 39, 40, 46]. See also the recent benchmarks [13, 37] for a comparison of the state-of-the-art ANN algorithms. Most existing research addresses the standard ANNS problem from the perspective of improving recall [32], scale and cost-efficiency [15, 39], distributed indexing [40], enabling real-time updates to the index [38], and designing algorithms with theoretical guarantees. With the increasingly central role of ANNS in semantic search/dense-retrieval, many application-critical requirements for ANNS are found lacking in research literature, one of which is that of *filtering* or *filtered queries* (used interchangeably).

There have been two recent works on filtered-ANNS. Analytic DB-V[44], Alibaba's real-world system integrates filtered ANNS queries on a SQL engine. It supports and optimizes for complex filters using a query-plan based on the specificity of the filter:

- high specificity: post-processing index
- moderate specificity: inline-processing IVF-PQ [22] index
- low specificity: inline-processing brute-force index

In this work, we limit to simpler filters – exact match with one filter. To support such searches at interactive latency and high recall, we develop new graph-based indices that can be updated. We can easily extend this to the disjunction (OR) of several filters by simply finding the answers corresponding to each individual filter, and aggregating and sorting these results by distance from the query. We leave the case of conjunctions (AND) of several filters and other more complicated expressions as a challenging avenue for future work. Note that when the possible set of predicates are known and not too large (thousands), we can label each vector with predicates it satisfies and build the graph to support filtering by those labels.

Another recent algorithm that supports filtered queries is the NHQ algorithm[42]. This is relevant to our work in that it is graph-based and actually modifies the indexing step: they encode the filter labels as vectors and append them to the real vector and index with ANNS algorithms such as NSW or kNN-graph [21]. However, this paper considers the setting where each data point has effectively only one filter label associated – i.e. the sets $P_f$ are completely disjoint, a scenario that can be handled by separate standard ANNS indices for each filter. Further, this technique could be ineffective in scaling to multiple filters/labels per point without significantly affecting recall. If a point in the index has three filter labels, and the query only has one, the distance in the other two label coordinates will adversely affect such candidates as compared to data points which have only the same query label. We demonstrate that our approaches outperform NHQ in the appendix, which in turn significantly outperforms Analytic DB-V [42].

**Algorithm 1:** FilteredGreedySearch($S$, $x_q$, $k$, $L$, $F_q$)

**Data:** Graph $G$ with initial nodes $S$, query vector $x_q$, search list size $\mathcal{L}$, and query filter(s) $F_q$.
**Result:** Result set $\mathcal{L}$ containing $k$ approximate nearest neighbors, and a set $\mathcal{V}$ containing all visited nodes.
**begin**
  Initialize sets $\mathcal{L} \leftarrow \emptyset$ and $\mathcal{V} \leftarrow \emptyset$.
  **for** $s \in S$ **do**
    **if** $F_s \cap F_x \neq \emptyset$ **then**
      $\mathcal{L} \leftarrow \mathcal{L} \cup \{s\}$
  **while** $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ **do**
    Let $p^* \leftarrow \arg\min_{p \in \mathcal{L} \setminus \mathcal{V}} \|x_p - x_q\|$
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$
    Let $N'_{\text{out}}(p^*) \leftarrow \{p' \in N_{\text{out}}(p^*) : F_{p'} \cap F_q \neq \emptyset, p' \notin \mathcal{V}\}$
    $\mathcal{L} \leftarrow \mathcal{L} \cup N'_{\text{out}}(p^*)$
    **if** $|\mathcal{L}| > L$ **then**
      Update $\mathcal{L}$ with the closest $L$ nodes to $x_q$.
  **return** [$k$ NNs from $\mathcal{L}$; $\mathcal{V}$]

**Algorithm 2:** FindMedoid($P$, $\tau$)

**Data:** Dataset $P$ with associated filters for all the points, threshold $\tau$.
**Result:** Map $M$ mapping filters to start nodes.
**begin**
  Initialize $M$ be an empty map, and $T$ to an zero map; *//T is intended as a counter*
  **foreach** $f \in \mathcal{F}$, *the set of all filters* **do**
    Let $P_f$ denote the ids of all points matching filter $f$
    Let $R_f \leftarrow \tau$ randomly sampled data point ids from $P_f$
    $p^* \leftarrow \arg\min_{p \in R_f} T[p]$
    update $M[f] \leftarrow p^*$ and $T[p^*] \leftarrow T[p^*] + 1$
  **return** $M$

## 3 THE FilteredVamana ALGORITHM

Graph-based ANNS indices are constructed so that greedy search quickly converges to the nearest neighbors of a query vector $x_q$. We first describe a natural adaptation of greedy search for filtered queries called FilteredGreedySearch (algorithm 1) and an index construction procedure (algorithm 4) that allows search to converge to the right answer with relatively few distance comparisons.

### 3.1 FilteredGreedySearch

Given a query $x_q$ and a set of labels $F_q$, we are required to output the $k$ approximate nearest neighbors of $x_q$, where each point in the output shares at least one label with $F_q$. The search procedure also takes in a set $S$ of start nodes. For a query with label set $F_q$, the set $S$ is typically $\{\text{st}(f) : f \in F_q\}$, where $\text{st}(f)$ is the designated start node for label $f \in \mathcal{F}$ computed during the index construction. In

**Algorithm 3:** FilteredRobustPrune($p$, $\mathcal{V}$, $\alpha$, $R$)

**Data:** Graph $G$, point $p \in P$, candidate set $\mathcal{V}$, distance threshold $\alpha \geq 1$, max outdegree bound $R$.
**Result:** $G$ is modified by setting at most $R$ out-neighbors for $p$.
**begin**
  $\mathcal{V} \leftarrow \mathcal{V} \cup N_{\text{out}}(p) \setminus \{p\}$
  $N_{\text{out}}(p) \leftarrow \emptyset$
  **while** $\mathcal{V} \neq \emptyset$ **do**
    $p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$
    $N_{\text{out}}(p) \leftarrow N_{\text{out}}(p) \cup \{p^*\}$
    **if** $|N_{out}(p)| = R$ **then**
      **break**
    **for** $p' \in \mathcal{V}$ **do**
      **if** $F_{p'} \cap F_p \not\subseteq F_{p^*}$ **then**
        **continue**
      **if** $\alpha \cdot d(p^*, p') \leq d(p, p')$ **then**
        Remove $p'$ from $\mathcal{V}$.

this paper, we benchmark queries where $F_q$ is a singleton set, but the algorithm can also be used for queries with $|F_q| > 1$.

The algorithm maintains a priority queue $\mathcal{L}$ of size at most $L$ (where $k \leq L$). At every iteration, it looks for the nearest unvisited neighbor $p^*$ of $x_q$ in $\mathcal{L}$. It then adds $p^*$ to a set of visited nodes $\mathcal{V}$. This is a useful invariant that we will refer to later on in this paper. We then *add only those out-neighbors of $p^*$ that have at least one label in $F_q$* to the list $\mathcal{L}$. Finally, if $|\mathcal{L}| > L$, we truncate $\mathcal{L}$ to contain the $L$ closest points to $x_q$. The search terminates when all nodes in $\mathcal{L}$ have been visited. The output consists of the $k$ nearest neighbors of $x_q$ from $\mathcal{L}$, as well as the set of visited nodes $\mathcal{V}$ which is useful for index construction (but not in user queries).

## 3.2 Index Construction

**Start Point Selection.** We require start nodes for each filter to satisfy two criteria: (a) the start point $s \equiv \text{st}(f)$ for a query with a single filter $f$ should be associated with that filter, i.e., $f \in F_s$, and (b) no point in $P$ should be the start point for too many filter labels. The load of routing queries with different filters should be shared across many points so that we can build a graph of small bounded maximum degree which caters to all filter labels. Indeed, if a single point served as the start point of many labels, there may be very few neighboring vertices with a certain label from the starting point, leading to poor search. We achieve this using a simple randomized load balancing algorithm described in algorithm 2.

**Incremental Graph Construction**. The FilteredVamana graph construction is an incremental algorithm. We first identify the start node $\text{st}(\cdot)$ for each filter label, and initialize $G$ to an empty graph. Then, for each data point $p \in P$, with associated filters/labels $f \in F_p$, we run FilteredGreedySearch($S_{F_p}, x_p, L, L, F_p$), with starting points $S_{F_p} = \{\text{st}(f) : f \in F_p\}$. This returns a set $\mathcal{V}_{F_p}$ of vertices visited in the search exploration. All visited nodes have some label $f \in F_p$.

Next, we prune the candidate set $\mathcal{V}_{F_p}$ with a call to the filter-aware pruning procedure in algorithm 3 with parameters $(x, \mathcal{V}, \alpha, R)$. This ensures the graph node corresponding to $x$ has at most $R$ out-neighbors, while also eliminating redundant edges to nearby vectors. The pruning procedure relies on the following principle:

**Algorithm 4:** FilteredVamana Indexing Algorithm

**Data:** Database $P$ with $n$ points where $i$-th point has coords $x_i$, parameters $\alpha, L, R$.
**Result:** Directed graph $G$ over $P$ with out-degree $\leq R$.
**begin**
  Initialize $G$ to an empty graph
  Let $s$ denote the medoid of $P$
  Let $\text{st}(f)$ denote the start node for filter label $f$ for every $f \in F$
  Let $\sigma$ be a random permutation of $[n]$
  Let $F_x$ be the label-set for every $x \in P$
  **foreach** $i \in [n]$ **do**
    Let $S_{F_{x_{\sigma(i)}}} = \{\text{st}(f) : f \in F_{x_{\sigma(i)}}\}$
    Let $[\emptyset; \mathcal{V}_{F_{x_{\sigma(i)}}}] \leftarrow$ `FilteredGreedySearch`($S_{F_{x_{\sigma(i)}}}$, $x_{\sigma(i)}, 0, L, F_{x_{\sigma(i)}}$)
    $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_{F_{x_{\sigma(i)}}}$
    Run `FilteredRobustPrune`($\sigma(i), \mathcal{V}_{F_{x_{\sigma(i)}}}, \alpha, R$) to update out-neighbors of $\sigma(i)$.
    **foreach** $j \in N_{\text{out}}(\sigma(i))$ **do**
      Update $N_{\text{out}}(j) \leftarrow N_{\text{out}}(j) \cup \{\sigma(i)\}$
      **if** $|N_{\text{out}}(j)| > R$ **then**
        Run `FilteredRobustPrune`($j, N_{\text{out}}(j), \alpha, R$) to update out-neighbors of $j$.

**Algorithm 5:** StitchedVamana Indexing Algorithm

**Data:** Database $P$ with $n$ points where $i$-th point has coords $x_i$, Database $F$ of labels, parameters $\alpha, L_{\text{small}}, R_{\text{small}}, R_{\text{stitched}}$.
**Result:** Directed graph $G$ over $P$ with out-degree $\leq R_{\text{stitched}}$.
**begin**
  Initialize $G = (V, E)$ to an empty graph
  Let $F_x \subseteq F$ be the label-set for every $x \in P$
  Let $P_f \subseteq P$ be the set of points with label $f \in F$.
  **foreach** $f \in F$ **do**
    Let $G_f \leftarrow$ Vamana($P_f, \alpha, R_{\text{small}}, L_{\text{small}}$)
  **foreach** $v \in V$ **do**
    FilteredRobustPrune ($v, N_{\text{out}}(v), \alpha, R_{\text{stitched}}$)

For any triplet of vertices $a, b, c$, and constant $\alpha \geq 1$, the directed edge $(a, c)$ can be pruned out of the graph if

(1) the edge $(a, b)$ is present,
(2) the vector $x_b$ is substantially closer to $x_c$ than $x_a$ is to $x_c$, i.e., $\|x_b - x_c\| \leq (1/\alpha)\|x_a - x_c\|$, and
(3) $F_b$ contains all common filter labels of $F_a$ and $F_c$, i.e., $F_a \cap F_c \subseteq F_b$.

Finally, we add backward edges from $y$ to $x$ for all $y \in N_{\text{out}}(x)$, and again, if the degree of any such $y$ exceeds $R$, we run the FilteredRobustPrune procedure on $y$.

# 4 THE StitchedVamana ALGORITHM

We now present a different algorithm for building an index called StitchedVamana (algorithm 5), which can only be used when the

| Dataset | Dim | # Pts. | # Queries | Source Data | Filters | Filters per Pt. | Unique Filters | 100pc. | 75pc. | 50pc. | 25pc. | 1pc. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Turing | 100 | 2,599,968 | 996 | Text | Natural | 1.09 | 3070 | 0.127 | $1.56x10^{-4}$ | $4.15x10^{-5}$ | $1.54x10^{-5}$ | $7.7x10^{-6}$ |
| Prep | 64 | 1,000,000 | 10000 | Text | Natural | 8.84 | 47 | 0.425 | 0.136 | 0.130 | 0.127 | 0.09 |
| DANN | 64 | 3,305,317 | 32926 | Text | Natural | 3.91 | 47 | 0.735 | 0.361 | 0.183 | 0.167 | 0.150 |
| SIFT | 128 | 1,000,000 | 10000 | Image | Random | 1 | 12 | 0.083 | 0.083 | 0.083 | 0.083 | 0.082 |
| GIST | 960 | 1,000,000 | 1000 | Image | Random | 1 | 12 | 0.083 | 0.083 | 0.083 | 0.083 | 0.082 |
| msong | 420 | 992,272 | 200 | Audio | Random | 1 | 12 | 0.083 | 0.082 | 0.082 | 0.082 | 0.082 |
| audio | 192 | 53,387 | 200 | Audio | Random | 1 | 12 | 0.085 | 0.084 | 0.083 | 0.082 | 0.081 |
| paper | 200 | 2,029,997 | 10000 | Text | Random | 1 | 12 | 0.083 | 0.083 | 0.083 | 0.083 | 0.082 |

**Table 1: Datasets used in the evaluation and their statistics. Top 3 rows are real-world datasets; the rest are semi-synthetic.**

point set is known ahead of time. For each $f \in F$, we build a graph index $G_f$ over points $P_f$ with label $f$ using the Vamana algorithm [39] with parameters $L_{\text{small}}$ and $R_{\text{small}}$. These parameters are smaller than the ones in previous algorithm for faster index construction. Then, since vertices can potentially belong to multiple indices $G_f$ (since a point $p \in P$ can belong to multiple $P_f$), we "stich" the graphs $G_f$'s together in to the graph $G$, whose edges are the the union of edge sets of each $G_f$. $G$ could have a large degree. We reduce its maximum out degree to $R_{\text{stitched}}$ using the FilteredRobustPrune procedure. The resulting graph is compatible with the FilteredGreedySearch procedure.

## 5 EVALUATION

We now compare the query performance and accuracy of these algorithms with several baselines representing inline and post-processing techniques using three real-world datasets from production scenarios and semi-synthetic datasets used in prior work. All experiments were run on an Azure E64dsv4 virtual machine with Intel(R) Xeon(R) Platinum 8272CL CPUs @ 2.60GHz with 64 vCPUs and 500GB of RAM. All query throughput measurements are reported for runs with 48 threads.

### 5.1 Datasets

Table 1 lists the datasets used for evaluation and provides statistics including the index size, the number of unique filters, the average number of filters associated with each point, and the specificity $|P_f|/|P|$ of the 100, 75, 50, 25 and 1 percentile filters as sorted in decreasing order of frequency. We measure the QPS (queries per second) and recall of different algorithms for filters with these specificities.

*Real-world datasets.* The Turing dataset consists of encodings of text from an enterprise corpus for query relevance, with the filters being sites associated with the text within the enterprise. The Prep and DANN datasets represent sponsored advertisements from a large ad corpus relevant across 47 regions (countries). Each ad can be served in one or more geographical regions based on advertiser preference. The vectors are derived from the twin-tower encoders [26, 31] applied to advertisements. We use the DANN and Prep datasets as the primary benchmarking datasets. The Turing dataset has a large range in terms of specificity of filters, which is helpful for analyzing the drawbacks of some popular approaches.

*Semi-Synthetic Datasets.* We also benchmark our algorithms on five datasets that were used to test the recent NHQ algorithm in [42]. These include one real-world dataset released in [42], and four datasets that are publicly available, with labels generated randomly via the method from [2]. These datasets are not as realistic because:

- For the latter four datasets, the filter for each point is fabricated or selected at random. In real-world datasets there could be correlation between the distribution of points and the set of labels that an ANNS algorithm could exploit.
- Each point in the index effectively has only one label. While it might appear at first glance that each data point and query in the NHQ datasets has 3 labels, we get a single label from the cartesian product of entries from three categories each with 3,2 and 2 distinct values. This gives a partitioning of the dataset into 12 disjoint sets, and it is therefore trivial to support filtered ANNS by creating separate indices over the partitions, and searching the relevant partition based on the query.

### 5.2 Algorithms and Parameters

We benchmark the algorithms described in this paper, as well as some of algorithmic approaches surveyed in the paper. We include a brief description of the parameters used and the source of the code below:

(1) StitchedVamana [36]: The index corresponding to each filter is built with parameters $R_{\text{small}} = 32$ and $L_{\text{small}} = 100$. The final pruning procedure is done with degree bound $R_{\text{stitched}} = 64$. The pruning threshold parameter is set to $\alpha = 1.2$. To generate the Recall/QPS curves, we use FilteredGreedySearch where $L$, the search parameter controling the tradeoff between accuray and speed, varies from 10 to 330 in increments of 20. These parameters generated the Pareto-optimal recall/QPS curve over a parameter sweep with $R_{\text{small}}, R_{\text{stitched}} \in \{32, 64, 96\}$ and $L_{\text{small}}$ between 50 and 100.

(2) FilteredVamana [36]: The index is built with $L = 90$ and a degree bound of $R = 96$. This was the Pareto-optimal choice for recall/QPS curve from a parameter sweep over $R \in \{32, 64, 96\}$ and $L$ between 50 and 100. To generate the search Recall/QPS curves, we use FilteredGreedySearch and vary $L$ from 10 to 650 in increments of 20.

(3) IVF Inline-Processing [1]: Since the Prep, Dann and Turing datasets had roughly 1-3 million points, and the recommended number of clusters is $\sqrt{n} \approx 2000$, we ran experiments

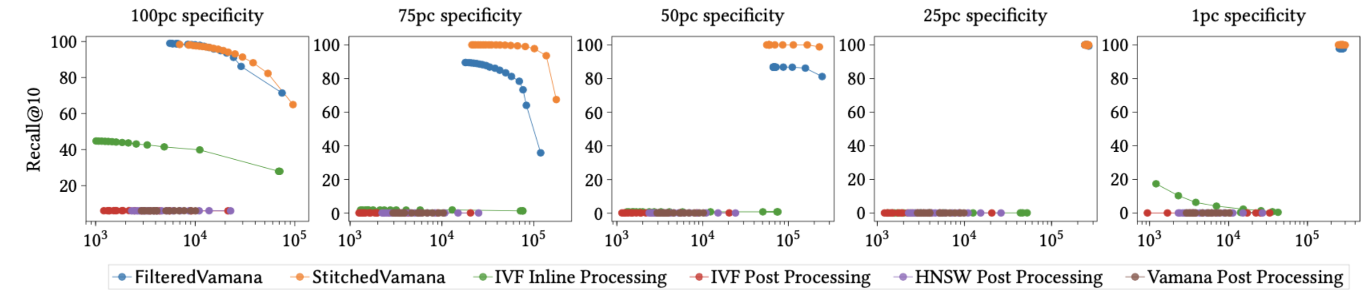


**Figure 1: Turing dataset: QPS (x-axis) vs recall@10 for various algorithms with filters of 100, 75, 50, 25 and 1 percentile specificity.**

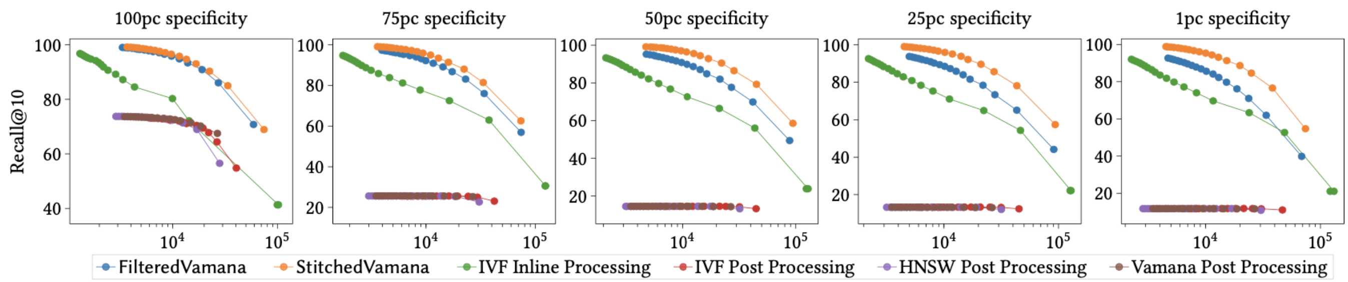


**Figure 2: Prep dataset: QPS (x-axis) vs recall@10 for various algorithms with filters of 100, 75, 50, 25 and 1 percentile specificity.**

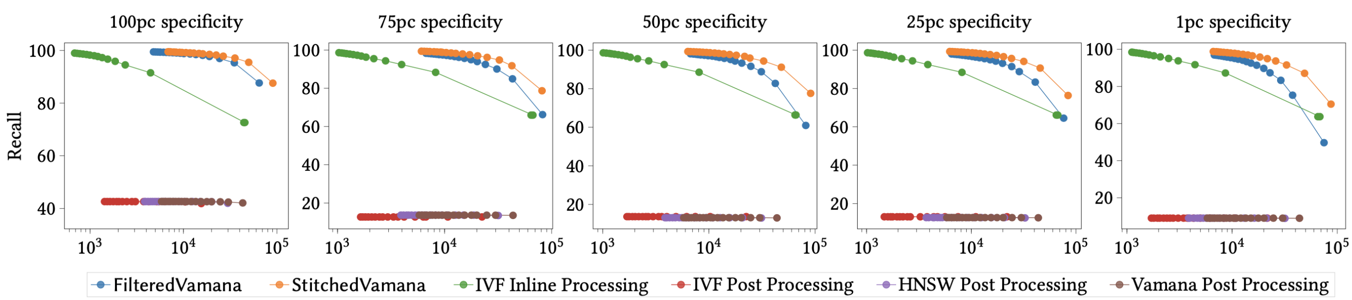


**Figure 3: DANN dataset: QPS (x-axis) vs recall@10 for various algorithms with filters of 100, 75, 50, 25 and 1 percentile specificity.**

with number of clusters in $\{1024, 2048, 4096, 8192\}$, and 4096 clusters had the best QPS/recall curve. The number of probes for searching was varied between 20 to 280.

(4) IVF post-processing with FAISS IVF [29]: 4096 clusters, with no. of probes varying from 10 to 350 in increments of 20.
(5) NHQ [4]: We use the build parameters recommended in [42]. To generate the Recall/QPS curves, we vary $L$ between 10 to 310 in intervals of 20. We have been unable to reproduce the results presented in [42]. See subsection A.1 for details.
(6) HNSW post-processing with FAISS HNSW [29]: built with the parameter efConstruction set 150 and $M$ set to 100, so that the build times were similar to (1) and (2). Search was done with $L$ from 10 to 350 in steps of 20.
(7) Milvus [3]: Parameters are described in subsection A.2

## 5.3 Comparison With Existing Approaches

We plot the tradeoff between recall and query throughput as measured in Queries per second (QPS) for the algorithms above. Index build times are reported in Table 2. Due to extremely low QPS, all Milvus and NHQ plots are left to the appendix, since it is difficult to plot them alongside other algorithms. In addition, post-processing approaches perform poorly across all evaluations in this scope, so comparison with them is omitted unless there is something of note.

*5.3.1 Filtered Queries on Turing.* Figure 1 shows the downside of the post-processing and the inline-processing approaches for filtered query on filters with extremely low specificity. These approaches have to search a large number of the space in order to find valid results. On the other hand, both FilteredVamana and StitchedVamana achieve 90%+ recall as specificity ranges from

| Alg./Data | Dann | Prep | Turing | Audio | SIFT |
|---|---|---|---|---|---|
| **FilteredVamana** | 159.8 | 66.6 | 103.4 | 1.3 | 44. |
| **StitchedVamana** | 469.9 | 222.6 | 295.9 | 1.6 | 24.4 |
| **NHQ** | NA | NA | NA | 1.1 | 24.4 |
| **Milvus HNSW** | 153.6 | 49.3 | NA | 5.5 | 72.0 |
| **Faiss HNSW** | 158.6 | 44.5 | 188.0 | 1.1 | 71.1 |

**Table 2: Build times in seconds for Filtered Vamana, Stitched Vamana, NHQ, Milvus HNSW and Faiss HNSW.**

$10^{-1}$ to $10^{-6}$, while the other approaches fail to achieve any meaningful accuracy, and have almost a 1000x lower QPS for the low specificity labels.

*5.3.2 Filtered Queries on PREP.* For 90% recall, Figure 2 shows that FilteredVamana performs 2.5x better than the next best prior technique – IVF inline processing – and StitchedVamana performs 6x better. Both the algorithms in this paper are substantially better than all prior techniques over a range of recall.

*5.3.3 Filtered Queries on DANN.* For 90% recall, Figure 3 shows that FilteredVamana performs around 3x better than IVF inline processing, and StitchedVamana performs around 7.5x better.

Overall, the results establish that both algorithms presented in this paper improve upon the recall to QPS ratio by an order of magnitude or more over a wide range of parameters and datasets.

## 5.4 Comparing FilteredVamana and StitchedVamana

*5.4.1 Dataset Comparisons.* StitchedVamana overlays per-label sub-graphs then prunes the overlaid graph, while FilteredVamana builds a single index where neighbors of a given vertex are decided based on both geometric structure as well as common labels. While both perform well on real-world datasets (Figure 3, Figure 2) StitchedVamana consistently offers better QPS for recall@90 by a factor of 2. The total indexing time for FilteredVamana is faster than StitchedVamana, across both datasets, as shown in Table 2.

*5.4.2 Examining Performance on Uncorrelated Labels.* Some existing ANNS solutions such as Milvus perform a pre-processing step wherein they rely on the distribution of the labels amongst the points for faster filtered search[41]. Such approaches will naturally experience some degradation or loss in efficiency if new queries do not follow this distribution. We show that while both FilteredVamana and StitchedVamana are robust to this possibility, FilteredVamana is slightly better.

We conducted a simple experiment to demonstrate this. Consider a dataset $P = \{x_1, \ldots, x_n\}$ and the associated label sets $\{F_{x_1}, \ldots, F_{x_n}\}$. Let $\mathcal{D}_1$ be the discrete distribution of the number of labels per point, and let $\mathcal{D}_2$ be the discrete distribution corresponding to the proportion of each label in the dataset. We then construct a new label set $\{F'_{x_1}, \ldots, F'_{x_n}\}$ in the following manner: for each point $x \in P$, sample the number of labels $x$ must have from the distribution $D_1$. Then sample labels without replacement from $\mathcal{D}_2$ until we obtain $|F'_x|$ labels. Label sets constructed in such a manner will have less correlation with the actual points and clusters in the dataset, and the labels themselves are assigned to each point somewhat independently.

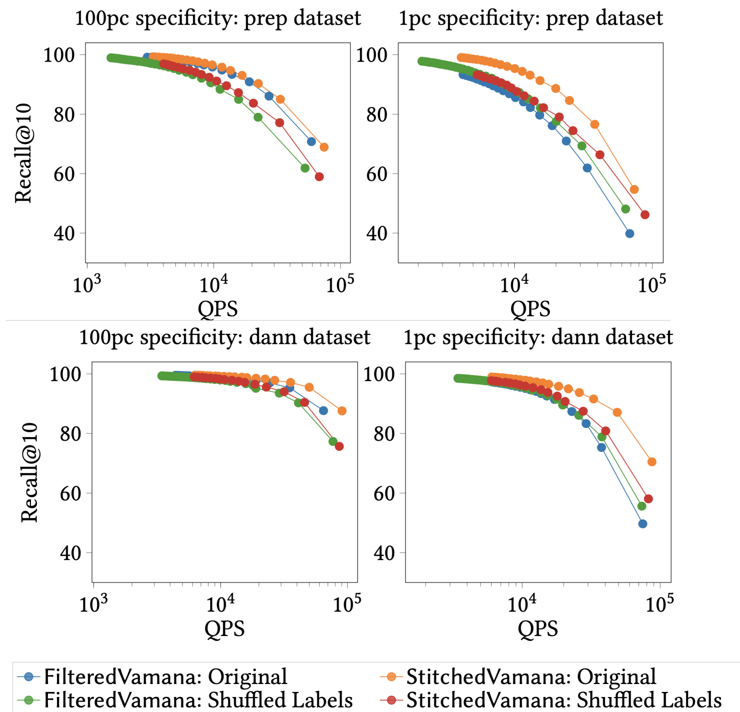


**Figure 4: Shuffled Experiment for Dann and Prep: QPS vs recall@10 for FilteredVamana and StitchedVamana with filters of 100 and 1 percentile specificity, but with shuffled labels.**

The results in Figure 4 show that for the Prep dataset, FilteredVamana shows more robustness to the shuffling of the labels. The recall/QPS curve barely changes in comparison to StitchedVamana, which has lower QPS after the shuffle. However, for the DANN dataset, there is minimal change for both approaches.

*5.4.3 Performance on Unfiltered Queries.* In addition to being fairly robust to the distribution of the labels, these algorithms also work relatively well for *unfiltered* search despite being designed for filtered search. Figure 5 compares both the algorithms we propose for Filtered search with Vamana, which is explicitly designed for unfiltered search. For both the Prep and DANN datasets, StitchedVamana supports 95% recall@10 at around 0.9 times the query throughput (QPS) of Vamana, while FilteredVamana is able to achieve the same recall at around 0.8x the QPS of Vamana.

*5.4.4 Streaming Indices.* While on QPS and recall, StitchedVamana outperforms FilteredVamana in most situations, FilteredVamana has an advantage that is likely to make it more useful in practice: dynamic index growth via point insertions. It is easier to ensure the principle of filter subgraph navigability for FilteredVamana: the set intersection requirement is inherently localized to the neighbors of a point, and it is easy to account for along with the geometric requirements in the dynamic setting. However, for StitchedVamana, we risk breaking the structure of the subgraphs, from which much of the performance advantage of StitchedVamana is gained over FilteredVamana. We leave a more detailed evaluation of the dynamic setting deletions as a possible avenue for future work.

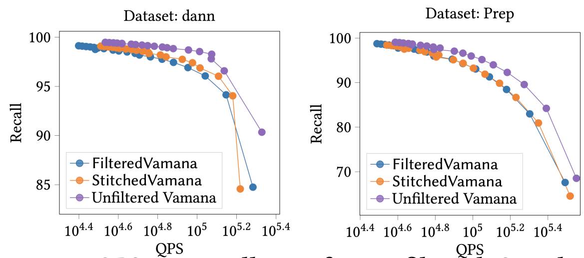


**Figure 5: QPS vs recall@10 for Unfiltered Search on** FilteredVamana **and** StitchedVamana **built on original labels.**

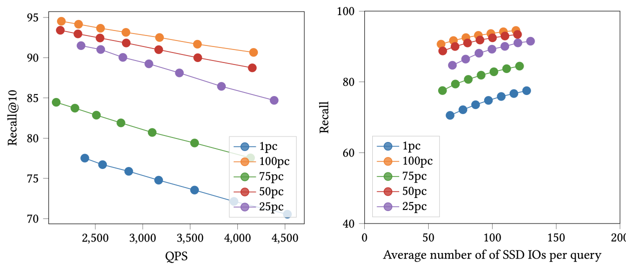


**Figure 6: Performance of Filtered-DiskANN on a larger 28 million point DANN dataset on filters of various specificity.**

## 5.5 SSD based indices

It is often necessary to index and query datasets much larger than the DRAM. The DiskANN [36, 39] system makes it is possible to do so cost-effectively by using a hybrid DRAM-SSD indices that require little DRAN. It internally uses the Vamana graph placed on SSDs and a compressed representation of points in the DRAM to answer queries accurately with latency. It is straight-forward to place the graph algorithms described in this paper in the DiskANN framework. In fact, several large scale deployments effectively use such a strategy which we term Filtered − DiskANN. Figure 6 demonstrates its performance on larger scale 28 million point Dann dataset. Filtered search was with run 24 threads on a machine with Intel E5-2673v3 CPUs and a local SSD with beamwidth 4 and search parameter $L$ varying between 40 and 100 in increments of 10.

# 6 ONLINE A/B TEST IN SPONSORED SEARCH

To measure the efficacy of FilteredVamana in an industrial setting, we conducted online A/B tests on live traffic of a sponsored search engine. Search engines generate most of their revenue via sponsored advertisements (ad). Each ad can be allowed to serve in one or more geographical regions (countries) based on advertiser's preference. Selecting relevant ads for a query is an important problem. Here relevance has multiple connotations, including intent-match between query and ads, targeting-match (e.g., match in location of user and allowed targeted regions for ads).

The production system uses ANNS index to select ads from a large ad corpus. We create one ANNS index with ads from 47 regions. Creating separate indices for each region is inefficient and expensive as a large fraction of ads are targeted in more than one region. A twin-tower encoder based on [26, 31] creates the dense embeddings for ads optimizing for intent-match. The baseline system uses post-processing to filter on target regions which helps towards targeting-match. As described earlier, post-processing has sub-optimal recall when strict latency budgets are to be met.

| Number of Different Regions | Pct. incr. in clicks | Pct. incr. in revenue |
|---|---|---|
| 47 | 34.61% (0.03) | 48.95% (0.009) |

**Table 3:** FilteredVamana **performance improvement over current production system for ANNS retrieval**

| Region's share in Index | Pct. incr. in clicks | Pct. incr. in revenue |
|---|---|---|
| 3-9% (10) | 25.54% | 28.61% |
| 1-2% (10) | 54.07% | 46.67% |
| <1% (27) | 70.67% | 79.77% |

**Table 4: Performance improvement on three subgroups of regions based on their along with subgroup-wise performance improvement. Number in brackets indicate subgroup size.**

We deployed FilteredVamana based indexes containing 47 filter labels (target regions) using the same encoder. Table 3 shows the relative improvement in clicks and revenue with respect to baseline production system. Numbers in brackets indicate the P-Value. P-Value below 5e-2 is considered significant in the production system. The data was collected over a period of two weeks and aggregated across all the target regions. The significant increase in clicks and revenue demonstrates the effectiveness of FilteredVamana.

Since the baseline system uses post-filtration, there is bias towards retrieving ads targeted in regions that have large index representation. This leads to heavy filtration downstream for queries targeting a region with smaller representation. FilteredVamana by design should work well for these smaller represented regions. To test this hypothesis, we further grouped target regions into 3 subgroups based on their index representation. Table 4 confirms that smaller regions see larger gains with FilteredVamana they now get a fair representation and all retrieval complies by targeting-match.

# 7 CONCLUSIONS AND FUTURE WORK

We have demonstrated that it is possible to build extremely efficient graph-based ANNS indices to support hybrid ANNS queries. The performance and accuracy improvements over baselines are significant and consistent across many real-world data sets and a range of values of filter specificity. This has a large positive impact on production systems. Support for filter sets larger than several thousands and support for more complex SQL-like filter expressions with the efficiency of graph indices remain challenging open problems. While ideas presented here may be relevant to the fullt dynamic setting with deletes (as in [38]), detailed evaluation remains future work.

# ACKNOWLEDGMENTS

We thank Gopal Srinivasa for help with deploying the code. We thank the Microsoft DLVS and Turing teams, specifically Fei Teng, Youngji Kim, Rachel Rong, Shi Zhang, Renan Santana, Mingqing Li, for helpful discussions and access to the Turing dataset.

# REFERENCES


[1] 2022. GRANN ANNS Library. https://github.com/rakri/grann. Commit: bce52e83896bb5af27942e9f20f117fa27db6ad4.

[2] 2022. Milvus-docs: Conduct a Hybrid Search. https://github.com/milvus-io/milvus-docs/blob/v2.1.x/site/en/userGuide/search/hybridsearch.md

[3] 2022. Milvus Repository. https://github.com/milvus-io/milvus Commit: 8ac30397dd7eef84251bf1e9bdb988a8f3946b75.

[4] 2022. NHQ. https://github.com/AshenOn3/NHQ Commit: a87d48be3e73fcec1078803d8efcb993c231c8e4.

[5] 2022. Vearch Doc Operation: Search. https://vearch.readthedocs.io/en/latest/use_op/op_doc.html?highlight=filter#search

[6] 2022. Vespa use cases: Semi-structured navigation. https://docs.vespa.ai/en/attributes.html

[7] 2022. Weaviate Documentation: Filters. https://weaviate.io/developers/weaviate/current/graphql-references/filters.html

[8] 2022. Weaviate: Filtered Vector Search. https://weaviate.io/developers/weaviate/current/architecture/prefiltering.html

[9] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *Commun. ACM* 51, 1 (Jan. 2008), 117–122. https://doi.org/10.1145/1327452.1327494

[10] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1225–1233. http://dl.acm.org/citation.cfm?id=2969239.2969376

[11] Alexandr Andoni and Ilya Razenshteyn. 2015. Optimal Data-Dependent Hashing for Approximate Near Neighbors. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing* (Portland, Oregon, USA) *(STOC '15)*. ACM, New York, NY, USA, 793–801. https://doi.org/10.1145/2746539.2746553

[12] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (Austin, Texas, USA) *(SODA '93)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 271–280. http://dl.acm.org/citation.cfm?id=313559.313768

[13] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020). http://www.sciencedirect.com/science/article/pii/S0306437918303685

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

[15] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. *CoRR* abs/1802.02422 (2018). arXiv:1802.02422 http://arxiv.org/abs/1802.02422

[16] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517. https://doi.org/10.1145/361002.361007

[17] Erik Bernhardsson. 2018. *Annoy: Approximate Nearest Neighbors in C++/Python*. https://pypi.org/project/annoy/ Python package version 1.13.0.

[18] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover Trees for Nearest Neighbor. In *Proceedings of the 23rd International Conference on Machine Learning* (Pittsburgh, Pennsylvania, USA) *(ICML '06)*. Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/1143844.1143857

[19] James Briggs. 2022. The Missing WHERE Clause in Vector Search. https://www.microsoft.com/en-us/research/blog/turing-bletchley-a-universal-image-language-representation-model-by-microsoft/

[20] Kenneth L Clarkson. 1994. An algorithm for approximate closest-point queries. In *Proceedings of the tenth annual symposium on Computational geometry*. 160–164.

[21] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. 577–586.

[22] Matthijs Douze, Jeff Johnson, and Hervé Jegou. 2017. Faiss: A library for efficient similarity search. [Online; accessed 29-March-2017].

[23] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *Proc. VLDB Endow.* 13, 3 (2019), 403–420. https://doi.org/10.14778/3368289.3368303

[24] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graphs. *PVLDB* 12, 5 (2019), 461 – 474. https://doi.org/10.14778/3303753.3303754

[25] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 4 (2014), 744–755. https://doi.org/10.1109/TPAMI.2013.240

[26] Jiafeng Guo, Yinqiong Cai, Yixing Fan, Fei Sun, Ruqing Zhang, and Xueqi Cheng. 2022. Semantic models for the first-stage retrieval: A comprehensive review. *ACM Transactions on Information Systems (TOIS)* 40, 4 (2022), 1–42.

[27] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (Dallas, Texas, USA) *(STOC '98)*. ACM, New York, NY, USA, 604–613. https://doi.org/10.1145/276698.276876

[28] Qing-Yuan Jiang and Wu-Jun Li. 2015. Scalable Graph Hashing with Feature Transformation. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, 2248–2254.

[29] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734* (2017).

[30] Wei Liu, Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2011. Hashing with graphs. In *ICML*.

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

[32] Yury A. Malkov and D. A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR* abs/1603.09320 (2016). arXiv:1603.09320 http://arxiv.org/abs/1603.09320

[33] M. Muja and D. G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (2014), 2227–2240.

[34] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. 2015. Neighbor-Sensitive Hashing. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 144–155. https://doi.org/10.14778/2850583.2850589

[35] Aviad Rubinstein. 2018. Hardness of Approximate Nearest Neighbor Search. *CoRR* abs/1803.00904 (2018). arXiv:1803.00904 http://arxiv.org/abs/1803.00904

[36] Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, Andrija Antonijevic, Dax Pryce, David Kaczynski, Shane Williams, Siddarth Gollapudi, Varun Sivashankar, Neel Karia, Aditi Singh, Shikhar Jaiswal, Neelam Mahapatro, Philip Adams, and Bryan Tower. 2023. DiskANN: Scalable, Efficient and Feature-rich Approximate Nearest Neighbor Search. https://github.com/Microsoft/DiskANN Version 0.5.

[37] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. 2022. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. https://doi.org/10.48550/ARXIV.2205.03763

[38] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *CoRR* abs/2105.09613 (2021). arXiv:2105.09613 https://arxiv.org/abs/2105.09613

[39] Suhas Jayaram Subramanya, Fnu Devvrit, Rohan Kadekodi, Ravishankar Krishnawamy, and Harsha Vardhan Simhadri. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 13748–13758. http://papers.nips.cc/paper/9527-rand-nsg-fast-accurate-billion-point-nearest-neighbor-search-on-a-single-node

[40] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming Similarity Search over One Billion Tweets Using Parallel Locality-Sensitive Hashing. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1930–1941. https://doi.org/10.14778/2556549.2556574

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

[42] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2022. Navigable Proximity Graph-Driven Native Hybrid Queries with Structured and Unstructured Constraints. *arXiv preprint arXiv:2203.13601* (2022).

[43] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.

[44] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165. https://doi.org/10.14778/3415478.3415541

[45] Wei Wu, Junlin He, Yu Qiao, Guoheng Fu, Li Liu, and Jin Yu. 2022. HQANN: Efficient and Robust Similarity Search for Hybrid Queries with Structured and Unstructured Constraints. *arXiv preprint arXiv:2207.07940* (2022).

[46] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A Fast and Accurate LSH Framework for High-Dimensional Approximate NN Search. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 643–655. https://doi.org/10.14778/3377369.3377374

# A APPENDIX

## A.1 Comparison with NHQ KGraph

In [42], the authors propose two graph algorithms for filtered ANNS: NHQ-NPG_NSW and NHQ-NPG_KGraph. In all their experiments, the KGraph algorithm had a much better Recall/QPS profile than the NSW algorithm. We thus benchmark FilteredVamana and StitchedVamana against KGraph on 5 of the datasets used in [42]. We also note that while [45] apparently offers an improvement over [42], we have not found publicly available code to evaluate their results.

Both FilteredVamana and StitchedVamana were run with the same build parameters as described in subsection 5.2, while KGraph was built with the default parameters as suggested in the NHQ codebase [4]. The search parameter $L$ for KGraph is varied from 50 to 130 in intervals of 10, and from 10 to 50 in intervals of 5 for FilteredVamana and StitchedVamana.

As seen in Figure 7, the QPS of the Vamana algorithms is an order of magnitude higher for 100 recall.

Further, we conduct a simple build normalized experiment. On the NHQ datasets, we modified the parameters of FilteredVamana to ensure similar build time as NHQ-KGraph. We observed that FilteredVamana has much higher QPS, as seen in Figure 8.

## A.2 Comparison with Milvus Algorithms

Here, we present the results of our experiments using some of the Milvus algorithms [3] with filtered search on several datasets, including the real world datasets Prep and Dann, as well as the NHQ datasets Audio, SIFT1M, Paper and Msong. We compare 4 Milvus algorithms with the build and search parameters listed below. Refer to the Milvus documentation [2] for further information about the Milvus parameters.

1. Milvus HNSW: The index was built with degree bound $M$ = 64 (the maximum permissable value) and efConstruction = 250, while the search parameter ef was varied from 10 to 50 in intervals of 5.
2. Milvus IVF FLAT: The index was built with number of clusters nlist = 2000, while the search parameter nprobe was varied from 10 to 450 in roughly intervals of 50.
3. Milvus IVF SQ8: The index was built with number of clusters nlist = 2000, number of factors of product quantization $m$ = 16 or 20 (depending on the dataset dimension) and the number of bits in which each low dimensional vector is stored nbits = 8), while the search parameter nprobe was varied from 10 to 450 in roughly intervals of 50.
4. Milvus IVF PQ: The index was built with number of clusters nlist = 2000, while the search parameter nprobe was varied from 10 to 450 in roughly intervals of 50.

The results of our Milvus experiments are seen in Figure 10, Figure 9 and Figure 11. Even with 48 threads, we were unable to get very high QPS for the Milvus algorithms. Since the QPS was less than 300 across datasets for the Milvus algorithms (orders of magnitude lower than the Vamana algorithms), we have omitted the Vamana curves here to avoid scaling issues with the figures.

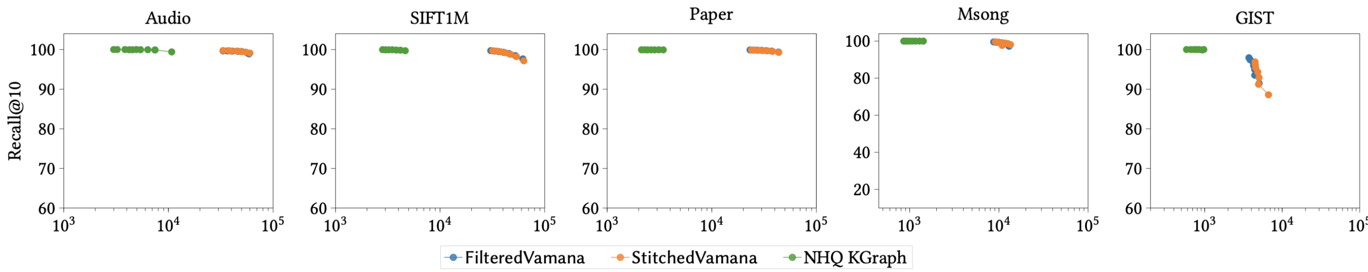


**Figure 7: KGraph on NHQ datasets: QPS (x-axis) vs recall@10 for NHQ KGraph,** FilteredVamana **and** StitchedVamana**.**

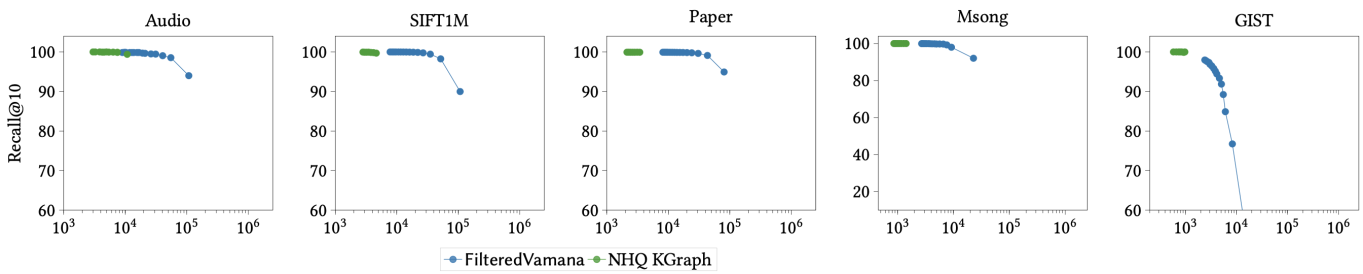


**Figure 8: KGraph and Filtered Vamana: QPS (x-axis) vs recall@10 on NHQ datasets (Build Normalized).**

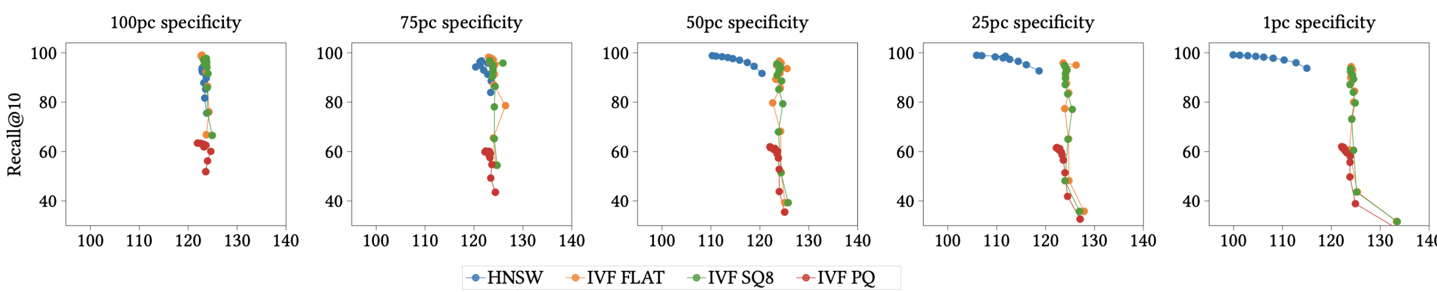


**Figure 9: Milvus algorithms on Prep dataset: QPS (x-axis) vs recall@10 with filters of 100, 75, 50, 25 and 1 percentile specificity.**

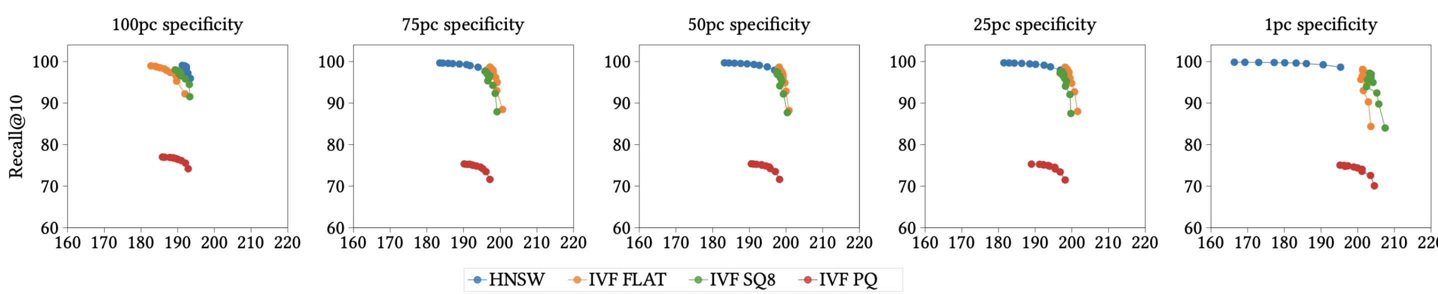


**Figure 10: Milvus algorithms on DANN dataset: QPS (x-axis) vs recall@10 with filters of 100, 75, 50, 25 and 1 percentile specificity.**

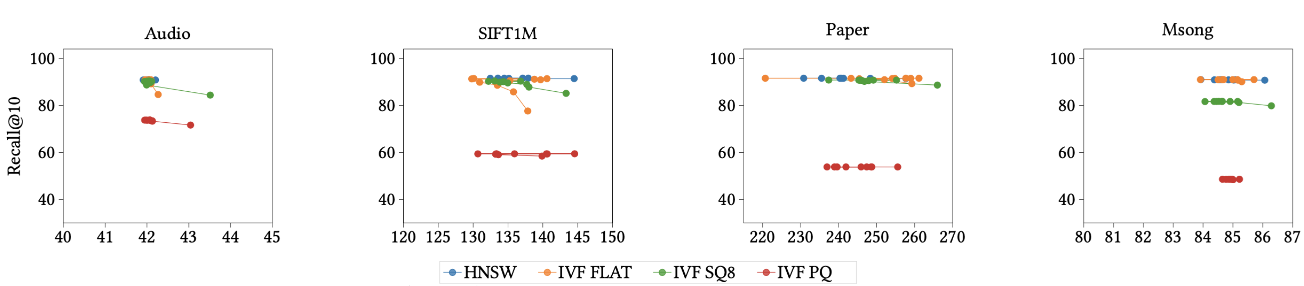


**Figure 11: QPS (x-axis) vs recall@10 for Milvus algorithms with 4 NHQ datasets.**