

# An In-Depth Study of Filter-Agnostic Vector Search on a PostgreSQL Database System: [Experiments & Analysis]

DUO LU\*, Brown University, USA

HELENA CAMINAL, Google, USA

MANOS CHATZAKIS\*, Université Paris Cité, LIPADE, France

YANNIS PAKONSTANTINO, Google, USA

YANNIS CHRONIS, ETH Zurich & Google, Switzerland

VAIBHAV JAIN, Google, India

FATMA ÖZCAN, Google, USA

Filtered Vector Search (FVS) is critical for supporting semantic search and GenAI applications in modern database systems. However, existing research most often evaluates algorithms in specialized libraries, making optimistic assumptions that do not align with enterprise-grade database systems. Our work challenges this premise by demonstrating that in a production-grade database system, commonly made assumptions do not hold, leading to performance characteristics and algorithmic trade-offs that are fundamentally different from those observed in isolated library settings. This paper presents the first in-depth analysis of filter-agnostic FVS algorithms within a production PostgreSQL-compatible system. We systematically evaluate post-filtering and inline-filtering strategies across a wide range of selectivities and correlations.

Our central finding is that the optimal algorithm is not dictated by the cost of distance computations alone, but that system-level overheads that come from both distance computations and filter operations (like page accesses and data retrieval) play a significant role. We demonstrate that graph-based approaches (such as NaviX/ACORN) can incur prohibitive numbers of filter checks and system-level overheads, compared with clustering-based indexes such as ScaNN, often canceling out their theoretical benefits in real-world database environments.

Ultimately, our findings provide the database community with crucial insights and practical guidelines, demonstrating that the optimal choice for a filter-agnostic FVS algorithm is not absolute, but rather a system-aware decision contingent on the interplay between workload characteristics and the underlying costs of data access in a real-world database architecture.

CCS Concepts: • **Information systems** → **Retrieval efficiency**.

Additional Key Words and Phrases: Vector Collections, Filtered Vector Search

## ACM Reference Format:

Duo Lu, Helena Caminal, Manos Chatzakis, Yannis Papakonstantinou, Yannis Chronis, Vaibhav Jain, and Fatma Özcan. 2026. An In-Depth Study of Filter-Agnostic Vector Search on a PostgreSQL Database System: [Experiments & Analysis]. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 134 (June 2026), 26 pages. <https://doi.org/10.1145/3802011>

\*The work was done while working at Google.

Authors' Contact Information: Duo Lu, [duo\\_lu@brown.edu](mailto:duo_lu@brown.edu), Brown University, Providence, Rhode Island, USA; Helena Caminal, [hcaminal@google.com](mailto:hcaminal@google.com), Google, Sunnyvale, California, USA; Manos Chatzakis, [manos.chatzaki@gmail.com](mailto:manos.chatzaki@gmail.com), Université Paris Cité, LIPADE, F-75006 Paris, France; Yannis Papakonstantinou, [yannisap@google.com](mailto:yannisap@google.com), Google, San Diego, California, USA; Yannis Chronis, [chronis@ethz.ch](mailto:chronis@ethz.ch), ETH Zurich & Google, Zurich, Switzerland; Vaibhav Jain, [jainva@google.com](mailto:jainva@google.com), Google, Bangalore, India; Fatma Özcan, [fozcan@google.com](mailto:fozcan@google.com), Google, Sunnyvale, California, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/6-ART134

<https://doi.org/10.1145/3802011>

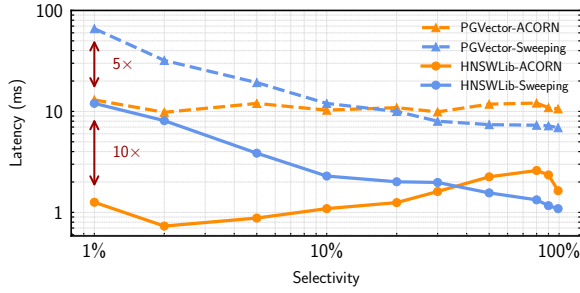


Fig. 1. Average latency for 100 filtered vector search queries on OpenAI-1M: HNSWLib (solid) vs. PGVector (dashed). System latency can be up to 10 $\times$  higher, and the gap varies with selectivity, motivating an in-depth comparison of FVS algorithms in a real DBMS.

## 1 Introduction

Vector embeddings enable semantic search over unstructured data. They underpin modern Generative AI and advanced information retrieval systems [25, 29], and their applications are rapidly proliferating across numerous domains [14, 36, 48].

Combining semantic search using vector embeddings with structured data filters enhances the quality of semantic query results [10], motivating the integration of *Approximate Nearest Neighbor vector Search (ANNS)* in conventional database systems. For example, consider an e-commerce query: a user wants a t-shirt "similar to this image" (vector search), but only those "available in size large" and "in stock locally" (filters on structured data). Enabling efficient *Filtered Vector Search (FVS)*, in other words, the efficient combination of relational filters, joins, and vector search, is a critical next step in blending traditional database systems with modern AI capabilities.

The baseline execution strategies of FVS are *pre-filtering*, which executes the filtering first, and *post-filtering*, which executes the vector search first. Pre-filtering and post-filtering are well understood and optimal only in some restricted scenarios. Going beyond them, blending filters and vector search has been shown to lead to efficient FVS [40]. There are multiple ways to order relational operators and vector search, and optimize the vector search itself.

Most existing research evaluates new FVS execution methods in a vector search library and not in a database system. This setup ignores system overheads and the cost of filter evaluation during the vector index traversal. These costs can dominate over the cost of distance computations, which is believed to be the most costly part of vector search. In the majority of existing studies, performance is often benchmarked solely by minimizing the number of distance computations, even if it requires more filter checks [39, 40].

Figure 2 shows significant differences in the performance characteristics of the same vector index traversal algorithms when implemented in a production database (PGVector [7]) compared to a standalone library (HNSWLib [4]). It is expected that the performance of an algorithm will differ when run in a standalone library compared to a fully fledged system. In our example, the total latency on the database system can be up to 10 $\times$  higher due to system overheads (Figure 1), which is expected. However, prior to this work, it was not evident that the performance gap between the two algorithms is not constant across different filter selectivities between the two setups (database and library). Consequently, the selectivity cross-over point—where one search algorithm becomes superior to another can change completely when filtered vector search is integrated into the database system.

Our observations show that minimizing distance computations alone is an unreliable proxy to improve the end-to-end FVS performance. As detailed in Table 1, this discrepancy stems from fundamental architectural differences between libraries and fully-fledged database systems. While libraries optimize for direct memory access, packing nodes contiguously and traversing graphs via raw pointers, database systems like PostgreSQL, introduce necessary layers of abstraction for durability and concurrency. In particular, neighbor filtering in a DBMS is rarely a single, lock-free memory dereference: it typically involves multiple engine layers (buffer manager and tuple access) before a candidate can be filtered and scored. As a result, algorithmic choices that look beneficial in libraries (e.g., more aggressive filtering to reduce distance computations) can become regressions once these system costs are accounted for.

It is well-established that the optimal approach for FVS depends on both filter selectivity and vector-predicate correlation [40]. However, a comprehensive comparison of methods across the full spectrum of selectivity-correlation combinations in a real system is absent. This gap prevents a complete understanding of when and why FVS strategies succeed or fail in a real database system.

Several existing approaches optimize vector indices for FVS by assuming specific filter patterns [15, 19, 23, 27, 32, 33, 41, 52, 56, 59, 61]. These approaches often limit the supported types and number of filter predicates, and thus, are not widely adopted for database workloads with arbitrary filter predicates on many attributes. Therefore, *filter agnosticism*—the ability to efficiently handle arbitrary, unknown filters at query time—is not just a desirable feature but a fundamental requirement for integrating vector search into a versatile database environment. To the best of our knowledge, ANN indices in commercial database systems are filter-agnostic [5, 8, 38, 49].

In this work, we evaluate filter-agnostic search strategies across two dominant indexing paradigms: graph-based and clustering-based, within a real database system. We use HNSW [35] to represent graph-based approaches for its widespread adoption [17, 22, 31, 53] and ScaNN [9], a performance-optimized clustering-based index. This selection allows us to contrast fundamental system-level trade-offs across index types. Our experiments use a commercially available PostgreSQL-compatible system, utilizing various filter selectivities and vector-predicate correlations on four different known vector datasets, ranging from 5 to 10 million vectors. Our goals are to inform a) the development of cost models for FVS queries and b) the design and optimization of vector search methods.

We summarize our contributions:

- We present filter vector search (with ANNS) evaluations conducted in a real DB system, focusing on filter-agnostic methods.
- We design a novel, dataset-agnostic workload generator that simulates workloads of different filter correlations and selectivities.
- We present guidelines and insights of filter-agnostic vector search algorithms, based on our observations.

## 2 Background on Filtered Vector Search

Databases blend relational operators with vector search. While database systems have long studied query planning and optimization across multiple operators, vector search poses a challenge. Most vector search use cases tolerate approximation in solution retrieval, enabling Approximate Nearest Neighbor Search (ANNS) via vector search indexes (e.g., HNSW [35], ScaNN [9]). The goal is to significantly reduce the data processed while maximizing recall.

### 2.1 Execution Strategies for FVS Queries

Based on the execution order of filter evaluation and vector search operators, we classify the strategies as follows (Figure 2):

Table 1. System breakdown for HNSW: architectural differences between standalone libraries and database systems.

| Arch. Aspect             | Library                                                                                                                       | System (PGVector)                                                                                                                                     | System overheads                                                                                                                    |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <b>Index Storage</b>     | <b>Compact representation:</b> Neighbors stored as raw pointers. Nodes are packed contiguously with no alignment constraints. | <b>Tuple-based storage:</b> Each neighbor reference is a 6-byte TID. Nodes are stored as tuples, with header overhead, in 8KB pages.                  | <b>Space amplification:</b> Page-level overhead (headers, alignment) reduces effective storage density, impacting cache efficiency. |
| <b>Node Access</b>       | <b>Direct deference:</b> The neighbor access is a CPU-level memory dereference.                                               | <b>Scattered page access:</b> The neighbors are stored as TIDs pointing to arbitrary pages.                                                           | <b>Random I/O amplification:</b> Each hop multiplies page overhead by neighbor fanout.                                              |
| <b>Filter Evaluation</b> | <b>Unified identifier:</b> A single ID/offset indexes both vector data and attribute data in memory.                          | <b>Indirection:</b> Index page $\rightarrow$ HeapTID $\rightarrow$ Heap page. Filter columns reside in the heap. Evaluating predicates needs HeapTID. | <b>Double lookup:</b> Every filtered candidate requires: (1) index access for vector+HeapTID, (2) heap access for filter columns.   |
| <b>Parallelism</b>       | <b>Flexible:</b> Intra-query parallelism via OpenMP, std::thread, or SIMD.                                                    | <b>Constrained:</b> One query per connection. No batch query primitive.                                                                               | <b>Arch. limit:</b> Executor/storage hinders fine-grained parallelism and batching.                                                 |

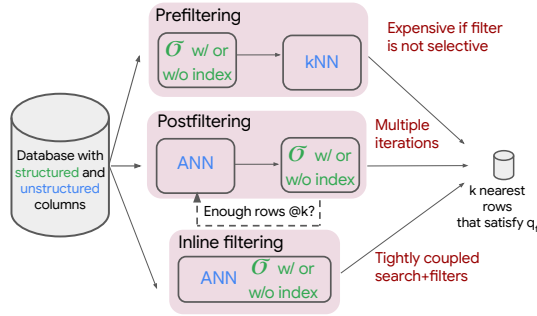


Fig. 2. Execution strategies for an FVS query.

- Pre-filtering: queries with extremely selective filters should filter first, then apply exact nearest neighbor (KNN) to surviving tuples.
- Post-filtering: if the filters are not selective, we should probe the ANN index, and then filter the result.
- Inline-filtering: tightly-coupled searching and filtering.

Note that to maintain an acceptable recall in post-filtering cases, we ought to retrieve  $k + \delta$  tuples from the vector index, to guarantee that after the filter is applied, there are at least  $k$  tuples left. Predicting the optimal  $\delta$  is non-trivial; however, both over- and under-provisioning have performance and quality implications. Alternatively, one may go back to probing the index, retrieving another round of NN points. We consider this last option (also known as *iterative post-filtering* or *iterative scan*) to belong to the inline-filtering family of execution strategies, since it merges filter evaluation and vector search.

In this paper, we focus on filter-agnostic inline filtering vector search methods. Filter-agnostic methods are better suited to database workloads that are dynamic and complex, and for which filters are not known a priori. Pre-filtering and inline filtering are complementary. After evaluating the filter predicates, query execution can choose the pre- or inline-filtering depending on the selectivity. Pre-filtering is faster when filters are very selective<sup>1</sup>. Pre-filtering, as an execution method, is straightforward for an optimizer to estimate its cost. In contrast, the performance of inline filtering is not well understood. We do not explicitly evaluate post-filtering because the iterative scan method subsumes it: once sufficient solutions are found in the first round of iterative scan, iterative scan and post-filtering are equivalent. The adaptability of iterative scan to the selectivity has made it a de-facto choice for FVS in PGVector systems. In conclusion, this study focuses on filter-agnostic inline filtering algorithms and their implementation in a real-world system.

## 2.2 Inline Filter-Agnostic Approaches

Filter-agnostic FVS decouples index construction from structured filters [39], offering three advantages: **① Arbitrary Filter Support:** Unlike specialized indexes limited to low-cardinality predicates, these handle any SQL filter—from equality to complex regex—without index modifications. **② Attribute-Blind Construction:** By ignoring attribute data during indexing, the structure remains valid and performant despite schema changes. **③ Update Resilience:** Modifications to attribute values never degrade the index or necessitate costly rebuilding, making it ideal for dynamic environments.

## 2.3 Filter-Agnostic FVS Algorithms

In our study, we focus on the two dominant paradigms for ANNS indexing: graph-based and clustering-based indexes. Specifically, we utilize Hierarchical Navigable Small World (HNSW) [35] and ScaNN [9], for their wide adoption.

**2.3.1 Unfiltered HNSW Search.** The HNSW search algorithm consists of two phases: **(i) Zoom-in Phase:** Starting at a top-layer entry point, the algorithm greedily traverses each layer to find the node closest to the query, using it as the entry point for the subsequent layer until reaching the base layer (layer 0). **(ii) Base Layer Search:** To find the  $k$  nearest neighbors, the search explores layer 0 using a min-priority queue of unvisited candidates ( $C$ ), a max-priority queue for the top- $ef$  results ( $W$ ), and a set of visited nodes ( $V$ ). At each step, the algorithm:

- Pops the top candidate from  $C$  and evaluates unvisited neighbors.
- Adds neighbors closer than the farthest node in  $W$  to both  $C$  and  $W$ , maintaining  $|W| \leq ef\_search$ .
- Stops when the top candidate in  $C$  is farther than  $W$ 's farthest.

The final result set contains the  $k$  closest vectors from  $W$ .

**2.3.2 Graph-based Filtered Search Algorithms.** Integrating filter predicates into the graph search algorithms introduces a fundamental design choice: when should we check if a node satisfies the filter predicates along the graph traversal? We categorize the evaluated three algorithms as:

**Filter-first (NaviX/ACORN)** [40, 45]: During traversal, it first checks if a node satisfies filters and, if it does not, it skips performing the distance computation.

**Traversal-first (Sweeping)** [44]: performs graph navigation using the original unfiltered structure, checking if a vector satisfies the filter only before moving candidates to the result priority queue.

**Iterative Scan:** [7] uses the index to produce candidate batches that are filtered after the graph traversal is complete.

Next, we deep dive into each of these methods.

<sup>1</sup>In some experiments, we have observed selectivity ranges below 0.01% benefiting from pre-filtering.

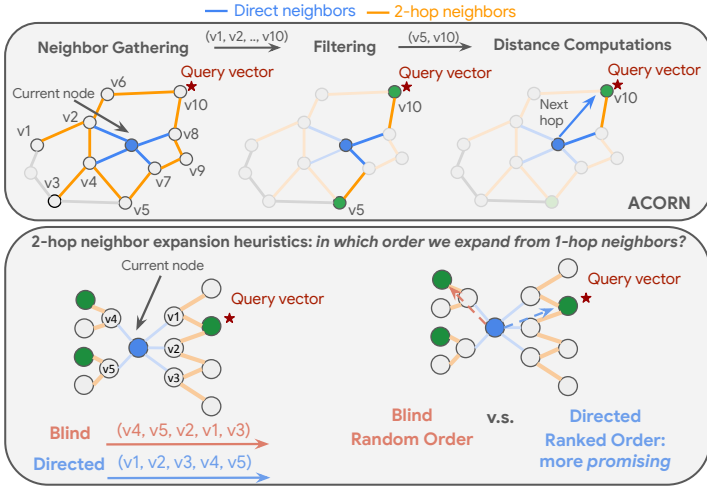


Fig. 3. Filter-first execution methods for an FVS query.

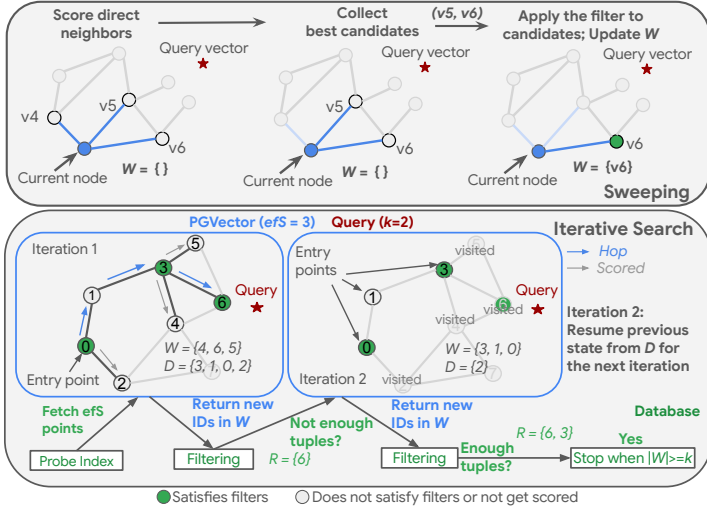


Fig. 4. Traversal-first execution methods for an FVS query.

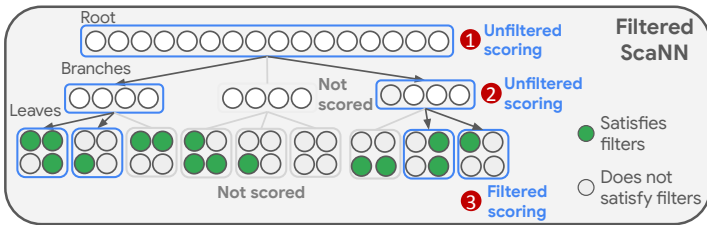


Fig. 5. Execution methods for an FVS query on ScaNN.

**2.3.3 ACORN.** Building on the HNSW index, ACORN was designed to address the inflexibility of filter-aware indexes.

**Predicate Subgraph Traversal.** At query time, the algorithm traverses the subgraph formed only by the nodes that pass the query's predicate (Figure 3-top). By ignoring neighbors that do not satisfy the filter, the search effectively steps across the graph to find paths within the predicate-specific subgraph.

Arbitrary predicate subgraphs can be too sparse or disconnected for efficient navigation. To maintain connectivity, ACORN employs *run-time 2-hop neighbor expansion* (Figure 3-top): at each traversal step, it gathers 1-hop and 2-hop neighbors, filters them, and computes distances only for qualifying nodes to determine the next hop. ACORN-1 relies solely on this search-time expansion, while ACORN- $\gamma$  also densifies the graph during construction, using compression to manage size.

**2.3.4 NaviX.** The NaviX algorithm design builds upon the ACORN-1 baseline but introduces a dynamic execution layer to optimize traversal under varying filter selectivities. The method defines three distinct heuristics: **Blind (Figure 3-bottom-left):** This strategy is generally equivalent to ACORN's 2-hop neighbor expansion. But it explores 1-hop neighbors first, followed by an expansion into 2-hop connections. **Directed (Figure 3-bottom-right):** A guided traversal that, when considering filtering 2-hop neighbors, prioritizes 2-hop neighbors from the closest (highest-ranked) 1-hop nodes, rather than exploring them indiscriminately. **Onehop-s:** A standard greedy search performed exclusively on filtered 1-hop neighbors. NaviX employs an *adaptive-local* mechanism to select the most appropriate heuristic at every traversal step based on the observed filter selectivity.

**2.3.5 Sweeping.** Sweeping [44] populates the result set ( $W$ ) based on proximity, as the non-filtered traversal, but conditionally to the points satisfying the filter (Figure 4).

**2.3.6 Iterative Scan.** It implements a "resumable" post-filtering strategy by decoupling the HNSW index traversal from the database (filter) executor. At each iteration, the algorithm maintains a *discarded candidate queue* ( $D$ ) containing visited nodes that were conceptually "paused" — either because (i) they were popped from the result set ( $W$ ) or (ii) were seen but not yet explored (Figure 4-bottom). If the initial batch yields fewer than  $k$  results after applying the filters, the search resumes execution by filling up its entry points and queues from the state preserved in  $D$ .

**2.3.7 Filtered ScaNN.** In a three-level tree, all root centroids are scored and used to select the closest branch candidates to the query vector (Fig. 5-①). Next, all these branch centroids are scored, similarly to the root centroids, to decide the best leaf candidates according to distance only (Fig. 5-②). Finally, in a filtered search, only points in leaves satisfying filters are fetched and scored (Fig. 5-③).

### 3 FVS Algorithms in PostgreSQL Database

In this section, we describe the system details that implement the previously described inline filtering methods on a PostgreSQL (PG) commercial-grade system.

#### 3.1 Graph-based Filter-First Methods

**Physical Layout.** In PG, data is persisted in fixed-size (of, by default, 8KB) pages. Index Pages in pgvector contain information about: (i) the node element: high-dimensional vector data and a pointer to the heap tuple, and (ii) information about its neighbors: pointers to neighbor pages.

**The implications of the Page Limit.** The current pgvector enforces a constraint for the neighbor information of a node to fit within a single page. This creates a hard structural limit on the graph topology:  $(L_{max} + 2) \cdot M \cdot S_{ptr} \leq S_{page}$ ; where  $M$  is the maximum number of connections per layer,  $L_{max}$  is the maximum layer height,  $S_{ptr}$  is the size of a neighbor pointer, and the constant 2 accounts for the base layer allowing  $2M$  connections (standard HNSW specification). This storage constraint

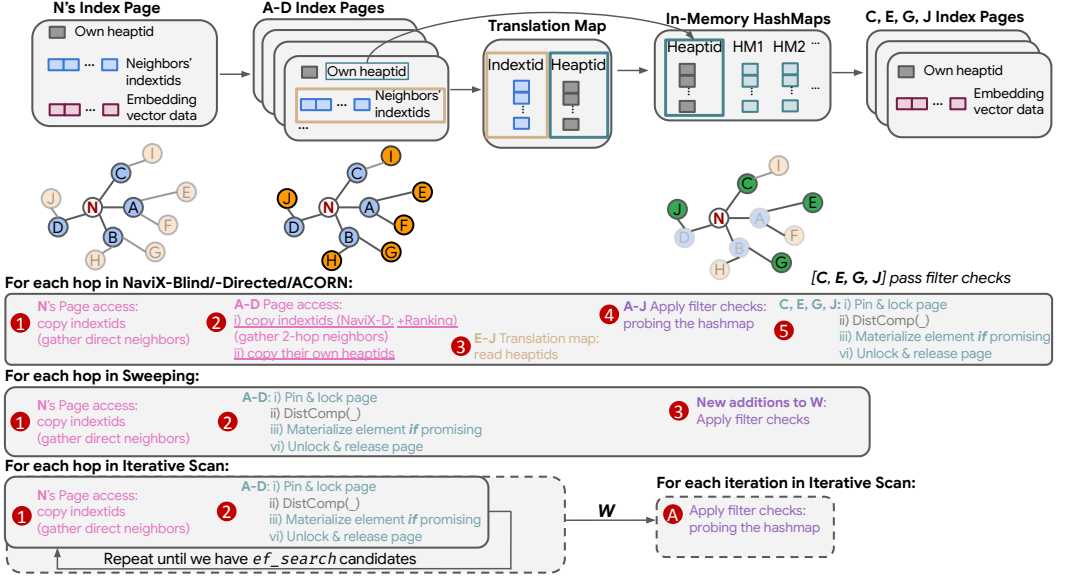


Fig. 6. NaviX, ACORN, Sweeping, and Iterative Scan implementations in PGVector.

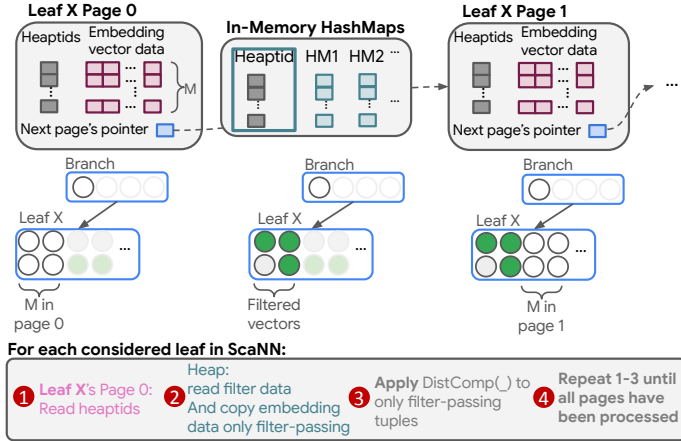


Fig. 7. Filtered ScaNN implementation in the system.

creates a zero-sum trade-off between the graph's connectivity ( $M$ ) and its navigability ( $L_{max}$ ). For example, with a standard configuration where  $M = 40$ ,  $L_{max} \approx 31$ . If we attempt to implement ACORN- $\gamma$  with  $\gamma = 2$  by simply doubling  $M$  to 80, the maximum hierarchy depth  $L_{max}$  drastically drops to  $\approx 14$ . For larger  $M$ , the graph rapidly collapses to merely 2 or 3 layers.

**TOASTing (PostgreSQL's The Oversized-Attribute Storage Technique) the neighbor lists is not an option** as visiting any node would trigger multiple random I/O requests just to materialize the node's neighbors and degrade the query latency significantly.

**System Level Costs.** A crucial and often overlooked factor in database system performance is the page retrieval and locking overhead associated with page access. In a row-based system like

PostgreSQL, a node's index page stores two types of identifiers: the *indextids* of its neighbors (pointing to other pages within the index structure) and the *heaptid* of the node itself (pointing to the row in the database table). Consequently, complex traversal algorithms that perform aggressive neighbor expansion, such as ACORN and the *Blind* or *Directed* heuristics of NaviX, incur a significant number of page accesses. To evaluate a candidate, the system cannot simply jump to the data; it must first fetch the candidate's index page to retrieve its *heaptid*. For strategies relying on two-hop expansion (e.g., ACORN and NaviX-Blind/Directed), this cost compounds geometrically. For a single node, the system must first fetch its index page to retrieve the *indextids* of its  $M$  neighbors (Figure 6-❶). Then, it must fetch each of those  $M$  neighbor pages to retrieve their *heaptids* for filtering and to discover the *indextids* of the two-hop neighbors (Figure 6-❷). Finally, to filter the (up to)  $M \times M$  two-hop neighbors, it must access their respective index pages to retrieve their *heaptids* (Figure 6-❸). In the worst case, this results in  $1 + M + (M \times M)$  page accesses, each requiring a buffer pool lookup and a page lock.

**Our Optimizations.** Porting NaviX/ACORN into a production row-oriented relational database necessitates adapting its design to the system's architectural constraints. A direct, naive implementation would be unacceptably slow due to the costs described previously. We introduced two critical optimizations, (i) *A pre-computed indextid-to-heaptid Translation Map*, implemented as an in-memory hash map (Figure 6-NaviX/ACORN's ❸). This map is generated during index build time. At query time, this map is loaded and used to perform the indextid to heaptid translation in-memory (Figure 6, Step 3). This optimization effectively eliminates an entire class of 2-hop neighbors' page accesses for a NaviX-Blind/NaviX-Directed/ACORN search step. (ii) *An adaptive 2-hop neighbor expansion (Hardening ACORN)*. We implemented a conditional strategy: if a 1-hop neighbor passes the filter, we skip the expensive 2-hop expansion for that branch. While this simple adaptive heuristic differs from NaviX's complex navigation logic, it is highly effective in the database system. It drastically reduces page access overheads in high-selectivity regions, ensuring ACORN serves as a competitive, optimized baseline.

### 3.2 Graph-based Traversal-First Methods

**Sweeping (Inline Filtering).** We implemented Sweeping [13] by modifying the internal HNSW loop to perform checks before adding candidates to the result set.

**Iterative Scan.** We utilize the implementation in PGVector 0.8.0. From a DBMS integration standpoint, Iterative Scan behaves fundamentally as a post-filtering method, following the logic described in Section 2.3.

### 3.3 Filtered ScaNN

As described in Section 2.3.7, the ScaNN-based filtered vector search performs the filtering at the leaf level only. The nature of cluster-based indexes allows for the implementation to be much more compact in size and simpler in number of steps: each leaf packs as many vectors as they fit in a single page (8KB) and maintains a linked list of pages of the same leaf (Figure 7, top). This allows for: a) simpler access to neighbors, and b) easier SIMD scoring. At the system level, once a leaf has been selected for scoring, the first page is accessed and the *heaptids* of the vectors in the leaf are used to read the in-memory hashmaps used for filtering (Fig. 7-❶ and ❷). At last, only tuples passing filters are scored using SIMD (Fig. 7-❸).

### 3.4 Steps in a Search Algorithm in a System

**3.4.1 Graph-based Indexes.** In summary, the types of actions a system may execute across search algorithms are as follows (Figure 6):

❶ **Accessing one-Hop Neighbors:** Access the current node's index page to read its direct neighbors' indexids (BlockNumber and OffsetNumber pair within the index file).

❷ **Gathering two-Hop Neighbors:** NaviX-Blind/ACORN: access the  $M$  direct neighbors index pages to gather their respective  $M$  neighbors' (two-hop neighbors) indexids. NaviX-Directed: access  $M$  direct neighbors index pages to score & rank all direct neighbors.

❸ **Translation for two-hop Neighbors:** NaviX-Blind / NaviX-Directed / ACORN: access the Translation Map (our optimization) to fetch the two-hop neighbors' heaptids.

**Apply Filter Checks** (❹ in NaviX/ACORN or ❸ in Sweeping or ❶ in Iterative Scan): Access the in-memory hashmaps (pre-generated, as we describe in more detail in Section 5).

**Vector Retrieval & Computing Distance:** (❺ in NaviX/ACORN (part of ❷ in the NaviX-Directed ranking phrase) or ❷ in Sweeping and Iterative Scan). This step scores a potential candidate node through a sequence of optimized operations designed to minimize memory overhead and unnecessary work. The process is not a single action but a conditional workflow, including page access, tuple retrieval, distance computation, conditional materialization, and resource release. Further details and profiling results are provided in Section 6.2.2.

Additionally, a Page Access involves multiple steps, including: (1) Page pinning in the memory buffer to prevent eviction and acquiring a shared lock to ensure a consistent read. (2) Reading the data directly from the shared page buffer. (3) The data for any given tuple, including its vector, resides on a shared buffer page that is only protected by a short-term lock. However, the search algorithm needs to maintain a working set of the best candidates found so far (the set  $W$ ) for the entire duration of the search on a given layer. To safely use a candidate's data across multiple comparisons — long after its original page lock is released — it must be copied into a private, query-local memory context. This materialization involves memory allocation (*palloc*) and copying the potentially large vector, making it an expensive operation. (4) Finally, the lock on the buffer is released, making the page available for other transactions.

**3.4.2 ScaNN Indexes.** ScaNN's index traversal is simpler to support in a PG system, compared to graphs. The root and branches traversal relies on walking the linked lists by accessing pages of the same cluster sequentially. Since there is no need to keep explicit neighbor information, the page can be filled with just multiple embedding vectors' data and the pointer to the next page. The leaves follow a similar flow to the one described in Section 3.4.1's **Vector Retrieval & Computing Distance** but in a batched way since a page references multiple tids for the filters. Only the embedding vectors of nodes passing filters are scored against the query vector.

## 4 Filtered Vector Workload Generator

### 4.1 The need for exhaustive correlation studies

Filters eliminate rows and thus the vectors that should be considered by the vector search of an FVS query. Filter agnostic vector indices are built on the unfiltered data, filters can make the index traversal significantly more difficult compared to the same search without filters. Concretely, filter selectivity and vector-predicate correlation change the distribution of the vectors satisfying the filters in the vector space, and change the vector search hardness (the effort to achieve the same recall).

Existing benchmarks based on real queries and datasets do not control selectivity and correlation simultaneously [3, 47], whereas synthetic benchmarks do not exhaustively evaluate queries with varying correlations [1, 6, 15, 19, 23, 26, 42, 45, 46, 52, 61].

To address this limitation, we developed a sophisticated filtered vector search workload generator that can be used to evaluate the behavior and performance of filter-agnostic FVS algorithms. Our goal is to evaluate the vector index search component of filter-agnostic inline-filtering vector search



Fig. 8. Illustration of the filter generator methodology. A Positive Correlation workload is then generated by preferentially sampling vectors from the closest neighbors, conversely, for the Negative Correlation.

methods using an execution strategy that first evaluates the filters and then passes the filter results as a bitmap to the vector index search.

Given a vector dataset, a vector query, a selectivity, and a correlation type, our generator produces a set of row IDs that satisfy the filters and meet the selectivity and correlation type specification. The list of rowIDs produced for each query and query specification simulates the result of evaluating the filter predicates, without the need to artificially generate structured data and filter predicates to achieve specific selectivity and correlation specifications. From the perspective of vector index search, this setup can simulate queries with a single complex predicate or multiple complex predicates.

## 4.2 Implementation

First, for every query, the generator calculates the distances of all the database vectors from the query vector. For each query, the database vectors are sorted by increasing distance and stored in an array. For each query selectivity and correlation combination, we generate the query result by sampling row IDs from the sorted array. The selectivity parameter controls the number of rowIDs sampled, and the correlation type determines the sampling method. Figure 8 illustrates how our generator takes into account the correlation type during sampling. For example, a positive vector-predicate correlation [40] means that the vectors closest to the query vectors have a higher probability of satisfying the filters. We consider four correlation types. In practice, we observe and expect most queries to have a positive correlation; therefore, we created three positive correlation types and one negative correlation type.

**High Positive Correlation:** For a high positive correlation, our generator samples from only the first third of the sorted array, in other words, the closest vectors to the query. In Figure 8, that would be the green vectors. We perform biased sampling (by applying softmax) based on the distance of each vector to the query vector, to ensure that the vectors closer to the query have a higher probability of being selected.

**Medium Positive Correlation:** Generating workloads with medium positive correlation follows the same process as the high positive correlation. The key difference is that for medium positive correlation, the generator considers the first half of the array for every query, rather than only the first third.

**Low Positive Correlation:** Generating workloads with low positive correlation follows the other positive correlation cases. However, for low positive correlation, the workload generator samples from the entire sorted array using the same biased sampling method.

Table 2. Comparison and characteristics of benchmark datasets.

| Dataset            | # Vec | Dims | Metric | Source            | Dist.-Filt. Rel. Cost | LID/LRC    |
|--------------------|-------|------|--------|-------------------|-----------------------|------------|
| sift10M [26]       | 10M   | 128  | L2     | Image Descriptors | 0.8x                  | 19.11/0.82 |
| openai5M [6]       | 5M    | 1536 | IP     | Text Embeddings   | 6x                    | 33.34/0.97 |
| cohere10M [1]      | 10M   | 768  | L2     | Text Embeddings   | 3.3x                  | 38.63/0.97 |
| text2image10M [46] | 10M   | 200  | L2     | Multimodal Emb.   | 1.3x                  | 52.47/0.98 |

Table 3. Index build time and size comparison.

| Dataset       | Time (min) |       | Size (GB) |       |
|---------------|------------|-------|-----------|-------|
|               | HNSW       | ScaNN | HNSW      | ScaNN |
| sift10M       | 24.80      | 1.87  | 9.58      | 1.73  |
| openai5M      | 49.10      | 10.36 | 38.15     | 1.89  |
| cohere10M     | 72.83      | 18.24 | 38.14     | 1.95  |
| text2image10M | 20.35      | 2.84  | 12.72     | 2.35  |

**Negative Correlation:** The negative correlation first multiplies all distances of the sorted array by  $-1$  and then proceeds identically to the low positive correlation case.

**No Correlation:** Non-correlated query workloads can be created by simply performing a random sampling over the dataset vectors.

## 5 Experimental Setup

**System.** All experiments were conducted on a server equipped with two 32-core AMD EPYC 7B13 CPUs (64 hyperthreads) and 240 GB of RAM, running Debian GNU/Linux (kernel 6.12.32). Our benchmark is implemented in a commercial-grade PostgreSQL-compatible database system using the PGVector [7] extension, and includes the ScaNN extension evaluated in the same PostgreSQL database environment. Since PGVector does not support intra-query parallelism, we evaluate throughput under load using workload-level parallelism: a client-side driver issues queries over multiple concurrent database connections. We use 16 client processes to execute queries from the workload in parallel.

**Index Configurations.** We next describe the index configurations and tuning methodology used throughout our evaluations.

**Quantization.** A key difference among the evaluated index families lies in their reliance on quantization. **(i) HNSW in PGVector:** As shown in Table 4, we evaluated PGVector’s available, off-the-shelf quantization options for HNSW—scalar quantization (halfvec/SQ16) and binary quantization with re-ranking—using a microbenchmark on *unfiltered* search at 95% recall@10. Across datasets, quantization substantially reduces index footprint and build time (up to  $\sim 12\times$  and  $\sim 3\times$ , respectively), but yields *no consistent QPS improvement* for recall (ranging from  $0.75\times$  to  $1.04\times$ ; in some cases failing to reach 95% recall within reasonable search parameters). This observation is consistent with prior reports from PGVector community [28]. This behavior is expected because PGVector-HNSW remains dominated by *random page accesses*: higher page density reduces bytes moved related to embedding data, but does not reduce the number of neighbor-related page fetches and buffer-manager interactions on the critical path. Therefore, since quantization does not improve latency/QPS at our target recall, we report HNSW results using the unquantized index and focus our tuning on run-time traversal parameters. **(ii) ScaNN:** In contrast, the clustering-based ScaNN design is inherently compatible with quantization: it scans leaves/clusters sequentially and can

Table 4. QPS speedup and reduction in index size and build time of 1,000 unfiltered vector search queries in PGVector-HNSW with 16 threads (Microbenchmark 95% Recall@10).

| Dataset       | Method    | QPS   | Index Size | Build Time |
|---------------|-----------|-------|------------|------------|
| sift10M       | Halfvec   | 0.97× | 1.34×      | 1.01×      |
|               | BQ+rerank | —*    | 2.00×      | 1.02×      |
| openai5M      | Halfvec   | 1.03× | 2.00×      | 1.11×      |
|               | BQ+rerank | 0.75× | 11.63×     | 2.86×      |
| cohere10M     | Halfvec   | 1.04× | 2.00×      | 1.24×      |
|               | BQ+rerank | 0.96× | 11.83×     | 2.33×      |
| text2image10M | Halfvec   | 0.99× | 1.53×      | 1.14×      |
|               | BQ+rerank | —*    | 2.55×      | 1.17×      |

\*Unable to reach 95% recall within reasonable search parameters.

Table 5. Latency speedup of different settings w.r.t. non-quantized non-PCAed ScaNN on 95% recall@10 with 16 threads (no correlation). Columns show increasing selectivity. Quantization uses SQ8.

| Dataset       | Method     | 1%<br>sel. | 5%<br>sel. | 20%<br>sel. | 50%<br>sel. | 80%<br>sel. |
|---------------|------------|------------|------------|-------------|-------------|-------------|
| sift10M       | Quant.     | 1.40×      | 1.42×      | 1.56×       | 1.49×       | 1.58×       |
| openai5M      | PCA        | 29.39×     | 5.79×      | 3.92×       | 3.22×       | 3.62×       |
|               | PCA+Quant. | 50.21×     | 8.61×      | 6.06×       | 5.02×       | 5.17×       |
| cohere10M     | PCA        | 5.24×      | 4.47×      | 3.48×       | 3.76×       | 3.83×       |
|               | PCA+Quant. | 9.60×      | 7.27×      | 6.55×       | 6.48×       | 5.52×       |
| text2image10M | Quant.     | 2.17×      | 1.51×      | 1.45×       | 1.87×       | 1.79×       |

\*sift10M and text2image10M are low-dimensional datasets. PCA is not applicable.

\*PCA vector dimension reduction (openai5M: 1536→193, cohere10M: 768→157).

leverage dense, quantized representations to achieve higher throughput (e.g., via SIMD scoring and more efficient use of memory bandwidth). Accordingly, we enable ScaNN quantization in the main experiments and choose corresponding available quantization settings based on the ablation in Table 5.

**Hyperparameter Tuning.** (i) **HNSW:** Our evaluation focuses on the runtime behavior of the search strategies for a fixed HNSW construction across datasets. All experiments use the same HNSW index built with fixed construction parameters ( $M=32$ ,  $ef\_construction=200$ ), following prior system settings [45]. For each algorithm, we tune its run-time parameters, primarily  $ef\_search$  and the ( $max\_scan\_tuples$  in PGVector), and use the configuration that yields the highest QPS at 95% recall. (ii) **ScaNN:** For filtered ScaNN, we tune the main run-time knobs that control the candidate budget and the accuracy/throughput trade-off: the number of leaves to scan and the reordering factor (to offset quantization error). We fix construction parameters:  $num\_leaves=10K$  and  $max\_num\_levels=1$  for OpenAI-5M, and  $num\_leaves=100K$  and  $max\_num\_levels=2$  for the other datasets. All results report the average over five independent runs.

**Database Configurations.** To eliminate the possible disk I/O variability, we utilized the *pg\_prewarm* extension to fully load both the table and the index into the database buffer cache before each experimental run. The scope of this work focuses on in-memory use cases. By removing disk latency,

we demonstrate that even under ideal memory-resident conditions, the architectural overhead of the database engine alone is enough to alter algorithmic trade-offs compared to libraries. The PostgreSQL instance we use is set with `shared_buffers = 64GB` and `maintenance_work_mem = 64GB`.

**Datasets.** To ensure a comprehensive and robust evaluation, we selected four datasets that span a wide range of size scales, distance metrics, vector dimensionalities, and intrinsic query difficulties (Table 2). Our selection includes datasets with varying costs for distance computations: low (`sift10M` [26], `text2image10M` [46]) and high (`cohere10M` [1], `openai5M` [6]), primarily driven by vector dimensionality. Furthermore, these datasets exhibit different levels of query hardness, providing a challenging testbed. Specifically, the `text2image10M` workload includes out-of-distribution (OOD) queries, which are difficult cases in which a query vector may not be close to any data points satisfying the filter.

**Relative Costs.** We have also calculated the relative cost of distance computations compared to bitmap probing (Table 2 Dist-Filt. Rel. Cost column). These results are from running filter checks and distance computations in isolation, without utilizing PostgreSQL, thereby emulating the relative cost incurred by queries served by a library (HNSWLib).

**Workloads.** For each dataset, we generate a comprehensive query workload designed to test algorithm performance across a spectrum of filter conditions. From a base set of 100 queries, we create variants covering nine distinct filter selectivities, ranging from highly-selective / low-selectivity (0.01) to non-selective / high-selectivity (0.9), and five vector-predicate correlations. This methodology yields 4,500 unique queries per dataset ( $100 \times 9 \times 5$ ), enabling fine-grained analysis of how each FVS method behaves across widely varying scenarios. Each reported data point represents the average performance across the 100 base queries for a given experimental configuration (i.e., correlation and selectivity).

**Query Execution in PostgreSQL.** In Section 4.2, we describe how we generate the queries. Each query is described by a query vector and a bitmap of the rowIDs that satisfy a filter with the specified selectivity and correlation. Our experimental setup simulates the execution of a query plan that applies the filters first, produces a bitmap of rowIDs that satisfy the filter, and then performs the vector search. During the HNSW graph traversal, checking if a vector satisfies the filter becomes a probe into this bitmap. For each query, we report the latency of the FVS algorithm by instrumenting the index access method, recording the wall-clock time from the beginning of the index scan to its completion. This measurement exclusively covers the time spent searching the index, excluding external overheads such as bitmap generation and query planning.

The execution strategy, where the filter is applied first in order to produce a bitmap, is commonly used to avoid switching between vector index traversal and filter evaluation, polluting the caches and the buffer pool [45]. If filters are expensive to apply, evaluating them for the entire input of the vector search might be inefficient. In this case, a query optimizer may evaluate the filters during the traversal of the vector index or choose a post-filtering method. In our evaluation, we focus on inline filtering methods that leverage precomputed bitmaps, modeling non-expensive filters.

**Query Hardness.** To formally quantify the inherent difficulty of the search task for each dataset, we adopt two key metrics, Local Intrinsic Dimensionality (LID) [16] and Local Relative Contrast (LRC) [24], which have been extensively used in prior work to characterize vector search workloads [17, 18, 20, 21, 50, 54, 55]. LID measures the effective dimensionality in the local neighborhood of a query point, with higher values indicating a more complex search space. LRC assesses the correlation between unfiltered nearest neighbors and true filtered nearest neighbors, where higher values between 0 and 1 suggest a harder filtered search task. Table 2 reports these values, confirming the diverse query hardness across our datasets, with `text2image10M` being particularly challenging.

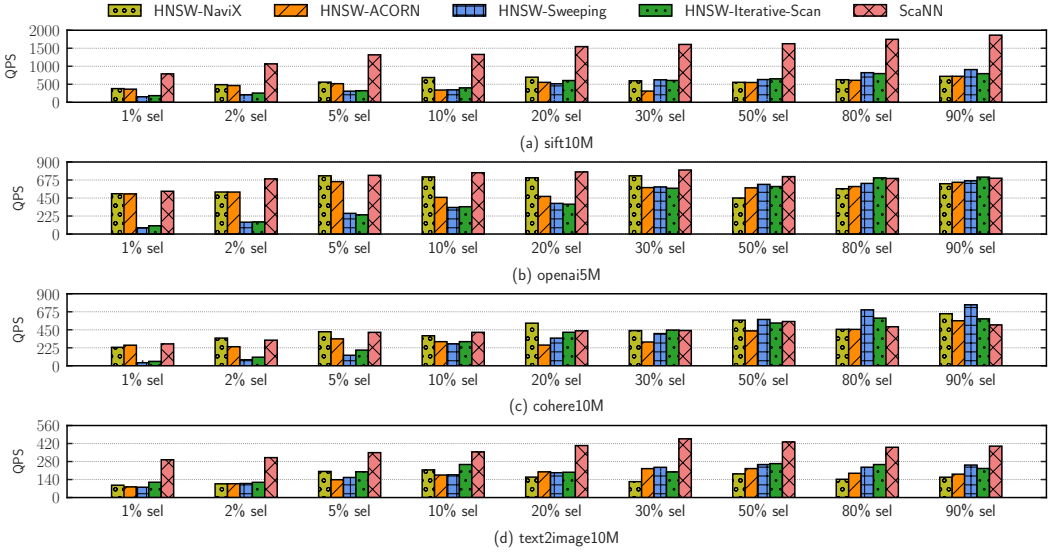


Fig. 9. QPS at 95% Recall@10 for methods across various selectivities on four datasets with no vector-predicate correlation.

Table 6. Breakdown of internal index statistics across varying selectivities on OpenAI-5M (no correlation). Colors indicate relative performance within each metric group (green=lower/better, red=higher/worse). Bold indicates the best value per row.

| Sel. | Distance Computations |            |             |       | Filter Checks |       |             |        | Hops / (Leaves & Reordering) |             |        |        | Page Access |             |             |       |
|------|-----------------------|------------|-------------|-------|---------------|-------|-------------|--------|------------------------------|-------------|--------|--------|-------------|-------------|-------------|-------|
|      | NaviX                 | ACORN      | Sweep.      | ScaNN | NaviX         | ACORN | Sweep.      | ScaNN  | NaviX                        | ACORN       | Sweep. | ScaNN  | NaviX       | ACORN       | Sweep.      | ScaNN |
| 1%   | 246                   | <b>238</b> | 23.0K       | 1.4K  | 71.8K         | 71.6K | <b>2.6K</b> | 143.9K | 17.8                         | <b>17.7</b> | 1.1K   | 150/62 | 1.2K        | <b>1.2K</b> | 24.0K       | 4.5K  |
| 2%   | <b>357</b>            | 408        | 9.4K        | 1.9K  | 66.6K         | 59.6K | <b>1.4K</b> | 96.0K  | <b>19.8</b>                  | 20.9        | 398    | 100/60 | <b>1.5K</b> | 1.6K        | 9.8K        | 3.2K  |
| 5%   | 627                   | <b>614</b> | 5.0K        | 3.8K  | 36.2K         | 34.9K | <b>910</b>  | 76.9K  | <b>13.0</b>                  | 23.0        | 197    | 80/60  | <b>1.1K</b> | 1.4K        | 5.2K        | 2.8K  |
| 10%  | <b>886</b>            | 1.1K       | 3.3K        | 4.8K  | 24.5K         | 40.4K | <b>359</b>  | 48.2K  | <b>12.8</b>                  | 21.9        | 107    | 50/95  | <b>1.2K</b> | 1.9K        | 3.4K        | 2.2K  |
| 20%  | <b>1.2K</b>           | 1.2K       | 2.2K        | 9.7K  | 12.6K         | 14.6K | <b>219</b>  | 48.2K  | <b>13.2</b>                  | 32.3        | 64.3   | 50/80  | <b>1.4K</b> | 1.8K        | 2.3K        | 2.1K  |
| 30%  | <b>1.2K</b>           | 1.5K       | 1.7K        | 14.5K | 6.8K          | 9.8K  | <b>178</b>  | 48.2K  | <b>12.8</b>                  | 41.7        | 48.3   | 50/80  | <b>1.4K</b> | 2.0K        | 1.8K        | 2.1K  |
| 50%  | 2.2K                  | 1.6K       | <b>1.5K</b> | 24.1K | 7.4K          | 4.7K  | <b>154</b>  | 48.2K  | <b>24.0</b>                  | 37.4        | 39.9   | 50/85  | 2.4K        | 1.8K        | <b>1.5K</b> | 2.2K  |
| 80%  | 1.5K                  | 1.5K       | <b>1.4K</b> | 30.9K | 2.5K          | 2.1K  | <b>149</b>  | 38.6K  | 37.1                         | <b>32.1</b> | 38.2   | 40/100 | 1.6K        | 1.6K        | <b>1.4K</b> | 2.0K  |
| 90%  | 1.4K                  | 1.4K       | <b>1.3K</b> | 30.5K | 1.7K          | 1.6K  | <b>138</b>  | 33.9K  | 37.5                         | <b>32.1</b> | 35.3   | 35/120 | 1.4K        | 1.4K        | <b>1.4K</b> | 2.0K  |

**Metrics.** We evaluate system performance using **Query Per Second (QPS) at 95% Recall@10**, a standard setting in related literature [21, 54]. To dissect sources of latency, we report four metrics: the number of Distance Computations, Filter Checks, Graph Hops (for graphs) and Leaves (for ScaNN), Reordering (for ScaNN), and 8KB Page Accesses.

## 6 Evaluations and Analysis

### 6.1 Performance Results

**Trends across datasets**– Figure 9 shows QPS performance for filter selectivities ranging from 1% to 90% (without correlation) across the four datasets described in Section 5. We see a few common trends across the different datasets:

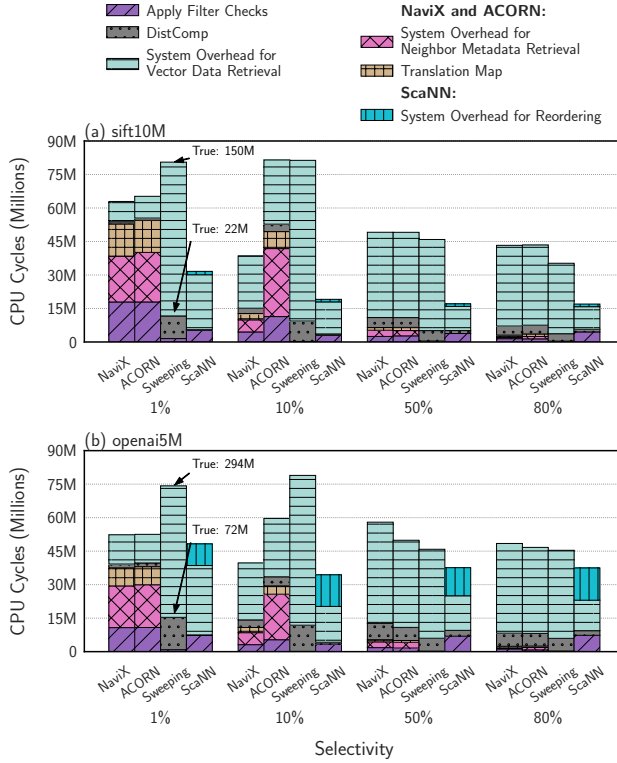


Fig. 10. Latency breakdown (CPU cycles) of a query running for 1, 10, 50, and 80% selectivities on the OpenAI-5M dataset without correlation. Bars (1% sel. for Sweeping) are clipped at the plot’s y-axis limit for readability; annotations “True: X” report the full (unclipped) cycle count.

- (1) Trend 1: The clustering-based approach (ScaNN) performs in general better (2-3x) than the graph approaches on low-dimensional vectors (sift10M and text2image10M). The difference is smaller for high-dimensional vectors (openai5M and cohere10M).
- (2) Trend 2: Filter-first methods (NaviX and ACORN) perform better for low selectivities compared to traversal-first methods (Sweeping and Iterative Scan), which perform better for larger selectivities.
- (3) Trend 3: The NaviX-Directed heuristic consistently outperforms ACORN at low-to-mid selectivities (5%–30%).

These trends highlight that optimal performance is driven by the interplay between algorithmic efficiency and the system-level overheads outlined in Table 1. First, regarding Trend 1, ScaNN’s superiority on low-dimensional vectors is explained by its sequential access pattern, which mitigates the *random memory access amplification* inherent in graph traversals. As dimensionality increases, for ScaNN, the number of vectors fitting on a page drops drastically (e.g., by 10×), which seriously affects the performance for the rescoring part of the ScaNN method that uses the full original dimensions and non-quantized values. Hence, the random access penalty of the graph is less amplified relative to the ScaNN, as even sequential approaches must fetch significantly more pages to retrieve the same number of vectors, causing performance to converge.

Second, regarding Trend 2, the results align with the motivational findings from Figure 1. We observe a distinct crossover point where the overhead of filter-first exploration outweighs its

benefits. Filter-first methods, such as ACORN, prioritize filtering neighbors before scoring them to reduce distance computation costs; however, as selectivity increases, valid neighbors become abundant. At this point, the cost of maintaining the predicate subgraph exceeds the savings, thereby making traversal-first methods more efficient. Within low to mid selectivities where NaviX-Directed heuristic is activated, it outperforms ACORN because the Directed heuristic effectively guides the search towards valid regions at fewer hops (Table 6: 5%-30%) than ACORN requires (Trend 3).

## 6.2 Where Does the Time Go?

To understand the performance characteristics of different FVS methods, we perform a detailed breakdown of query execution time (Figure 10) and internal search metrics (Table 6).

**6.2.1 Search metrics.** (i) For graph-based methods, filter-first approaches (NaviX/ACORN) perform fewer hops than traversal-first (Sweeping/Iterative Scan) to reach the same recall, since NaviX/ACORN can filter and score two-hop neighbors without first hopping to their immediate neighbors (two-hop neighbor expansion). At low selectivity (1%), they perform significantly fewer distance computations (NaviX: 246, ACORN: 238) compared to Sweeping (23.0K), achieving up to 100× reduction through predicate-subgraph navigation. However, this reduction comes at the cost of more filter checks: NaviX and ACORN evaluate 71.8K and 71.6K filters, respectively, compared to Sweeping's 2.6K, as graph traversal opportunistically checks many candidates during navigation. Page access patterns provide an additional direct explanation for performance differences: at 1% selectivity, Sweeping incurs 24.0K page accesses, while NaviX and ACORN maintain only 1.2K through selective navigation. (ii) For ScaNN, it exhibits different behavior due to its cluster-based architecture. At low selectivity, it incurs significant number of filter checks (143.9K at 1%) because it evaluates filters for every page of every opened leaf. As selectivity increases, ScaNN requires fewer leaves to reach the target recall, so filter checks decrease while distance computations increase. For page access, ScaNN achieves 4.5K at 1% selectivity and improves to ~2K at mid selectivities (10–30%), but still larger than the optimal graph-based method (NaviX), at this selectivity range. These internal metrics explain only part of the story. Hops between filter-first and traversal-first methods do not incur the same per-step cost, and metrics are not uniformly better for one method across all selectivities. Moreover, page accesses across methods do not follow the same sequence of steps, either. *This makes these metrics not fully comparable across methods, motivating a more in-depth study of where time is actually spent, which we present next.*

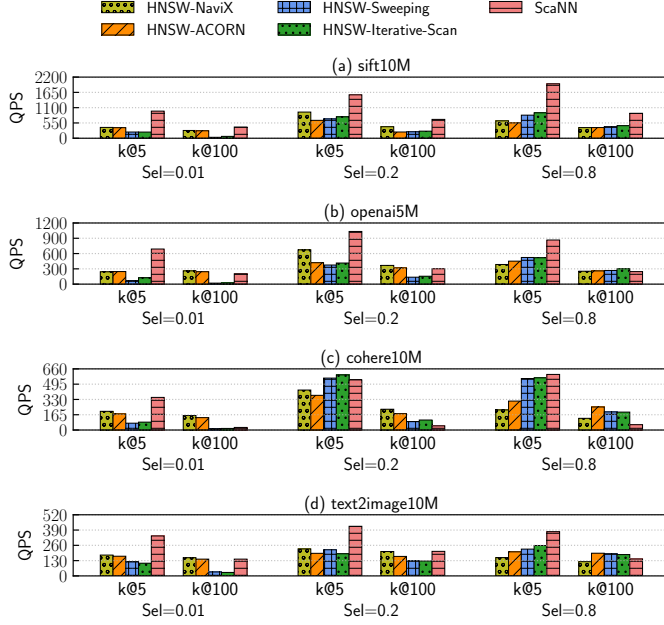
**6.2.2 Where the Time Really Goes.** Figure 10 shows the time breakdown for each of the search methods (Iterative Scan exhibits similar performance to Sweeping, therefore we don't show its breakdown). Each color-coded component in the legend corresponds to a numbered operational step illustrated in Figures 6 and 7.

As described in Section 3.4, each search algorithm is composed of several actions. For **graph-based methods**, these range from operations like accessing an index page to retrieving neighbor's indexTIDs (step ❶ for all graph methods) and accessing a direct neighbor's index page to acquire 2-hop neighbors (step ❷ in filter-first), to accessing the Translation Map (step ❸ in filter-first), applying filter checks (step ❹ in filter-first), and more complex multi-stage process of distance computation and element materialization (step ❺ in filter-first). For **ScaNN**, the operational flow is different. A leaf may logically contain multiple pages. For each page, ScaNN first retrieves all heapTIDs for probing bitmaps (step ❶ for ScaNN), then executes relatively simpler steps: filtering (step ❷ for ScaNN) and distance computation (step ❸ for ScaNN). Unlike graph methods, ScaNN does not require complex traversal steps such as neighbor storage/lookups and maintaining various navigation lists. Our profiling reveals that *the vast majority of query time for all algorithms is spent on low-level system overheads such as page access and data handling operations*. These costs are

not redundant; they are a direct consequence of maintaining the transactional consistency and isolation guarantees of an ACID-compliant database. This presents a fundamental challenge when implementing high-throughput search algorithms within a database storage manager.

Figure 10 shows that at low selectivities, filter-first approaches divide their latency among many more components than traversal-first methods. For example, NaviX and ACORN’s latency at 1% selectivity is distributed across Translation Map access, neighbor metadata retrieval, filter checks, and distance computations. In contrast, Sweeping’s performance is mainly dominated by system overhead for vector data retrieval rather than distance computation itself — at 1% selectivity, OpenAI-5M, vector retrieval consumes over 300M CPU cycles (shown truncated with “True: 300M” annotation), far exceeding the actual computational work. These results signify that although NaviX/ACORN’s overall performance is preferred at low selectivities, it is affected by many system-level factors not present in the HNSWLib library. The proportion by which these factors affect overall performance correlates with filter selectivity, explaining the shift in crossover point between the HNSWLib and PGVector results (Figure 1). For **ScaNN**, one distinct component is the reordering step that requires (i) retrieval of full-precision vectors from the heap (system overhead for reordering) and (ii) sorting by distance to identify the top-k candidates. This is especially significant for OpenAI-5M, where quantization compresses vectors from  $1536 \times 4$  bytes to approximately 193×1 byte ( $\approx 32\times$  compression), requiring nearly one page access ( $8196 / 1536 \times 4 \approx 1$ ) per vector during reordering.

**6.2.3 System Implications: Graph v.s. Tree.** Taking metrics and profiling results together, we observe a phenomenon: ScaNN incurs more filter checks and page accesses at low selectivities, and more distance computations at high selectivities, compared to graph-based methods, yet it often consumes fewer CPU cycles overall. This paradox can be explained by considering the fundamental differences between graph-based and tree-based index structures in a page-based storage system. (i) Memory access patterns: random vs. sequential. Graph methods (HNSW) exhibit **random access patterns** during traversal. Each hop to a neighbor node requires loading different pages distributed across the buffer pool, resulting in poor spatial locality and preventing effective hardware prefetching. In contrast, ScaNN’s tree-based clustering enables **sequential access within partitions**. Once a cluster leaf is selected, ScaNN scans vectors within that partition sequentially across buffer pool pages, benefiting from hardware prefetching and cache line fills. This sequential pattern reduces memory latency despite touching more data. (ii) The “system tax” for complex heuristics. Graph methods impose a **system tax** for their sophisticated navigation heuristics. The Translation Map overhead (5–15M cycles) and neighbor metadata retrieval (15–25M cycles) represent fixed costs of maintaining the HNSW structure at low-mid selectivities. (iii) Hardware utilization. ScaNN’s sequential vector scanning is **highly amenable to SIMD vectorization**. Filter evaluation on cluster members enables batched bitmap probing, and distance computations within a cluster can leverage SIMD instructions for parallel processing. The uniform access pattern (load cluster pages → filter → compute distances) maps naturally to modern CPU vector units. In contrast, graph methods exhibit poor hardware utilization due to irregular access patterns (at page granularity). Each graph hop involves a conditional decision (which neighbor to visit next?), making it difficult to batch operations. Data-dependent control flow further limits vectorization opportunities. This explains why graph methods exhibit higher end-to-end latencies despite fewer distance computations: the work cannot be effectively parallelized. (iv) Index footprint and cache locality. Graph methods require storing the full HNSW structure (vertices and neighbor lists), resulting in substantial memory overhead beyond the raw vectors. The random access pattern means that graph metadata frequently evicts useful data from cache. This is particularly problematic for large-scale indexes where the graph structure cannot fit in the L3 cache, forcing frequent main memory accesses.

Fig. 11. Varying LIMIT  $k$  on all datasets (no correlation).

**Takeaway.** The graph vs. tree tradeoff reveals that *system-level efficiency can offset algorithmic efficiency*. For instance, filter-first methods can achieve fewer distance computations and lower page access – yet ScaNN’s sequential access patterns, SIMD-friendly operations, and cache locality, yield competitive overall throughput. Dataset characteristics (e.g., vector dimensionality) can tip the balance, allowing graph methods to bridge the gap in certain workloads. Therefore, no single structure dominates across all scenarios. *This highlights that any correlation between lightweight libraries and PostgreSQL system performance is, at best, coincidental. We encourage the community to be aware of system-level implications when proposing new methods targeting vector search within database systems.*

### 6.3 Sensitivity to Result Size (LIMIT $k$ )

We evaluate  $k \in \{5, 100\}$  at 95% Recall@ $k$  (Figure 11). When the requested result size increases, the query returns more *filter-valid* neighbors and makes the search procedure perform more work – either by traversing more of the graph (graph methods) or by scanning more leaves (ScaNN).

This growth is modest for filter-first methods (NaviX/ACORN), which traverse the predicate-induced subgraph, but substantial for traversal-first baselines. At 1% selectivity, OpenAI-5M, for example, NaviX’s hop count is changed reasonably when moving from  $k=5$  to  $k=100$  (12.0  $\rightarrow$  22.4, +86%), while Sweeping increases sharply (991.6  $\rightarrow$  6307.6, +536%). In terms of graph hops, while NaviX grows by +106%, Sweeping grows by +599%. ScaNN similarly scales up its effort with  $k$  (leaves scanned: 122  $\rightarrow$  391, +220% at 1% selectivity; +294% on average), which narrows its advantage in the high- $k$ , low-selectivity setting.

### 6.4 Non-Monotonic Performance

An observation from our results (Figure 9) is that the performance of adaptive indices (NaviX/ACORN) is not strictly monotonic with respect to selectivity. This phenomenon is explained by

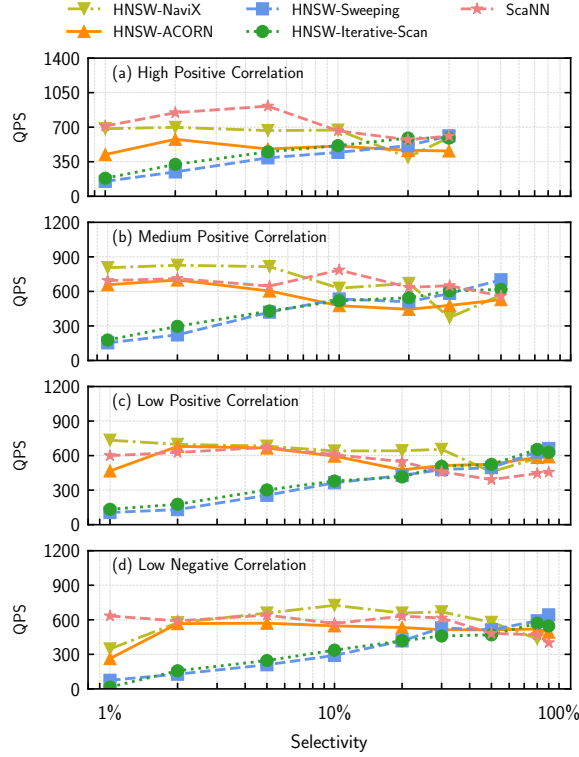


Fig. 12. QPS v.s. Selectivity on OpenAI-5M, showing how vector-predicate correlation alters the performance and crossover point of each FVS algorithm.

the interaction between *adaptive heuristics* and the different *system overheads* of each of these techniques. As detailed in Table 6, the search passes through distinct phases: i) **Aggressive filtering (1–5%)**: Two-hop neighbors *Filter Checks* minimize *Distance Computations*. ii) **Transition to Directed heuristic (NaviX) / Skip some 2-hop (ACORN) (5–30%)**: As selectivity increases, the graph exploration expands (increasing distance computations), but the heuristic has not yet fully switched to a less filter-focused search strategy, causing filter checks to remain high. **Converge to normal graph-traversal ( $\geq 50\%$ )**: The algorithm detects sufficient density and switches to skipping checks or standard 1-hop traversal. Filter checks drop sharply, and the bottleneck shifts entirely to distance computations and system overheads. Figure 10 visualizes these opposing forces: as selectivity increases, filter-related costs shrink, while scoring and system-related costs expand. The same observation can be found in the original NaviX [45] across selectivities.

For ScaNN, we observe a slight QPS drop after 50% selectivity because we execute inline-filtering throughout the search. As selectivity increases, the bitmap sizes grow significantly and can no longer fit in cache easily; thus, we observe increasing overhead for filtering (Figure 10) even as the number of filter checks decreases.

## 6.5 Correlation Effects

Vector-predicate correlation is a query property that significantly alters the search effort needed to achieve the target recall. Figure 12 illustrates the performance of five methods on the OpenAI-5M

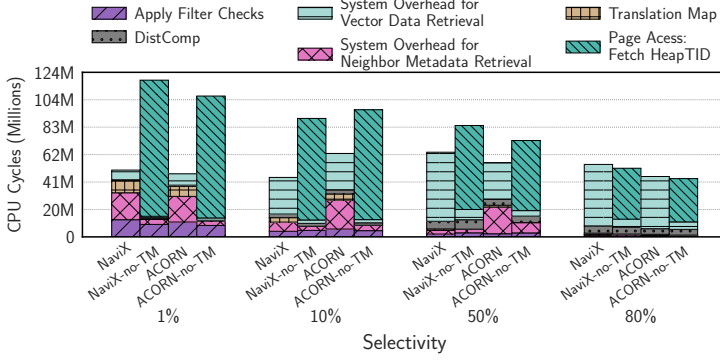


Fig. 13. Latency breakdowns with/without a translation map on the OpenAI-5M dataset (no correlation).

dataset across selectivities for four different correlation settings. For positive correlation workloads (Figure 12 (a)–(c)), where the nearest neighbors to a query vector are more likely to satisfy the predicate, NaviX consistently outperforms ACORN by 1.2–1.7 $\times$ , while traversal-first methods like Sweeping and Iterative Scan improve significantly as selectivity increases, becoming competitive with NaviX at selectivities as low as 10%. ScaNN achieves strong throughput at low selectivities (711–913 QPS under high positive correlation) but degrades at higher selectivities where its filter-then-rank approach incurs overhead from processing more candidates.

Negative correlations in workloads reveal fundamental differences among algorithm architectures. As shown in Figure 12(d), when nearest neighbors fail the predicate, graph-based performance degrades at 1% selectivity: NaviX drops by 53%, ACORN by 44%, and Iterative Scan by 89% relative to the positive correlation. These methods waste resources exploring graph regions where vectors are close to the query but do not satisfy the predicate. In contrast, ScaNN demonstrates remarkable robustness—its throughput increases by 6% under negative correlation, making it the optimal choice for this setting. Because ScaNN’s tree-based partitioning does not rely on graph proximity, it avoids the exploration penalty that graph traversal methods incur. NaviX shows a dip around 50–80% selectivity, where its adaptive heuristic may skip beneficial 2-hop expansion (at 50%) mostly, and there is not enough connectivity for solely looking at one-hop neighbors; at 90%, abundant qualifying 1-hop neighbors make the heuristic efficient again.

Despite its sensitivity to negative correlation at very low selectivity, NaviX recovers to become the best-performing method from 5–50% selectivity even under negative correlation. These results confirm that more adaptive graph-based NaviX provides superior performance across most settings.

## 6.6 Sources of System Overhead

We analyze two DBMS-specific effects behind the “system tax”: ID indirection (via the translation-map ablation) and concurrency.

**6.6.1 Ablation breakdowns of the translation map.** Figure 13 breaks down CPU cycles for NaviX and ACORN with and without the translation map (TM). When TM is disabled, fetching `heaptid` dominates: across 1%–50% selectivity, it accounts for roughly 60%–75% of total cycles. This overhead stems from the DBMS-specific ID indirection in the system (Table 1): 2-hop filtering needs the candidate’s heap tuple identifier (`heaptid`), which is stored on the index page and thus requires page accesses to resolve. Enabling TM replaces these repeated page accesses with an in-memory lookup. We also observe that, with TM enabled, the relative shares of other components (e.g.,

Table 7. Comparison of average CPU cycles breakdown between single- v.s. multiple- threads (1T v.s. 16T) at 10% selectivity on OpenAI-5M with 95% recall@10 (no correlation).

| Method | Threads | Total Cycles | SysOH% | DistComp% | Filter% |
|--------|---------|--------------|--------|-----------|---------|
| NaviX  | 1T      | 24.1M        | 55.9%  | 13.0%     | 12.3%   |
| NaviX  | 16T     | 35.6M (+48%) | 73.5%  | 7.5%      | 9.5%    |
| Sweep. | 1T      | 45.0M        | 81.8%  | 18.6%     | 0.2%    |
| Sweep. | 16T     | 75.7M (+68%) | 91.0%  | 11.9%     | 0.2%    |
| ScaNN  | 1T      | 20.7M        | 84.4%  | 3.1%      | 12.5%   |
| ScaNN  | 16T     | 32.9M (+59%) | 86.9%  | 2.5%      | 10.5%   |

Note: For NaviX, SysOH includes both Vector Data Retrieval and Neighbor Metadata Retrieval. For ScaNN, SysOH includes both Vector Data Retrieval and Reordering (Retrieval Vector Data from Heap). Translation map overhead (8-17%) is omitted from the table for clarity.

vectors/neighbors retrieval) become more visible. This is primarily a cost-shift effect: disabling TM may warm the relevant index pages and benefit subsequent steps via temporal locality, but the metadata-fetch page accesses still dominate end-to-end runtime. Once TM removes this dominant component, the remaining inherent work becomes the new bottleneck. At 80% selectivity, the advantage of TM narrows because the search increasingly relies on 1-hop neighbors and performs substantially fewer 2-hop expansions, reducing the frequency of heap tid resolutions.

**6.6.2 Isolating contention effects.** Our earlier results used 16 concurrent threads. To understand whether contention dominates system overheads and how much concurrent execution (e.g., lock conflicts and cache interference) amplifies these costs, we compare single-thread vs. multi-thread performance. Table 7 demonstrates the statistics for one configuration (10% selectivity, OpenAI-5M, 95%recall@10, no correlation) that exhibits representative behavior. System overheads (buffer reads, tuple access), excluding productive computation (distance calculations, bitmap probings), dominate performance even in single-threaded execution: NaviX (55.9%), Sweeping (81.8%), ScaNN (84.4%) of total CPU cycles. This single-thread baseline establishes that high overhead is inherent to *algorithms' intrinsic architectural costs*, independently of the concurrency contention. Scaling to 16 threads indeed increases total CPU cycles per query (NaviX: +48%, ScaNN: +59%, Sweeping: +68%) as cache sharing and buffer management introduce additional costs. These results highlight that concurrency amplifies the system overhead costs differently for each method, indicating that some methods may benefit from more system optimizations than others.

## 7 Observations and Insights

**1. System Implications Invalidate the Assumptions.** Performance in PostgreSQL is governed by *page-level data retrieval and metadata management* (buffer lookups, TID indirection) rather than distance computation alone. This shift can change library-observed “viability zones” for algorithms: techniques that appear attractive in libraries, because metadata access is a single pointer dereference, can become *memory-bound* in a DBMS, where every neighbor dereference triggers multiple page accesses. Filter-first traversals may translate into higher *system work per visited node*, which can erode or even nullify their algorithmic advantages, making their end-to-end gains in PG negligible without explicit metadata decoupling. Recent work has begun to adopt this decoupling perspective in PostgreSQL [34].

**2. No Clear Winner: Graph vs. Tree.** In systems, neither graph nor clustering indexes dominate universally. Our experiments show: (i) **Dimensionality:** High-dimensional vectors reduce ScaNN’s sequential-scan advantage, narrowing (or eliminating) the gap with HNSW-style graphs. (ii) **Resilience to  $k$ :** Graph-based filter-first methods are more robust as  $k$  increases (Figure 11); while ScaNN must visit more leaves, filter-first methods efficiently identify high-quality qualifying candidates on the predicate subgraph.

**3. Fewer Hops as the Core Design of Graph.** Reducing hops remains the core advantage of graph-based filter-first designs: they can “tunnel” through the predicate-induced subgraph to quickly reach filtered candidates. However, this only translates to end-to-end wins if the hop-reduction heuristic does *not* introduce prohibitive per-hop overhead (e.g., extra metadata lookups). Guided expansion (e.g., NaviX-Directed) is the key lesson: it preserves the “fewer hops” benefit, thus reducing wasted exploration.

**4. A Call for System-Aware Algorithm Design.** Our study proposed that many FVS ideas that look strong in library settings can degrade in production DBMSs unless they explicitly account for system overheads. The actionable direction is not “new optimizations in isolation,” but *co-design*: algorithms that (i) expose and minimize system taxes (metadata layout and page access patterns), and (ii) adapt online across selectivity/correlation and query hardness.

## 8 Related Work

**Algorithms for Filtered Vector Search.** There have been multiple efforts to integrate filters on graph indexes [13, 15, 19, 23, 27, 32, 33, 40, 44, 52, 56, 57, 59]. Such approaches could be classified into filter-aware and filter-agnostic. The former integrates the filter attributes as part of the index building process, evaluating filter information to decide which connections to build, given the common filters among points [23] or by modifying the distance metric [52] to incorporate filter similarity. The latter family of mechanisms (also called inline filtering methods) is unaware of the filter attribute columns and operations at build time [40, 44, 45]. While they still utilize an index for searching, they combine applying the filters as they traverse the index. While pre- and post-filtering approaches are technically also filter-agnostic, they remain complementary to the inline set of solutions. In this work, we focus on filter-agnostic mechanisms since we are interested in serving a wide variety of use cases that cannot assume the filter attribute columns or operations are known or can be used at build time. See [39] for more details on the diversity of filtered vector search methods.

**Vector Data Management Systems.** The growing interest in the applications of vector search in recent years has led to an increasing need for systematic approaches in vector data processing. Such approaches are either added as additional functionalities on top of classic database systems [7, 10, 37, 49, 60], or have led to the emergence of standalone, vector-native database systems [13, 43, 51]. In fact, some works already study the overheads of such general systems when adding vector search features [60], although they study vector search in isolation, without filters. Production-level database systems that support vector search operations enable vectors to coexist in tables alongside structural data, which can serve as filters in Filtered Vector Search processes. In such systems, filter-agnostic vector search algorithms provide the most viable solutions, as the continuous addition of new data leaves no room for assumptions about the data that might serve as filters.

**Filtered Vector Search Evaluations.** Current works evaluate different kinds of filtered vector search algorithms using them as standalone libraries, running in isolation from other system-related aspects that might greatly affect their performance in a real-world production-level database system [15, 30, 40, 45, 58]. At the same time, public information about the performance of those algorithms in production-level database systems usually provides limited results and insights [2, 11–13]. To the best of our knowledge, our work unleashes novel critical performance insights in a

complex and general database system, with a breakdown of the manifold sources of performance cost. In addition, our evaluation is the first one to deep-dive into various costs in a real system, together with the implications of filter selectivities and correlations, all combined, which determines the optimal search algorithm for each case.

## 9 Future Work and Conclusions

Our benchmark evaluates filter-agnostic FVS algorithms across diverse selectivities and correlations within a PostgreSQL database system. We show that the system-level overheads create important performance trade-offs. These findings provide a cost-based framework for practitioners and highlight the necessity for future research to shift from isolated library evaluations to the complexities of system-integrated algorithm design. There is a clear need for coordinated innovation in algorithms and databases to deliver the best of both worlds: matching the performance of best-in-class in-memory vector search algorithms when RAM is plentiful, while leveraging the transactional consistency and reliability of a database for out-of-core datasets. A promising approach in this direction is the use of columnar engines to host both the index as well as the columns that participate in filtered vector search. Columnar engines have shown their ability to drastically reduce the page access overhead and other system overheads while maintaining transactional consistency. More broadly, it is important that databases combine top main memory performance (when there is enough memory) with ACID properties and the ability to operate from secondary storage as well.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive feedback.

## References

- [1] 2022. Cohere Wikipedia Embeddings Dataset. <https://huggingface.co/datasets/Cohere/wikipedia-22-12/tree/main/en>.
- [2] 2024. Pinecone vs. Postgres pgvector: For vector search, easy isn't so easy. <https://www.pinecone.io/blog/pinecone-vs-pgvector/>.
- [3] 2024. wiki-ann. Hugging Face. <https://huggingface.co/2024anonymous/wiki-ann>
- [4] 2025. hnswlib. <https://github.com/nmslib/hnswlib>.
- [5] 2025. kNN search in Elasticsearch. Elastic Docs. <https://www.elastic.co/docs/solutions/search/vector/knn>
- [6] 2025. OpenAI5M. <https://huggingface.co/datasets/allenai/c4>.
- [7] 2025. pgvector: Open-source vector similarity search for Postgres. GitHub repository. <https://github.com/pgvector/pgvector>
- [8] 2025. Run Vector Search Queries (Atlas Vector Search stage). MongoDB Documentation. <https://www.mongodb.com/docs/atlas/atlas-vector-search/vector-search-stage/>
- [9] 2025. ScaNN. [github.com/google-research/google-research/tree/master/scann](https://github.com/google-research/google-research/tree/master/scann).
- [10] 2025. ScaNN for AlloyDB. [https://services.google.com/fh/files/misc/scann\\_for\\_alloydb\\_whitepaper.pdf](https://services.google.com/fh/files/misc/scann_for_alloydb_whitepaper.pdf).
- [11] 2025. Supercharging vector search performance and relevance with pgvector 0.8.0 on Amazon Aurora PostgreSQL. <https://aws.amazon.com/blogs/database/supercharging-vector-search-performance-and-relevance-with-pgvector-0-8-0-on-amazon-aurora-postgresql/>.
- [12] 2025. Vector Search in the Real World: How to Filter Efficiently Without Killing Recall. <https://milvus.io/blog/how-to-filter-efficiently-without-killing-recall.md>.
- [13] 2025. Weviate. <https://weaviate.io/blog/speed-up-filtered-vector-search>.
- [14] Google AI. 2023. Gemini: A Large Language Model. <https://geminilang.google>
- [15] 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. *Proc. ACM Manag. Data* (2025).
- [16] Laurent Amsaleg, Oussama Chelly, Teddy Furon, Stéphane Girard, Michael E. Houle, Ken-ichi Kawarabayashi, and Michael Nett. 2015. Estimating Local Intrinsic Dimensionality. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 29–38.
- [17] Martin Aumüller and Matteo Ceccarello. 2021. The role of local dimensionality measures in benchmarking nearest neighbor search. *Information Systems* 101 (2021), 101807.

- [18] Ilias Azizi, Karima Echihiabi, and Themis Palpanas. 2025. Graph-based vector search: An experimental evaluation of the state-of-the-art. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–31.
- [19] Yuzheng Cai, Jiayang Shi, Yizhuo Chen, and Weiguo Zheng. 2024. Navigating Labels and Vectors: A Unified Approach to Filtered Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data (SIGMOD)* 2, 6 (2024), 1–27.
- [20] Matteo Ceccarello, Alexandra Levchenko, Ioana Ileana, and Themis Palpanas. 2025. Evaluating and generating query workloads for high dimensional vector similarity search. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 5299–5310.
- [21] Manos Chatzakis, Yannis Papakonstantinou, and Themis Palpanas. 2025. DARTH: Declarative Recall Through Early Termination for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 4 (2025), 1–26.
- [22] Karima Echihiabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2020. Return of the lernaean hydra: Experimental evaluation of data series approximate similarity search. *arXiv preprint arXiv:2006.11459* (2020).
- [23] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. 2023. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters (*WWW '23*).
- [24] Junfeng He, Sanjiv Kumar, and Shih-Fu Chang. 2012. On the difficulty of nearest neighbor search. *arXiv preprint arXiv:1206.6411* (2012).
- [25] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2553–2561.
- [26] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 861–864.
- [27] Mengxu Jiang, Zhi Yang, Fangyuan Zhang, Guanhao Hou, Jieming Shi, Wenchao Zhou, Feifei Li, and Sibao Wang. 2025. DIGRA: A Dynamic Graph Indexing for Approximate Nearest Neighbor Search with Range Filter. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.
- [28] Jonathan Katz. 2024. Scalar and Binary Quantization for pgvector Vector Search and Storage. <https://jkatz.github.io/post/postgres/pgvector-scalar-binary-quantization/>. Accessed: 2026-01-27.
- [29] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
- [30] Mocheng Li, Xiao Yan, Baotong Lu, Yue Zhang, James Cheng, and Chenhao Ma. 2025. Attribute Filtering in Approximate Nearest Neighbor Search: An In-depth Experimental Study. *arXiv:2508.16263 [cs.DB]*
- [31] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.
- [32] Zhaocheng Li, Silu Huang, Wei Ding, Yongjoo Park, and Jianjun Chen. 2025. SIEVE: Effective Filtered Vector Search with Collection of Indexes. *arXiv preprint arXiv:2507.11907* (2025).
- [33] Anqi Liang, Pengcheng Zhang, Bin Yao, Zhongpu Chen, Yitong Song, and Guangxu Cheng. 2025. UNIFY: Unified Index for Range Filtered Approximate Nearest Neighbors Search. *Proceedings of the VLDB Endowment* 18, 4 (2025), 1118–1130.
- [34] Jiayi Liu, Yunan Zhang, Chenzhe Jin, Aditya Gupta, Shige Liu, and Jianguo Wang. 2026. Fast Vector Search in PostgreSQL: A Decoupled Approach. In *Proceedings of the 16th Annual Conference on Innovative Data Systems Research (CIDR '26)*. Chaminade, USA. <https://www.cidrdb.org/papers/2026/p2-liu.pdf>
- [35] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* (2020).
- [36] OpenAI. 2024. ChatGPT (November 2024 version). <https://openai.com>
- [37] Oracle Corporation. 2025. Oracle AI Vector Search. <https://www.oracle.com/database/ai-vector-search/>.
- [38] Oracle Corporation. 2025. Oracle Vector Search Manual. <https://docs.oracle.com/en/database/oracle/oracle-database/26/vecse/ai-vector-search-users-guide.pdf>.
- [39] Yannis Papakonstantinou, Anastasia Ailamaki, Yannis Chronis, Helena Caminal, and Fatma Ozcan. 2025. Filtered Vector Search: State-of-the-art and Research Opportunities.
- [40] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proc. ACM Manag. Data* (2024).
- [41] Zhencan Peng, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2025. Dynamic Range-Filtering Approximate Nearest Neighbor Search. *Proceedings of the VLDB Endowment* 18, 10 (2025), 3256–3268. doi:10.14778/3748191.3748193

- [42] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.
- [43] Pinecone. 2025. Pinecone: The Vector Database for AI Search and Retrieval. <https://www.pinecone.io/>.
- [44] Abdel Rodriguez, John Trengrove, and Joon-Pil (JP) Hwang. 2024. How we speed up filtered vector search with ACORN. <https://weaviate.io/blog/speed-up-filtered-vector-search>. Accessed: 2024-11-19.
- [45] Gaurav Sehgal and Semih Salihoğlu. 2025. NaviX: A Native Vector Index Design for Graph DBMSs With Robust Predicate-Agnostic Search Performance. *Proc. VLDB Endow.* (2025).
- [46] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, et al. 2022. Results of the NeurIPS’21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR, 177–189.
- [47] 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.
- [48] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [49] Nitish Upreti, Harsha Vardhan Simhadri, Hari Sudan Sundar, Krishnan Sundaram, Samer Boshra, Balachandar Perumal-swamy, Shivam Atri, Martin Chisholm, Revti Raman Singh, Greg Yang, Tamara Hass, Nitesh Dudhey, Subramanyam Pattipaka, Mark Hildebrand, Magdalen Manohar, Jack Moffitt, Haiyang Xu, Naren Datha, Suryansh Gupta, Ravishankar Krishnaswamy, Prashant Gupta, Abhishek Sahu, Hemeswari Varada, Sudhanshu Barthwal, Ritika Mor, James Codella, Shaun Cooper, Kevin Pilch, Simon Moreno, Aayush Kataria, Santosh Kulkarni, Neil Deshpande, Amar Sagare, Dinesh Billa, Zishan Fu, and Vipul Vishal. 2025. Cost-Effective, Low Latency Vector Search with Azure Cosmos DB. (2025).
- [50] Sairaj Voruganti and M Tamer Özsu. 2025. MIRAGE-ANNS: Mixed Approach Graph-based Indexing for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–27.
- [51] 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 (*SIGMOD* ’21).
- [52] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongang Ni. 2023. An efficient and robust framework for approximate nearest neighbor search with attribute constraint (*NIPS* ’23).
- [53] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph-and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *IEEE Data Eng. Bull.* 46, 3 (2023), 3–21.
- [54] Zeyu Wang, Qitong Wang, Xiaoxing Cheng, Peng Wang, Themis Palpanas, and Wei Wang. 2024. Steiner-hardness: A query hardness measure for graph-based ann indexes. *Proceedings of the VLDB Endowment* 17, 13 (2024), 4668–4682.
- [55] Jiuqi Wei, Xiaodong Lee, Zhenyu Liao, Themis Palpanas, and Botao Peng. 2025. Subspace collision: an efficient and accurate framework for high-dimensional approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–29.
- [56] Yuexuan Xu, Jianyang Gao, Yutong Gou, Cheng Long, and Christian S. Jensen. 2024. iRangeGraph: Improvising Range-dedicated Graphs for Range-filtering Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–26.
- [57] Ziqi Yin, Jianyang Gao, Pasquale Balsebre, Gao Cong, and Cheng Long. 2025. DEG: Efficient Hybrid Vector Search Using the Dynamic Edge Navigation Graph. *Proceedings of the ACM on Management of Data* 3 (2025), 1 – 28. <https://api.semanticscholar.org/CorpusID:279379684>
- [58] Chaoqun Zhan, Mengzhao Wang, Lingwei Lv, Yitong Geng, Bin Wu, and Sheng Wang. 2025. BigVectorBench: Heterogeneous Data Embedding and Compound Queries are Essential in Evaluating Vector Databases. *Proceedings of the VLDB Endowment* 18, 5 (2025), 1536–1549.
- [59] Fangyuan Zhang, Mengxu Jiang, Guanhai Hou, Jieming Shi, Hua Fan, Wenchao Zhou, Feifei Li, and Sibao Wang. 2025. Efficient Dynamic Indexing for Range Filtered Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* (2025).
- [60] Yunan Zhang, Shige Liu, and Jianguo Wang. 2024. Are There Fundamental Limitations in Supporting Vector Data Management in Relational Databases? A Case Study of PostgreSQL. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*.
- [61] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: segment graph for range-filtering approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

Received October 2025; revised January 2026; accepted February 2026