GAS: A Lightweight Framework for Filtered Search over Wide-table Vectors

Ziyuan He
Beihang University
Beijing, China
hzy_he@buaa.edu.cn

Yuxiang Wang
Beihang University
Beijing, China
yuxiangwang@buaa.edu.cn

Yu Sun
Nankai University
Tianjin, China
sunyu@nankai.edu.cn

Zijie Ma
Beihang University
Beijing, China
zijie_ma@buaa.edu.cn

Hui Li
Xidian University
Xi'an, China
hli@xidian.edu.cn

Qian Tao
Beihang University
Beijing, China
qiantao@buaa.edu.cn

Yu Li
Beihang University
Beijing, China
carp84@gmail.com

Yongxin Tong
Beihang University
Beijing, China
yxtong@buaa.edu.cn

ABSTRACT

Wide-table vectors, where each embedding is linked with numerous structured attributes, are prevalent in applications such as autonomous driving and multimodal data processing for large-model training. Efficiently retrieving semantically similar vectors under attribute filters is crucial for these tasks, a problem addressed by Filtered Approximate Nearest Neighbor Search (FANNS). Recent approaches follow two paradigms: (1) building per-attribute dedicated indexes that integrate attribute information, which incurs prohibitive build time and storage in wide-table settings; or (2) building an attribute-agnostic general index and applying predicates at query time, which often degrades search efficiency. Consequently, neither paradigm adequately supports wide-table scenarios. We aim to achieve good query performance with low upfront cost by incorporating information from many attributes into a single graph index, avoiding prohibitive overhead. Our key observation is that graph-traversal information from past queries can be reused to optimize future queries with the same filter attribute. Based on this insight, we devise Graph with Adaptive Shortcuts (GAS), a framework that leverages historical query logs to build lightweight auxiliary structures, enhancing search efficiency over a single base graph with minimal overhead. Extensive experiments on real-world datasets show that GAS consistently outperforms existing general indexes in wide-table scenarios, achieving up to 42.1× speedup on datasets with thousands of structured attributes.

PVLDB Reference Format:
Ziyuan He, Yuxiang Wang, Yu Sun, Zijie Ma, Hui Li, Qian Tao, Yu Li, and Yongxin Tong. GAS: A Lightweight Framework for Filtered Search over Wide-table Vectors . PVLDB, 19(11): 3133 - 3145, 2026.
doi:10.14778/3836663.3836678

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

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836678

1 INTRODUCTION

Vector embeddings capturing semantic meaning of unstructured data, are increasingly important in modern AI applications [32, 43]. In domains such as autonomous driving and multimodal data processing for large-model training, an embedding may be linked to numerous sensor signals or extracted features, forming a *wide-table* with a large number of attributes [3, 19, 40]. Efficiently retrieving semantically similar vectors subject to attribute constraints is crucial in these tasks [18, 40], motivating research on the problem known as Filtered Approximate Nearest Neighbor Search (FANNS) from both academia [20, 23] and industry (including Microsoft [14], Apple [34], and NVIDIA [47]).

Many approaches have been designed for the FANNS problem, where most works are based on the graph due to its superior search performance and widespread adoption in real-world systems [7, 44]. Depending on whether attribute information is integrated into the graph index, existing approaches can be broadly categorized into two paradigms: attribute-integrated *dedicated indexes* [6, 14, 22, 25, 45, 49, 53] and attribute-agnostic *general indexes* [33, 39].

Dedicated graph indexes integrate vectors with a specific attribute to construct a tailored graph structure optimized for filtering on that attribute. By explicitly incorporating attribute data during index construction, these methods can prune irrelevant vectors during search [22, 25, 49]. However, since the graph structure is tightly coupled to a specific attribute, these approaches must build a separate index per attribute, resulting in prohibitive construction time and storage overhead in wide-table scenarios.

In contrast, general graph indexes are built purely on vector similarity, decoupling the graph structure from attribute data [33, 39]. A single index can therefore support arbitrary attribute predicates applied at query time, avoiding per-attribute reconstruction. However, because attributes are not encoded in the structure, search may traverse many candidates that are later discarded by filtering, making general indexes substantially less efficient than dedicated methods [20], including in wide-table settings (see Sec. 5.2).

Consequently, neither paradigm adequately supports wide-table scenarios: dedicated indexes incur prohibitive space overhead, while general indexes often suffer substantial performance degradation. Given this, we aim to achieve good query performance with the limited space cost, and a natural intuition is to incorporate information from many attributes into a single graph index, without incurring prohibitive construction overhead. In this sense, we devise Graph

with Adaptive Shortcuts (GAS), a framework for efficient FANNS over wide-table vectors that augments a single base graph with lightweight shortcut overlays. We reuse graph-traversal information from past queries to construct these shortcuts and attach them to the same graph. This design allows search performance to improve over time with minimal additional overhead and no large upfront build cost, even with numerous attributes. We introduce two types of shortcuts: *attribute-aware shortcuts* that connect nodes sharing the same predicate to increase valid visits, and *structure-aware shortcuts* that bypass detours to shorten traversal paths, yielding robust speedups regardless of which attribute is filtered. To exploit these shortcuts effectively, we further design traversal and pruning strategies that prioritize shortcut expansions toward nearby valid nodes and avoid exploring nodes unlikely to satisfy the filter.

Example 1. *Fig. 1 illustrates a real-world wide-table example of targeted driving-scene retrieval in autonomous driving, where data are sampled from the COMMA driving dataset [37] (see Sec. 5.1 for details). Each driving segment is represented with a semantic embedding of its visual content, together with numerous structured signals such as speed, gear, and brake/accelerator pedal measurements from the CAN bus and other onboard sensors. As illustrated in Fig. 1a, retrieving scenes of "braking before a crosswalk" requires matching the crosswalk context in vector space while filtering on the brake signal to ensure the vehicle is braking.*

Dedicated indexes such as UNIFY [22] can answer this query quickly by using a pre-built index tailored to the brake *signal to prune vectors that do not satisfy the brake predicate. However, they incur substantial upfront build time and space overhead before serving such queries, because they must build and store a separate index for* speed, gear, *and many other attributes, as shown in Fig. 1b. General indexes such as NaviX [39] build a single index over all vectors in the wide table to serve this query. However, because the index does not encode* brake *predicates, the search may traverse many nodes that match the crosswalk scene but do not satisfy braking, as shown in Fig. 1c, wasting computation and degrading performance.*

As for our solution, GAS also builds on a single graph index, but accelerates this query by leveraging shortcuts learned from past queries, as shown in Fig. 1d. These shortcuts guide traversal toward nodes that satisfy the brake predicate and reduce unnecessary detours, jointly improving search performance.

Our major contributions are as follows:

- We study FANNS over wide-table vectors and propose Graph with Adaptive Shortcuts (GAS), tailored to this setting. GAS is a lightweight framework that leverages query logs to augment a single graph index, improving search efficiency with minimal construction and storage overhead. (Sec. 2)
- We identify the key limitations of single-graph FANNS and leverage graph-traversal information from historical queries to construct two types of shortcuts that address these limitations. (Sec. 3)
- We design specialized search strategies for GAS that exploit the constructed shortcuts to efficiently support both single-filter and multi-filter queries. We further provide theoretical analysis showing that GAS reduces overall search complexity. (Sec. 4)

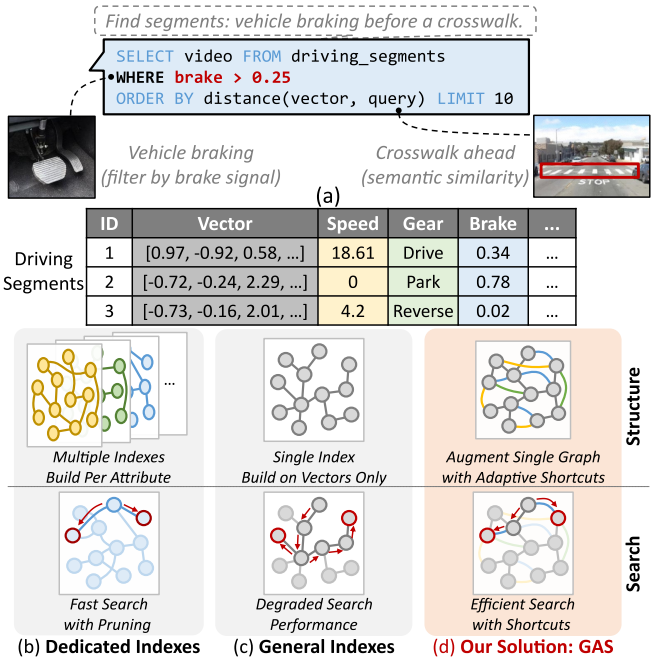


Figure 1: Example wide-table scenario from the real-world driving dataset COMMA [37]. (a) shows a FANNS query for targeted scene retrieval. (b)-(d) illustrate the index structures and corresponding schematic search processes of different methods for this query.

- We evaluate GAS on seven real-world vector datasets with up to 2,381 structured attributes. The results show that GAS consistently outperforms general indexes, achieving up to 42.1× speedup in the wide-table setting. It also achieves performance comparable to dedicated indexes with far lower space overhead, reaching 92.4% of dedicated QPS using only 9.3% of the space in the multi-attribute setting. (Sec. 5)

2 BACKGROUND

In this section, we first review the graph-based index preliminaries needed for our method and define the problem of FANNS over wide-table vectors, then provide an overview of GAS.

2.1 Preliminaries and Problem Statement

As introduced in Sec. 1, FANNS retrieves vectors that are close to a query in embedding space while satisfying an attribute constraint, and graph-based vector indexes are the preferred approach for FANNS due to their strong performance [20].

Specifically, a graph can be defined as $G = (V, E)$, where V represents the set of nodes, each corresponding to a vector in the dataset, and E denotes the set of edges connecting each node to its neighbors under certain neighborhood conditions. The neighbors of a node u are denoted by $G[u]$. Then the search algorithm is typically designed on top of the graph index [2, 4, 22, 33, 49, 53].

We study the FANNS problem over graph-based indexes in the wide-table setting, where each vector is associated with a large number of attributes. Formally, we consider a dataset $D = \{v_1, \ldots, v_N\}$

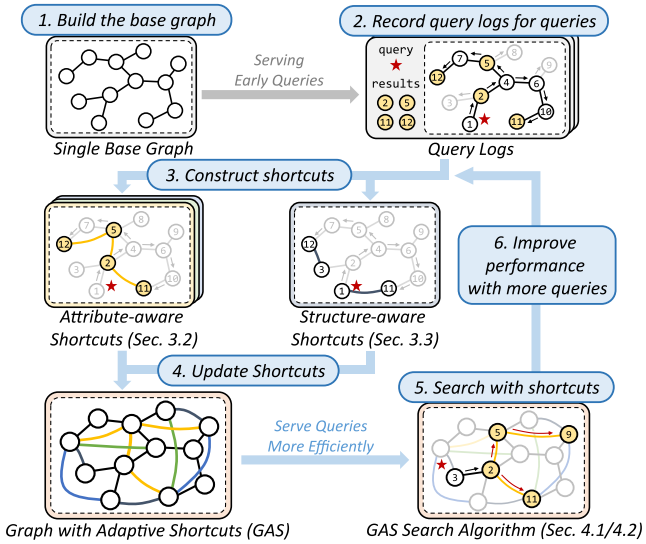


Figure 2: Overview of our GAS framework.

of *wide-table vectors*: each v_i is a high-dimensional vector associated with w structured attributes $\{a_1, \ldots, a_w\}$, where w can reach hundreds or even thousands in practice. For example, in Fig. 1, each driving segment is represented by an embedding vector produced by a deep learning model and is attached with numerous structured attributes collected from various onboard sensors. A query specifies a query vector and a filter predicate on a designated attribute among the w attributes; only vectors satisfying the predicate are eligible. Given the query vector and the filter, FANNS over wide-table vectors returns the nearest vectors within this eligible subset. We now define the problem and our design goals.

Definition 1 (Filtered Approximate Nearest Neighbor Search over Wide-table Vectors (FANNSWV)). *Let D denote a dataset of wide-table vectors, where each vector is associated with w structured attributes (with w potentially very large). A query consists of a query vector q, an integer k, and a filter predicate f on a designated attribute, aiming to retrieve k vectors that are both valid under the filter and closest to the query vector.*

Formally, let $D_f = \{v \in D \mid f(v) = \text{True}\}$ denote the subset of vectors in D that satisfy the filter f. The FANNSWV returns a set $R \subseteq D_f$ with $|R| = k$ that minimizes the total distance to the query $\sum_{v \in R} dis(v, q)$, where $dis(\cdot)$ denotes the Euclidean distance.

Design goals. Given the potentially large number of attributes w, building attribute-specific (dedicated) indexes becomes impractical, since the construction time and space overhead can scale with w, leading to prohibitive upfront cost in the FANNSWV problem. Therefore, we target a solution that incorporates attribute information into *a single graph index*, with the following two goals:

- **Low construction overhead under large w:** keep build time and space overhead low even when w is large.
- **High query efficiency at a target accuracy:** maximize QPS (queries per second) subject to a target recall $\frac{|R \cap R^*|}{k}$, where R^* is the ground-truth result set.

Table 1: Summary of major notations.

Notations	Description
D	dataset of wide-table vectors
N	total vector number in D
w	structured attribute number for wide-table vector
q	query vector
k	number of results to return
k'	parameter controlling the search scope
f	filter predicate
G	graph index
m	max degree of graph index
SC_a, SC_s	the set of attribute/structure-aware shortcuts
m_a, m_s	per-node limit for attribute/structure-aware shortcuts
L	query log
$L.results$	the valid results in query log
$L.pre(v)$	the predecessor of node v in query log
$L.dis_q(v)$	data vector v's distance to query q

In this paper, we focus on two widely used attribute types: numerical [22, 49, 53] and categorical [6, 14, 25]. A numerical filter predicates on a numerical attribute, where v_i satisfies the filter $[\min, \max]$ if its attribute $v_i.a_x$ falls within the range, *i.e.*, $\min \le v_i.a_x \le \max$. A categorical filter predicates on a categorical attribute, where v_i satisfies the filter if its attribute $v_i.a_x$ belongs to the valid category set S_{valid}, *i.e.*, $v_i.a_x \in S_{\text{valid}}$. We further elaborate on these two types of filters in Example 2.

Example 2. *Following Example 1, consider driving-scene retrieval over the segment dataset in Fig. 1. Suppose the three segments in the figure are the nearest candidates under vector similarity. Applying a numerical filter for low speed (*e.g., *Speed $<$ 5) retains $\{v_2, v_3\}$, while applying a categorical filter for non-parking gears (*e.g., *Gear $\in$ {Drive, Reverse}) retains $\{v_1, v_3\}$.*

Throughout the paper, we refer to vectors that satisfy the query filter as *valid*, and those that do not as *invalid*. In addition, we define a filter's *selectivity* as the proportion of vectors in the dataset that satisfy the filter, following the prior work [43, 46]. Table 1 summarizes the major notations used in this paper.

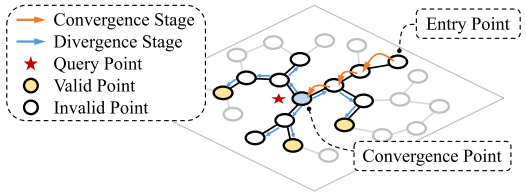
2.2 Overview of GAS

We now present a workflow overview of our GAS framework (in Fig. 2) before diving into its details. Given a wide-table dataset and a query workload, GAS improves query performance over time as more queries are processed.

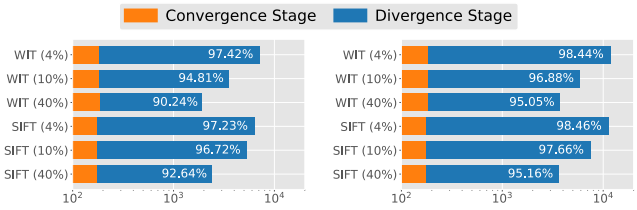
Step 1: Build the base graph. GAS first builds a single base graph to serve early queries, enabling fast startup in the wide-table scenario. The base graph is augmented with lightweight auxiliary shortcut structures, introduced in the subsequent steps.

Step 2: Record query logs for queries. While serving queries, GAS records traversal traces in query logs. These traces expose search inefficiencies and provide opportunities to address them. We detail this step in Sec. 3.1.

Step 3: Construct shortcuts from query logs. GAS processes the query logs in background to incrementally construct two types of shortcuts that address the bottlenecks identified in Sec. 3.1: *Attribute-aware shortcuts* increase the valid-visit ratio by linking



(a) Two stages of FANNS on a single graph index.



(b) Fixed 90% Recall **(c) Fixed 95% Recall**

Figure 3: Bottleneck analysis of FANNS on a single graph index. (a) illustrates the convergence and divergence stages in the search process. (b)-(c) report the total distance computations in each stage on WIT [41] and SIFT [17] datasets under different selectivity levels, with $k = 10$.

nodes that satisfy the same filter predicate, enabling search to bypass nodes that do not meet the filter condition (Sec. 3.2). *Structure-aware shortcuts* mitigate traversal detours by repairing inefficient traversal segments in query logs, allowing future searches to bypass detours and reduce redundant explorations (Sec. 3.3).

Step 4: Update shortcut structures. All constructed shortcuts are attached to the base graph and stored in auxiliary structures. To control space overhead, we selectively retain shortcuts with the highest potential utility. We detail this step in Sec. 3.4.

Step 5: Search with shortcuts. Building on the shortcut overlays from the previous step, the GAS search algorithm exploits these shortcuts to improve performance by prioritizing promising candidates and pruning unlikely ones (Sec. 4.1). The search procedure can be extended to support multi-filter queries, which impose filters over more than one attribute (Sec. 4.2).

Step 6: Improve performance with more queries. As more queries are processed, new traversal logs are fed back into Step 3 to refine the shortcut overlays, enabling GAS to improve search performance over time without rebuilding the base graph.

3 CONSTRUCTION OF GAS

This section describes the construction of GAS. We first analyze the bottlenecks of FANNS on a single graph, which motivates our shortcut design, then present attribute-aware and structure-aware shortcuts, and finally describe how we maintain them over time.

3.1 From Bottlenecks to Shortcut Design

To guide shortcut design, we analyze bottlenecks of FANNS on a single graph by quantifying the cost breakdown of vanilla HNSW [1] on WIT [41] and SIFT [17] datasets, measuring the number of distance computations required to reach a fixed recall (see Sec. 5.1).

As illustrated in Fig. 3a, the search can be divided into two stages: (1) **Convergence**, which converges from an entry point toward the query by repeatedly moving to closer nodes until reaching the closest visited node, called the *convergence point*; and (2) **Divergence**, which diverges from the convergence point to explore nearby regions and collect valid results.

As shown in Fig. 3b-3c, divergence dominates the search cost, accounting for over 90% of all distance computations, with its dominance becoming more pronounced as selectivity decreases. Building on this breakdown, we further analyze the divergence stage and identify two key limitations: (1) the search visits a *low fraction of valid nodes*, which forces repeated scope expansion and incurs substantial redundant distance computations; (2) traversal frequently falls into *detours* near the query, failing to reach valid nearest neighbors through local expansions and only discovering them after the search scope has grown and the traversal reaches a distant node.

Our Insights. During a FANNS query on a single graph, the search procedure yields valuable graph-traversal traces that not only reveal these limitations but also provide opportunities to address them:

- *Query Results*: Since those results are the nearest neighbors that satisfy the filter, they tend to be close in embedding space and have similar attribute values. We can leverage these results to add shortcut edges that connect such nodes, guiding traversal toward nodes more likely to satisfy the filter and mitigating the low valid-visit ratio.
- *Path to Query Result*: The traversal path to each returned vector can expose detours by examining the already computed distances of intermediate nodes to the query. These traces can also be used to add direct shortcut connections that bypass these detours.

We store each query's traversal trace in a query log L, including the result set $L.results$, the predecessor $L.pre(u)$ of each visited node u for reconstructing full traversal paths, and its distance $L.dis_q(u)$ to the query; these records are later used for shortcut construction.

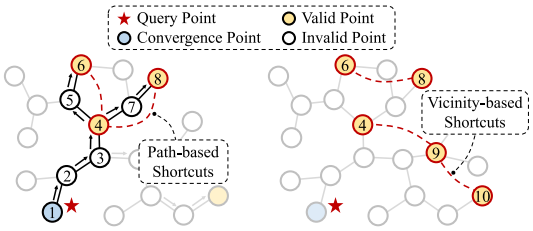
Algorithm 1: ***Construct Attribute-aware Shortcuts***

Input: graph index G, query log L
Output: set of attribute-aware shortcuts SC_a

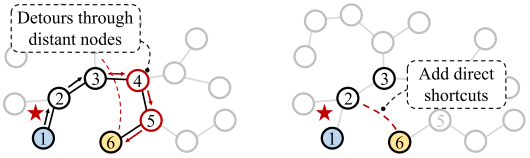
```
SC_a ← {};
foreach u ∈ L.results do
    prev ← L.pre(u);
    while prev is not the entry point of G do
        if prev ∈ L.results then
            Add shortcut ⟨u, prev⟩ to SC_a ;      // Path-based Shortcuts
            break;
        prev ← L.pre(prev);
foreach u ∈ L.results do
    NN_u ← the μ nearest neighbors of u within L.results;
    foreach v ∈ NN_u do
        Add shortcut ⟨u, v⟩ to SC_a ;            // Vicinity-based Shortcuts
return SC_a;
```

3.2 Attribute-aware Shortcuts

Rationale. A FANNS query aims to find nearest neighbors that satisfy the query filter. Consequently, the results recorded in the query log tend to be similar in both vector space and attribute



(a) Constructing attribute-aware shortcuts.



(b) Constructing structure-aware shortcuts.

Figure 4: Examples of shortcut construction.

space. We leverage this observation by introducing *attribute-aware shortcuts* that create direct connections among these related nodes. For subsequent queries with similar filters, these shortcuts enable efficient transitions between valid nodes, bypassing standard graph traversal over numerous invalid nodes.

We now describe how attribute-aware shortcuts are constructed from the query log. For each query, GAS leverages the recorded search paths and the result set to identify pairs of valid nodes that are closely related in the traversal trace, and adds shortcuts between them. These shortcuts provide more direct routes among valid nodes, reducing wasted expansions on invalid nodes.

Path-based Attribute-aware Shortcuts. We construct path-based shortcuts from the traversal paths recorded in the log. For each valid result node u, we reconstruct its traversal path using the logged predecessor information. We then scan backward from u and add a shortcut between u and the nearest preceding valid node on this path (Lines 5-7 in Algorithm 1). This shortcut allows future searches to skip invalid intermediate nodes on the same route. The scan stops after adding a shortcut or reaching the path start.

Vicinity-based Attribute-aware Shortcuts. Path-based shortcuts can be sparse and may provide limited benefit when different queries share little overlap in their traversal paths. To complement them, we also add shortcuts within the final valid result set recorded in the log. Intuitively, these returned nodes often form a valid neighborhood around the query, and linking them creates direct routes among nearby valid nodes once any one of them is reached. As shown in Lines 9-12 of Algorithm 1, each node in the result set is connected to μ nearest neighbors within the same set, where μ controls the number of vicinity-based shortcuts per node.

Example 3. *Fig. 4a illustrates how GAS adds attribute-aware shortcuts. We first reconstruct the traversal path to each valid result using the logged predecessors and connect the result to the nearest preceding valid node on its path. For example, node* 6 *is reached from entry node* 1*, and we add a shortcut from* 6 *to node* 4*. We apply the same rule to other valid results. We then add vicinity-based shortcuts within the final result set by connecting each valid result to its* μ *nearest neighbors in that set. When* $\mu = 1$*, node* 9 *is the nearest neighbor of nodes* 4 *and* 10*, so we add shortcuts from both to node* 9*.*

Algorithm 2: ***Create Structure-aware Shortcuts***

Input: graph index G, query log L
Output: set of structure-aware shortcuts SC_s

1 $SC_s \leftarrow \{\}$;
2 **foreach** $u \in L.results$ **do**
3 $\quad path \leftarrow$ reconstruct path to u by tracing $L.pre(\cdot)$;
4 $\quad v_{start} \leftarrow$ the point closest to q in $path$;
5 $\quad$ **foreach** cur *in* $path$ **do**
6 $\quad\quad$ **if** $L.dis_q(cur) > L.dis_q(u)$ **then**
7 $\quad\quad\quad cand \leftarrow$ points from v_{start} to cur in $path$, excluding cur;
8 $\quad\quad\quad nbrs \leftarrow$ select γ neighbors for u from $cand$;
9 $\quad\quad\quad$ **foreach** $v \in nbrs$ **do**
10 $\quad\quad\quad\quad$ Add shortcut $\langle v, u \rangle$ to SC_s;
11 $\quad\quad\quad$ **break**;
12 **return** SC_s;

3.3 Structure-aware Shortcuts

Rationale. Traversal may take detours in the divergence stage, where the search visits nodes farther from the query before reaching closer neighbors. These detours indicate routing inefficiency in the base graph and can be observed from historical query logs through traversal paths and the distances of visited nodes to the query. Motivated by this, we introduce *structure-aware shortcuts*, which add direct edges to bypass detour segments observed in past traversals and reduce redundant expansions in the divergence stage.

Adding Structure-aware Shortcuts. We add structure-aware shortcuts by detecting detours in historical traversal paths. As shown in Algorithm 2, the algorithm reconstructs and checks the divergence stage path of each valid result (Lines 3-5). If all intermediate nodes on this path are closer to the query than u, no shortcut is needed. Otherwise, when a node farther from the query than u is found, the nodes preceding this detour are collected as candidates, and γ of them are selected to form shortcuts to u (Lines 6-11), where γ is a parameter controlling the number of structure-aware shortcuts added per valid node. We adopt the RNG-based neighbor-selection strategy [7, 44] to choose candidates and form these shortcuts.

Example 4. *Fig. 4b illustrates the process of adding structure-aware shortcuts. In the search path from convergence point* 1 *to valid node* 6*, the query log shows that nodes* 4 *and* 5 *are farther from the query than node* 6*, indicating a detour. Nodes before this detour that are closer to the query (nodes* 1-3*) are collected as candidates, from which the algorithm selects and adds a shortcut, for instance from node* 2 *to* 6*, to bypass the inefficient segment.*

3.4 Maintenance of Shortcuts

We now describe how GAS maintains shortcuts by selectively retaining those with the highest potential utility, while bounding the total number of shortcuts to control space overhead. Specifically, each node stores at most m_a attribute-aware shortcuts for each attribute and at most m_s structure-aware shortcuts in total. In the following, we detail the maintenance procedures for attribute-aware and structure-aware shortcuts, respectively, and analyze the space complexity of GAS.

Attribute-aware Shortcuts. To prioritize shortcuts that are most useful during search, we rank candidate shortcuts by their probability of leading from a reached valid node to another valid node during the query processing. Specifically, for any $u \in V$ and $v \in SC_a[u]$, we define this probability as

$$p_{u\to v} := \Pr\big(f(v) = \text{True} \mid f(u) = \text{True}\big), \tag{1}$$

which represents the likelihood that v is also valid, conditioned on having reached a valid node u during search.

Although $p_{u\to v}$ cannot be computed exactly, we can estimate a confidence interval for it from historical query logs. Specifically, suppose there are $n_{u\to v}$ historical queries in which the search reaches a valid node u and subsequently visits node v. Let $\hat{p}_{u\to v}$ denote the empirical probability that v is also valid under these queries. Using Hoeffding's inequality, we associate the shortcut with a confidence interval for $p_{u\to v}$:

$$p_{u\to v} \in \left[\hat{p}_{u\to v} - \sqrt{\frac{\ln(4/\delta)}{2n_{u\to v}}},\ \hat{p}_{u\to v} + \sqrt{\frac{\ln(4/\delta)}{2n_{u\to v}}}\right] \tag{2}$$

Here, the parameter δ controls the aggressiveness of shortcut replacement: larger values of δ lead to more aggressive replacement of existing shortcuts with newly observed candidates.

Using the confidence interval for $p_{u\to v}$, we maintain at most m_a attribute-aware shortcuts for each node u under each attribute. If $|SC_a[u]| < m_a$, a newly observed shortcut is inserted directly. Otherwise, for a candidate shortcut v, we compare its lower-bound with the smallest upper-bound among the existing shortcuts of u; if the former is larger, we replace that shortcut with v, and otherwise discard v. This confidence-bound-based rule balances bounded space and shortcut utility: less frequently observed but promising shortcuts can still be preserved through wider confidence intervals, while frequently observed shortcuts are evaluated using confidence-adjusted estimates rather than raw frequency alone. We analyze this update strategy in Sec. 4.3 and show that it leads to better search complexity as more queries are processed over time.

Structure-aware Shortcuts. GAS maintains structure-aware shortcuts in a single auxiliary structure shared by all queries, with at most m_s shortcuts per node. Unlike attribute-aware shortcuts, these shortcuts are not tied to specific attributes or filters, so they can also benefit other attributes with few or no prior queries. When the count limit for a node u is reached, the shortcut farthest from u is discarded, ensuring that the remaining shortcuts are those most effective at repairing detours in the graph. After the shortcuts are incorporated, the corresponding query logs are discarded to avoid additional storage overhead.

Space Complexity. Under these shortcut count limits, each node in GAS stores three types of edges: m base-graph edges, up to m_a attribute-aware shortcuts for each attribute, and up to m_s structure-aware shortcuts. Therefore, each node stores at most $m + wm_a + m_s$ edges in total, where w is the attribute number. For a dataset with N vectors, the overall space complexity is $O(mN + wm_aN + m_sN)$.

4 SEARCH WITH GAS

This section presents the GAS search algorithm using the shortcut overlays from Sec. 3, followed by its theoretical analysis.

Algorithm 3: ***GAS Search Algorithm***

Input: graph index G, query q, result size k, filter f, search scope parameter k', attribute/structure-aware shortcuts SC_a, SC_s
Output: a set of k nearest neighbors of q satisfying f

1 Initialize *pool*, *ann*, $pool_{sc}$ with entry point of G;
2 **while** *pool is not empty* **do**
3 $u \leftarrow$ pop closest point to q in *pool*;
4 $v \leftarrow$ furthest point from q in *ann*;
5 **if** $dis(q,u) > dis(q,v)$ and $|ann| = k'$ **then break**;
  // Visiting Original Edges (Low Priority)
6 $v^* \leftarrow k$-th closest point to q in *ann*;
7 **if** $dis(q,u) < dis(q,v^*)$ *or* u *has valid neighbor* **then**
8 **foreach** *unvisited point* $o \in G[u] \cup SC_s[u]$ **do**
9 Mark o as visited and add to *pool*;
10 **if** $|ann| < k'$ or $dis(q,o) < dis(q,v)$ **then**
11 **if** o satisfies f **then** Add o into *ann* and $pool_{sc}$;
  // Visiting Attribute-aware Shortcuts (High Priority)
12 **while** $pool_{sc}$ *is not empty* **do**
13 $s \leftarrow$ pop closest point to q in $pool_{sc}$;
14 **foreach** *unvisited valid point* $o \in SC_a[s]$ **do**
15 Mark o as visited and add to *pool*;
16 **if** $|ann| < k'$ or $dis(q,o) < dis(q,v)$ **then**
17 Add o into *ann*;
18 **if** $dis(q,o) < dis(q,s)$ **then** Add o into $pool_{sc}$;
19 Trim *ann*'s size to k' by removing furthest points;
20 **return** k *points closest to* q *in* *ann*;

4.1 GAS Search Algorithm

We now present the GAS search algorithm, which integrates the proposed shortcuts into the search process to improve the divergence stage. For attribute-aware shortcuts, we design a *dual-queue traversal* strategy that separately traverses shortcut edges and base-graph edges while coordinating both to prioritize promising candidates. Structure-aware shortcuts are used as regular graph edges to improve routing. Moreover, we introduce a *candidate pruning strategy* to filter out less promising candidates in divergence and further reduce unnecessary visits to distant or invalid nodes.

Dual-queue Traversal Strategy. Attribute-aware shortcuts can improve efficiency by directly diverging to nearby valid nodes, thereby accelerating divergence. However, shortcuts that are effective for previous queries may not benefit the current one, so blindly prioritizing them can lead to unnecessary visits to distant valid nodes and increased computation. To address this, we introduce a dual-queue traversal strategy that prioritizes shortcuts leading to nearby valid nodes. Specifically, we maintain two queues: the original candidate queue and a high-priority shortcut queue. Shortcuts targeting closer valid nodes are pushed to the shortcut queue and explored first, while the rest are deferred to the candidate queue.

As shown in Algorithm 3, at each iteration, the main search pops a node from the candidate queue and first explores its standard graph neighbors (Lines 3-11). When a valid neighbor is found, it is added to both the result set and the *shortcut queue* (Line 11) for later shortcut exploration. The result set maintains the k' closest nodes found so far (Line 19), where $k' > k$ extends the search

scope for better accuracy. The algorithm then pops nodes from the shortcut queue and explores their valid shortcut neighbors (Lines 12-18). If a shortcut leads to a closer valid node, it is inserted into the shortcut queue (Line 18), forming a traversal chain that quickly reaches better results, while shortcuts to farther nodes are deferred to the main queue for later exploration. By alternating between the two queues, the algorithm quickly reaches nearby valid nodes while avoiding unnecessary visits to invalid ones.

Accelerating Search with Candidate Pruning. Although the dual-queue strategy quickly discovers valid nearest points, it may still explore invalid points in divergence that are closer to the query than the current valid results, causing redundant distance computations. To reduce this overhead, we design a candidate pruning strategy that skips invalid points unlikely to lead to better results.

Unlike the original filtered search algorithm on graph index, which uniformly explores all candidates within the expanded scope k', our pruning algorithm focuses on a smaller core region defined by the top-k results within k'. We perform equal exploration only within this core region and selectively explore nodes outside it only when they have valid neighbors that may lead to better results. Specifically, if a node lies outside the core region and none of its neighbors are valid, we skip exploring its neighbors (Lines 6-7 of Algorithm 3). This selective pruning can effectively reduce redundant computations while maintaining high accuracy.

4.2 GAS for Multi-Filter Queries

Beyond single-filter queries, GAS also supports queries with filters on more than one attribute. We describe how GAS handles conjunctive and disjunctive queries, which retrieve the nearest vectors satisfying all or at least one of the attribute filters, respectively.

For conjunctive queries, GAS selects the constrained attribute with the most developed shortcut structure to guide traversal using its attribute-aware shortcuts, while enforcing all filter conditions and retaining only nodes that satisfy every constraint. For disjunctive queries, GAS leverages attribute-aware shortcuts from all filtered attributes to guide traversal, and treats any node satisfying at least one filter predicate as eligible for the result set. In addition, query logs generated from conjunctive and disjunctive queries can be used to augment the future search. Specifically, each such query is decomposed into per-attribute traversal logs, which are then used for shortcut construction following the procedure in Sec. 3.

4.3 Theoretical Analysis

This section analyzes the search complexity of FANNS over graph indexes and shows that GAS reduces this complexity.

Search Complexity of FANNS on Graph Index. As discussed in Sec. 3.1, the performance bottleneck of FANNS lies in the divergence stage. Hence, prior analyses focusing only on the convergence path length [44] are insufficient to characterize FANNS's complexity. This part extends the analysis to account for the divergence stage. We assume an HNSW base graph [28], and that valid results collection begins after the convergence stage completes.

Lemma 1. *Consider a graph index G with average degree m and the average valid-visit ratio during the search is ρ. The expected number of visited nodes required to return k valid neighbors is $O(m \log N + k/\rho)$.*

Proof. Consider the two stages defined in Sec. 3.1 separately. In convergence stage, the expected search path length for HNSW is $O(\log N)$ [28], and searching each node involves visiting m neighbors, leading to $O(m \log N)$ time in convergence stage. Divergence stage aims at collecting k valid results. Given the average valid-visit ratio ρ, the expected number of visited nodes is k/ρ. Summing both stages yields $O(m \log N + k/\rho)$. □

Lemma 1 shows that the query complexity of FANNS on a graph index increases notably as the valid visit ratio ρ decreases, aligning with the findings from our quantitative analysis in Sec. 3.1.

Analysis of Searching on GAS. We now analyze the valid-visit ratio on GAS, which impacts the expected number of nodes visited in the divergence stage. For analysis, we assume single-filter queries are drawn from a fixed workload distribution, reflecting scenarios where the underlying query distribution remains relatively consistent over time [15, 21, 30, 31]. Theorem 1 shows that, under the shortcut maintenance strategy in Sec. 3.4, the probability of reaching a valid node via an attribute-aware shortcut increases as more queries are processed.

Theorem 1. *For any node u in the graph, the probability of visiting a valid point through its attribute-aware shortcuts (*i.e.*, $p_{u\to v}$ for $v \in SC_a[u]$) improves after each update of the shortcut list using the maintenance strategy in Sec. 3.4, with probability at least $1-\delta$.*

Proof. Consider a fixed query workload, and there are $n_{u\to v}$ historical queries in which the search reaches node u and subsequently visits node v. For each such query, define an indicator random variable that equals 1 if visiting v leads to a valid result and 0 otherwise. Under independent queries from the fixed workload, these $n_{u\to v}$ observations are *i.i.d.* Bernoulli random variables with mean $p_{u\to v}$. Let $\hat{p}_{u\to v}$ denote the empirical mean estimated from the query logs. By Hoeffding's inequality, for any $\varepsilon > 0$,

$$\Pr\left(|\hat{p}_{u\to v} - p_{u\to v}| \geq \varepsilon\right) \leq 2\exp(-2n_{u\to v}\varepsilon^2) \tag{3}$$

Substituting $\varepsilon = \sqrt{\frac{\ln(4/\delta)}{2n_{u\to v}}}$ from Equation (2) into Equation (3) yields:

$$\Pr\left(p_{u\to v} \in \left[\hat{p}_{u\to v} - \sqrt{\frac{\ln(4/\delta)}{2n_{u\to v}}},\ \hat{p}_{u\to v} + \sqrt{\frac{\ln(4/\delta)}{2n_{u\to v}}}\right]\right) \geq 1 - \frac{\delta}{2} \tag{4}$$

Therefore, under the maintenance strategy in Sec. 3.4, replacing v with v' as u's neighbor improves the probability of visiting a valid point, with probability at least $1 - \frac{\delta}{2}\cdot 2 = 1-\delta$, which concludes our proof. □

With Theorem 1, we now analyze the valid-visit ratio ρ in GAS, which largely influences the search complexity of FANNS queries. Consider a search path l in the graph with length $|l|$, and let ρ_0 denote the valid-visit ratio when the search expands only original graph edges. Along this path, expanding original edges evaluates $|l|\cdot m$ candidates, among which $|l|\cdot m\rho_0$ are valid in expectation.

In our GAS framework, attribute-aware shortcuts are explored only when the current node is valid, which occurs for a ρ_0 fraction of the $|l|$ visits in expectation. For each such valid visit at node u, the expected number of valid shortcut endpoints is $P(u) =$

$\sum_{v \in SC_a[u]} p_{u \to v}$. Therefore, the expected number of candidates (and valid nodes) contributed by attribute-aware shortcuts along l is $\rho_0 \sum_{u \in l} P(u)$. Combining the contributions from original edges and shortcuts yields the overall valid-visit ratio:

$$\rho_a = \frac{|l| \cdot m\rho_0 + \rho_0 \sum_{u \in l} P(u)}{|l| \cdot m + \rho_0 \sum_{u \in l} P(u)} = 1 - \frac{|l| \cdot m(1-\rho_0)}{|l| \cdot m + \rho_0 \sum_{u \in l} P(u)} \quad (5)$$

According to Theorem 1, the average value of $P(u)$ increases with high probability as more queries are processed. As a result, $\sum_{u \in l} P(u)$ grows over time, which in turn increases ρ_a as shown in Equation (5). Therefore, $\rho_a > \rho_0$ and continues to increase as more queries are processed. By Lemma 1, this increase directly translates into lower search complexity, demonstrating that GAS achieves lower search complexity than the base graph and that its performance continues to improve with additional queries.

5 EXPERIMENTAL STUDY

This section evaluates GAS on standard benchmarks and wide-table datasets. We summarize the key results as follows:

- We evaluate GAS comprehensively across all representative settings: single-attribute, multi-attribute, and wide-table vectors with up to 2,381 attributes. GAS consistently outperforms general indexes, improving QPS by up to 5.6×, 6.5×, and 42.1×, respectively. In the single- and multi-attribute settings, GAS matches dedicated-index performance with far lower space overhead (*e.g.*, 92.4% of dedicated QPS using 9.3% of the space in the multi-attribute setting), while dedicated indexes do not scale to wide-table settings.
- We conduct extensive experiments to evaluate GAS from multiple angles, including multi-filter queries, cold-start and warm-up behavior, robustness to workload shifts (query correlation and order), compatibility with different base graphs, sensitivity to result size, scalability to large datasets, and ablation studies.

5.1 Experiment Setup

Datasets. We use seven real-world datasets in our experiments, including three wide-table datasets with hundreds to thousands of structured attributes: **(1) COMMA [37]**: 768-dimensional ViT embeddings [9] extracted from road-facing camera frames, each with 1,310 numerical/categorical attributes, such as IMU/GPS measurements and CAN-bus signals. **(2) EMBER [5]**: 768-dimensional SBERT embeddings [36] extracted from program snippets, each accompanied by 2,381 numerical features, such as counts of functions and symbols. **(3) WDC [35]**: 768-dimensional SBERT embeddings [36] extracted from computer product descriptions, each with 140 numerical/categorical product attributes.

We also include four standard datasets: **(4) WIT [41]**: 2,048-dimensional image features extracted from Wikipedia, each with 17 numerical/categorical attributes, such as image size and language. **(5) YFCC [42]**: 192-dimensional CLIP embeddings derived from Flickr, each with 15 numerical/categorical attributes, such as upload time and license name. **(6) SIFT [17]**: 128-dimensional SIFT descriptors from the INRIA Holidays dataset, with synthetic attributes generated following prior work [33, 53]. **(7) MNIST [23, 24]**: 784-dimensional SVM-based vectors of a handwritten digit, each with a categorical label.

We use 1M-vector subsets from WIT, SIFT, and MNIST, and a 10M-vector subset from YFCC. For the wide-table datasets, we construct embeddings from the full original datasets, yielding 0.5M vectors for COMMA and WDC, and 0.8M vectors for EMBER. For each dataset, an additional 100K vectors are held out as query workloads. By default, we report top-k results with $k = 10$.

Baselines. We compare GAS with representative state-of-the-art methods from two paradigms. For general index paradigm, we compare against a single graph FANNS baseline using a **vanilla HNSW [1]** as the base graph. Note that this baseline outperforms those used in prior work [25, 33] by applying filtering during the search, rather than filtering after retrieving a fixed number of nearest neighbors. We also include recent general-index methods, **ACORN [33]**, **NaviX [39]** and **VBASE [51]**, in our comparison. For dedicated index paradigm, we compare against recent state-of-the-art FANNS methods, categorized by the types of attributes they support: **SeRF [53]**, **iRangeGraph [49]**, and **UNIFY [22]** for numerical attributes, **RWalks [6]** and **FilterGraph [25]** for categorical attributes. In addition, we also include **Brute-force** method, which first performs a binary search on attribute-sorted data, followed by a sequential scan over the valid vectors. For all baseline methods, we use their default parameters for evaluation.

Implementation. All methods are implemented in C++, and compiled with GCC 11.4.0. We implement GAS on top of the HNSWLib [1], with the default HNSW construction parameters set as $m = 32$ and $ef_{\text{construction}} = 200$. By default, we set GAS parameters to $\mu = 4$, $\gamma = 2$, $\delta = 0.1$, $m_a = 12$, and $m_s = 4$. Experiments of GAS start from the initial base graph without any shortcuts, and performance is recorded from this cold-start state as shortcuts are gradually added during query processing. For all methods, we vary the search parameter k' to generate QPS-recall curves.

Environment. All experiments are carried out on a server with Intel Xeon(R) Gold 6240 2.60GHz CPUs and 768GB RAM.

5.2 Comparison with Existing Methods

This section compares GAS with existing FANNS solutions under three settings: single-attribute, multi-attribute, and wide-table.

5.2.1 Single-attribute setting. We evaluate the single-attribute setting on three standard vector datasets: WIT [41], YFCC [42], and SIFT [17]. These datasets provide both numerical and categorical attributes, enabling comparisons between dedicated and general-index baselines under the same workload. For each dataset, we consider one numerical and one categorical attribute, running 100K queries at four selectivity levels (1%/4%/10%/40%), and report QPS, recall rate, index build time, and index size.

Search Performance. Fig. 5 reports query performance across datasets and selectivity levels. GAS delivers on-par performance to dedicated indexes while outperforming all general indexes, with up to 5.6× higher throughput on WIT at 95% recall. This gain comes from continuously leveraging query logs to add shortcuts that accelerate subsequent queries. Although GAS starts from the base graph before sufficient shortcuts are available, it warms up

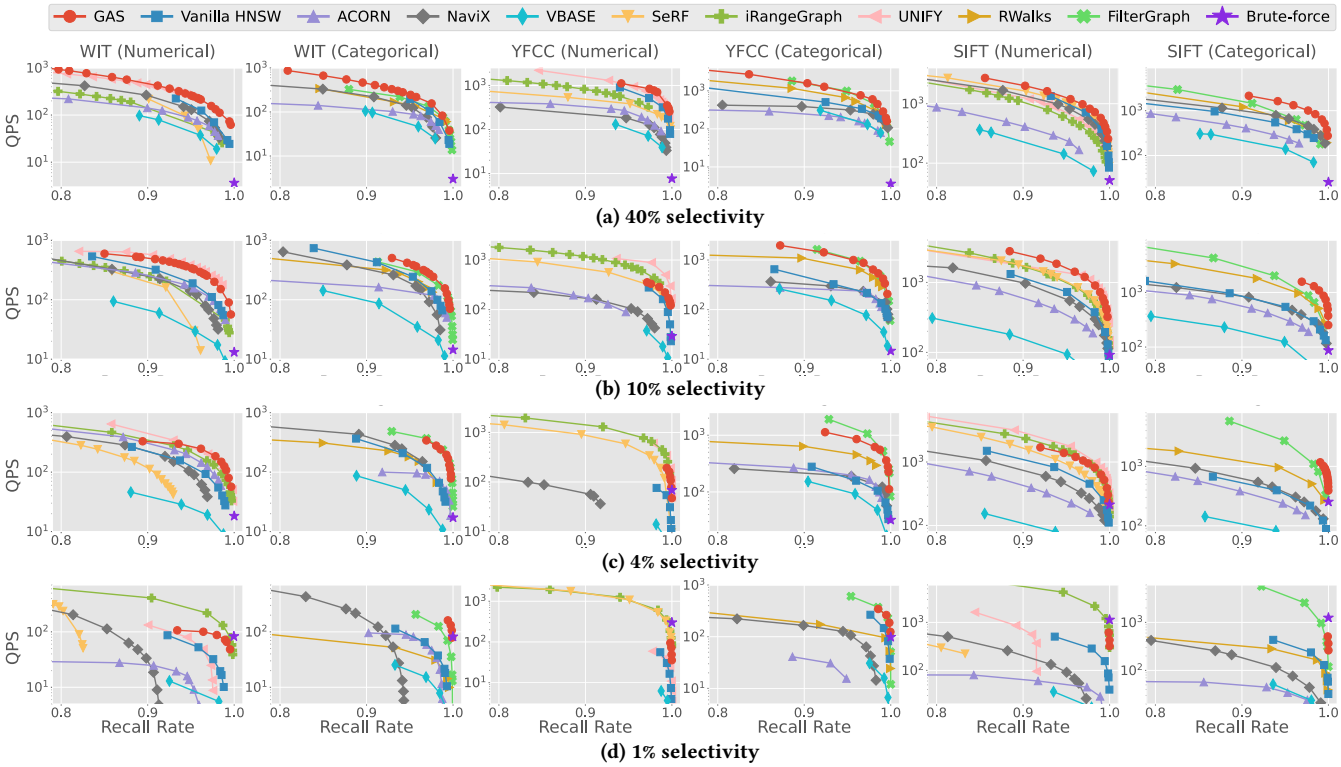


Figure 5: Search performance on three datasets in the single-attribute setting under varying selectivity levels.

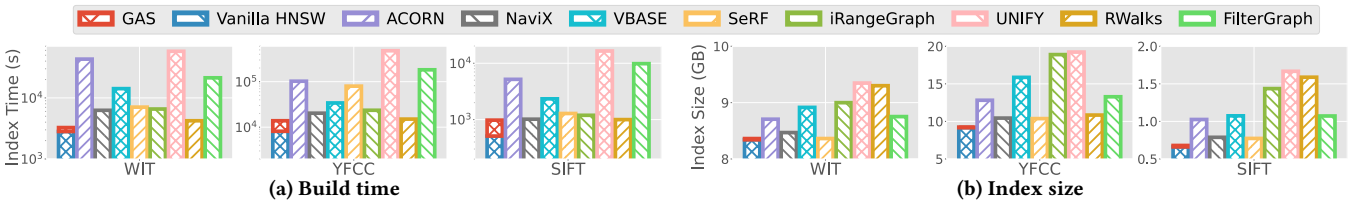


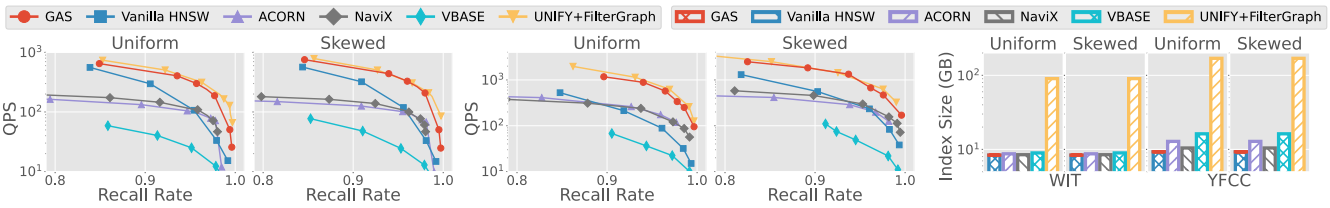
Figure 6: Index construction time and index size of all methods in the single-attribute setting across datasets.

quickly, reaching 80% of the best dedicated index's performance after 1K-16K queries across datasets. Dedicated indexes achieve strong performance by explicitly incorporating attribute information: SeRF, iRangeGraph, and UNIFY build range-specific subgraphs for numerical filters, while RWalks and FilterGraph add category-specific structures such as attribute vectors or extra connections, at the cost of higher construction overhead (see Fig. 6). In contrast, general indexes degrade due to inefficient traversal: ACORN incurs detours within the valid-node subgraph, whereas NaviX, VBASE and vanilla HNSW waste computation exploring invalid nodes.

Index Construction. Fig. 6 reports index construction performance. Since GAS builds the same HNSW base graph as vanilla HNSW, we report only the additional time and space overhead introduced by the shortcut structures after processing all query logs. On WIT, the shortcut structures increase build time by only 15.9% and index size by 0.3%, because GAS reuses query-processing results during construction and uses a maintenance strategy to bound space overhead. Across all datasets, GAS consistently incurs lower build time and index size than dedicated indexes, which construct heavy attribute-aware structures, such as range-specific subgraphs (iRangeGraph, UNIFY) or category-specific vectors/connections (RWalks, FilterGraph). SeRF reduces storage via lossy compression but can degrade accuracy in some cases (Fig. 5). ACORN and VBASE incur higher build time and index size than NaviX and vanilla HNSW, likely because ACORN requires a higher graph degree to preserve filter-specific connectivity, whereas VBASE additionally maintains an underlying table structure.

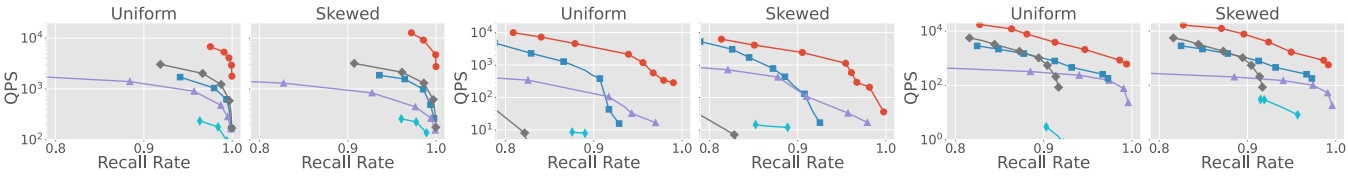
5.2.2 Multi-attribute setting. We conduct multi-attribute experiments on two standard datasets with multiple real-world attributes, WIT and YFCC, where each query filters on one of ten attributes. We generate two mixed-selectivity workloads: *uniform*, where all attributes are queried equally often, and *skewed*, where attribute frequencies follow a Zipf distribution with $\alpha = 2$. We compare GAS with general indexes and a composite baseline that uses the best-performing dedicated indexes from the single-attribute setting for numerical and categorical filters, *i.e.*, UNIFY and FilterGraph.

Search Performance. Fig. 7a-7b report query performance under two multi-attribute workloads. Across both workloads, GAS



(a) Search performance on WIT (b) Search performance on YFCC (c) Index size

Figure 7: Search performance and index size in the multi-attribute setting under varying query workloads.



(a) COMMA (b) EMBER (c) WDC

Figure 8: Search performance in the wide-table setting under varying query workloads.

achieves performance comparable to dedicated indexes and consistently outperforms general indexes, with up to 6.5× higher throughput. Under the uniform workload, GAS reaches 80% of the composite baseline's performance after 3K-20K queries and surpasses all general indexes after only 1K queries. The composite baseline of UNIFY and FilterGraph achieves the highest throughput by maintaining per-attribute indexes, but incurs substantially higher storage cost (see Fig. 7c). Under the skewed workload, GAS achieves higher average performance, reaching 92.4% of the dedicated index's QPS, because queries concentrate on a few attributes and GAS quickly forms effective shortcuts for them. In contrast, ACORN, NaviX, VBASE, and vanilla HNSW are less effective in multi-attribute settings because they lack attribute-specific guidance, leading to inefficient filtered traversal.

Index Construction. Fig. 7c reports total index size in the multi-attribute setting with 10 attributes. GAS and the general indexes, including vanilla HNSW, ACORN, NaviX, and VBASE, show only modest growth over the single-attribute case, because GAS adds only lightweight shortcuts and the others store only additional attribute values. In contrast, the composite baseline grows sharply because it builds separate indexes for each attribute. GAS remains among the smallest indexes, second only to vanilla HNSW, at only 9.3% of the dedicated index size with 10 attributes, highlighting the lightweight nature of our shortcuts.

5.2.3 Wide-table setting. We further evaluate the wide-table setting on COMMA [37], EMBER [5], and WDC [35]. Following the multi-attribute setup, we consider both uniform and skewed workloads with mixed selectivities. We compare GAS only against general indexes, since per-attribute dedicated indexes are impractical in wide-table scenarios due to prohibitive space overhead.

Search Performance. As shown in Fig. 8a-8c, GAS outperforms all general indexes across datasets and workloads after only 1K queries, achieving 2.1×-42.1× higher throughput at the same recall. NaviX, VBASE, and vanilla HNSW underperform and sometimes fail to reach 95% recall because diverse predicates and attribute distributions cause frequent detours and prevent them from reaching enough valid neighbors within a fixed search budget. ACORN can

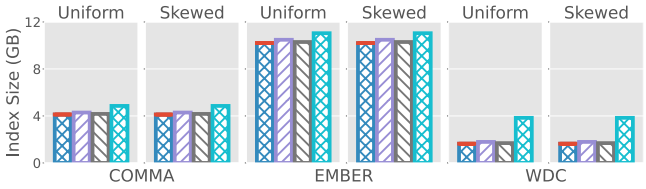


Figure 9: Index size in the wide-table setting.

reach 95% recall, but remains slow because its valid-node subgraph becomes less effective under diverse filters. Dedicated indexes such as UNIFY and FilterGraph do not scale to wide-table settings, as discussed in Sec. 1, since per-attribute constructions incur prohibitive space overhead even with far fewer attributes (Fig. 7c). Overall, GAS significantly improves over the base graph, confirming the effectiveness of shortcut overlays for wide-table workloads.

Index Construction. Fig. 9 reports total index size in the wide-table setting. GAS and the general indexes (ACORN, NaviX, VBASE, and vanilla HNSW) have similar sizes, since all maintain a single graph and the main increase over the multi-attribute setting comes from storing additional attributes or a small number of shortcuts. Dedicated indexes are omitted because per-attribute indexes for hundreds to thousands of attributes would incur prohibitive storage overhead. GAS adds only a small extra overhead for its auxiliary structures, yielding the second-smallest index after vanilla HNSW and demonstrating its lightweight design.

5.3 Evaluation of Proposed Method

In this section, we conduct a series of experiments to evaluate our GAS framework from multiple angles.

Exp-1: Search Performance of Multi-Filter Queries. We evaluate multi-filter queries in Sec. 4.2 using 100K two-attribute conjunctive queries on WIT and YFCC at 4% and 10% selectivity. As shown in Fig. 10, GAS consistently outperforms the base graph, with larger gains at lower selectivity. This is because GAS build shortcuts for constrained attributes from conjunctive-query logs and prioritizes attributes with richer shortcuts during search, improving valid visit ratio when the base graph degrades at low selectivity.

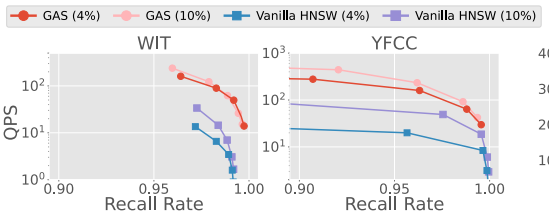
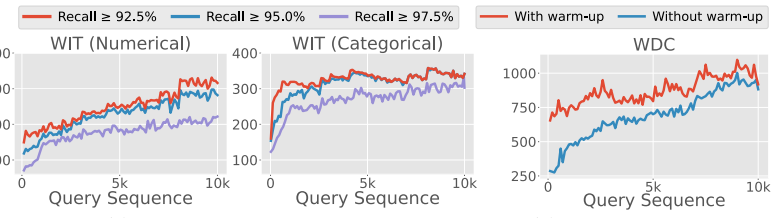


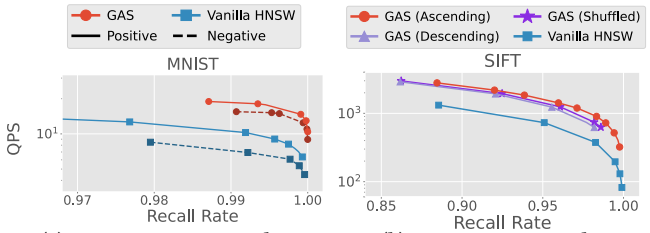
Figure 10: Multi-filter query performance.



(a) Cold-start performance

(b) Effect of Warm-up

Figure 11: Cold-start and warm-up behavior of GAS.



(a) Varying query correlation **(b) Varying query order**

Figure 12: Robustness of GAS under workload shifts.

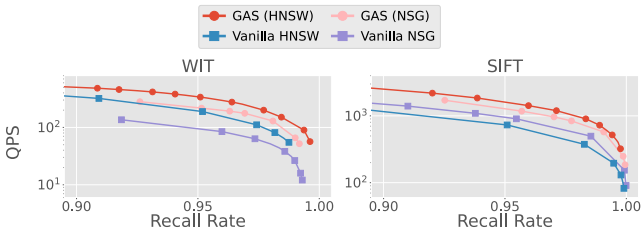


Figure 13: Compatibility of GAS with different graph indexes.

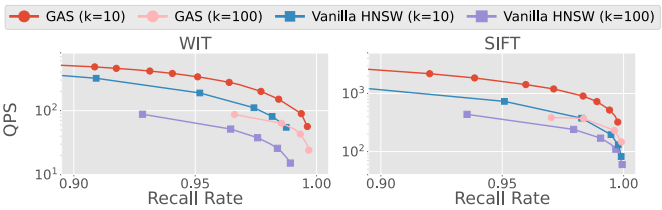


Figure 14: Search performance under different result sizes k.

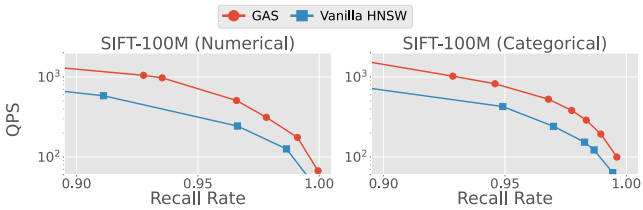


Figure 15: Search performance in the scalability test.

Exp-2: Cold-start Behavior and Warm-up. Since shortcuts in GAS are constructed from query logs, the system initially relies more on the base graph before sufficient queries are observed. We evaluate this cold-start behavior using 10K mixed-selectivity queries on WIT. We divide the query sequence into segments and report QPS and recall for each segment. As shown in Fig. 11a, GAS steadily improves QPS as more queries are processed under the same recall targets, with up to 3.1× speedup for numerical filters.

To mitigate the initial cold-start overhead, we explore a practical warm-up strategy: running generated queries on a warm-up attribute and adding the resulting shortcuts to the graph index, which can benefit all attributes through shared structure-aware shortcuts. To verify this effect, we evaluate GAS on WDC with 10K mixed-selectivity queries on an unseen attribute. We compare two settings: starting from scratch, and starting after 5K warm-up queries on other attributes. The latter setting contains structure-aware shortcuts derived from the warm-up queries. As shown in Fig. 11b, at 97.5% recall, these shortcuts improve search performance on the unseen attribute and reduce the warm-up burden in wide-table workloads. These results are consistent with Sec. 4.3: as more shortcuts are added, GAS reaches valid results more efficiently and needs a smaller search scope to achieve the same recall.

Exp-3: Robustness under Varied Query Correlation. To study robustness under different query correlations [33], we follow prior work and evaluate on MNIST dataset [23, 24]. We issue 10K *positively* correlated queries, where the target label matches the query digit, and 10K *negatively* correlated queries, where the target label differs. As shown in Fig. 12a, GAS improves performance in both settings. Under negative correlation, the base graph degrades more because valid nodes are less similar in embedding space, reducing the valid-visit ratio. Attribute-aware shortcuts allow GAS to reach valid nodes more directly, making it far less sensitive to this effect.

Exp-4: Robustness under Different Query Orders. We evaluate 100K SIFT queries under three orders based on their appearance in the query list: ascending order, descending order, and a random shuffle. Fig. 12b compares GAS across these orders, alongside the base graph, which is invariant to query order. The results show that GAS is robust to query order: When a useful shortcut is absent, early queries tend to expose similar traversal inefficiencies and construct similar shortcuts; thus, later queries can still benefit, and the overall performance converges across different orders.

Exp-5: Compatibility with Different Graph Indexes. We evaluate GAS on another widely used graph index, NSG [10, 11], and further include HNSW for direct comparison in Fig. 13. The results show that GAS consistently improves FANNS performance on top of both graph indexes. This demonstrates that the effectiveness of GAS is not tied to a specific base graph, but comes from alleviating common FANNS bottlenecks through query-adaptive shortcuts, making GAS compatible with different graph indexes.

Exp-6: Search Performance with Varying Result Size k. We further evaluate GAS with a larger result size ($k = 100$) to examine how its search performance scales with k. As shown in Fig. 14, QPS

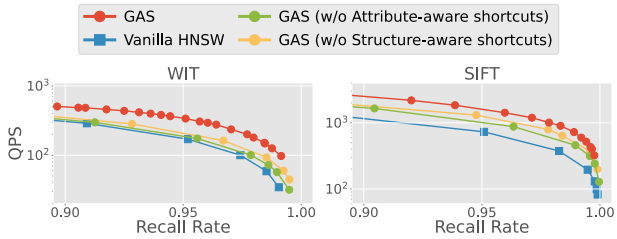


Figure 16: Ablation study of two types of shortcuts in GAS.

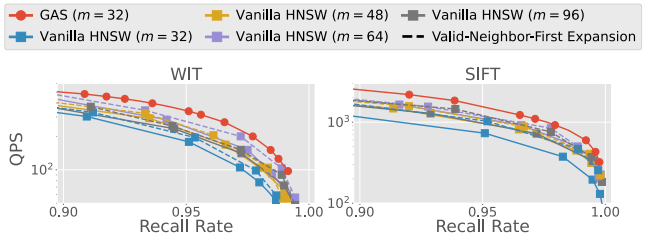


Figure 17: Comparing GAS with higher-degree base graphs.

drops for both GAS and the base graph, since retrieving more valid results requires expanding and verifying more candidate nodes in the divergence stage. Nevertheless, GAS retains its advantage, achieving at least 1.7× speedup at the same recall by using shortcuts to reduce ineffective divergence expansions.

Exp-7: Scalability Test. We evaluate scalability on the SIFT dataset at the 100M-vector scale [17], using 100K mixed-selectivity queries. As shown in Fig. 15, GAS scales well, achieving up to 2.5× higher throughput than the base graph. We also find that at this scale, although convergence becomes longer due to the larger dataset size, the divergence stage on the base graph still dominates computation, accounting for 96.6% of all distance evaluations, which is consistent with our bottleneck analysis in Sec. 3.1.

Exp-8: Effectiveness of Two Types of Shortcuts. We evaluate the two shortcut types in GAS through ablation on WIT and SIFT using 100K mixed-selectivity queries. As shown in Fig. 16, disabling either type reduces the gain over the base graph, confirming that they address complementary bottlenecks: attribute-aware shortcuts reduce wasted expansions on invalid nodes, while structure-aware shortcuts reduce routing detours during traversal.

Exp-9: Comparison with Higher-Degree Graph. To further assess shortcut effectiveness, we compare GAS with higher-degree HNSW and a stronger hybrid variant that uses offline attribute-sorted neighbor lists to prioritize valid neighbors during search. As shown in Fig. 17, GAS consistently outperforms both baselines. This suggests that simply increasing degree and prioritizing valid neighbors still cannot fully address FANNS inefficiencies identified in Sec. 3.1. Therefore, static graph augmentation remains less effective than GAS, further demonstrating the advantage of query-log-driven shortcuts over offline-added edges.

6 RELATED WORK

Approximate Nearest Neighbor Search. Existing ANNS indexing methods can be broadly classified into three categories. Partition-based methods [8, 13, 26, 27] divide vectors into partitions and search within the potential partitions. Quantization-based methods [12, 17, 29] compress vectors into compact quantization codes to approximate distances efficiently. Graph-based methods [11, 28, 48] have emerged as the preferred choice in practice due to their superior performance [7].

Constructing vector indexes often incurs large initial overhead, motivating research on reducing build cost [27, 50, 52]. Among these, CrackIVF [27], inspired by database cracking [16, 38], also leverages query-time computation to reduce initial indexing overhead. However, it differs fundamentally from our work in both problem definition and index design, as it studies partition-based index for unfiltered ANNS.

Filtered Approximate Nearest Neighbor Search. As categorized in Sec. 1, existing works can be divided into dedicated indexes [6, 14, 22, 25, 45, 49, 53] and general indexes [33, 39]. General indexes build a single vector index without incorporating attribute information, and enforce attribute filters during search. Owing to their simplicity and low overhead, this design is widely adopted in modern vector search libraries and database systems [2, 4, 43, 46]. Methods such as ACORN [33] and NaviX [39] further augment the search by leveraging 2-hop neighbors to improve accuracy on a single graph, and we include them as baselines in our evaluation.

Recent work has explored dedicated indexes for numerical [22, 49, 53] and categorical attributes [6, 14, 25]. For instance, SeRF [53] constructs subgraphs for all possible numerical ranges and compresses them into a single index, trading high construction overhead for improved query efficiency. Similarly, UNIFY [22] partitions the dataset by attribute, builds subgraphs for each partition, and links them through cross-partition connections to enable search within valid subgraphs. While such designs achieve high search efficiency, they are typically limited to specific attribute types and incur large construction and space overhead [20], making them unsuitable for wide-table scenarios. In our experiments, we compare GAS against several leading methods in this category [6, 22, 25, 49, 53].

7 CONCLUSION

In this paper, we present Graph with Adaptive Shortcuts (GAS), a lightweight framework for efficient FANNS over wide-table vectors. GAS augments a single graph with lightweight auxiliary structures built incrementally from query logs, achieving minimal space and construction overhead while improving search performance as more queries are processed. Experiments on real-world datasets show that GAS consistently outperforms existing general indexes across all settings, while delivering performance comparable to dedicated indexes with far lower space overhead.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (NSFC) (Grant Nos. 62425202, 62336003), the Beijing Natural Science Foundation (Z230001), the Fundamental Research Funds for the Central Universities No. JKF-2026007844235, and the State Key Laboratory of Complex & Critical Software Environment (SKLCCSE). Qian Tao's work is partially supported by Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing. Yongxin Tong is the corresponding author.

REFERENCES

[1] 2023. HNSWLIB. https://github.com/nmslib/hnswlib Accessed: 2026-07-22.

[2] 2025. Knowhere: Vector Search Engine Inside Milvus. https://github.com/zilliztech/knowhere Accessed: 2026-07-22.

[3] 2025. Modern ML Workloads. https://lancedb.com/blog/designing-a-table-format-for-ml-workloads/#modern-workloads Accessed: 2026-07-22.

[4] 2025. Qdrant Library. https://github.com/qdrant/qdrant Accessed: 2026-07-22.

[5] Hyrum S Anderson and Phil Roth. 2018. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637* (2018).

[6] Anas Ait Aomar, Karima Echihabi, Marco Arnaboldi, Ioannis Alagiannis, Damien Hilloulin, and Manal Cherkaoui. 2025. RWalks: Random Walks as Attribute Diffusers for Filtered Vector Search. In *SIGMOD*, Vol. 3. 1–26.

[7] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-based vector search: An experimental evaluation of the state-of-the-art. In *SIGMOD*, Vol. 3. 1–31.

[8] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient billion-scale approximate nearest neighborhood search. In *NeurIPS*, Vol. 34. 5199–5212.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *ICLR*. 1–21.

[10] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. The FAISS Library. *IEEE Big Data* (2025), 1–17.

[11] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2018. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *PVLDB* 12, 5 (2018), 461–474.

[12] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. In *SIGMOD*, Vol. 2. 1–27.

[13] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.

[14] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, and Premkumar Srinivasan. 2023. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *WWW*. 3406–3416.

[15] Ville Hyvönen, Elias Jääsaari, and Teemu Roos. 2022. A multilabel classification framework for approximate nearest neighbor search. In *NeurIPS*, Vol. 35. 35741–35754.

[16] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*, Vol. 7. 68–78.

[17] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE TPAMI* 33, 1 (2010), 117–128.

[18] Minjoo Ki, Daejung Kim, Kisung Kim, Seon Joo Kim, and Jinhan Lee. 2025. CARIM: Caption-Based Autonomous Driving Scene Retrieval via Inclusive Text Matching. In *ICCV*. 22036–22045.

[19] Jesper Knapp, Klas Moberg, Yuchuan Jin, Simin Sun, and Miroslaw Staron. 2024. A Multi-model Approach for Video Data Retrieval in Autonomous Vehicle Development. *arXiv preprint arXiv:2410.03580* (2024).

[20] Mocheng Li, Xiao Yan, Baotong Lu, Yue Zhang, James Cheng, and Chenhao Ma. 2026. Attribute Filtering in Approximate Nearest Neighbor Search: An In-depth Experimental Study. In *SIGMOD*. 1–26.

[21] Zhaoheng Li, Silu Huang, Wei Ding, Yongjoo Park, and Jianjun Chen. 2025. SIEVE: Effective Filtered Vector Search with Collection of Indexes. *PVLDB* 18, 11 (2025), 4723–4736.

[22] Anqi Liang, Pengcheng Zhang, Bin Yao, Zhongpu Chen, Yitong Song, and Guangxu Cheng. 2025. UNIFY: Unified Index for Range Filtered Approximate Nearest Neighbors Search. *PVLDB* 18, 4 (2025), 1118–1130.

[23] Yanjun Lin, Kai Zhang, Zhenying He, Yinan Jing, and X Sean Wang. 2025. Survey of Filtered Approximate Nearest Neighbor Search over the Vector-Scalar Hybrid Data. *arXiv preprint arXiv:2505.06501* (2025).

[24] Gaëlle Loosli, Stéphane Canu, and Léon Bottou. 2007. Training invariant support vector machines using selective sampling. *Large scale kernel machines* 2, 1 (2007), 301–320.

[25] Jiarui Luo, Miao Qiao, Chaoji Zuo, and Dong Deng. 2025. Tag-Filtered Approximate Nearest Neighbor Search. In *ICDE*. 3642–3654.

[26] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *VLDB*. 950–961.

[27] Vasilis Mageirakos, Bowen Wu, and Gustavo Alonso. 2025. Cracking Vector Search Indexes. *PVLDB* 18, 11 (2025), 3951–3964.

[28] Yury A Malkov and Dmitry A Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE TPAMI* 42, 4 (2020), 824–836.

[29] Julieta Martinez, Shobhit Zakhmi, Holger H Hoos, and James J Little. 2018. Lsq++: Lower running time and higher recall in multi-codebook quantization. In *ECCV*. 491–506.

[30] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-throughput vector similarity search in knowledge graphs. In *SIGMOD*, Vol. 1. 1–25.

[31] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.

[32] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *The VLDB Journal* 33, 5 (2024), 1591–1615.

[33] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. In *SIGMOD*, Vol. 2. 1–27.

[34] Jeffrey Pound, Floris Chabert, Arjun Bhushan, Ankur Goswami, Anil Pacaci, and Shihabur Rahman Chowdhury. 2025. MicroNN: An On-device Disk-resident Updatable Vector Database. In *SIGMOD*. 608–621.

[35] Anna Primpeli, Ralph Peeters, and Christian Bizer. 2019. The WDC Training Dataset and Gold Standard for Large-Scale Product Matching. In *WWW*. 381–386.

[36] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP*. 3982–3992.

[37] Harald Schafer, Eder Santana, Andrew Haden, and Riccardo Biasini. 2018. A commute in data: The comma2k19 dataset. *arXiv preprint arXiv:1812.05752* (2018).

[38] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2016. An experimental evaluation and analysis of database cracking. *The VLDB Journal* 25, 1 (2016), 27–52.

[39] Gaurav Sehgal and Semih Salihoglu. 2025. NaviX: A Native Vector Index Design for Graph DBMSs With Robust Predicate-Agnostic Search Performance. *PVLDB* 18, 11 (2025), 4438–4450.

[40] Jun Song, Jingyi Ding, Irshad Kandy, Yanghao Lin, Zhongjia Wei, Zilong Zhou, Zhiwei Peng, Jixi Shan, Hongyue Mao, and Xiuqi Huang. 2025. Magnus: A Holistic Approach to Data Management for Large-Scale Machine Learning Workloads. *PVLDB* 18, 12 (2025), 4964–4977.

[41] Krishna Srinivasan, Karthik Raman, Jiecao Chen, Michael Bendersky, and Marc Najork. 2021. WIT: Wikipedia-Based Image Text Dataset for Multimodal Multilingual Machine Learning. In *SIGIR*. 2443–2449.

[42] Bart Thomee, David A Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. 2016. Yfcc100m: The new data in multimedia research. *Commun. ACM* 59, 2 (2016), 64–73.

[43] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD*. 2614–2627.

[44] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *PVLDB* 14, 11 (2021), 1964–1978.

[45] Yuxiang Wang, Ziyuan He, Yongxin Tong, Zimu Zhou, and Yiman Zhong. 2025. Timestamp Approximate Nearest Neighbor Search over High-Dimensional Vector Data. In *ICDE*. 3043–3055.

[46] 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. *PVLDB* 13, 12 (2020), 3152–3165.

[47] Jingyi Xi, Chenghao Mo, Ben Karsin, Artem Chirkin, Mingqin Li, and Minjia Zhang. 2025. VecFlow: A High-Performance Vector Data Management System for Filtered-Search on GPUs. In *SIGMOD*, Vol. 3. 1–27.

[48] Jiadong Xie, Jeffrey Xu Yu, and Yingfan Liu. 2025. Graph Based K-Nearest Neighbor Search Revisited. *TODS* 50, 4 (2025), 1–30.

[49] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. In *SIGMOD*, Vol. 2. 1–26.

[50] Shuo Yang, Jiadong Xie, Yingfan Liu, Jeffrey Xu Yu, Xiyue Gao, Qianru Wang, Yanguo Peng, and Jiangtao Cui. 2025. Revisiting the index construction of proximity graph-based approximate nearest neighbor search. *PVLDB* 18, 6 (2025), 1825–1838.

[51] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, and Fan Yang. 2023. VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *OSDI*. 377–395.

[52] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *PVLDB* 16, 8 (2023), 1979–1991.

[53] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search. In *SIGMOD*, Vol. 2. 1–26.