Tag-Filtered Approximate Nearest Neighbor Search

Jiarui Luo
Rutgers University
jl3288@scarletmail.rutgers.edu

Miao Qiao
The University of Auckland
miao.qiao@auckland.ac.nz

Chaoji Zuo
Rutgers University
chaoji.zuo@rutgers.edu

Dong Deng
Rutgers University
dong.deng@rutgers.edu

***Abstract*—Approximate Nearest Neighbor Search (ANNS) plays an important role in the search and recommendation of objects represented with high-dimensional vectors. For objects that are associated with tags such as the origin location, color, and type, it is common to perform ANNS with tag constraints, i.e., conduct search on objects that carry the query tags. We call such search Tag-Filtered Approximate Nearest Neighbor Search (TFANNS). The state-of-the-art TFANNS method Filtered-DiskANN is a graph-based method which suffers from a low recall for queries with low-to-medium frequent tags. Pre-filtering on these tags could boost the recall but lead to a large memory footprint. To address this issue, we propose three strategies in constructing a graph that strikes a balance between the performance and memory footprint; note that we are the first work on tag-frequency-aware graph-based indexing for TFANNS. Our extensive experiments show the superiority of our proposed methods over existing baselines: under ≥ 0.9 recall, our QPS is up to 13 times that of the best baseline.**

***Index Terms*—ANNS, tag frequency, graph-based indexing**

I. Introduction

Given a set P of n vectors in d-dimensional space $\mathbb{R}^d$, Nearest Neighbor Search (NNS) reports, for a query vector q in $\mathbb{R}^d$, k vectors in P with the shortest distances to q. With deep learning effectively embedding objects such as videos, images, and products, into high dimensional space such that similar objects have similar representations, NNS becomes a routine for applications such as document retrieval, image search, and recommendation systems [1]–[4]. Objects in these applications are often associated with metadata in the form of keywords or attributes such as the type of an image, the product category, or the music genres. Therefore, NNS can be raised with tag constraints. For example, in recommendation systems, an NNS query may specify preferences, genres, or product categories to retrieve personalized suggestions; in large collections of publications, an NNS may need to report semantically similar documents with a specified topic or author. For these scenarios, each vector p in P is associated with a set $\mathsf{t}(p)$ of tags and an NNS query may require the NNS to be conducted among the vectors whose tag sets satisfy specific constraints. We call the NNS with tag constraints Tag-Filtered NNS: For a query $Q(k, q)$ with a query vector q that is associated with a tag set $t(q)$, and an integer k, report the k nearest neighbors of q in the vectors $P[\mathsf{t}(q)]$ that carry tag $\mathsf{t}(q)$ where $P[\mathsf{t}(q)] = \{p \in P | \mathsf{t}(q) \subseteq \mathsf{t}(p)\}$. An efficient solution to Tag-Filtered NNS serves as a building block for, and is essential to, NNS with complicated tag constraints.

Approximation is introduced to NNS on large datasets to enable a shorter delay. This is because exact NNS has an inherent complexity barrier – asymptotically, NNS can hardly be processed significantly faster than a linear scan under strong exponential hypothesis [5]. Approximate Nearest Neighbor Search (ANNS) has been extensively studied [6], [7]. Graph-based ANNS such as HNSW [8] has been widely adopted due to its superior performance.

This paper considers Tag-Filtered Approximate Nearest Neighbor Search (TFANNS) where the aim is to efficiently report, for a query $Q(k, q)$, a set R of k vectors with a high recall to the real k-nearst neighbors (kNN) of q in $P[\mathsf{t}(q)]$. The state-of-the-art TFANNS is StitchedVamana [9]. It constructs a graph-based index (based on DiskANN [10]) with a tag-sensitive pruning strategy. Specifically, in DiskANN, the nearest neighbors S of a vector p undergo a pruning process, after which the remaining vectors are retained as the neighbors of p in the navigation graph of DiskANN. The pruning process lies on a domination relation: for two nodes u and v in S, u dominates v if 1) p is closer to u than to v and 2) v is closer to u than to p. In such a case, v will be pruned. The pruning process eliminates the dominated vectors in S and the M (M is a parameter indicating the maximum degree of the navigation graph) remaining nodes that are closest to q will be the neighbors of p. The critical change StitchedVamana made to the pruning process of DiskANN is to add an additional tag-related condition to the domination relation, i.e., 3) $\mathsf{t}(v) \subseteq \mathsf{t}(u)$. In other words, u dominates v requires 1)-3). With this tag-sensitive pruning, StitchedVamana conceptually stitches the indexes built for different tags together into a single graph. StitchedVamana performs well for tags that are frequent enough; for tags with low-to-medium frequency, however, StitchedVamana cannot ensure the connectivity of the graph with regard to the tags.

Example 1: Figure 2(a) illustrates a scenario where the dataset contains two tags blue and green of colors. The maximum degree M of the graph of StitchedVamana is 3. There are 6 vectors with only the blue tag and 3 vectors with both tags blue and green. The subgraph induced by vectors with the blue tag is well-connected. The subgraph corresponding to the green tag has no edge. This is because the vectors with green tag are not among each other's top-3 nearest neighbors. Consequently, the green tag subgraph suffers from poor connectivity, which can result in a low recall for queries involving the green tag. Such a scenario often appears on tags with low-to-medium frequency.

A straightforward remedy is to perform pre-filtering over low-to-medium frequency tags; however, it leads to a large

memory footprint and is thus not practical. Specifically, we construct, for each tag t whose frequency is lower than a given cutoff value, a graph index on $P[t]$ and resort to StitchedVamana for the other high frequency tags. Figure 1 shows the query performance with cutoff values 0, 1K, 10K, and 100K. We used the single-tag queries on a real-world dataset YFCC [11]. In the figure, the y-axis is the recall, while the x-axis shows the frequency of the query tag. To get a recall above 60%, the cutoff needs to be $\geq$ 100K, leading to an index size of 82G, 11 times the index size of StitchedVamana (i.e., when the cutoff is 0). It means for StitchedVamana+prefiltering, without paying massive memory overhead, it is hard to achieve an acceptable recall consistently for all the queries.

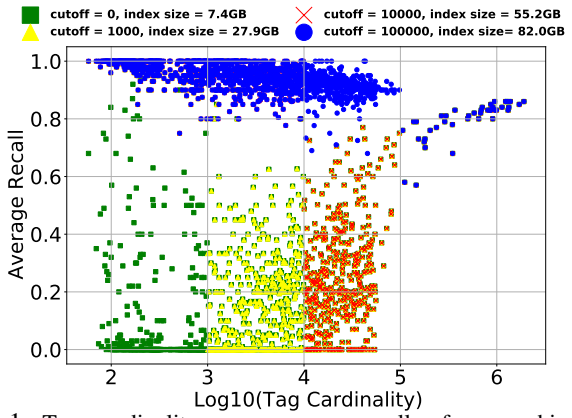
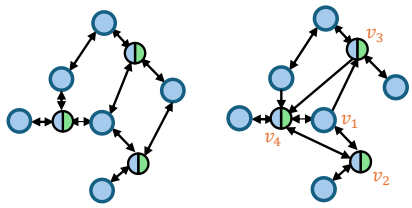


Fig. 1 Tag cardinality v.s. average recall of a combination of pre-filtering and StitchedVamana. For a cutoff value in $\{0, 10^3, 10^4, 10^5\}$, tags with frequencies smaller than the cutoff are coped with pre-filtering, while tags with frequencies larger than the cutoff are processed with StitchedVamana.

The poor performance in low-to-medium frequency tags is not unique to StitchedVamana. Other TFANNS approaches such as NHQ [12], ACORN [13], and Post-filtering HNSW [14] all suffer from this issue. The main reason is that existing approaches do not pay attention to the tag frequency, i.e., the construction of their index is tag-frequency-insensitive. To improve the performance for low-to-medium frequency tags, the indexing should allocate more resources to these tags to keep the connectivity of the graph w.r.t. these tags. However, it is yet to be known how to improve the recall for low-to-medium frequency tags while keeping the index size small.

This paper proposes three graph-based TFANNS approaches that are sensitive to the tag frequencies to boost the performance of queries with low-to-medium frequency tags. Specifically, we propose three different schemes of node degree allocation based on tag frequencies in the index construction. 1) The local method builds the index in a node-centric way, i.e., each node has the maximum degree M and the allocation of M to the neighbors of a node p is based on the frequency of tags of p. 2) The global method builds the index in a tag-centric way, i.e., each tag will have a maximum degree (applied to all the vectors with this tag) proportional to a function over the tag frequency. 3) The packing method saves space by considering several vectors together to reduce the impact of variance of the number of tags of vectors: within each pack, we use tag-centric indexing, across different packs, we apply pack-centric indexing, i.e., each pack has the same size for index.



(a) StitchedVamana (b) Local

Fig. 2 Compare the graph structure of StitchedVamana with that of our Local edge allocation strategy when the maximum degree $M = 3$. There are two tags in the data, blue and green. (a) In the StitchedVamana graph, there does not exist any edge between two vectors with green tag. (b) In the Local allocation method, some edges exist between green tag vectors while the subgraph of vectors of blue tag remain connected.

Our contribution is summarized as below.

1) This is the first work discussing the impact of frequency distribution of the tags in indexing and their performance of TFANNS, especially for low-to-medium frequency .
2) We propose three graph-based TFANNS approaches to improve the recall especially for low-to-medium frequency tags with low memory overhead.
3) We conducted extensive experiments to evaluate the performance of our proposed approaches, and results show that they significantly outperform existing methods.

This paper is organized as follows. Section II defines the problem and introduces the building blocks of our approach. Section III introduces the local algorithm, Section IV the global algorithm and Section V the packing strategy. Section VII presents the related work, Section VI shows the experiment, and Section VIII concludes the paper.

II. Preliminary

Let $\mathcal{T}$ be the domain of all tags. Let $d > 0$ be an integer. Let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of n data vectors where each $p \in \mathcal{P}$ is a vector in $\mathbb{R}^d$ with a set of tags $\mathtt{t}(p) \subseteq \mathcal{T}$. For each tag $t \in \mathcal{T}$, its frequency is $\mathtt{f}(t) \doteq |\{p \in P | t \in \mathtt{t}(p)\}|$.

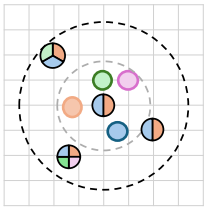


Fig. 3 TFANNS

Definition 1 (Tag-Filtered-NNS): A Tag-Filtered Nearest Neighbor Search (Tag-Filtered-NNS) query $Q = (k, q)$ consists of an integer k and a query vector q in $\mathbb{R}^d$ with a set of query tags $\mathtt{t}(q)$. It reports a set of data vectors $R \subseteq P$ with size $|R| = k$ such that they all carry tags $\mathtt{t}(q)$, i.e., $\forall v \in R, \mathtt{t}(q) \subseteq \mathtt{t}(v)$, and all vectors carry tags $\mathtt{t}(q)$ are more distant to q than nodes in R, i.e., for $\forall v \in P \setminus R$, either $\mathtt{t}(q) \not\subseteq \mathtt{t}(v)$ or $d(v, q) \geq d(u, q)$ for $\forall u \in R$.

Definition 2 (Tag-Filtered Approximate NNS (TFANNS)): Consider a dataset P of n vectors, a Tag-Filtered-NNS query $Q = (k, q)$. Let S be the exact result of Q. Consider a set R of vectors in P. Define the recall of R w.r.t. Q as $Recall@k = \frac{|R \cap S|}{k}$. The problem of (TFANNS) is to find a set $R \subseteq P$ that maximizes the recall with minimal delay.

Algorithm 1: PRUNE

Input: p: a vector; ANN: a set of p's approximate nearest neighbors; M: the maximum number of neighbors.
Output: $neighbors \subseteq$ ANN: p's neighbors after pruning.

```
neighbors ← ∅;
foreach v ∈ ANN in the ascending order of δ(p, v) do
    not_dominated ← true ;
    foreach u ∈ neighbors do
        if u dominates v w.r.t. p then
            not_dominated ← false and break ;
    if not_dominated then add v to neighbors;
    if |neighbors| ≥ M then break;
return neighbors;
```

Figure 3 shows an example of TFANNS. Each vector represents a vector in the 2D space. Each color represents a tag. The center vector is our query vector with two tags: blue and orange. To find out the three nearest neighbors of the query vector with the two tags, although there are four vectors close to the query vector in the gray circle, they do not satisfy the tag requirement. TFANNS expects to retrieve the three vectors between the gray and black circles, which are closest to the query vector while satisfying the tag constraint.

A. HNSW

The main structure of HNSW is a navigation graph G where each vector in P corresponds to a node in G. Given a query vector q, the search process starts from an entry node, and uses a priority queue of a fixed length efSearch to conduct a priority search – the priority of a node is its distance to q. The HNSW graph can be constructed by sequential vector insertion. When a new element p is inserted, HNSW first conducts an ANNS on p to fetch a set ANN of candidate neighbors and apply pruning algorithm to the candidates (Algorithm 1) based on a domination relation. For vector p, a node v is dominated by u if $d(p,v) > d(p,u)$ and $d(p,v) > d(u,v)$. The pruning process eliminates nodes in ANN that have been dominated by other nodes in ANN. At most M (a parameter) vectors will be kept in ANN, i.e., the remote ones may be truncated. As a result, the space complexity of HNSW graph is $O(nM)$.

III. LOCAL METHOD

This section describes an effective incremental algorithm for TFANNS called the local method. The core idea is to allocate node degrees (each node has degree M) to different tags such that each tag's subgraph is sufficiently connected for search. This idea is derived from the following observation.

Observation 1: Consider a tag t of a large frequency and and t' with a small frequency. Let v be a node with both tags t and t' who has a large number of neighbors carrying tag t and no neighbor carrying t'. Connecting v with an additional node with tag t has smaller marginal benefit than connecting v with a node with t' to keep the graph connected for TFANNS. In other words, in allocating the degree of each node as a resource to different tags, low-to-medium frequency tags are more important in keeping the graph connected for TFANNS.

To empirically confirm this observation, we randomly sampled vectors in SIFT1M with 7 different sizes and varied the maximum degree parameter M of HNSW, and show search performance. Figure 4 shows that when the recall is required to be ≥ 0.95, a small but sufficiently large maximum degree (e.g., $M = 16$) can achieve a search speed comparable to that of large maximum degree (e.g., M = 128). Thus, if we see the degree as resources (related to memory footprint of the index) the marginal gain of giving higher maximal degree resources to large frequency tags is diminishing. On the other hand, a reasonable degree allocation strategy should allow tags with low to medium cardinalities to "survive" the connectivity with a minimum degree (e.g., 1) without compromising the query performance for large-cardinality tags.

Algorithm 2: ALLOCATEDEGREE

Input: p: a vector with tags $\mathtt{t}(p)$; $\mathtt{f}(\cdot)$: the frequency function of all the tags inserted; θ: threshold parameter; M: the maximum degree of a vector.
Output: $\mathtt{M}_p = \{m_p(t)$: maximum degree of tag $\forall t \in \mathtt{t}(p)\}$.

```
len ← 0;
foreach t ∈ t(p) ∧ f(t) ≥ θ do
    l(t) ← ⌊log2(f(t))⌋; len+ = l(t);
foreach t ∈ t(p) ∧ f(t) ≥ θ do
    m_p(t) ← ⌊M*l(t) / len⌋;
return M_p = {m_p(t) | ∀t ∈ t(p)}
```

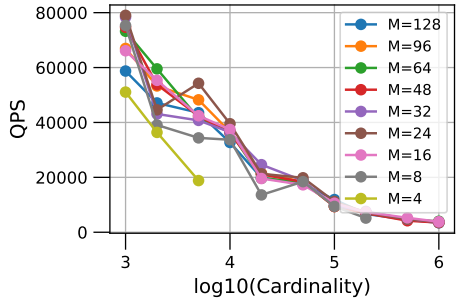


Fig. 4 Search performance comparison on different allocated degrees for different sizes of sampled data.

This observation leads us to our local method. It fixes the degree of each vector to be M, which limits the space complexity to $O(Mn)$. Intuitively, if a tag has more data vectors carrying it, the larger graph is needed to provide good search accuracy for the tag. Therefore, for a specific vector v, if one of its tags t has larger cardinality compared to its other tags, then t ought to have a higher priority in "degree allocation" of v, i.e., t should have a larger "maximum-degree". If a tag t has a small cardinality, as long as the frequency is larger than a threshold θ (i.e., we cannot afford the bruteforce search over $P[t]$), it should have at least 1 degree. How to allocate the degree? A heuristic way to do such allocation is to define the priority based on the cardinality of the tag. One design choice is to allocate proportional to the cardinality of the tag. However, consider two tags, one with cardinality 1K and another with cardinality 1M. In this case, the former small tag could hardly get 1 degree in the graph, which may result in poor performance on these tags. Therefore, this design choice may weight too much on large tags. The design we choose is to allocate degrees proportional to the log of the cardinality. The *log* function reduce the gaps between large cardinality and low cardinality and could

Algorithm 3: SEARCH

Input: q: query vector; $\mathsf{t}(q)$: query tags; k: output size; efSearch: Graph search parameter; $\mathcal{T}$: inserted tags; $\mathsf{f}(\cdot)$: the frequency function of all the tags inserted; G: the navigation graph; P: data vectors.

Output: the set of k approximate nearest neighbors of q.

1 $t \leftarrow$ The tag in $\mathsf{t}(q)$ with the lowest frequency;
2 $est \leftarrow$ the estimated # of vectors in P carrying all tags in $\mathsf{t}(q)$;
3 **if** $est < \theta$ **then**
4 Scan $P[t]$ to **Return** the k closest neighbor to q among the vectors in set $\{p \in P[t] | \mathsf{t}(q) \subseteq \mathsf{t}(p)\}$;
5 $L \leftarrow \emptyset$;/* Priority queue on pairs (vector, distance) */
6 $S \leftarrow \emptyset$; $V \leftarrow \emptyset$;
/* Randomly choose 5 entry vectors to start search */
7 **for** *a small number (e.g., 5) times* **do**
8 $x \leftarrow$ a random vector in $P[t]$;
9 $V \leftarrow V \cup x$; $L \leftarrow L \cup x$;
10 **if** $\mathsf{t}(q) \subseteq \mathsf{t}(x)$ **then** add x to S;
11 **while** $L \neq \emptyset$ **do**
12 $p \leftarrow$ the vector of the pair in L with minimum distance;
13 Delete the pair with the minimum distance from L;
14 **foreach** $r \in N_p(t) \setminus V$ **do**
15 $V \leftarrow V \cup r$;
16 Add $(r, \delta(q, r))$ to L;
17 **if** $\mathsf{t}(q) \subseteq \mathsf{t}(r)$ **then** Add r to S;
18 **if** $|L| >$ efSearch **then**
19 Delete the pair in L with the maximum distance;
20 **return** the set of k vectors in S that are closest to q;

provide reasonable allocation results. We empirically justified this design choice, please see Section VI C for details.

Based on this design choice, for a vertex p, the number of edges allocated for a "large" tag $t \in \{t' \in \mathsf{t}(p) | \mathsf{f}(t') \geq \theta\}$ is proportional to the portion of priority of t among all "large" tags $\{t' \in \mathsf{t}(p) | \mathsf{f}(t') \geq \theta\}$. We call this way Log Allocation, as Algorithm 2 describes. Mathematically, the degree of tag t for a vertex p will be

$$\mathsf{m}_p(t) = \begin{cases} 0 & \text{if } \mathsf{f}(t) < \theta, \\ \frac{M * \log \mathsf{f}(t)}{\sum_{t \in \{t' \in \mathsf{t}(p) | \mathsf{f}(t') \geq \theta\}} \log \mathsf{f}(t)} & \text{if } \mathsf{f}(t) \geq \theta. \end{cases}$$

Log Allocation increases the chance for small tags to receive edges, and as a result, abundant edges are provided in small tags' subgraphs to guarantee the search process. At the time, the search performance on large tags will not be influenced significantly since degree provided for them are sufficient.

Example 2: Figure 2(b) demonstrates our proposed local method. For vectors with only the blue tag (e.g., v_1 in the figure), the local method utilizes all 3 of its degrees to connect with blue-tagged neighbors. For vectors with both blue and green tags (v_2, v_3 and v_4), the method allocates degrees proportionally based on the log of tag cardinality: 2 degrees are assigned to blue-tagged neighbors and 1 degree to green-tagged neighbors, given the presence of 9 blue-tagged vectors and 3 green-tagged vectors. This ensures that each vector with both tags connects to its nearest green-tagged neighbor, thereby enabling search within the green-tag subgraph.

Index structure. The local algorithm maintains a structure of $(\mathcal{T}, \mathsf{f}(\cdot), G, P, M, \theta))$. $\mathcal{T}$ keeps all the tags that have appeared on inserted vector where all the inserted vectors and their tags are in P. f is a function that maps each tag $t \in \mathcal{T}$ to its frequency $\mathsf{f}(t)$, i.e., the number of vectors in P carrying tag t. The navigation graph G has nodes P – each vector in P is also a node in G. M is a parameter indicating that each vector in P will have at most M neighbors in the navigation graph. θ is a threshold indicating that only tags with frequency exceeding θ can appear in graph G.

Each node p of G has at most M neighbors, i.e., its maximum degree is M. A scheme $\mathtt{M}_p = \{m_p(t) | t \in \mathsf{t}(p)\ with\ \mathsf{f}(t) \geq \theta\}$ indicates the allocation of the degree to different tags where $\Sigma_{t \in \mathsf{t}(p)} m_p(t) \leq M$. The neighbors of p are also categorized based on tags, i.e., $\{N_p(t) | t \in \mathsf{t}(p)\ with\ \mathsf{f}(t) \geq \theta\}$. We store the index structure in a node-centric way, where the vector data of a node is stored followed by its M neighbors. In order to distinguish the tags these M neighbors belong to, we first store neighbors sharing the same tag continuously and then map each tag to its neighbor's starting index.

Search with the index. Because every node p in G has divided its neighbors into groups where each group $N_p(t)$ corresponds to the neighbors carrying a tag $t \in \mathsf{t}(p)$ of p with $f(t) \geq \theta$. We can divide the search process into three phases, query plan preparation, plan selection and plan execution. Algorithm 3 shows the search algorithm.

In query plan preparation, we pick the tag t in $\mathsf{t}(q)$ with the smallest frequency $f(t)$ as the base tag (Line 1). Since every vector satisfying $t(q)$ must contain tag t, picking the tag with smallest frequency can help us narrow down the search range from all the vectors to the vectors with tag t. This way, we generate two query plans. One brute-forcely scans all the vectors in $P[t]$ and use post-filtering to report the results; the other one searches with the graph index, i.e., the subgraph of G on the vectors carrying tag t.

To select between the two plans, we need to estimate the number of vectors in P that satisfy the constraints of $\mathsf{t}(q)$ (Line 2). Note that we are not aware of the cardinality of vectors that satisfy the $t(q)$ constraint since $t(q)$ may contain multiple tags. We use existing estimation method [15] (Line 1) to estimate the cardinality. The detailed estimation will be elaborated at the end of the section. The estimated cardinality should be no less than $\mathsf{f}(t)$ as otherwise the estimation can be trivially refined to $\mathsf{f}(t)$, and this is also because the vectors satisfying $t(q)$ must be a subset of vectors containing tag t.

With the estimated cardinality, we choose the query plan by comparing the estimation with the threshold θ, the threshold of index construction. If the estimation is no more than θ, we choose the brute-force scanning plan. Otherwise, we must have $f(t) \geq \theta$, i.e., the vectors of $P[t]$ has been index. We then use the graph index to search. During the execution of graph search, specifically, we use a priority queue L to start with a small constant number (5 in our case) of starting vectors randomly selected from P. The queue has a maximum length of efSearch, once it exceeds the limit, the candidate with the maximum distance to q will be removed (Lines 18-19). Apart from this, the search iteratively pick the vector p in L that has the smallest distance to q (Line 12) to expand L to all p's

Algorithm 4: LocalIns

Input: p: a vector with tags $\mathsf{t}(p)$; efCons: graph search parameter; $\mathcal{T}$: the inserted tags; $\mathsf{f}(\cdot)$: the frequency function of all the tags inserted; G: the navigation graph; P: vector set; M: the maximum degree of a node; θ: threshold parameter.
Output: Insert p to the structure $(\mathcal{T}, \mathsf{f}(\cdot), G, P, M, \theta)$.

1 **foreach** $t \in \mathsf{t}(p)$ **do**
2 **if** $t \notin \mathcal{T}$ **then** Add t to $\mathcal{T}$; $\mathsf{f}(t) \leftarrow 1$; **else** $\mathsf{f}(t) \leftarrow \mathsf{f}(t) + 1$;
3 **if** $\mathsf{f}(t) < \theta$ **then continue**;
4 $l(t) \leftarrow \lfloor \log_2(\mathsf{f}(t)) \rfloor$;
5 Add p to P; $\mathtt{M}_p \leftarrow$ AllocateDegree$(p, \mathsf{f}(\cdot), \theta, M)$;
/* Construct index for tags with frequency reaches θ */
6 **foreach** $t \in \mathsf{t}(p)$ *with* $f(t) = \theta$ **do**
7 **foreach** $r \in P[t]$ **do**
8 $\mathtt{M}_r \leftarrow$ AllocateDegree$(r, \mathsf{f}(\cdot), \theta, M)$;
/* If the # of neighbors of tag $t' \neq t$ exceeds the maximum degree of t' of r, prune */
9 **foreach** $t' \in \mathsf{t}(r) \setminus \{t\}$ *s.t.* $|N_r(t')| > m_r(t)$ **do**
10 $N_r(t') \leftarrow$ Prune$(r, N_r(t'), m_r(t'))$;
/* Populate the neighbors of r for tag t */
11 $candidates \leftarrow$ Search$(r, \{t\}, L,$ efCons$)$;
12 $N_r(t) \leftarrow$ Prune$(r, candidates, m_r(t))$;
13 **foreach** $u \in N_r(t)$ **do**
14 $N_u(t) \leftarrow N_u(t) \cup p$;
15 **if** $|N_u(t)| > M_u(t)$ **then**
 $N_u(t) \leftarrow$ Prune$(u, N_u(t), m_u(t))$;
/* For tags with existing indexes, insert p */
16 **foreach** $t \in \mathsf{t}(p)$ *such that* $f(t) > \theta$ **do**
17 $candidates \leftarrow$ Search$(p, \{t\}, L,$ efCons$)$;
18 $N_p(t) \leftarrow$ Prune$(p, candidates, M_p(t))$;
19 **foreach** $r \in N_p(t)$ **do**
20 $N_r(t) \leftarrow N_r(t) \cup p$;
21 **if** $|N_r(t)| > m_r(t)$ **then**
 $N_r(t) \leftarrow$ Prune$(r, N_r(t), m_r(t))$;

unvisited neighbors baring the root tag t (Lines 14-17). If a visited vector carries all the tags of $\mathsf{t}(q)$, then it will be added to S as candidates (Line 17) while only the top k vectors will be reported (Line 20).

Index Construction. Algorithm 4 shows the algorithm. When a vector p with tags $\mathsf{t}(p)$ arrives to the system, after maintaining the tag set $\mathcal{T}$ and frequency function f (Lines 1-3), we calculate $l(t)$ for a tag t if its frequency reaches the threshold θ (Line 4), in other words, it should be reflected in the navigation graph. We adopt the log function over the frequency. Line 5, i.e., Algorithm 2, allocates the degree of p, i.e., M, to different tags-above-the-threshold proportional to their $l(\cdot)$ values and shapes a scheme $\mathtt{M}_p = \{m_p(t) | t \in \mathsf{t}(p), \mathsf{f}(t) \geq \theta\}$. The scheme is used in constructing the neighbors of p in Lines 16-21. Lines 17-18 finds the neighbors of p w.r.t. t and Lines 19-21 add reverse edges to p's neighbors.

When a tag t's frequency reaches θ for the first time, we construct the index w.r.t. t (Lines 6-15): For each vector r with tag t, we squeeze the degree from other tags (Lines 9-10) if necessary before populating the neighbors (Lines 11-12); add reverse edges to the graph via Lines 13-15.

Cardinality Estimation. In order to select a query plan between these two methods, we used a simple cardinality estimator and query optimizer [15]. Consider query q's tags $t(q) = \{t_0, t_1, \cdots, t_m\}$ with cardinality $\mathsf{f}(t_0) < \mathsf{f}(t_1) < \cdots < \mathsf{f}(t_m)$. We estimate the cardinality $\mathsf{f}(\mathsf{t}(q))$ by the following formula $\mathsf{f}(\mathsf{t}(q)) = |\mathcal{P}| \prod_{i=0}^{m} (\mathsf{f}(t_i)/|\mathcal{P}|)^{1/2^i}$. Based on this cardinality estimation, query plan selection becomes easy. Pre-filtering is chosen when $\mathsf{f}(t(q)) < \theta$, where θ is the graph building threshold. $\mathsf{f}(t_0)$ is also taken into account because $\mathsf{f}(t(q)) \leq \mathsf{f}(t_0)$ always hold. Post-filtering is adopted otherwise.

(a) Local (b) Global

Fig. 5 Different edge allocation strategies: the center vector containing 4 tags, maximum degree $M = 3$. (a) Local method cannot deal with the case when the number of tags a vector (i.e., the center vector) has is larger than maximum degree (b) In global method, an extra edge budget is allocated to this vector to improve connectivity.

IV. Global Method

Although the local method can cope with TFANNS problem quite well in most real-world scenarios, the connectivity of tag's graph cannot be guaranteed. Therefore, there exists extreme scenarios when local method fails to report any result.

Example 3: Figure 5(a) shows a scenario where a vector (e.g., point v in the figure) has the number of tags larger than the maximum degree M. In this case, there must be tags (e.g., the green tag) that cannot have edges. For a query with $\mathsf{t}(q) = \{\mathsf{green}\}$, v has no neighbors though it has a green node in the surrounding which refrains the search performance. To solve this problem, more degrees are needed for the data vector (Figure 5(b)) to guarantee the connectivity of graph respected to low frequency tags. This section describes a global method to solve this problem, which uses the HNSW as a building block and build index in a tag-centric way.

Index structure. The global algorithm maintains a structure $(\mathcal{T}, \mathsf{f}(\cdot), \mathcal{G}, P, M_0, \theta, len)$. There are two differences between the global algorithm and the local algorithm. Firstly, instead of maintaining one navigation graph G, we maintain, for each tag t with frequency exceeding θ, a navigation graph, and the collection of which is denoted as $\mathcal{G} = \{G_t | t \in T, \mathsf{f}(t) > \theta\}$. Secondly, we allocate a global degree of M_0 to all the tags proportional to the log of their frequencies. Specifically, we set for each tag $t \in \mathcal{T}$ with $\mathsf{f}(t) \geq \theta$, $\mathsf{m}(t)$ maximum degree and maintain a HNSW index G_t with $M = \mathsf{m}(t)$ for tag t. The value of $\mathsf{m}(t)$ can vary within a range:

$$\mathsf{l}(t) \doteq \lceil \log_2 \mathsf{f}(t) \rceil, \text{ for } \forall t \in \mathcal{T};\ len \doteq \sum_{t \in \mathcal{T}, \mathsf{f}(t) > \theta} \mathsf{l}(t);$$

$$\mathsf{m}(t) < 2 \frac{M_0 \cdot \mathsf{l}(t)}{len}. \tag{1}$$

Note that for $\mathsf{f}(t) \geq \theta$, $\mathsf{l}(t) \in [\log_2 \theta, \log_2 n]$. For example, for $\theta = 1024$, $b = 2$, and $n < 10^9$, $\mathsf{l}(t) \in [10, 31]$. To ensure that there is an integer in $(0, 2\frac{M_0 \cdot \mathsf{l}(t)}{len})$ for $\mathsf{m}(t)$ to choose if $\mathsf{f}(t) = \theta$, the setup of θ and M_0 must ensure that

$$2M_0 \cdot \lceil \log_2 \theta \rceil > len. \tag{2}$$

The values of $\mathsf{l}(t)$, $\mathsf{m}(t)$ and len are maintained during the index construction.

Index construction. The main challenge of building the index for global method is to maintain the structure that up holds Equations 1- 2. Note that when a new vector arrives to the system (Line 1), for each tag $t \in \mathsf{t}(p)$, we maintain the tag set and the frequency function. When the frequency firstly reaches θ, we setup the values for $\mathsf{l}(t)$, maintains len, determine $\mathsf{m}(t)$ (Line 6), and then construct the HNSW graph G_t (Line 7). Due to the truncation when computing the value of $\mathsf{l}(t)$, only when $\mathsf{f}(t)$ is a power of 2 can increment the value of $\mathsf{l}(t)$, which may potentially trigger the reconstruction of the HNSW graph G_t (Lines 9-11). Most of the time $\mathsf{m}(t)$ can remain unchanged. When Equation 2 no longer holds, we may need to rebuild the entire index with a larger θ. The value of θ is doubled every time to find a valid value (Lines 13-16). After finding a working value of θ, we rebuild the entire index over all the tags with frequency over θ (Lines 17-20).

Lemma 1 (Space Complexity): Denote by f the total frequency of $\mathcal{T}$. The space of $\mathcal{G}$

$$\sum_{t \in T, \mathsf{f}(t) > \theta} \mathsf{f}(t)\mathsf{m}(t) \leq M_0 \min\{\frac{f}{\theta}, 2n\}.$$

The space complexity is thus $O(nM_0)$.

Proof 1: Because $\sum_{t \in \mathcal{T}, \mathsf{f}(t) \geq \theta} \mathsf{m}(t) < 2M_0$, we have

$$\sum_{t \in T, \mathsf{f}(t) \geq \theta} \mathsf{f}(t)\mathsf{m}(t) < 2nM_0.$$

Search with the index. The search is generally the same with Algorithm 3. The difference lines in that the navigation graph used for conducting the search of Lines 11-19 is on the graph G_t with tag t selected based on Line 1.

V. Packing Strategy

While the global method can cope with corner cases, it is not sufficiently cost-effective in index time. To this end, we propose packing strategy to combine the benefits of both local and global methods.

Packing strategy packs data vectors into blocks and treating tags with large frequency into a global mask tag. By packing data vectors into blocks, degree resources are shared among vectors inside the same block. As a result, when a data vector has many tags and needs more than M degrees, it could "borrow" degrees from data vectors in the same block that have few tags and need few degrees. Therefore, it could improve the utilization of degree resources and thus improve the search performance. On the other hand, packing large tags into a global mask tag avoids wasting degrees on the tags that are already frequent enough to ensure the graph connectivity. This introduce a new threshold parameter θ' controls how large the tags should be packed to the mask tag. The cardinality of a mask tag is defined as the large cardinality of tags.

Based on these two designs, the degree allocated to a tag of a vector is proportional to the size of the tag in the block.

Algorithm 5: GlobalInsert

Input: p: a vector with tags $\mathsf{t}(p)$; $\mathcal{T}$ inserted tags; $\mathsf{f}(\cdot)$: the frequency function of all the tags inserted; $\mathcal{G} = \{G_t | t \in \mathcal{T}, f(t) \geq \theta\}$: a collection of HNSW graphs for frequent tags; P: the node set; M_0 and θ: parameters. len: global value dependent on the structure, initially to be 0.
Output: Insert p to the structure $(\mathcal{T}, \mathsf{f}(\cdot), \mathcal{G}, P, M_0, \theta)$.

1 Add p to P;
2 **foreach** $t \in \mathsf{t}(p)$ **do**
3 **if** $t \notin \mathcal{T}$ **then** Add t to $\mathcal{T}$; $\mathsf{f}(t) \leftarrow 1$; **else** $\mathsf{f}(t) \leftarrow \mathsf{f}(t) + 1$;
4 **if** $\mathsf{f}(t) < \theta$ **then continue**;
5 **if** $\mathsf{f}(t) = \theta$ **then**
6 $\mathsf{l}(t) \leftarrow \lceil \log_2(\theta) \rceil$; $len+ = \mathsf{l}(t)$; $\mathsf{m}(t) \leftarrow \lceil \frac{M_0 \cdot \mathsf{l}(t)}{len} \rceil$;
7 Build HNSW graph G_t on $P[t]$ with $M = \mathsf{m}(t)$;
8 **else**
9 **if** $\mathsf{f}(t)$ *is a power of* 2 **then**
10 $\mathsf{l}(t) += 1$; $len += 1$; $\mathsf{m}(t) \leftarrow \lceil \frac{M_0 \cdot \mathsf{l}(t)}{len} \rceil$;
11 Build HNSW graph G_t on $P[t]$ with $M = \mathsf{m}(t)$;
12 $rebuild \leftarrow$ **false**;
13 **while** $2M_0 \cdot \lceil \log_2 \theta \rceil \leq len$ **do**
14 $rebuild \leftarrow$ **true**; $\theta \leftarrow \theta \times 2$; $len \leftarrow 0$;
15 **foreach** $t' \in \mathcal{T}$ *with* $f(t') > \theta$ **do**
16 $\mathsf{l}(t') \leftarrow \lceil \log_2(f(t')) \rceil$; $len+ = \mathsf{l}(t')$;
17 **if** *rebuild or* len *doubles since the last rebuild* **then**
18 Empty $\mathcal{G}$;
19 **foreach** $t' \in \mathcal{T}$ *with* $f(t') > \theta$ **do**
20 $\mathsf{m}(t') \leftarrow \lceil \frac{M_0 \mathsf{l}(t')}{len} \rceil$; Build the HNSW graph $G_{t'}$ on $P[t']$ with $M = \mathsf{m}(t')$;

Algorithm 6: AllocateDegreeForBlock

Input: B: a set of vectors; $\mathsf{f}(\cdot)$: the frequency function of all the tags inserted; θ: parameter threshold ; M: degree of a vector.
Output: $\mathtt{M}_p = \{m_p(t)$: maximum degree of $t \in \mathsf{t}(p)\}$, for $\forall p \in B$.

$len \leftarrow 0$;
foreach $p \in B$ **do**
  foreach $t \in \mathsf{t}(p) \wedge \mathsf{f}(t) > \theta$ **do**
    $l(t) \leftarrow \lfloor \log_2(\mathsf{f}(t)) \rfloor$; $len+ = l(t)$;
foreach $p \in B$ **do**
  foreach $t \in \mathsf{t}(p) \wedge \mathsf{f}(t) > \theta$ **do**
    $m_p(t) \leftarrow \lfloor \frac{M * |B| * l(t)}{len} \rfloor$;
return $\mathtt{M}_p = \{m_p(t) | \forall t \in \mathsf{t}(p)\}$, *for* $\forall p \in B$;

Mathematically, packing strategy allocates degree for tag t of a data vector p in a block B is

$$\mathsf{m}_p(t) = \begin{cases} 0 & \text{if } \mathsf{f}(t) < \theta, \\ \frac{M * k * \log \mathsf{f}(t)}{\sum_{p' \in B} \sum_{t \in \{t' \in \mathsf{t}(p') | \mathsf{f}(t') \geq \theta\}} \log \mathsf{f}(t)} & \text{if } \mathsf{f}(t) \geq \theta. \end{cases}$$

Index structure. $(\mathcal{T}, \mathsf{f}(\cdot), G, P, M, \theta, \theta', \mathsf{pack}, k, \mathsf{gTags}, \mathsf{maxf})$, the structure has an additional threshold θ', where any tag with frequency exceeding θ' will be considered as the dummy masktag in the resource allocation. The frequency of the dummy tag is set as the maximum frequency maxf of all the tags that have been inserted so far while all the frequent tags are kept in the set of gTags.

Index construction. The indexing process, as Algorithm 7 shows, is similar to Algorithm 4 while having the following differences. Line 1 buffers vectors and when there are k vectors in the buffer, Line 9 allocates degree using Algorithm 6. Lines 3−7 maintains the structure while the tags who should

Algorithm 7: PACK

Input: o: a vector with tags $\mathsf{t}(o)$; $\mathcal{T}$: inserted tags; $\mathsf{f}(\cdot)$: the frequency function of all the tags inserted; G: the index graph; P: the set of nodes; M, θ and θ': parameters; pack: A buffer of vectors; k: the number of vectors to be packed; gTags: the high frequency tags; maxf: the maximum frequency in gTags.

Output: Add o to the structure $(\mathcal{T}, \mathsf{f}(\cdot), G, P, M, \theta, \mathsf{pack}, k)$.

1 Add o to pack; **if** $|\mathsf{pack}| < k$ **then return**;
2 $bordertags \leftarrow \{\}$;
3 **foreach** $p \in \mathsf{pack}$ *and* $t \in \mathsf{t}(p)$ **do**
4 Line 2 to Line 4 from Algorithm 4;
5 **if** $f(t) == \theta$ **then** Add t to $bordertags$;
6 **if** $f(t) == \theta'$ **then** Add t to gTags;
7 **if** $\mathsf{f}(t) > \mathsf{maxf}$ **then** $\mathsf{maxf} \leftarrow \mathsf{f}(t)$;
8 Mask, in the calculations below, all tags in gTags with a dummytag t_0 with frequency $\mathsf{f}(t_0) = \mathsf{maxf}$;
9 $[\mathtt{M}_p, \forall p \in \mathsf{pack}] \leftarrow$ ALLOCATEDEGREEFORBLOCK$(\mathsf{pack}, \mathsf{f}(\cdot), \theta)$;
10 **foreach** $p \in \mathsf{pack}$ **do**
11 **foreach** $t \in \mathsf{t}(p) \wedge t \notin bordertags$ **do**
12 Line 17 to Line 21 from Algorithm 4;
13 **foreach** $t \in bordertags$ **do**
14 $relblocks \leftarrow \{\}$;
15 **foreach** *vector* $p \in P[t]$ **do**
16 Add the block that containing p to the set $relblocks$;
17 **foreach** *block* $B \in relblocks$ **do**
18 For each $\forall r \in B$, let $\mathtt{M}'_r \leftarrow \mathtt{M}_r$;
19 $[\mathtt{M}_r, \forall r \in \mathsf{pack}] \leftarrow$ ALLOCATEDEGREEFORBLOCK$(r, \mathsf{f}(\cdot), \theta)$;
20 **foreach** $r \in B$ **do**
21 Line 10 to Line 16 from Algorithm 4;

have new indexes are kept in the set of $bordertags$. In local method, when there is a tag needing to build index, degree re-allocation is needed for all vectors involving this tag, while in packing strategy, degree re-allocation is needed for all blocks containing vectors involving this tag (Lines 14-21). Note that, we treat all tags in $gtags$ as the same tag, the frequency of which is maxf, in the indexing process. The starting position of the storage of each vector's neighbor list is indexed for efficient retrieval.

Search with the index. The search follows Algorithm 3 with changes regarding high frequency tags. High frequency tags are indexed under a mask tag t_0. The search will traverse along neighbors regarding t_0 and apply post-filtering to exclude irrelevant vectors.

VI. EXPERIMENT

In this section, we evaluate our proposed methods (source code available at https://github.com/SpaceIshtar/FilterGraph.) and compare them with several baseline methods using both real-world datasets and semi-synthetic datasets. All experiments were conducted using a single thread on a server with an Intel(R) Xeon(R) Gold 6212U CPU @ 2.40GHz and 64GB of RAM. The SIMD optimization was enabled for all methods.

A. Datasets

The datasets we used vary in their sources and distributions. Table I presents the datasets' information including data dimensions, types, dataset sizes, number of queries, data representation, filter distributions, average number of filters per vector, and the total number of unique filters. The filter distributions include three types: 'Natural', 'Uniform', and 'ZIPF'. The 'Natural' distribution indicates that the dataset is a real-world dataset, with tags generated based on metadata information. The 'Uniform' distribution means that tags are generated for each embedding, with nearly equal cardinality for each tag. The 'ZIPF' distribution also involves generating tags, but the cardinality of tags follows the Zipf distribution.

Table II provides details about the filter distribution for these datasets. For the base dataset, we present the selectivity of the 100th, 75th, 50th, 25th, and 1st percentile filters, sorted in ascending order of cardinality. For the query dataset, we report the average query filter selectivity and the number of unique query filters present in each dataset.

Dataset Generation. We study both real-world and semi-synthetic datasets for performance evaluation. For real-world datasets, we extract tags from the original datasets; for semi-synthetic datasets, we generate tags as described below.

We used the SIFT1M and GIST1M datasets from image domain, introduced by Jegou et al. in [16], and Paper dataset from text domain, for evaluation. The SIFT dataset contains 128-dimensional SIFT descriptors of images, while the GIST dataset comprises 960-dimensional GIST descriptors. Both datasets have been widely used in evaluating ANNS algorithms. Paper dataset is a 2-million dataset consisting embeddings of research papers. To enable tag-filtered ANNS on these datasets and assess algorithm performance across different filter distributions, we assigned each vector a random integer tag. Tags are either uniformly distributed over the range 1-12 or follow a Zipfian distribution over the range 1-50 (assuming the integer represents frequency rank). These distributions align with those used in previous works, such as NHQ [12] and Filtered-DiskANN [9]. Unlike the uniform distribution, Zipfian better resembles real-world data.

YFCC is a large-scale dataset introduced by Thomee et al. in [17], comprising millions of media objects (photos and videos), each represented by a 192-dimensional UINT8 vector derived from the CLIP model [18]. The dataset also includes metadata such as the owner's name and camera model. The Big-ann-benchmark [19] provides a 10 million sample of this dataset, where each vector is associated with a set of tags – words extracted from the image's description, camera model, year, and country. The resulting tags have a total vocabulary of size 200,386. We used this sampled data for evaluation.

The YouTube dataset, introduced in [20], consists of audio and video content collected from the YouTube website. Feature vectors are extracted from these audios and videos using the Inception network [21]. Each resulting object includes a 128-dimensional audio vector, a 1024-dimensional video vector, and tags associated with the video (which are categories of the video). We first randomly sampled one million objects from this dataset, then randomly sampled the remaining objects to produce queries, ensuring that the tags of each query are a subset of the tags in the sampled data. The sampled dataset

TABLE I Dataset Information.

Dataset	Dim	Data Type	# Pts.	# Queries	Source Data	Filter Distribution	Filters/Pts.	Unique Filters
SIFT Zipf Attr	128	float	1M	10000	Image	ZIPF	1	50
GIST Zipf Attr	960	float	1M	1000	Image	ZIPF	1	50
SIFT Uniform Attr	128	float	1M	10000	Image	Uniform	1	12
GIST Uniform Attr	960	float	1M	1000	Image	Uniform	1	12
SIFT Zipf Tag	128	float	1M	10000	Image	ZIPF	3.154	50
Paper Uniform Attr	200	float	2M	10000	Text	Uniform	1	12
YFCC	192	uint8	10M	10000	Media	Natural	10.821	200386
Youtube-Audio	128	float	1M	42568	Audio	Natural	3.027	3862
Youtube-Video	1024	float	1M	42568	Video	Natural	3.027	3862
LAION	768	float	960K	4322	Image	Natural	3	30

TABLE II Filter/Tag Selectivity: Sort filters/tags (a multiset) in ascending order of their selectivities, show selectivity percentiles

	Filter/Tag Selectivity Percentile in the Base Dataset					Filter/Tag Selectivity Percentile in Query Set					
Selectivity	100pc.	75pc.	50pc.	25pc.	1pc.	100pc.	75pc.	50pc.	25pc.	1pc.	the # of distinct query filters
Uniform Attr	0.083	0.083	0.083	0.083	0.084	0.083	0.083	0.083	0.083	0.084	12
Zipf Attr	0.004	0.006	0.009	0.016	0.222	4.4e-03	0.013	0.044	0.111	0.222	50
Zipf Tag	0.014	0.019	0.028	0.054	0.700	0.014	0.043	0.140	0.350	0.700	50
YFCC	1e-07	3e-06	5.4e-06	1.4e-05	0.339	5.8e-06	7.1e-04	2.9e-3	0.015	0.191	7910
Youtube	1.5e-05	6.1e-05	1.1e-04	3.1e-04	0.198	1.5e-05	5.3e-04	2.8e-03	0.023	0.198	3287
LAION	1.5e-04	0.015	0.047	0.115	0.612	5.3e-04	0.182	0.612	0.612	0.612	30

contains 3,862 unique tags, with each vector associated with an average of 3.027 tags.

The LAION [22] dataset comprises one million 768-dimensional image embeddings paired with caption descriptions. These image embeddings are generated using CLIP, a multi-modal language-vision model [18]. For each embedding, we generate a keyword list as tags. Specifically, we use the 30 most common words from the dataset's descriptions as the tags. For each embedding, the 3 words with the highest text-to-image CLIP score become its tags.

B. Algorithms and Parameters

Next we briefly introduce the algorithms evaluated and their parameter settings.

Local Method: Our local method has three key index parameters: the degree for each vector (M), the construction candidate list size (efCons), and the threshold θ. For SIFT and GIST, we selected $M = 64$ and efCons $= 128$, which are typical values for these datasets. For YFCC, LAION, and YouTube, we varied M in {64, 96, 128} and efCons in {128, 200, 256} to identify the optimal combination (the one with the highest QPS at 0.9 recall for the search process). As for the threshold θ, we simply fixed it to be 1024 for all the experiments. During the search phase, we varied the search parameter efSearch from 10 to 150 to control the trade-off between QPS and recall.

Global Method: Our global method has two index parameters: the total degree budget (M_0) and the construction candidate list size (efCons). We chose M_0 for each dataset to ensure the index size is comparable to that of the local method. The efCons parameter was set to the same value as used in the local method. During the search process, efSearch was varied from 10 to 150 to balance the trade-off between QPS and recall.

Packing Strategy: Packing strategy has the same parameters as the local method. Besides, it also has a parameter k to control how many vectors in each block as well as a threshold θ' to control the dummy masktag. For all the experiments, we fixed k to be 10 and θ' to be 20% of the size of the dataset. In addition, the degree of each vector (M) and the efCons parameter were configured identically to the local method. We adjusted efSearch between 10 and 150 to manage the trade-off between QPS and recall.

Filtered-DiskANN: We evaluated both algorithms proposed in the paper [9], namely StitchedVamana and FilteredVamana. For both algorithms, we set the vector degree (R) and candidate list size (L) to match those of our local method, ensuring a fair comparison in terms of memory consumption. Following the guidelines from Filtered-DiskANN, we set the pruning threshold (α) to 1.2. During the search phase, we varied L from 10 to 150 to trade-off QPS with recall.

NHQ: NHQ integrates filter information into vectors and then constructs a proximity graph. However, it only supports scenarios where query filters exactly match the data vectors, making it unsuitable for our real-world datasets. To evaluate its performance, we used semi-synthetic datasets. NHQ has two key index parameters: the vector degree (M) and the construction candidate list size (efCons). For these semi-synthetic datasets, we set $M = 64$ and efCons $= 128$, which are consistent with the settings used for our proposed methods. During the search process, we varied the search parameter (efSearch) from 10 to 500.

ACORN: ACORN has two variants, ACORN-γ and ACORN-1. The ACORN methods were governed by three index parameters: M, M_β, and γ. For both methods, we set $M = 64$ for the semi-synthetic datasets, $M = 96$ for YouTube-Video, YouTube-Audio, and LAION, and $M = 128$ for YFCC. As recommended in the paper, we varied M_β between M and $2M$ for ACORN-γ and fixed $M_\beta = M$ for ACORN-1. For ACORN-γ, we set $\gamma = 30.0$ for LAION and YFCC and $\gamma = 12.0$ for the other datasets.

Post-filter IVF: The post-filter IVF method first partitions the data vectors into $nlist$ clusters using k-means. The search process has a parameter $nprobe$, which is the number of clusters to traverse during the search process. To conduct post-filtering, we asked the index to return 1000 candidate neighbors and dropped those that did not satisfy the filters to produce the final top-k neighbors. We varied $nlist \in \{500, 2000, 10000, 20000\}$ and $nprobe \in \{100, 500, 1000, 2000\}$ to tradeoff QPS with recall.

Post-filter HNSW: We conducted experiments on post-filter HNSW by varying the degree M in {32, 64, 128} and the corresponding index construction parameter efCons in {64,

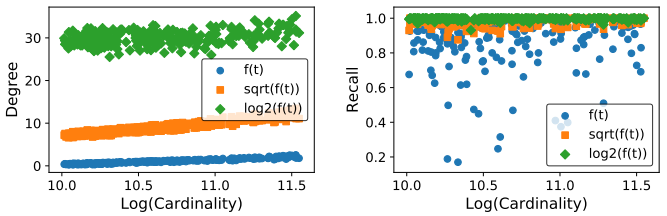


Fig. 6 YouTube-Audio: The difference in degree and recall among different filters when we use different allocation functions on packing strategy. (M = 96, efSearch = 10)

128, 256} to identify the configuration with the best query performance. During the search process, we varied efSearch, which controls the candidate list size, from 10 to 150. We instructed the HNSW index to return the top efSearch candidates, then performed filtering on these candidates to retain the top-k neighbors that meet the search criteria.

Faiss-IVF: The Faiss-IVF method integrates both pre-filtering and post-filtering search strategies. It uses a threshold to determine which search plan to apply. If the smallest query filter cardinality is below the threshold, it first filters out all unsatisfactory vectors and then performs a brute-force search. Otherwise, it searches using a pre-trained IVF index and filters out unsatisfactory results afterward. We created 1024 clusters in IVF and used scalar quantization during the indexing phase and varied the number of clusters to traverse from 1 to 96 during the search phase.

C. Evaluating Proposed Methods

In this section, we evaluate the three proposed methods, the local method as described in Section III, the global method presented in Section IV, and the packing strategy (Section V). Specifically, we first evaluate the impact of allocation functions and then the index size, index time, query performance, and scalability of our methods.

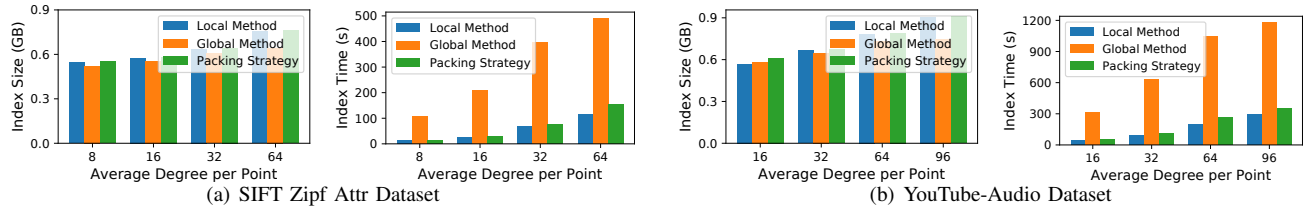
Degree Allocation Function Comparison. In this experiment, we compare our allocation function $\log(f(t))$ with two alternatives: $f(t)$ and $\sqrt{f(t)}$. We first built indexes on the YouTube-Audio dataset using the packing strategy, ensuring an average degree of 96 per vector using these three allocation functions. We then performed searches on these indexes with efSearch set to 10, and recorded the average degrees and recall of various tags. Figure 6 shows the results on tags with cardinality from 1000 to 3000. We can see that using $f(t)$ for allocation results in very few degrees (fewer than 5 edges) for smaller tags, leading to low recall rates. In contrast, $\sqrt{f(t)}$ allows these smaller tags to have more degrees, resulting in recall rates exceeding 0.9 for most tags. The $\log(f(t))$ function, however, provides even more degrees for smaller tags, achieving near 1.0 recall for these tags. This focus on smaller tags is crucial because our query optimization directs queries with both large and small tags to the graph of small tags, improving graph connectivity and search performance. It is worth noting that the base of the logarithm does not significantly impact degree allocation, as the proportion of a tag among a set of tags remains unaffected by the logarithm base.

Index Construction. We then compare the index cost of the three methods. Figure 7 shows the index time and index size of these methods by varying the average degree per data vector. The local method and packing strategy had almost the same index time, while the global method took much longer. For example, on the Youtube-Audio dataset, the local method and packing strategy respectively took 198 and 265 seconds when the average degree per vector was 64, while the global method took 1048 seconds. This is because the global method needs to rebuild a tag's graph to match the degree of the graph with the tag's frequency when a tag's frequency shifts. In terms of index size, the global method had a slightly smaller index size than the other two. For example, the index size of the global method was 0.73 GB while those of the local method and packing strategy were both 0.79 GB. This is mainly because the local method and packing strategy need space for some allocated, but not used, degrees.

Query Performance. We evaluate the query performance of our methods. We varied one of the two parameters, the average degree M and the search candidate size efSearch, and fixed the others. For YouTube-Audio and SIFT Zipf Attr, we varied M from 16 to 96 respectively. In addition, we varied efSearch from 10 to 50 for both datasets. Figure 8 shows the results. As we can see, with the increase of the average degree, the recall improved rapidly while QPS decreased sublinearly. For instance, on YouTube-Audio, when the average degree increased from 16 to 96, the recall grew from 0.75 to 0.95 for the local method, from 0.75 to 0.92 for the global method, and from 0.8 to 0.95 for the packing strategy, while the QPS dropped from 10,000 to 6,132 for the local method, from 8,300 to 6,104 for the global method, and from 9,600 to 6,445 for the packing strategy. This is because the enhanced graph connectivity with a higher average degree improves recall at the cost of exploring more vectors, thus reducing QPS. A similar effect is observed with increased search candidate size, efSearch, on the YouTube-Audio Dataset.

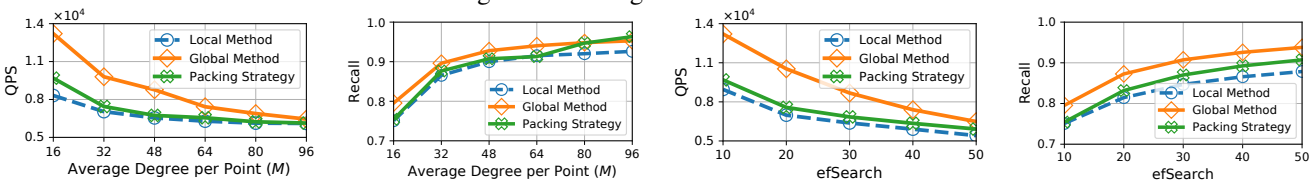
Notice that the increase in recall varies across methods and datasets. On the YouTube-Audio dataset, the global method achieved the highest recall and QPS, indicating its effectiveness in finding accurate results with fewer explorations. This is due to the global method's frequent adjustments to degree allocation and proximity graph rebuilding, which enhances graph quality. However, this advantage comes with higher index costs, as discussed previously. In contrast, the benefit of the global method is less pronounced on the SIFT dataset. Although its superior graph quality improves recall, it requires more data vectors to be explored, resulting in lower QPS compared to other methods. Therefore, we will use the packing strategy in the subsequent experiments due to its low index cost and comparable query performance to the global method.

Another parameter of our algorithm is the threshold which controls if we use graph search or brute-force search. We varied the threshold in set $\{512, 1024, 2048\}$ and tested our performances on Youtube-Audio datasets. Figure 9 shows the results. When the threshold increases, our algorithms use brute-force more frequently and also have higher degree resources to allocate among the tags, recall tends to increase and QPS tends to drop. For example, when the threshold varied from 512 to 2048, for the global method, QPS dropped

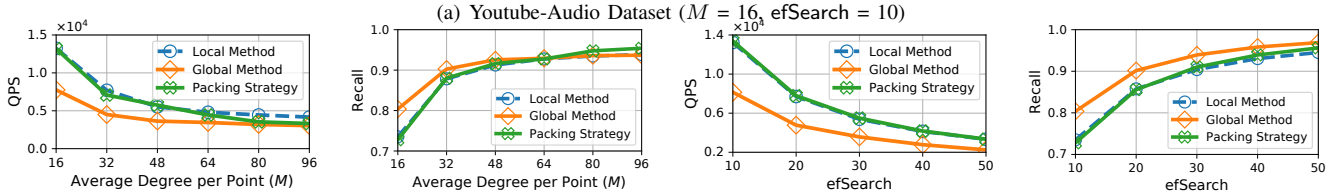


(a) SIFT Zipf Attr Dataset (b) YouTube-Audio Dataset

Fig. 7 Evaluating index construction.



(a) Youtube-Audio Dataset (M = 16, efSearch = 10)



(b) SIFT Zipf Attr Dataset (M = 16, efSearch = 10)

Fig. 8 Evaluating query performance.

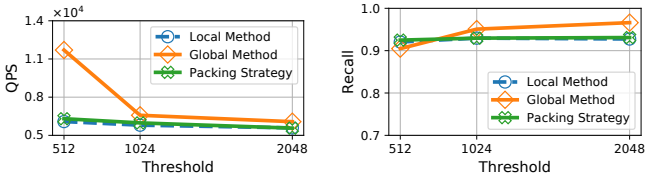


Fig. 9 Evaluate The Influence of Threshold.

from over 11000 to 6000 when $efSearch = 10$ while recall increased from 0.90 to 0.96.

As we discussed in Section V, the proposed packing strategy introduces an additional data structure to maximize the use of the degree allocated. This added data structure incurs negligible overhead while significantly enhancing search quality. For instance, when the average degree is 96 and $sfSearch$ is set to 10, the packing strategy increases the index size by only 1% yet improves recall by 5% under the same QPS.

Scalability. We evaluate the scalability of our methods by varying the number of data vectors in SIFT Zipf Tag from 100,000 to 50 million while maintaining an average degree of 96. We evaluated scalability using million-scale datasets because billion-scale datasets cannot fit in our memory. For the packing strategy, the number of vectors per block was set to 10. As shown in Figure 11, the index size increased linearly with the dataset size due to our strict memory constraints during index construction. As for the query performance, although the recall decreased as the dataset size grew, it remains high enough, with all three methods achieving a recall above 0.8. This is because, with a fixed average degree per vector, the graph's connectivity diminishes as the dataset size grows. Consequently, QPS decreased sublinearly with the increase of dataset size. Specifically, as the dataset size grew from 100,000 to 50 million, QPS dropped from around 3,000 to 400. This decline occurs because a larger dataset size necessitates more iterations for the search process to complete, leading to increasing distance computations. These findings highlight the robustness of our methods on varying dataset scales.

Insertion Order. We evaluate the robustness of our algorithms under the changes in tag distribution on Youtube-Audio dataset. To facilitate significant changes in tag distribution, we reordered the insertion orders of data points so that data points sharing the same tags will be inserted together. We used two insertion orders, one is to insert tags with high frequency first, and the other is to insert tags with low frequency first. The first two figures in Figure 10 show the selectivity changes of the 5 most frequent tags during the insertion process. We tested the query performance of our algorithms on the snapshots when 25%, 50%, 75%, 100% of data has been inserted. Since some query tags may not be inserted when progress has not reached 100%, we only keep queries whose query tags have been inserted in the snapshot. The last two figures in Figure 10 show the QPS of our proposed methods when they achieve 0.90 recall under each insertion order. We can see the QPS drops from around 15000 to around 6000 — the graph becomes larger during the insertion process. This experiment shows that our algorithms keep high recall (> 0.90) at high QPS during the insertion process regardless of the insertion order.

D. Comparison with Existing Approaches.

The objective of this part is to compare our proposed algorithm with the state-of-the-art methods. The experiments are divided into three parts: the first part utilizes datasets with queries containing a single filter while the second part employs datasets with queries that include multiple filters. This division is necessary because some methods like NHQ do not support multi-filter queries. In the last part, we investigate how different tag distributions influence the query performance.

Evaluating Single-Filter Queries. Figure 12 compares the QPS and recall of our algorithm (packing strategy) with the state-of-the-art algorithms on datasets whose queries having single-filters. Our method achieved both the highest QPS and the highest recall in most scenarios. For example, on semi-

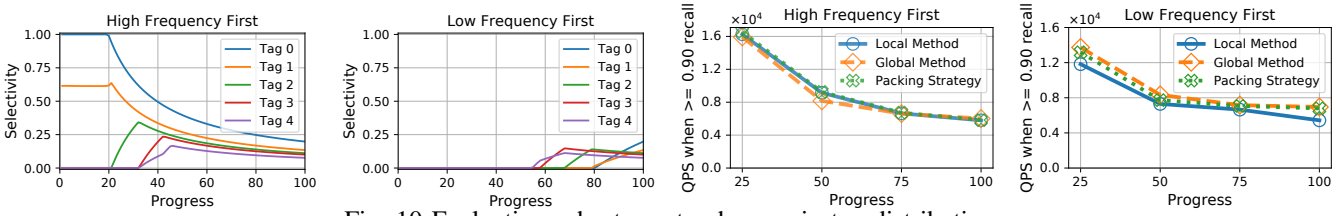


Fig. 10 Evaluating robustness to changes in tag distribution.

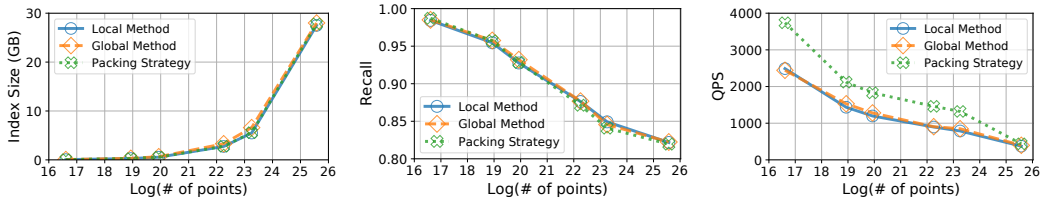
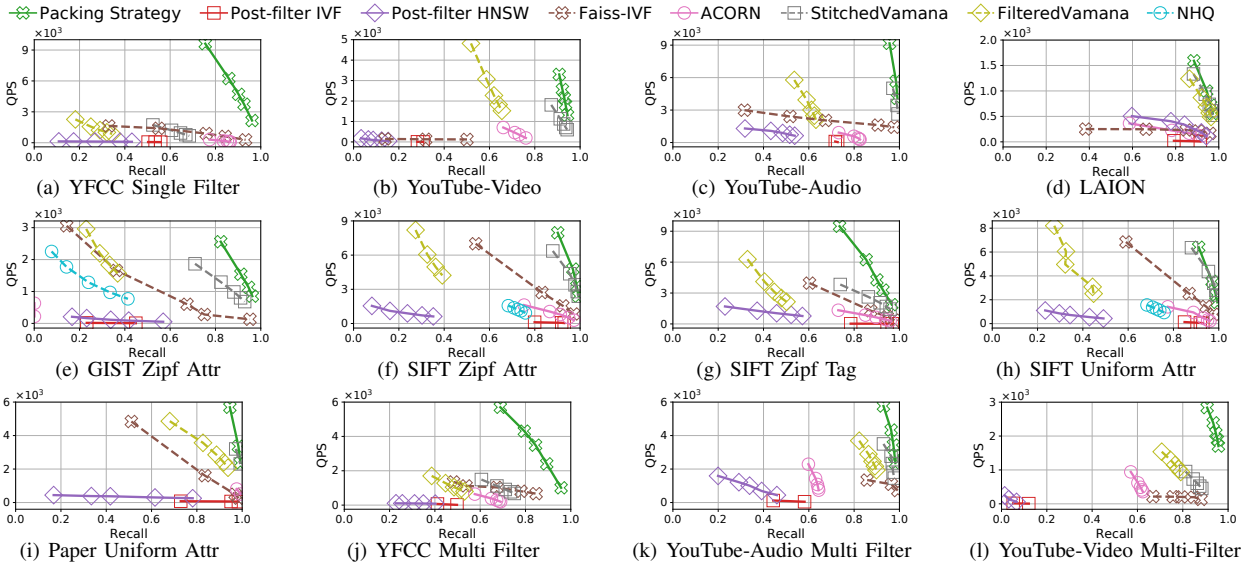


Fig. 11 Evaluating the scalability by varying the number of vectors in SIFT Zipf Tag from 100,000 to 50,000,000.



(a) YFCC Single Filter (b) YouTube-Video (c) YouTube-Audio (d) LAION

(e) GIST Zipf Attr (f) SIFT Zipf Attr (g) SIFT Zipf Tag (h) SIFT Uniform Attr

(i) Paper Uniform Attr (j) YFCC Multi Filter (k) YouTube-Audio Multi Filter (l) YouTube-Video Multi-Filter

Fig. 12 Comparison with the state-of-the-art algorithms.

TABLE III Index Size (GB) and Index Construction Time (s).

Alg./Dataset	SIFTUniform	GISTUniform	SIFTZipf	GISTZipf	YFCC	YouTube-Audio	YouTube-Video	LAION
Packing Strategy	0.75 / 96	3.95 / 245	0.79 / 153	3.95 / 215	8.60 / 7980	0.91 / 350	4.28 / 354	3.16 / 906
StitchedVamana	0.71 / 52	3.83 / 298	0.72 / 259	3.82 / 423	7.46 / 8777	0.87 / 247	4.27 / 885	3.13 / 423
FilteredVamana	0.74 / 84	3.83 / 500	0.72 / 435	3.84 / 435	7.57 / 3310	0.85 / 170	4.25 / 776	3.12 / 441
NHQ	0.72 / 22	3.90 / 126	0.72 / 25	3.90 / 131	-	-	-	-
ACORN	1.37 / 562	4.47 / 3235	1.37 / 560	4.47 / 3240	19.22 / 58552	1.39 / 683	4.73 / 2393	3.62 / 174

synthetic dataset GIST Zipf Attr, the recall of our method, StitchedVamana, FilteredVamana, Post-filter IVF, Post-filter HNSW, Faiss-IVF, ACORN and NHQ were respectively 0.96, 0.92, 0.37, 0.44, 0.57, 0.95, 0.01 and 0.41, while the QPS were respectively 853, 682, 1568, 6.6, 52, 129, 220 and 770. Same results can be observed on real-world datasets. For example, on YouTube-Video, our method achieved 0.95 recall at a QPS of 1307, while the only other algorithm could achieve this recall, StitchedVamana, was at a QPS of 600. The most significant improvement came from YFCC, a real-world dataset with a larger number of low-to-medium frequency tags (as shown in Table II, YFCC has 200,386 unique tags and 75% of them have a selectivity of less than 1%). On this dataset, we achieved approximately 13 times speed up compared to the second best algorithm Faiss-IVF when the recall was at 0.9. This significant improvement was attribute to the following reasons. First, compared to post-filtering methods, our method filters inappropriate candidates during graph search, which reduces the number of distance computations. Second, compared to other incremental algorithms like FilteredVamana and NHQ, our algorithm maintains the connectivity of the subgraph of each filter during index construction, which results in better search performance. Finally, compared to other graph-based methods like StitchedVamana, FilteredVamana, NHQ and ACORN, our high query performance came from our reasonably allocated degrees, which makes the index better utilizes the limited memory resource.

Table III lists the index time and the index size of graph-based methods. The index sizes of our packing strategy were comparable to those of StitchedVamana, FilteredVamana and

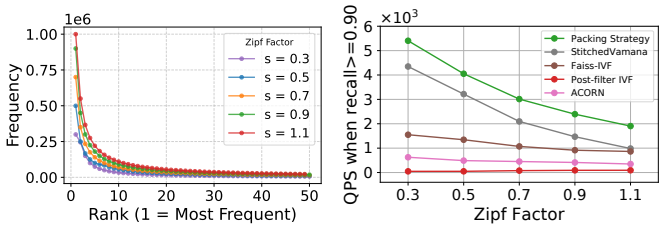


Fig. 13 The Influence of Tag Distribution.

NHQ. For example, on YouTube-Video, our packing strategy took 4.28GB, while StitchedVamana and FilteredVamana occupied 4.27GB and 4.25GB respectively. ACORN needed much more storage as its degree of a vector could be at most $M \times \gamma$. As for the index time, our packing strategy used less time than StitchedVamana and FilteredVamana on SIFT Zipf Attr, GIST, YFCC, and YouTube-Video, but more time on SIFT Uniform Attr and YouTube-Audio, which suggests that our packing strategy was less sensitive to data dimension and more adaptive to complicated filter distributions.

Evaluating Multi-Filter Queries. Figure 12 also compares our algorithm with the state-of-the-art algorithms where the queries could have multiple filters – data vectors reported should satisfy all the query filters. Since FilteredVamana and StitchedVamana do not support multi-filter search, we implement our multi-filter search strategy on their index, where we first find the smallest-selectivity tag in the query tags, and then conduct graph search inside this tag's subgraph; only data points that matches the query tags will be added to the result queue. NHQ is excluded because even our search strategy cannot be used on NHQ to support multi-filter search. Among these algorithms, our algorithm shows superior performance. For example, on YFCC, our algorithm achieved a recall of 0.95 with QPS 1008, while StitchedVamana, FilteredVamana, Post-filter IVF, Post-filter HNSW, Faiss-IVF and ACORN could only achieve 0.75, 0.56, 0.49, 0.40, 0.84 and 0.69 recall respectively with no more than 670 QPS.

The Influence of Tag Distribution. In this experiment, we varied the Zipf factor from 0.3 to 1.1 and compared the performance of different algorithms on SIFT dataset by the QPS they achieve when recall reaches 0.90. The figure on the left of Figure 13 shows how different zipf factors influence the distribution of the tags by recording the cardinality of each tag. When zipf factor increases, tags with larger frequency will appear. At the same time, the average number of tags each data point has will also increase. The right hand side figure in Figure 13 shows the QPS of each algorithm when they achieves 0.90 recall. Some algorithms like FilteredVamana and Post-filter HNSW are not included because they failed to achieve this recall in the search scope. We can see from this figure that when zipf factor increased, QPS declined for all Graph-based methods, including Packing Strategy, StitchedVamana and ACORN. This is because when zipf factor increases, the subgraph on each tag also increases, resulting in a larger search scope for graph-based methods. Post-filtering method like post-filter IVF, on the contrary, has slightly larger QPS for large zipf factor. This is because tags with larger frequency makes it easier for post-filtering methods to find candidates that match the query filters. This experiment also demonstrates that our proposed method can have superior performance than other methods on various tag distributions.

VII. Related Work

Attribute-Filtered Approximate Nearest Neighbor Search. The analytic database system AnalyticDB-V [23] proposes a cost model to select the optimized query plan for *hybrid queries* (e.g., search over a table containing attribute columns and a vector column). Focusing on the same problem, Milvus extends the query plans developed in AnalyticDB-V by introducing a partition-based query plan [24]. Mohoney et al. combined vector similarity queries with predicates over relational attributes while proposing workload-awareness optimization [25]. NHQ [12] leverages self-defined distance metrics to fuse the embedding vector with the attributes together. The follower HQANN [26] proposed another fusion distance metric, which emphasizes more on attributes rather than vector distances. These two methods are inefficient when the number of attributes is large. SeRF [27] specializes in the *range-filtered* ANNS for total order attributes, compressing n^2 ANNS graphs into one index. Joshua et al. [28] studied a similar problem as SeRF using another term: *Window Filtered*. They adopted a tree-based structure to store multiple ANNS indices for arbitrary window filters.

Tag-Filtered Approximate Nearest Neighbor Search. To support TFANNS, industry vector databases like Milvus [24], Faiss [3], and PGVector [29] adopted post-filtering strategy on the HNSW index, suffering from visiting too many unsatisfied vectors. Filtered-DiskANN [9] considers attribute information in its heuristic prune procedure. However, their proposed strategy will be degraded to distance-only pruned when the distribution of attributes is skewed, which avoids underrepresented attributes. ACORN [13] introduces a metadata-agnostic graph index to support arbitrary filtering predicates, including tag-based filtering. Instead of relying on a distance-based heuristic for pruning, ACORN uses an attribute-based online pruning strategy, which can result in reduced search speed compared to other proximity-graph-based methods.

VIII. Conclusion

This paper investigates the problem of TFANNS and identifies a common problem of low recall on queries with low-medium frequency tags on existing approaches, which cannot be easily remedied by pre-filtering. To address this problem, we provide three tag-frequency-aware approaches. Local algorithm is node-centric, global algorithm is tag-centric and packing strategy combines node-centric and tag-centric to strike a better balance between index size and query performance. Our extensive experiments have justified the superiority of our proposed approach compared to existing TFANNS approaches on real-world dataset.

Acknowledgments. Dong Deng* was supported by the National Science Foundation grant #2212629 and a gift from Adobe Research.

REFERENCES

[1] P. S. H. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[2] X. Li, J. Yang, and J. Ma, "Recent developments of content-based image retrieval (CBIR)," *Neurocomputing*, vol. 452, pp. 675–689, 2021. [Online]. Available: https://doi.org/10.1016/j.neucom.2020.07.139

[3] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," *CoRR*, vol. abs/2401.08281, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.08281

[4] A. Kusupati, G. Bhatt, A. Rege, M. Wallingford, A. Sinha, V. Ramanujan, W. Howard-Snyder, K. Chen, S. M. Kakade, P. Jain, and A. Farhadi, "Matryoshka representation learning," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/c32319f4868da7613d78af9993100e42-Abstract-Conference.html

[5] A. Rubinstein, "Hardness of approximate nearest neighbor search," *CoRR*, vol. abs/1803.00904, 2018. [Online]. Available: http://arxiv.org/abs/1803.00904

[6] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 8, pp. 1475–1488, 2020. [Online]. Available: https://doi.org/10.1109/TKDE.2019.2909204

[7] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 1964–1978, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1964-wang.pdf

[8] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *TPAMI*, vol. 42, no. 4, pp. 824–836, 2018.

[9] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan *et al.*, "Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters," in *WWW*, 2023, pp. 3406–3416.

[10] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," in *NeurIPS 2019*, November 2019. [Online]. Available: https://www.microsoft.com/en-us/research/publication/diskann-fast-accurate-billion-point-nearest-neighbor-search-on-a-single-node/

[11] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L. Li, "YFCC100M: the new data in multimedia research," *Commun. ACM*, vol. 59, no. 2, pp. 64–73, 2016. [Online]. Available: https://doi.org/10.1145/2812802

[12] M. Wang, L. Lv, X. Xu, Y. Wang, Q. Yue, and J. Ni, "An efficient and robust framework for approximate nearest neighbor search with attribute constraint," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/32e41d6b0a51a63a9a90697da19d235d-Abstract-Conference.html

[13] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia, "ACORN: performant and predicate-agnostic search over vector embeddings and structured data," *Proc. ACM Manag. Data*, vol. 2, no. 3, p. 120, 2024. [Online]. Available: https://doi.org/10.1145/3654923

[14] J. Engels, B. Landrum, S. Yu, L. Dhulipala, and J. Shun, "Approximate nearest neighbor search with window filters," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=8t8zBaGFar

[15] J. Sack, "Optimizing your query plans with the sql-server 2014 cardinality estimator," 2014.

[16] H. Jégou, M. Douze, and C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, Jan. 2011. [Online]. Available: https://inria.hal.science/inria-00514462

[17] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li, "Yfcc100m: the new data in multimedia research," *Communications of the ACM*, vol. 59, no. 2, p. 64–73, Jan. 2016. [Online]. Available: http://dx.doi.org/10.1145/2812802

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

[19] "Big-ann benchmarks: Neurips 2023 competition," https://big-ann-benchmarks.com/neurips23.html, accessed: 2024-09-01.

[20] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "Youtube-8m: A large-scale video classification benchmark," *arXiv preprint arXiv:1609.08675*, 2016.

[21] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, ser. JMLR Workshop and Conference Proceedings, F. R. Bach and D. M. Blei, Eds., vol. 37. JMLR.org, 2015, pp. 448–456. [Online]. Available: http://proceedings.mlr.press/v37/ioffe15.html

[22] C. Schuhmann, R. Beaumont, R. Vencu, C. Gordon, R. Wightman, M. Cherti, T. Coombes, A. Katta, C. Mullis, M. Wortsman, P. Schramowski, S. Kundurthy, K. Crowson, L. Schmidt, R. Kaczmarczyk, and J. Jitsev, "LAION-5B: an open large-scale dataset for training next generation image-text models," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/a1859debfb3b59d094f3504d5ebb6c25-Abstract-Datasets_and_Benchmarks.html

[23] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, "Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data," *PVLDB*, vol. 13, no. 12, pp. 3152–3165, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3152-wei.pdf

[24] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, "Milvus: A purpose-built vector data management system," in *SIGMOD*. ACM, 2021, pp. 2614–2627. [Online]. Available: https://doi.org/10.1145/3448016.3457550

[25] J. Mohoney, A. Pacaci, S. R. Chowdhury, A. Mousavi, I. F. Ilyas, U. F. Minhas, J. Pound, and T. Rekatsinas, "High-throughput vector similarity search in knowledge graphs," *SIGMOD*, vol. 1, no. 2, pp. 1–25, 2023.

[26] W. Wu, J. He, Y. Qiao, G. Fu, L. Liu, and J. Yu, "HQANN: efficient and robust similarity search for hybrid queries with structured and unstructured constraints," in *CIKM*, 2022, pp. 4580–4584. [Online]. Available: https://doi.org/10.1145/3511808.3557610

[27] C. Zuo, M. Qiao, W. Zhou, F. Li, and D. Deng, "Serf: Segment graph for range-filtering approximate nearest neighbor search," *Proc. ACM Manag. Data*, vol. 2, no. 1, pp. 69:1–69:26, 2024. [Online]. Available: https://doi.org/10.1145/3639324

[28] J. Engels, B. Landrum, S. Yu, L. Dhulipala, and J. Shun, "Approximate nearest neighbor search with window filters," *CoRR*, vol. abs/2402.00943, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.00943

[29] "Pgvector," https://github.com/pgvector/pgvector.