

VecFlow: A High-Performance Vector Data Management System for Filtered-Search on GPUs

JINGYI XI^{*†} and CHENGHAO MO^{*}, SSAIL Lab, UIUC, USA

BEN KARSIN, Nvidia, USA

ARTEM CHIRKIN, Nvidia, Switzerland

MINGQIN LI, Microsoft, USA

MINJIA ZHANG, SSAIL Lab, UIUC, USA

Vector search and database systems have become a keystone component in many AI applications. While many prior research has investigated how to accelerate the performance of generic vector search, emerging AI applications require running more sophisticated vector queries efficiently, such as vector search with attribute filters. Unfortunately, recent filtered-ANNS solutions are primarily designed for CPUs, with few exploration and limited performance of filtered-ANNS that take advantage of the massive parallelism offered by GPUs. In this paper, we present VecFlow, a novel high-performance vector filtered search system that achieves unprecedented high throughput and recall while obtaining low latency for filtered-ANNS on GPUs. We propose a novel label-centric indexing and search algorithm that significantly improves the selectivity of ANNS with filters. In addition to algorithmic level optimization, we provide architecture-aware optimizations for VecFlow's functional modules, effectively supporting both small batch and large batch queries, and single-label and multi-label query processing. Experimental results on NVIDIA A100 GPU over several public available datasets validate that VecFlow achieves 5 million QPS for recall 90%, outperforming state-of-the-art CPU-based solutions such as Filtered-DiskANN by up to 135 times. Alternatively, VecFlow can easily extend its support to high recall 99% regime, whereas strong GPU-based baselines plateau at around 80% recall.

CCS Concepts: • **Information systems** → **Data management systems**; **Semi-structured data**.

Additional Key Words and Phrases: Vector Database, Filtered ANNS, GPUs

ACM Reference Format:

Jingyi Xi, Chenghao Mo, Ben Karsin, Artem Chirkin, Mingqin Li, and Minjia Zhang. 2025. VecFlow: A High-Performance Vector Data Management System for Filtered-Search on GPUs. *Proc. ACM Manag. Data* 3, 4 (SIGMOD), Article 271 (September 2025), 27 pages. <https://doi.org/10.1145/3749189>

1 Introduction

Dense vectors have become a key form of data representation in the era of AI. Deep learning models encode various entities, such as text [38, 41], images [36, 50], and code snippets [33], into dense continuous vectors, enabling a growing number of vector-based online applications, including image search [31], web search [27, 66], question and answering [64], ad-hoc retrieval [14, 25, 43], mobile search [3], and product search [58]. More recently, vector search has also become an integral component for building generative AI systems, such as retrieval-augmented generation for large

^{*}Both authors contributed equally to this research.

[†]Work done while intern at UIUC.

Authors' Contact Information: Jingyi Xi; Chenghao Mo, SSAIL Lab, UIUC, USA, flotherxi@gmail.com, cmo8@illinois.edu; Ben Karsin, Nvidia, USA, bkarsin@nvidia.com; Artem Chirkin, Nvidia, Switzerland, achirkin@nvidia.com; Mingqin Li, Microsoft, USA, mingqli@microsoft.com; Minjia Zhang, SSAIL Lab, UIUC, USA, minjiaz@illinois.edu.



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

© 2025 Copyright held by the owner/author(s).

ACM 2836-6573/2025/9-ART271

<https://doi.org/10.1145/3749189>

language model (LLM)-based applications [35, 39]. To provide an interactive user experience at scale, these applications require high-performance deployment to achieve high throughput and recall with low query latency. As a result, there have been numerous studies on optimizing vector search performance by developing compute and memory-efficient approximate nearest neighbor search (ANNS) algorithms [5, 7, 18, 20, 21, 30, 32, 40, 44, 45].

Despite many advancements, prior work often focuses on optimizing *unconstrained top-K retrieval*, where the algorithm searches a vector dataset for the K closest vectors based purely on their geometric distance from the query. However, real-world and emerging applications often require finding Top-K results under certain *filtering* or *label* constraints. For example, Google Multisearch [23] allows users to search images with additional text hints, advertising engine apply filters to display region-relevant ads, and enterprise search places access control to displayable documents based on users' permission levels.

One straightforward approach to enable ANNS with filters is to combine ANNS with *post-processing*, where one first runs a standard vector search over the entire dataset to obtain a list of candidates and then applies a filter operator on the given label to select the top-K elements with the given filter. This approach allows one to leverage existing high-performance solutions developed for unconstrained top-K ANNS. However, it is very challenging to predict exactly how many returned candidates will pass the filter operation. Therefore, the number of final candidates that match the query's labels can be much less than K, or in the worst-case scenario, none at all.

In practice, post-processing methods perform poorly. As such, academia and industry companies such as Milvus [42], Weaviate [61] and Pinecone [49] that offer ANNS-as-a-service have explored alternative methods to support ANNS with filters. Alternatively, Filtered-DiskANN [22] proposes building a graph index with enhanced navigability for filtered searches and achieves state-of-the-art results with tens of thousands of query-per-second (QPS) at >0.9 recall. One gap in this approach is that Filtered-DiskANN [22] assume that a query only has one label while, in practice, queries can contain multiple labels, involving *OR* and *AND* operations. This is an important use case, so new filtered ANNS approaches should consider multiple query labels in their design.

ANNS index construction and search are computationally expensive operations, so modern GPU architectures offer an opportunity to significantly improve performance. Researchers have developed several GPU-based vector search algorithms for unconstrained top-K ANNS [46, 57, 68]. The state-of-the-art GPU-based vector search system from NVidia, CAGRA [46], builds GPU-friendly graph indices to accelerate the vector search on GPUs, offering over an order of magnitude higher QPS than CPU-based solutions. Despite achieving high ANNS speed on GPUs, few studies have explored high-performance design and implementations optimized for GPU-based filtered ANNS, due to several challenges. First, the hardware architecture of GPUs is very different from CPU. While the GPU has a much higher parallelism and memory bandwidth than CPU, one has to redesign the parallelism strategy for an algorithm to map it to the GPU architecture to extract sufficient parallelism to fully utilize the hardware resources. Second, directly applying filter-aware partitioning algorithms leads to huge memory redundancy, yet GPU global memory is a scarce resource. Therefore, it is important to optimize the memory efficiency of GPU-based filtered-ANNS. Third, existing GPU-based ANNS algorithms primarily focus on maximizing throughput when the batch size is large. However, in online serving scenarios, queries are often coming in through small batch sizes, making it difficult to achieve high GPU utilization while maintaining a low query latency.

In this paper, we investigate the following research question: *How to build a high-performance system for filtered ANNS that can effectively leverage GPU hardware?* To address the aforementioned

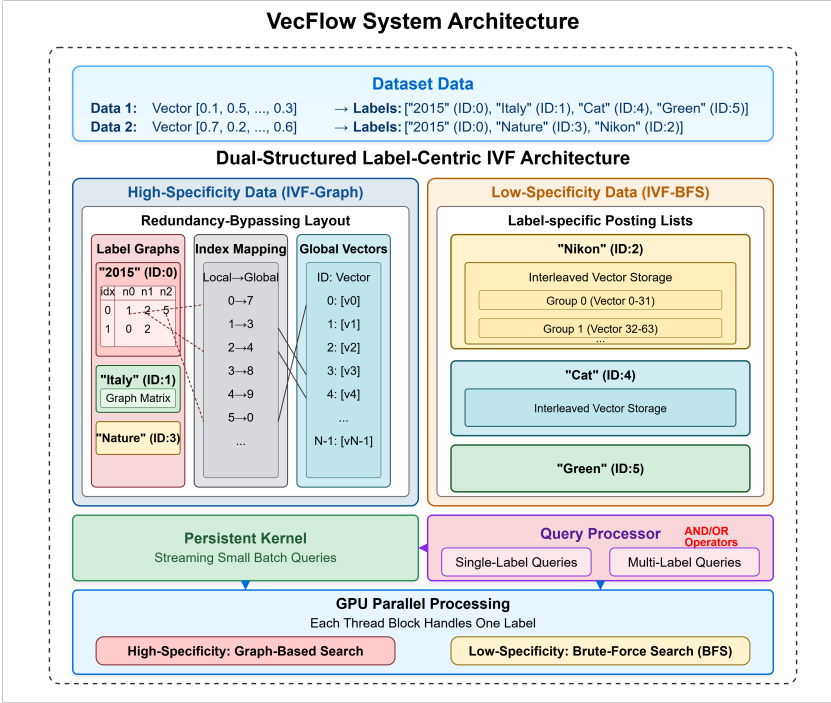


Fig. 1. System overview of VecFlow. The architecture shows how vectors with labels are managed through VecFlow's Dual-Structured Label-Centric IVF design. High-specificity data are handled via graph-based indexing with a redundancy-bypassing layout (IVF-Graph), while low-specificity data use an optimized brute-force index (IVF-BFS). The system includes GPU-tailored optimizations such as GPU-centric parallel search, persistent kernels for streaming search of small batch sizes, supports for both efficient single-label and multi-label queries with AND/OR logic.

challenges, we present VecFlow¹, a high-performance vector filtered-search system for GPUs. Fig. 1 shows the overview of VecFlow's design. Different from recent graph-based solutions [22, 46], VecFlow adopts a label-centric IVF (inverted file indexing) design to improve the selectivity of ANNS with filters, where data points are grouped based on shared labels instead of spatial proximity or distance. VecFlow partitions the IVF posting lists into high-specificity and low-specificity groups, each characterized by distinct label distribution patterns. § 4.1.1 provides detailed definition of specificity, which is a measure of how uniquely a label identifies data points. We design a dual-structured index along with tailored search algorithms for the two groups, maximizing GPU utilization to enable fast and accurate filtered-ANNS (§ 4.1). In addition, VecFlow designs high-performance GPU kernels to achieve (1) redundancy-bypassing IVF-Graph (§ 4.2.1) that reduces memory usage by avoiding redundant vector replication, (2) interleaved-scan-based IVF-BFS (§ 4.2.2) that maximizes GPU memory bandwidth efficiency for brute-force distance computations, and (3) persistent kernels for streaming small batches (§ 4.2.3) that eliminate kernel launch overhead and maintain high GPU utilization. Together, these optimizations deliver unprecedented performance for filtered ANNS. Finally, unlike prior work that assumes each query only has one label, VecFlow introduces a high-performance algorithm to efficiently tackle queries with multiple labels with OR and AND conditions (§ 4.3).

¹Source code for VecFlow is available at <https://github.com/Supercomputing-System-AI-Lab/VecFlow>

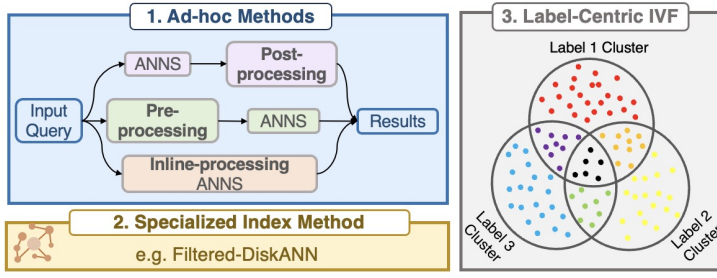


Fig. 2. Three common approaches to enable filtered-ANNS.

Our experiments show that VecFlow achieves significantly speedups over state-of-the-art filtered-ANNS solutions on both CPU and GPU over both real-world and semi-synthetic filtered search datasets (§ 5). In particular, VecFlow achieves 5 million QPS for recall 90%, outperforming state-of-the-art CPU-based solutions such as Filtered-DiskANN by up to 135 times. VecFlow can easily extend its support to reach the high recall regime ($>99\%$), whereas strong GPU-based baselines plateau at around 80% recall for these use cases.

2 Background

2.1 Filtered ANNS

As one of the core components for data management, there are numerous algorithms for ANNS with diverse index construction methods and a range of trade-offs concerning indexing time, search time, and recall [5, 7, 18, 20, 21, 30, 32, 40, 44, 45]. Several work have done an excellent survey and comparison of ANNS algorithms [4, 37, 47, 60]. However, most existing research addresses the unconstrained ANNS problem from the perspective of improving search efficiency [18, 40], reducing the memory and compute costs [30, 51], enabling fresh updates to the index [55]. With the increasingly central role of ANNS in dense retrievers, new capabilities are required for ANNS, including *search with filters* (filtered-ANNS).

Filtered-ANNS extends ANNS by allowing users to search for similar vectors that also satisfy specific metadata constraints, combining both vector similarity and attribute-based filtering to deliver more precise and contextually relevant results. In filtered-ANNS, each data point contains both a vector embedding and metadata labels. As shown in Fig. 1, the data point (a vector, e.g., $[0.1, 0.5, \dots, 0.3]$) is associated with a list of labels (e.g., $["2015" (ID:0), "Italy" (ID:1), "Cat" (ID:4), "Green" (ID:5)]$). During search, the system combines vector similarity with label constraints, e.g., "Find images visually similar to this reference image, but only consider those taken in 'Italy' in '2015'." Here, vector similarity determines visual resemblance, while labels "2015" and "Italy" constrain the search space to return only results matching both labels.

There has been several recent works on filtered-ANNS, such as vbase [67], NHQ [59], IVF² [6], and Filtered-DiskANN [22]. We categorize existing approaches to filtered-ANNS into three major types, as illustrated in Fig. 2. Ad-hoc methods integrate filtering through pre-/post-processing or inline filtering on top of existing ANNS systems. For example, FAISS-IVF [16] supports inline filtering by associating metadata with index entries to skip mismatched points during traversal. Weaviate [61] and Milvus [42] support inline-processing by maintaining inverted file indexing (IVF) where each label constraint has an "approved list" containing data IDs that fulfill the label constraints. Systems like pgvector [1], PASE [63], Analytic DB-V [62] and SingleStore-V [10] use similar mechanism on top of general-purpose ANNS and build unified SQL and vector search backends. However, all ad-hoc methods just modify the search process, do not modify the index building.

Filtered-DiskANN [22] builds a specialized filtered-ANNS index. It achieves excellent results by building filtered search on top of Vamana graph-based ANNS solution DiskANN [29]. In particular, Filtered-DiskANN builds separate proximity graph for data points associated to each label and stitches multiple small graphs together to eliminate redundant edges. By leveraging the geometric relation between points and incorporating label information during index construction, Filtered-DiskANN achieves much better search efficiency than prior filtered-ANNS methods based on IVF/LSH indices. More recently, IVF² introduces a label-centric IVF-based (inverted file indexing) method that wins the NeurIPS'23 Big-ANN Competition Filter Track[9]. Unlike these existing filtered-ANNS solutions, ours is the first to explore achieving optimal filtered-ANN performance on GPUs. Furthermore, different from prior work that only considers simple filters, such as exact matches with one filter in Filtered-DiskANN, we address more complex filtering conditions, including efficient search with the disjunction (OR) and conjunction (AND) of multiple filters, which have been less explored in previous research.

2.2 GPU-based Vector Search

The high computational throughput of modern GPU architectures has made it a key resource to solve computationally difficult problems [26]. In recent years, increasingly computationally difficult operations, such as ANNS, have become commonplace in database systems, necessitating hardware acceleration [17, 52]. Early work to efficiently accelerate ANNS using GPUs focused on developing fast k-selection algorithms based on product quantization [12, 31]. Subsequently, several studies proposed to build graph-based search algorithm on GPUs [24, 57, 65, 68, 69]. For example, GGNN builds hierarchical kNN graphs on GPUs [24]. BANG [57] employs a heterogeneous CPU-GPU vector search solution, using compressed data for distance computations on GPUs while maintaining the graph and actual data points on the CPU. Although these methods achieve promising speedups, they primarily focus on adapting or optimizing GPU resource utilization over existing CPU-designed graphs, without fully leveraging the GPU's capabilities. More recently, Ootomo et al. [46] proposed CAGRA, an enhanced GPU algorithm for solving k-ANNS. CAGRA designs hardware-friendly proximity graph structures to fully leverage the compute and memory bandwidth of modern GPU, outperforming both well-known CPU-based ANNS such as HNSW [40] and state-of-the-art GPU-based ANNS, such as GGNN [24] and GANNS [65], achieving the state-of-the-art results for k-ANNS. Different from those work, VecFlow focuses on solving the filtered-ANNS problem on GPUs. In addition, although VecFlow leverages CAGRA graphs, it is fundamentally different. First, CAGRA can only perform k-ANNS without filters. Although CAGRA can be combined with post-processing to support filtered-ANNS, the performance is quite sub-optimal. In contrast, VecFlow's novel index and search algorithms are specifically designed to enable significantly higher QPS and recall on filtered-ANNS. Second, while CAGRA achieves high throughput with very large batch sizes, VecFlow introduces a persistent kernel-based method to achieve filtered-ANNS high throughput in streaming scenarios with small incoming query batch sizes.

3 Problem Formulation

DEFINITION 1 (FILTERED NEAREST NEIGHBOR SEARCH (FILTERED-NNS)). *Given a finite data point set X of N points in a vector space $\mathcal{R}^D$, each data point (a.k.a. vector) $x \in X$ has an associated set of labels $L_x \subseteq \mathcal{L}$, where $\mathcal{L}$ is a finite set of labels. The goal of filtered-NNS is to answer a given query point $q \in \mathcal{R}^D$ with labels $L_q \subseteq \mathcal{L}$ by finding the closest point $x \in X$ with L_q , i.e., points in X that have the label L_q associated with them.*

Note that the query point q may or may not be in the point set X , and the above definition generalizes naturally to *Top-K Filtered-NNS*, i.e., finding $A_{topk} = \{x \mid \text{top-K nearest points to } q \text{ in } X\}$.

Given that N is often quite large and the service level objective (SLO) is stringent, we define an approximated form of the above problem.

DEFINITION 2 (FILTERED APPROXIMATED NEAREST NEIGHBOR SEARCH (FILTERED-ANNS)). *The goal of ϵ -filtered-NNS is to design an algorithm that answers a given query point $q \in \mathcal{R}^D$ with labels $L_q \subseteq \mathcal{L}$ by maximizing the recall of finding the top- k closest points $A_{topk} \in X$ with L_q while minimizing the search time spent to return A_{topk} .*

Assume the ground truth top- K nearest neighbors of q is GT_{topK} , the *recall* is defined as follows:

$$Recall = \frac{|A_{topK} \cap GT_{topK}|}{|GT_{topK}|} = \frac{|A_{topK} \cap GT_{topK}|}{K} \quad (1)$$

4 Design and Implementation of VecFlow

As described in § 1, the core of VecFlow is the Dual-Structured Label-Centric IVF index and search algorithm, which processes data differently based on label specificity. This design enables VecFlow to flexibly and efficiently handle the full spectrum of label distributions. The remainder of this section details the key design components of VecFlow: § 4.1 presents the dual-structured IVF approach and motivates the separation of high- and low-specificity labels. § 4.2 describes our GPU-optimized indexing and search kernel design, including redundancy-bypassing data layout for IVF-Graph, interleaved memory organization for IVF-BFS and techniques for streaming small-batch queries via persistent kernels. § 4.3 extends the design to multi-label queries with support for both AND and OR operations utilizing parallel GPU search.

4.1 Dual-Structured Label-Centric IVF

We start by considering two popular approaches for ANNS with filters, namely the graph traversal-based approach such as Filtered-DiskANN [22], and the clustering-based method employed in Milvus [42] and Weaviate [61], which relies on inverted file indexing (IVF) [34]. Filtered-DiskANN proposes to exploit both geometric relationship between data points and the labels that each point has to construct a navigable graph index and perform graph traversal with inline-filtering, which achieves excellent performance in Filtered-ANNS and outperforms filtered-ANNS methods based on IVF/LSH indices on CPU.

Despite its promising results, Filtered-DiskANN uses a single graph index for all points, meaning that multiple queries with different labels must use the same method to search this single graph index. When the number of data points with associated labels is high, the single index approach performs similarly to unfiltered search. However, we observe that the label distribution in many datasets exhibits a *long tail* distribution, with a few frequent labels and many rare ones, as shown in Figure 3. Challenges arise when data points with certain rare labels fall into the long tail, as most distance computations in the index search process become unnecessary, negatively impacting query performance. In addition, it is difficult to ignore vectors that do not pass the filter during graph traversal, as this can break the connectivity that the graph-based index relies on to achieve high recall. When the graph is large and the frequency of data points corresponding to a given label is very low, the search performance of Filtered-DiskANN becomes problematically low. As we later show in the evaluation, FilteredVamana achieves a maximum recall of only 50%, while StitchedVamana reaches at most just 70% on the YFCC dataset. Both methods struggle to reach high recall because YFCC has many rare labels, making it challenging to predict how many top- K returned candidates will include those rare labels. Consequently, the number of final candidates that match the query's labels is less than K . Moreover, a large graph degree is required to maintain search quality (96 for FilteredVamana and 64 for StitchedVamana, as reported in [22]). This further impacts search efficiency, as a higher graph degree results in slower search performance. This

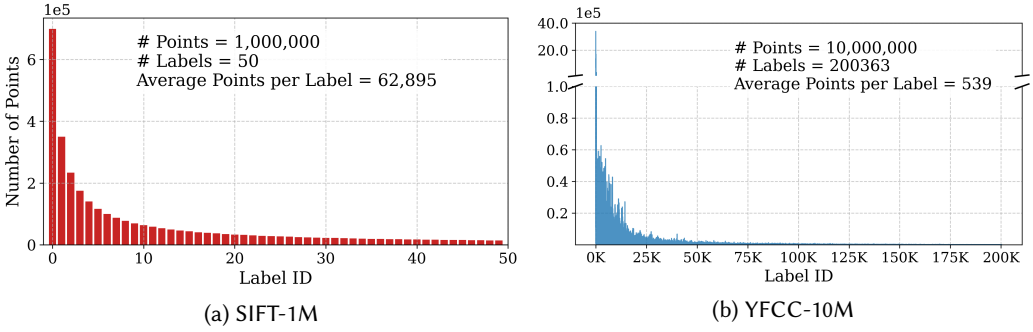


Fig. 3. Label specificity distribution in different datasets.

motivates us to consider algorithms that leverage the high computational throughput of GPUs to achieve high recall even on these cases with rare labels, which may be prohibitively costly on CPU systems.

Different from graph-based methods, IVF indexing is another fundamental technique in ANNS that partitions vectors into clusters to reduce the search space [11, 30]. In traditional IVF methods, vectors are clustered based on their distance to the cluster centroids (typically using k-means clustering), creating "inverted lists" where the cluster centroid becomes the keyword, i.e., the descriptor of the cluster, and the posting list comprises all vectors in the corresponding cluster. During a search, the system first identifies the most promising clusters by comparing the query vector with centroids, then performs exact searches only within those selected posting lists. In contrast to traditional IVF, one can construct an inverted index where the keyword is the label, and the posting list contains vectors that share the same label [6, 49]. This label-centric IVF knows exactly which posting list to search for each query based on its labels, eliminating the cluster selection overhead. This approach works effectively like setting $n_probes = 1$ in traditional IVF search, but without the accuracy trade-off, as it guarantees searching the exact cluster containing the matching vectors. Prior studies show that IVF-based filtered-ANNS solutions can also achieve competitive performance compared to graph-based solutions such as Filtered-DiskANN [54].

To address the issue, we introduce a *dual-structured label-centric IVF* that leverages the high computational throughput and parallelism of GPUs for high-performance filtered-ANNS. VecFlow adopts a label-centric IVF design to improve the selectivity of ANNS with filters, because a query only needs to search the posting list that has the query's label. In addition, we define label specificity, a metric for quantifying the density of a given label in a dataset, and we divide the IVF posting lists into distinct groups, each with its own characteristics of label distribution. VecFlow employs a dual-structured index and search algorithms to achieve fast and accurate filtered ANNS. For each group, we design high-performance GPU kernel optimizations to maximize GPU resource utilization. We first describe this algorithm and then introduce several optimizations to map the algorithm to GPUs.

4.1.1 Specificity Definition We start with the definition of *specificity*, a property that measures the fraction of data points in X that have the label $l \in \mathcal{L}$ associated with them. More specifically:

$$\text{specificity}(l) = \frac{\text{number of data points with label } l}{\text{total number of data points } N}$$

This definition is similar to prior work [29, 48]. Different from prior work, we further classify a label as:

- High-specificity (HS): $\text{specificity}(l) \geq T/N$
- Low-specificity (LS): $\text{specificity}(l) < T/N$

where T is a chosen specificity threshold. A data point x therefore can also be categorized according to its labels:

- High-specificity data: x has at least one high-specificity label
- Low-specificity data: x has at least one low-specificity label

Note that in scenarios where a data point can have multiple labels, a data point can have both high- and low-specificity labels simultaneously.

4.1.2 Decoupling High/Low-Specificity Labels with Label-Centric IVF and Dual-Structured Indices

We construct an inverted index where the *keyword* is the label, and the posting list contains data point indices. Specifically, for each label $l \in \mathcal{L}$, we define $C_l = \{i \mid l \in L_{X_i}, 0 \leq i < |X|\}$, which stores the global indices of points in X associated with label l . Using C_l , we can efficiently retrieve the actual vectors for label l as $X_l = \{X_i \mid i \in C_l\}$ without scanning the entire data point set. Once we build the Label-Centric IVF, we divide the posting lists into two groups: the high-specificity group (HS) and the low-specificity group (LS), and develop a dual-structured index construction and search algorithm to enable efficient filtered-ANNS for both HS and LS. We note that classifying data based on labels is a long-standing idea in database systems [13, 19]. However, our approach mainly focuses on optimizing filtered-ANNS on GPUs, given the long-tailed distribution of real-world filtered search scenarios. For the HS partition, we build a separate GPU-friendly graph index (e.g., CAGRA [46]) over an individual posting list C_l (IVF-Graph). This allows VecFlow to avoid an exhaustive search within a posting list while taking advantage of the hardware-friendly GPU-based ANNS solution, achieving performance similar to unfiltered search. For the LS partition, we directly resort to brute force search (IVF-BFS).

The rationale for adopting this dual-structured index is that for LS data, each IVF list contains a relatively small number of points, making the construction and maintenance of a graph-based index unnecessarily complex and potentially counterproductive. For example, CAGRA's search behavior is primarily governed by its internal top-k (*itopk*) parameter rather than the actual dataset size. This parameter determines both the number of iterations and the number of distance computations performed, regardless of cluster size. Our experimental analysis demonstrates this scaling inefficiency – when testing across different cluster sizes from 100 to 20,000 points with constant *itopk*(32), CAGRA's throughput remains relatively flat (2.6M QPS for 100-point clusters vs. 2.3M QPS for 20,000-point clusters), as shown in Fig. 4(a).

Despite showing excellent performance, graph traversal incurs iterative neighbor expansion that leads to additional synchronization overhead. Fig. 4(b) demonstrates that for queries with low-specificity labels in the YFCC dataset, an average of 71.29% of the CAGRA search time (*itopk*=32) is allocated to actual distance computations, while the remaining time is distributed across graph traversal overhead, including initialization (8.04%), parent node selection (5.41%), and internal top-k maintenance (14.79%). Consequently, when a label corresponds to only a small number of points, it becomes more efficient to directly compare the query vector against all points in the IVF list, avoiding the overhead of initializing and traversing a graph.

For low-specificity data, brute-force search (BFS) offers advantages over graph-based methods because BFS can be more easily parallelized to efficiently leverage modern GPUs, especially when the number of data points is relatively small. Essentially, the distance computation between a query and a set of vectors can be effectively formulated as a *batched matrix-vector multiplication (GeMV)* operation, which can be tiled across Streaming Multiprocessors (SMs) on a GPU. This approach helps to maximize the utilization of the GPU's compute and memory bandwidth. GeMV is primarily a memory bandwidth bound operation and the peak memory bandwidth for a GPU (e.g., Nvidia A100) is often over 2 Terabytes per second (2TB/s), which is over 20x higher than modern CPU bandwidth (often <100GB/s). Thus, BFS on GPUs offers significant speedups over CPUs, on which

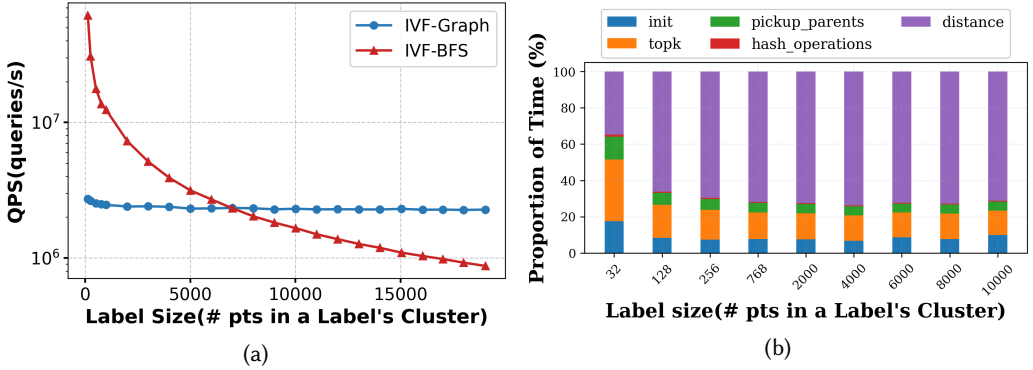


Fig. 4. (a) Scaling efficiency of IVF-Graph and IVF-BFS varying the number of data points within a cluster. (b) Time decomposition of CAGRA search process.

BFS may be prohibitively slow. By employing BFS, execution time is optimized by focusing on the distance computation, eliminating graph traversal overhead entirely. This makes BFS a more efficient choice for low-specificity scenarios where the benefits of graph traversal are minimal.

During the search, whether a query searches the HS or LS partition is determined by the specificity threshold T . Given a query with label l , we route the query based on the size of its corresponding cluster $|C_l|$:

$$\text{method} = \begin{cases} \text{IVF-BFS} & \text{if } |C_l| < T \\ \text{IVF-Graph} & \text{otherwise} \end{cases} \quad (2)$$

Finding the optimal threshold T analytically is non-trivial as it depends on many architectural parameters and specificity distribution. However, it is not necessary in practice. T can be carefully chosen to represent the crossover point where the brute force approach becomes less efficient than graph traversal for a given number of data points. This is determined through offline profiling query performance across different cluster sizes and auto-tuning, e.g., by comparing the performance of HS and LS search to identify a threshold T that maximizes the overall query throughput.

4.2 Index and Search Strategies of VecFlow on GPUs

4.2.1 Bottom-Level IVF-Graph with Redundancy-Bypassing for HS

Indexing. While combining IVF with graphs significantly improves the search speed, building an individual graph index for each IVF list leads to one major challenge – the memory consumption becomes much higher compared to a single index for all data points. We observe that directly building IVF-Graph index for each posting list leads to a $10.8\times$ increase in memory compared to a single graph index over the entire YFCC dataset. Why does the memory consumption increase dramatically? We dive deep into how graph-based ANNS index is constructed for GPUs and find out that this is caused by memory redundancy in IVF-Graph. Taking CAGRA as an example, for a given set of data points, CAGRA builds a k -NN graph with fixed out-degree using NN-descent [15] on GPUs and performs rank-based reordering to improve the graph's navigability. The final CAGRA-based IVF-Graph list for each label l , consists of two components (1) vectors X_l requiring $|C_l| \times D$ memory, and (2) a graph structure consisting geometric relation requiring $|C_l| \times R$ memory, where D is the vector dimension and R is the graph out-degree. When a data point X_i belongs to multiple labels (e.g., $i \in C_{l_1}$ and $i \in C_{l_2}$), its vector must be duplicated in both X_{l_1} and X_{l_2} . Since vectors typically dominate memory consumption (e.g., $D = 192$ in YFCC dataset versus $R = 16$), this redundant storage of vectors across multiple IVF lists causes the substantial memory increase.

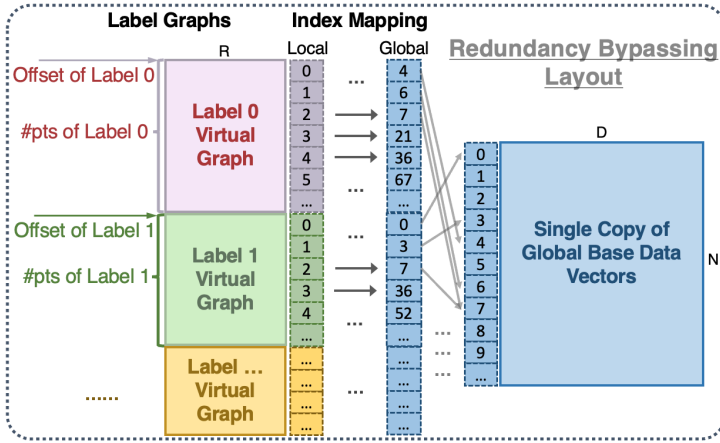


Fig. 5. The data layout used in VecFlow's redundancy-bypassing IVF-Graph index and search.

To address the redundancy issue, we propose an approach called IVF-Graph with Redundancy-Bypassing. For each label l , VecFlow builds a *local virtual graph* G_l , where each virtual graph only contains the graph structure information expressed via local vertex IDs. Alongside these graphs, VecFlow has a single *global* vectors X that contains all data points. In addition, VecFlow maintains a **local-global** index mapping table M_{HS} that translates these local vertex IDs (within each G_l) to their corresponding global indices in the global vectors. This mapping mechanism allows VecFlow to maintain only a single copy of data point set while having multiple virtual graphs to share the same global vectors.

Current graph-based ANNS algorithms typically use a single graph structure represented by adjacency lists, where each row uniquely corresponds to the neighbors of a specific data point. However, VecFlow introduces multiple virtual graphs to enable filtered search. If these virtual graphs were stored separately, processing queries with different labels would require frequent index switching, hindering the parallel performance of methods like CAGRA for large batches. On the other hand, storing multiple graphs together in a single index is not supported by existing ANNS libraries. To address the issue, we design a readily integrable and GPU-efficient index structure to implement our approach. In particular, we compact all virtual graphs into a single, continuous memory space, organized and ordered by their label l . This compacted structure not only avoids the overhead of switching between indices but also represents multiple virtual graphs as a unified structure. As a result, it seamlessly integrates into any graph-based ANNS algorithm by replacing the original single-graph structure, enabling filtered searches without requiring significant modifications to the index structure. Furthermore, since our combined index is compatible with single-graph structures, any optimization techniques designed for single indices can be directly applied to our unified graph.

The layout of VecFlow's compacted Label Graphs G_{HS} is illustrated in Fig. 5. To make this structure functional, VecFlow maintains three key metadata components: (1) Label Sizes S_{HS} : the number of data points in each posting list C_l , (2) Label Offsets O_{HS} : the starting position of label's virtual graph G_l within the compacted Label Graphs G_{HS} , which together help VecFlow efficiently locate G_l , and (3) Index Mapping M_{HS} : which translates local vertex IDs in G_l to their global indices in X . As shown in Algorithm 1, to manage labels, VecFlow's indexing follows a two-pass process: first, VecFlow collects all data points' indices associated with each label l into posting list C_l . Then, VecFlow builds separate virtual graphs for each high-specificity label while establishing metadata for each label that includes size, offset, and mapping information. This approach allows VecFlow

Algorithm 1: VecFlow Index Construction**Input:** Data point set X ; L_x for each $x \in X$; Graph Degree R ; specificity-threshold T .**Output:** IVF-Graph-Index; IVF-BFS-Index.

```

1  $\mathcal{L} \leftarrow \bigcup_{x \in X} L_x$   $\triangleright$  Set of unique labels
2  $C_l = \{i \mid l \in L_{X_i}, 0 \leq i < |X|\}$  for all  $l \in \mathcal{L}$ 
3 foreach  $l \in \mathcal{L}$  do
4    $X_l = \{X_i \mid i \in C_l\}$ 
5   if  $|C_l| \geq T$  then
6     SetMetadata( $C_l, S_{HS}[l], O_{HS}[l], M_{HS}[l]$ )
7      $G_{HS} \leftarrow \text{vstack}(G_{HS}, \text{BuildCAGRAGraph}(X_l, R))$ 
8   else
9     SetMetadata( $C_l, S_{LS}[l], O_{LS}[l], M_{LS}[l]$ )
10     $X_{LS} \leftarrow \text{StoreInterleaved}(X_{LS}, X_l)$ 
11 return  $(X, G_{HS}, S_{HS}, O_{HS}, M_{HS}), (X_{LS}, S_{LS}, O_{LS}, M_{LS})$ 

```

Algorithm 2: VecFlow Search Algorithm**Input:** Query batches Q ; L_q for each $q \in Q$; IVF-Graph-Index($X, G_{HS}, S_{HS}, O_{HS}, M_{HS}$);IVF-BFS-Index($X_{LS}, S_{LS}, O_{LS}, M_{LS}$). specificity-threshold T ;**Output:** Top- k nearest neighbors for each query.

```

1  $(Q_{HS}, L_{HS}), (Q_{LS}, L_{LS}) \leftarrow \text{ClassifyQueries}(Q, L_q, T)$ 
2 foreach  $(q, l)$  from  $(Q_{HS}, L_q)$  in parallel do
3    $G_l \leftarrow G_{HS}[O_{HS}[l] : O_{HS}[l] + S_{HS}[l] - 1]$ 
4    $topM, candidates \leftarrow \text{InitializeSearch}(G_l)$ 
5   for search iterations do
6      $dists \leftarrow \text{GetDist}(candidates, q, l, X, M_{HS}, O_{HS})$ 
7      $TopM \leftarrow \text{UpdateTopM}(TopM, candidates, dists)$ 
8      $candidates \leftarrow G_l[topM[\text{first unvisited node}]]$ 
9 foreach  $(q, l)$  from  $(Q_{LS}, L_q)$  in parallel do
10  BruteForceSearch(IVF-BFS-Index,  $q, l$ )
11 return Merge results and map to global IDs

```

index to construct multiple virtual graphs into a compacted Label Graphs G_{HS} referencing a single global vectors X , thereby reducing memory consumption while preserving search efficiency.

Search. For queries with high-specificity labels, VecFlow routes the query to the HS partition and performs the graph-based search in the posting lists that have the query's labels. Algorithm 2 demonstrates how VecFlow utilizes its indexing structures during search. The key step $G_l \leftarrow G_{HS}[O_{HS}[l] : O_{HS}[l] + S_{HS}[l] - 1]$ extracts precisely the portion of local virtual graph G_l for label l , using both the *Label Sizes* and *Label Offsets* metadata. The actual search follows the top-M list and candidate list expansion procedure in CAGRA, where VecFlow starts with random sampling (e.g., VecFlow uses *Label Sizes* to confine the range of random sampling) to get the initial candidate list and performs iterative graph traversal iterations, i.e., sort itopk, pick up next parents, compute distance for child node, expand the candidate list until the top-M list converges, i.e., the top-M list remains unchanged from the previous iteration. Along the search process, VecFlow uses the same

elemental technologies from CAGRA [46] to boost the GPU utilization for the graph traversal on GPUs, including (1) warp splitting that divides a warp into fine-grained thread groups (a.k.a, teams) to allow all threads within each warp to maximize the bandwidth utilization through 128-bit load instructions, (2) single warp-level bitonic sort to quickly sort the priority queue in the registers of a single warp without the shared memory footprint, and (3) the forgettable hash table management to keep the hash table in the fast shared memory for managing the visited node list.

Like CAGRA search, all of our search operations, such as top- k computations and internal list updates, are performed using local vertex IDs within G_l with values in the range $[0, S_{HS}[l]]$. Only when computing distances, VecFlow's *Index Mapping* $M_{HS}[l]$ translates the local vertex IDs to corresponding global indices in the global vectors.

4.2.2 Bottom-Level GPU-Friendly Brute Force Search for LS While BFS for LS is conceptually straightforward, requiring only target vectors and query vectors, performing BFS on GPUs presents challenges, especially when processing multiple queries in parallel (e.g., batch size > 1), as each query in the batch must search different parts of the dataset based on its labels. Recent advancements in GPU-based KNN search introduced in the NVidia cuVS library [2], provide two efficient BFS implementations with bitmap filtering. For these approaches, each row of the bitmap represents a different query's label and each column represents a data point in the dataset, with bits set to 1 indicating label matches between queries and data points. In the first method, a tiled brute force kNN search implementation is used to achieve high compute efficiency between the queries and the entire data points, followed by applying the bitmap as a post-processing filter to get the final results. However, this approach becomes computationally expensive with large datasets like YFCC, which has 10M data points, as it requires computing distances between each query and all data points regardless of whether they match the query's labels or not. The second approach transforms the kNN search into a sparse computation by converting the bitmap into the Compressed Sparse Row (CSR) format, which only stores the coordinates of the matching pairs. This enables masked matrix multiplication that computes distances only between each query vector and its label-matching data vectors. The implementation processes all queries together in a single masked matrix multiplication operation, followed by a k -selection step to find the nearest neighbors for each query. This approach requires both extra memory accesses to load the CSR mask and leads to scattered memory accesses from sparse computation patterns, which are less efficient than the contiguous accesses typical in dense operations. As a result, it is unclear how to achieve high performance filtered kNN on large-scale datasets for large batches.

Unlike these cuVS brute-force search methods, VecFlow introduces an interleaved scan based IVF-BFS method for low-specificity data. This method is specifically designed for high-throughput distance computations on GPUs. The approach is built on top of cuVS IVF-Flat. During the index construction phase, VecFlow employs an interleaved memory layout where vectors within each posting list of IVF are organized into blocks of interleaved components. As shown in Figure 6, VecFlow divide vectors into groups of 32 to match the GPU warp size. Within each group, instead of storing vectors sequentially $((v_0[0 : D], v_1[0 : D], \dots))$, VecFlow stores them in an interleaved pattern where components are grouped based on $veclen = 16/\text{sizeof}(T)$, where T is the data type. For example, with float32 vectors ($\text{sizeof}(T) = 4$), $veclen = 4$, so VecFlow stores $(v_0[0 : 3], v_1[0 : 3], \dots, v_{31}[0 : 3], v_0[4 : 7], v_1[4 : 7], \dots)$. This organization enables efficient memory access patterns in two ways. First, when 32 threads in a warp process different vectors simultaneously, they access contiguous memory locations, allowing the GPU to coalesce these accesses into fewer memory transactions. Second, each thread can use wide load instructions (LD.E.128) to fetch 16 bytes at once, as the components for each vector are stored in chunks sized according to $veclen$. This dual-optimization of coalesced access across threads and vectorized loading within threads significantly

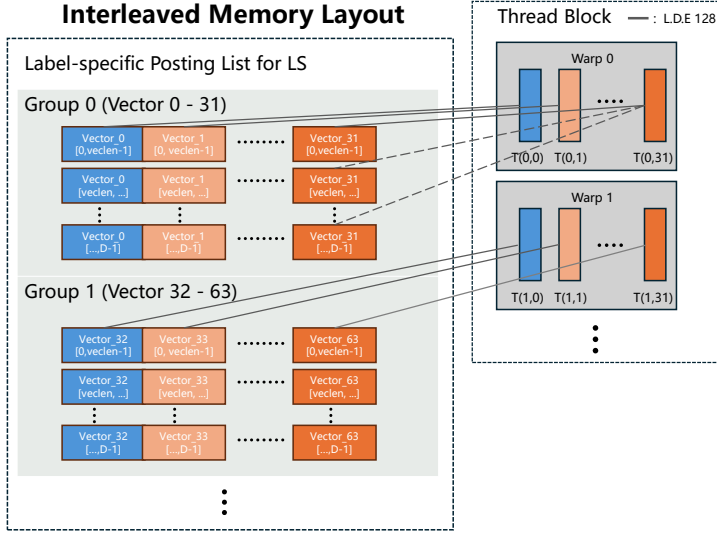


Fig. 6. Interleaved memory layout for VecFlow's IVF-BFS over LS partition.

improves utilization of GPU memory bandwidth during distance computations. In addition, this interleaved layout improves instruction-level parallelism and instruction pipeline efficiency by letting threads begin performance distance computations while vector coordinates continue to be fetched from memory.

For the search process, VecFlow uses `ivfflat_interleaved_scan` kernel by cuVS which achieves high throughput through hierarchical parallelism and efficient distance computation. At the grid level, each CUDA block processes one query that searches inside its corresponding label-specific posting list from the IVF index. Within each block, the query vector is first loaded into shared memory for fast access by all threads. The block then processes vectors in its assigned posting list through coordinated warp execution. Each warp processes one group of 32 interleaved vectors from the memory layout described in Figure 6. Within a warp, each thread computes the distance between one vector from the current group and the shared query vector. Another important step during the brute force search is to filter out the top-k candidates. The kernel fuses an optimized block-select-k operation into the distance computation to identify nearest neighbors efficiently. This approach outperforms cuVS's brute force search implementations by avoiding extra distance computations through label-specific IVF posting lists and achieving better memory bandwidth utilization with optimized interleaved memory layout. This hierarchical organization naturally aligns with the structure of small label-based clusters: blocks can independently process different query-label pairs, while warps provide efficient parallel distance computation and sorting within each cluster. Overall, this ensures complete coverage for exhaustive search without the overhead of graph traversal.

Same as IVF-Graph Redundancy-Bypassing, IVF-BFS maintains three key metadata components for each label, which are *Label Sizes* S_{LS} , *Label Offsets* O_{LS} and *Indexing Mapping* M_{LS} . Algorithm 1 and Algorithm 2 also show the index construction and search process of IVF-BFS. We note that, for low-specificity data points, VecFlow introduces additional memory overhead compared to using base vectors directly. This overhead arises because vectors are stored within each label's IVF list in a continuous, interleaved layout to optimize GPU memory bandwidth utilization. Since a single data point may be associated with multiple low-specificity labels, its vector needs to be stored in multiple IVF lists. The total memory overhead can be expressed as $N \times D \times F$, where F represents

the average number of low-specificity labels per point. For example, in the YFCC dataset, about 5.95M points (59.5% of the dataset) have at least one low-specificity label (e.g., labels associated with fewer than 1,000 points), with these points appearing on average in 3.54 low-specificity IVF lists (21.1M total entries across low-specificity IVF lists divided by 5.95M points), leading to 3.54x memory overhead compared to the original vector storage for LS partition.

4.2.3 Persistent Kernel-based Search for Small Batch Queries Since all queries within a batch are independent, VecFlow uses a single-kernel implementation for handling batched queries but use an individual CUDA thread-block (single-CTA) within a kernel to handle large batch sizes, similar as how CAGRA handles large batch sizes. Overall, when the batch size is large (e.g., >100), each query is mapped to a single thread block, and multiple of these blocks are executed in parallel. This approach effectively harnesses the massive parallelism of GPUs, maximizing their compute potential. However, when the batch size is small (e.g., <100), the GPU resource can be left underutilized with the single-CTA implementation.

In practice, vector search services often must execute queries in small batches and maintain low latency. However, traditional GPU-based methods like CAGRA are optimized for large batches and suffer from high overhead when processing small batches due to repeated kernel launching overhead. One may wonder whether it is possible to hide the kernel launching overhead to some extent by launching a few kernels in parallel using streams, thereby overlapping the launch overhead with the execution of other kernels. However, this strategy also does not work very well in practice because of CPU thread synchronization overheads. Modern GPUs require hundreds of thread blocks running in parallel to achieve maximum throughput. At this scale, even a few CUDA host API calls per-thread can become a bottleneck due to shared CUDA context locking. To address this, VecFlow introduces a persistent kernel approach that maintains a continuously running kernel on the GPU to process queries as they arrive, which not only eliminates the repeated kernel launch overhead but also avoid synchronization at all costs by using atomic-based control structures.

Fig. 7 shows our persistent kernel design, which uses a job queue system with atomic ring buffers to manage incoming queries and a worker queue to track available GPU thread blocks. When a query arrives, it is assigned a job ID and mapped to an available worker. The persistent kernel continuously monitors for new work, with each thread block handling one query independently. This approach enables efficient parallel processing while maintaining high GPU utilization despite small batch sizes.

4.2.4 Memory Consumption. We express the memory consumption as the sum of bytes required to store both the HS and LS partition indices. Let F denote the average number of labels per point, with F_{HS} and F_{LS} representing the averages for the HS and LS partitions, respectively. Without applying the redundancy-bypassing algorithm, the memory consumption of the HS partition is $N \times F_{HS} \times (D + R) \times b$, where R is the length of the out-degree, and b is the number of bytes of data type. After applying the redundancy-bypassing optimization, VecFlow stores a single shared copy of the base data vectors while maintaining separate neighbor lists for each label-specific graph. This reduces memory usage significantly because D is often much larger than R . The memory consumption of the HS partition becomes $N \times (D + F_{HS} \times R) \times b$. Furthermore, the average cluster size per label is much smaller than N in practice. For example, in the YFCC dataset, this average label size reduces from 10M to 540. Consequently, we use a much smaller out-degree R' , which shows no performance degradation based on empirical observation. Thus, the final memory consumption for the HS partition is $N \times (D + F_{HS} \times R') \times b$. Instead, we store only the base data in the LS partition, resulting in a memory consumption of $N \times D \times F_{LS} \times b$. Thus the total memory consumption is $N \times (D + F_{HS} \times R') \times b + N \times D \times F_{LS} \times b$. Assume we have 100 million points, with the following parameters: $D = 128$, $R = 64$, $F = 3.17$, $F_{HS} = 3.0$, $F_{LS} = 0.17$, $b = 4$, $R' = 16$. Substituting the values,

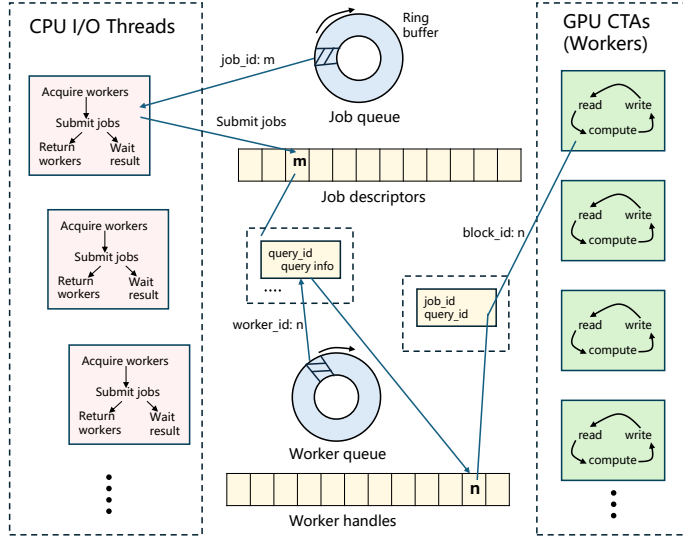


Fig. 7. Persistent kernel design in VecFlow.

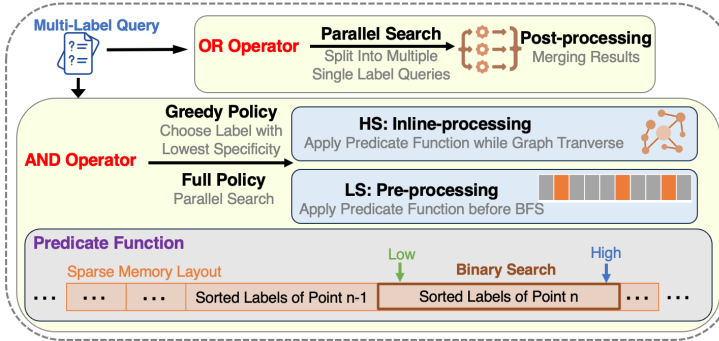


Fig. 8. Multi-label query processing in VecFlow.

the memory consumption is $65.57 \text{ GB} + 8.11 \text{ GB} = 73.67 \text{ GB}$, which is slightly higher than a single index's memory consumption of 71.53 GB , calculated as $N \times (D + R) \times b$. For metadata, the total memory consumption for local-global index mapping is $N \times F \times b$, which is 1.18 GB . Regarding label metadata (*Label Size* and *Label Offset*, 2 integers for each label), since the number of labels is typically in the range of several thousands, its memory usage is negligible. For example, even in the YFCC dataset with 200K labels (which already represents a relatively large label set), the label metadata only accounts for 1.49 MB . This value is sufficiently small that it can be ignored in most cases.

4.3 Multi-Label Query Processing with Predicate Filtering and Early Stopping

Prior work often assumes queries have a single label. However, in practice, one query is often associated with multiple labels, which introduces challenges. VecFlow's label-centric indexing handles multi-label queries efficiently through GPU parallelism, where each thread block processes one query label. As shown in Fig. 8, VecFlow implement different strategies for *OR* operations (parallel processing with result merging) and *AND* operations (either parallel search with filtering or selective greedy search, while both using predicate verification).

4.3.1 Handling OR Operations For multi-label query connected by OR operations, we can simply convert it into single-label search problem, treating each label as a separate search request and launch multiple standard single-label searches *in parallel*, using IVF-Graph search for high-specificity labels and IVF-BFS for low-specificity labels, followed by merging results to find the nearest neighbors that satisfy any of the query labels.

4.3.2 Handling AND Operations It becomes challenging to handle AND operations efficiently because a point must satisfy all query labels simultaneously. For example, when looking for points that satisfy both label A AND B, even if label A and B individually have many matching points, the set of points satisfying both constraints may be sparsely distributed in the single graph index. This forces the search to explore many irrelevant paths and points before finding valid results that satisfy all constraints. Our key insight to support efficient AND Operations is that it is *not necessary to check all paths* to identify candidates that satisfy A AND B. Instead, we develop a predicate function that efficiently verifies whether a point contains all specified labels and uses this predicate function to obtain a *selective early-stopping* search strategies for both HS and LS partitions to handle AND queries effectively.

Predicate function for efficient multi-label verification. To efficiently verify whether a data point contains all specified labels, we use a compact data structure based on three arrays: a global label array, a label offset array, and a label size array. The global label array stores the label of all data points sequentially, keeping each data point's labels contiguous and sorted. The label offset array marks where each point's labels begin in the global array, and the label size array records how many labels each point has. We choose this structure over a matrix representation $N \times |\mathcal{L}|_{\max}$ (where $|\mathcal{L}|_{\max}$ represents the max label size), because of highly variable label sizes. In the YFCC dataset, while the average is 10.8 labels per point, some points have up to 1,500 labels. A matrix approach would waste memory by allocating 1,500 columns for every point. Our array-based approach stores only the actual labels each point has, reducing memory fragmentation and enabling efficient GPU parallel processing through a compact, contiguous memory layout.

For each data point, we employ a *binary search* algorithm to retrieve its portion of labels from the global label array. For instance, if a data point's offset is 1000 and it has 15 labels, we search within positions 1000 to 1014 of the global array. The search adjusts two pointers iteratively based on comparisons with the target label. If found, it returns true and the label's position; otherwise, it returns false, indicating the point fails the AND predicate.

For queries with multiple labels, we optimize the search process by first sorting the query labels. It then searches from the smallest and largest query labels to establish the valid range that must contain all other query labels. For example, suppose a point has labels [1,3,5,7,9,11,13,15] and the algorithm needs to verify query labels [5,9,11]. It first searches for the smallest label 5, finding it at index 2. If label 5 is not found, it immediately returns false. Otherwise, it searches for the largest label 11, finding it at index 5. If label 11 is not found, it returns false. After finding both boundary labels, it establishes that all remaining query labels must exist within range [3,4]. For the middle label 9, it only needs to search this smaller range. This approach maintains the worst-case complexity of $O(|\mathcal{L}_q| \log F)$, but typically performs better in practice as intermediate searches operate on a much smaller range. The function returns true only if all query labels are found.

IVF-Graph AND search for HS. For queries with multiple labels connected by AND operations in the high-specificity partition, we develop a *greedy search policy* that combines IVF list selection with inline-processing during graph traversal. The key idea is to minimize the search space while maintaining high accuracy through two main components. Our search space selection is straightforward: given a query with multiple labels connected by AND operations, the search

selects the IVF list corresponding to the label with the *lowest* specificity. For example, if a query asks for points with labels A associated with 10,000 points *AND* B associated with 1,000 points *AND* C associated with 5,000 points, the search chooses label B's IVF list. This ensures we search within the smallest possible search space. For each iteration of graph traversal, we employ the inline-processing approach that integrates label verification directly into the search process. After computing distances at each graph expansion iteration, we apply our predicate function to check whether candidate points contain all the rest higher-specificity labels from the query. Points that fail this verification step are removed from consideration before the next iteration begins. This filtering strategy ensures that graph traversal expands only from points that satisfy all label requirements.

This combined approach provides performance benefits through improved selectivity. By starting from the IVF list of the lowest-specificity label and using higher-specificity labels as filters during traversal, we increase the relative selectivity to points that satisfy all label constraints. For example, when searching for points with labels A *AND* B where A is associated with 50,000 points and B with 5,000 points, searching in B's IVF list reduces the initial search space by 90% compared to searching in A's list. Meanwhile, since a point in the dataset has a higher probability of having label A, filtering for label A during graph traversal becomes more effective. This means when we verify if a point has label A during traversal, we have a better chance of finding valid neighbors, making the search more efficient in exploring the relevant regions of the graph.

If very high accuracy is desired, we also provide a full *parallel search policy* for handling *AND* queries in the HS partition. This approach converts *AND* queries into single-label search problems, similar to what we do for *OR* operators, but incorporates inline processing with our predicate function. Instead of starting from the IVF list with the lowest specificity, we launch parallel searches in each label's IVF list. Each search applies the predicate function during graph traversal to verify if points contain all other labels in the query. For instance, when searching for points with labels A *AND* B, two parallel searches are launched: one in label A's IVF list filtering for label B, and another in label B's IVF list filtering for label A. Each search is executed by a dedicated CTA. After all CTAs complete their searches, the final results are obtained by merging the top-K candidates from all parallel searches. This method delivers higher QPS, particularly when the recall target is high (e.g., >90%).

IVF-BFS AND search for LS. For queries with multiple labels connected by *AND* operations in the low-specificity partition, we select the IVF list corresponding to the label with the lowest specificity. This choice is sufficient since BFS will traverse all points in the selected list. Before performing a brute-force search within the selected IVF list, we apply our predicate function to verify whether each point satisfies all query labels. This pre-processing strategy effectively eliminates unnecessary distance computations for points that do not meet all label constraints.

5 Evaluation

In this section, we experimentally evaluate VecFlow.

5.1 Evaluation Methodology

Datasets. We use public datasets from prior work to evaluate our system, as listed below:

- **YFCC.** The YFCC-10M dataset [56], utilized in the NeurIPS'23 Big-ANN Competition Filter Track [9], contains 10 million images transformed into CLIP [50] embeddings with 192 dimensions. Each is associated with labels extracted from multiple sources including image descriptions, camera model information, capture year, and location data. The dataset features 200,386 unique labels ($|\mathcal{L}| = 200,386$) following a highly skewed distribution – a small number of labels appear frequently while most appear rarely. On average, each data point is associated with 10.8 labels.

- **WIKI-ANN.** Wiki-ANN [28] is a dataset introduced in [8], designed to evaluate AND predicates. Each query in the dataset is associated with two labels connected by an AND operator. The dataset contains 35 million Wikipedia passages transformed into embeddings with 768 dimensions. The set of all possible labels $\mathcal{L}$ consists of the 4,000 most frequent words in passages, where each passage's labels are those words from $\mathcal{L}$ that appear in its text. With the total embedding size exceeding 100GB, which surpasses single GPU memory capacity, we use a 1-million-point subset for our experiments, with each data point having an average of 22.5 labels.
- **Semi-synthetic dataset.** Following the approach used in Filtered-DiskANN [22], we employ Zipf's law to generate labels for the base dataset, effectively simulating the skewed distributions commonly observed in real-world applications. For the base dataset, we use **SIFT-1M**($D=128$), with a set of $|\mathcal{L}| = 50$ unique labels, and an average of 3.17 labels assigned per data point. Additionally, to assess the scalability of our algorithm, we use the same method to generate $|\mathcal{L}| = 2,500$ unique labels for the **DEEP-50M**($D=96$) dataset, with an average of 5.88 labels per data point.

Metrics. We evaluate Filtered-ANNS search efficiency using recall and queries processed per second (QPS). The recall measures the fraction of the top- K query results retrieved by the Filtered-ANNS that match exact nearest neighbors with query labels(measured by brute force search). In practice, we usually choose $K = 10$ and take QPS at 90% recall as an important metric. In addition, we also measure average latency (time to run a single batch) in latency mode (single-batch mode), and memory footprint consumption of our algorithm.

Baselines. We compare with the following baseline methods:

- Filtered-DiskANN [22]: We include the two algorithms from Filtered-DiskANN: FilteredVamana and StchedVamana. We use the parameters listed in the Filtered-DiskANN paper to build the index and search.
- FAISS [16]: The baseline from the NeurIPS'23 Competition BigANN Filter track.
- IVF² [6]: The best-performing solution from the NeurIPS'23 Competition BigANN Filter track that uses CPU for filtered-ANNS. It is tailored for the YFCC dataset.
- CAGRA + Post-processing (CAGRA-Post) [46]: We include a strong baseline by extending CAGRA with post-processing. We build a CAGRA graph for the entire dataset, query as usual, and employ post-processing to select only those results returned by the index that match the query filter.
- CAGRA + Inline-processing (CAGRA-Inline) [46]: This offers a stronger baseline by extending the state-of-the-art GPU-based ANNS solution CAGRA with on-the-fly filtering, i.e., the graph traversal skips points in the graph that do not match the filters of the query.

Testbeds Experiments are conducted on nodes with NVIDIA A100 GPU (40GB) and AMD EPYC 7763 CPU (2.45 GHz, 128 cores, 256GB memory) and NVIDIA Grace Hopper GH200 Superchip (96GB).

5.2 Main Result

Our first experiment demonstrates the search performance across different datasets. We compare VecFlow with the state-of-the-art CPU-based Filtered-DiskANN, IVF², and our GPU-based baselines CAGRA-Inline and CAGRA-Post. For Filtered-DiskANN, we build graphs with $L = 90$ and $R = 96$ for FilteredVamana, $R_{small} = 32$, $L = 100$ and $R_{stitched} = 64$ for StchedVamana; For IVF² we use parameters suggested in [53]. For CAGRA-Inline and CAGRA-Post we build a single CAGRA graph with $R = 32$ as suggested in [46]. For VecFlow, we set the specificity threshold $T = 2000$, and we build the CAGRA graphs with $R = 16$ for the HS group.

Our results, as shown in Fig. 9 that follows the style of [6, 22, 46], highlight VecFlow’s significant advantage in QPS-vs-recall across different datasets. We achieve these plots by varying search parameters for each algorithm to improve recall and reduce throughput: search width (L_s from [22]) for FilteredVamana and StitchedVamana, and *itopk* for the CAGRA-based approaches and VecFlow. Overall, it is worth noting that VecFlow achieves million-scale QPS at 90% recall ($K=10$), which is one to two orders of magnitude higher than both CPU and GPU baselines.

On the semi-synthetic SIFT-1M dataset, FilteredVamana achieves a maximum recall of 60% even with a large L_s value. This suggests that traditional proximity graph with inline-filtering cannot effectively capture label-specific information within the graph’s connections. StitchedVamana performs better than FilteredVamana, achieving 37K QPS at 90% recall and capable of reaching 100% recall. This is because the stitched graph retains some label-related connections, which helps improve filtered graph search. CAGRA with post-processing and CAGRA with inline-processing both outperform StitchedVamana, thanks to CAGRA’s highly optimized search capability on GPUs. In comparison to those existing methods, VecFlow achieves an impressive 5M QPS at 90% recall, which is 135 times higher than StitchedVamana. VecFlow is able to achieve high performance because the dataset has only 50 labels, all of which are high specificity (the least specific label cluster contains 14,000 data points). This causes each query to only search a small IVF list, resulting in significantly reduced distance computation. In addition, VecFlow leverages CAGRA’s high performance within each IVF, achieving high throughput and high recall by leveraging GPU-friendly graph traversal. This use case illustrates that VecFlow performs very well for datasets without extreme label distributions.

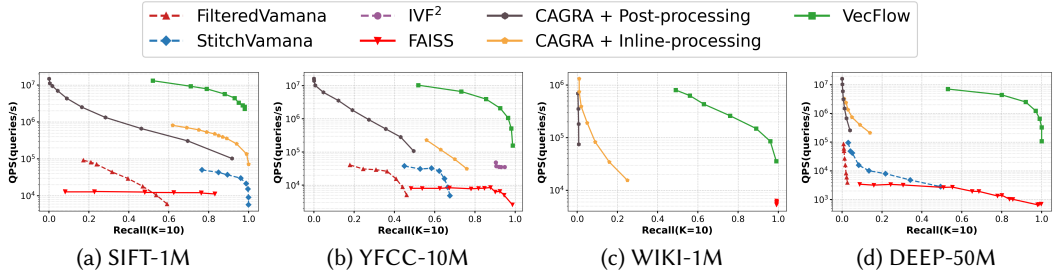


Fig. 9. The QPS vs. recall comparison results on (a) SIFT, (b) YFCC, (c) WIKI and (d) DEEP respectively.

Despite the more extreme filter labels for the YFCC-10M dataset, we observe similar trends with the performance of various methods. FilteredVamana struggles with lower recall levels even with large L_s , highlighting its limitations in capturing label-specific connections on a larger dataset. StitchedVamana also begins to struggle to achieve high recall, getting less than 70% even with very large L_s . This is because YFCC contains over 200K labels, many of which exhibit low specificity. FAISS achieves high recall but with very low QPS because it needs to explore a larger portion of vectors. For this dataset, IVF² performs the best among the CPU baselines, achieving 48K QPS at 90% recall. This is because it employs a label-centric IVF method similar to ours, which is particularly well-suited for datasets like YFCC that have a large number of labels with low specificity. CAGRA + Post-processing and CAGRA + Inline-processing outperform StitchedVamana again, leveraging GPU efficiency for rapid search, but are unable to achieve more than 80% recall. Finally, VecFlow continues to outperform all other methods, achieving a remarkable QPS of 2.6M at 90% recall, demonstrating its scalability and effectiveness on a real dataset as large as YFCC-10M and as complex as 200K labels.

The WIKI-1M dataset is particularly challenging due to the multi-label search queries. FilteredVamana and StitchedVamana are not included in the results because they do not support multi-label

filter searches. FAISS achieves high recall by performing BFS for almost every query due to the extremely small intersection between two labels. However, the time taken to find the intersection for each query is too slow. Both CAGRA Post-processing and Inline-processing methods achieve nearly 0 recall due to the extremely small intersection between two labels, making it unlikely to find points with the same label as the query. In contrast, VecFlow still delivers high performance, achieving 150K QPS at 90% recall. This is attributed to our highly efficient predicate function and optimized search policy for multi-label queries described in § 4.3. Finally, our evaluation on DEEP-50M, which is the largest dataset that can accommodate on a NVidia A100 40GB GPU, reveals that VecFlow achieves around 3M QPS at 90% recall, whereas other methods struggle to surpass 50% recall and many fail to even reach 20% recall, which is consistent with our previous analysis over SIFT-1M.

5.3 Analysis Results

Next we will justify each part of our design by conducting related performance evaluations. We also evaluate VecFlow's effectiveness over other type of GPUs.

How does redundancy-bypassing IVF-Graph bring benefits? In § 4.2.1 we introduce the IVF-Graph with redundancy-bypassing (RB) optimization. We evaluate its effectiveness over SIFT-1M and WIKI-1M datasets by comparing (i) single-index graph-based approach using CAGRA (ii) IVF-Graph without RB, and (iii) IVF-Graph with RB, in the same graph degree of 32, where "without RB" means without additional layers of indirection but with data vectors copied for each IVF-Graph. In addition, we also report IVF-Graph with reduced degree. Fig. 10 shows that without RB, the memory consumption of IVF-Graph increases dramatically, especially when each data point is associated with a large number of labels. In contrast, the RB optimization drastically reducing index size to 1.45 $\times$ and 1.89 $\times$ for the single-index baseline for SIFT-1M and WIKI-1M, respectively. This is achieved through our local virtual graph plus local-global mapping that enables sharing of the embedding vectors across IVF lists, effectively eliminating redundancy.

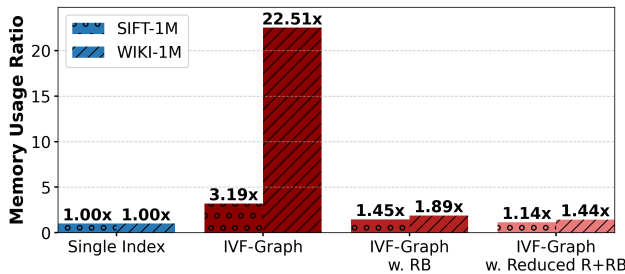


Fig. 10. Memory efficiency of redundancy-bypassing.

What is the speed-memory trade-off of redundancy-bypassing? We study VecFlow with and without RB on the SIFT dataset for filtered search. Fig. 11 shows that at recall 0.925, VecFlow with RB achieves 4.7M QPS while VecFlow without RB achieves 5.3M QPS, indicating that the additional layer of indirection causes approximately 11.6% performance loss. We consider this a desirable trade-off given that VecFlow with RB uses only 45% of the memory required by VecFlow without RB as shown in Fig. 11.

Is it necessary to perform IVF-BFS for search in LS? Fig. 12 presents the performance comparison between IVF-BFS and two baseline methods on the YFCC dataset for clusters with sizes less than 2000. The results show that IVF-BFS achieves 26M QPS, which is 2167 times and 9455 times faster compared to baseline BFS search with CSR format and BFS search with postprocessing, respectively.

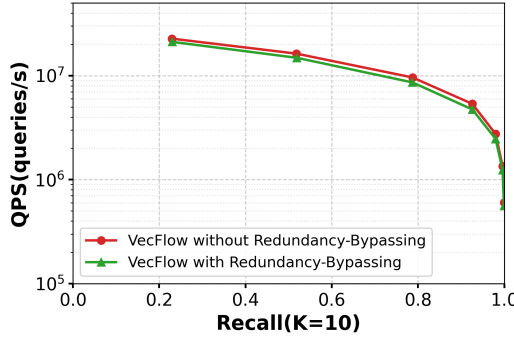


Fig. 11. Comparison results between VecFlow with Redundancy-bypassing and VecFlow without Redundancy-bypassing.

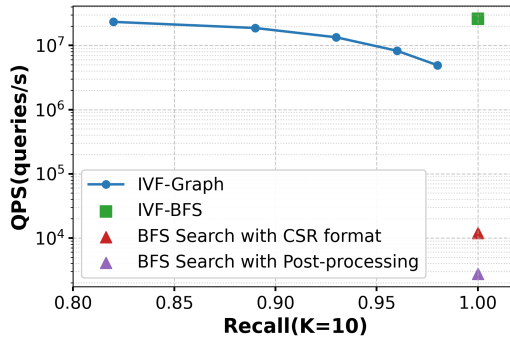


Fig. 12. IVF-BFS performance analysis on YFCC dataset (T = 2000).

Additionally, it is 1.32 times faster than IVF-Graph at 90% recall. This performance advantage comes from our interleaved scan-based kernel optimization, which makes IVF-BFS suitable for accelerating search in LS.

How does VecFlow's multi-label search policy perform? Fig. 13 illustrates the comparison between the two search strategies we proposed for the AND query. Here, we discuss the advantages and disadvantages of both approaches. The results show that the parallel strategy outperforms the greedy policy, especially in the high-recall region. For instance, on the wiki dataset, the greedy policy achieves a maximum recall of only 94% even with the largest *itopk*. However, with the same *itopk* size, greedy policy searches only the cluster with the lowest specificity label, resulting in reduced computational costs while achieving faster search speeds. As shown in the figure, with the same *itopk*, the QPS of the greedy policy is significantly higher than that of the parallel strategy, with minimal loss in accuracy.

We also evaluate single-label and double-label queries separately on the YFCC dataset. As shown in Fig. 16, both types of queries can achieve high recall of > 97%. Regarding QPS, single-label queries are 4 times faster than double-label queries at 90% recall, as the inline-processing slows down the search for double-label queries.

How does persistent kernel improve the performance of small batch queries? We test VecFlow on single-batch mode. For QPS, the persistent kernel achieves an average speedup of 5.68x and a maximum speedup of 6.39x on the SIFT dataset than the baseline, as well as an average speedup of 6.72x and a maximum speedup of 7.08x on the YFCC dataset. For the average latency per query, the persistent kernel achieves an average speedup of 1.54x and a maximum speedup of 1.71x on

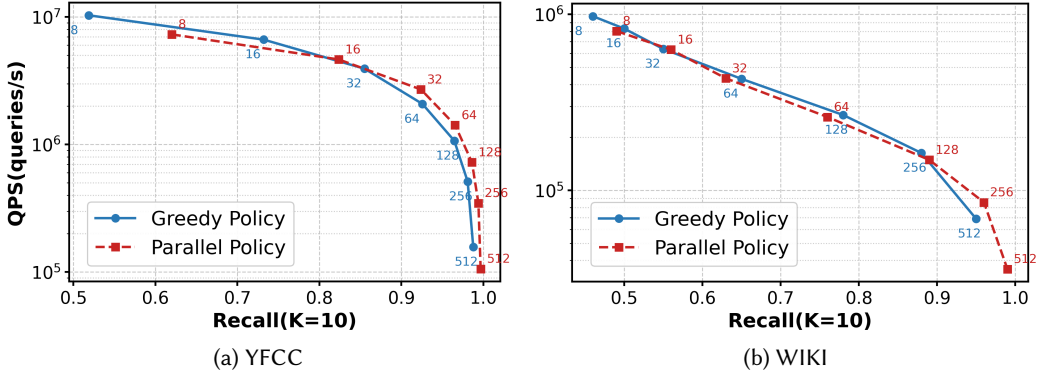


Fig. 13. The comparison of multi-label search policy in VecFlow. The *itopk* size used for each point is labeled next to the corresponding data point.

the SIFT dataset, and an average speedup of 1.73x and a maximum speedup of 1.82x on the YFCC dataset, as shown in Figure 14. Persistent kernel-based search improves the QPS when batch size is small because it uses a single-kernel to process incoming queries, without launching additional kernels, largely reducing the kernel launching overhead. In addition, by continuously accepting new queries to process, it effectively increases the GPU utilization, achieving high QPS.

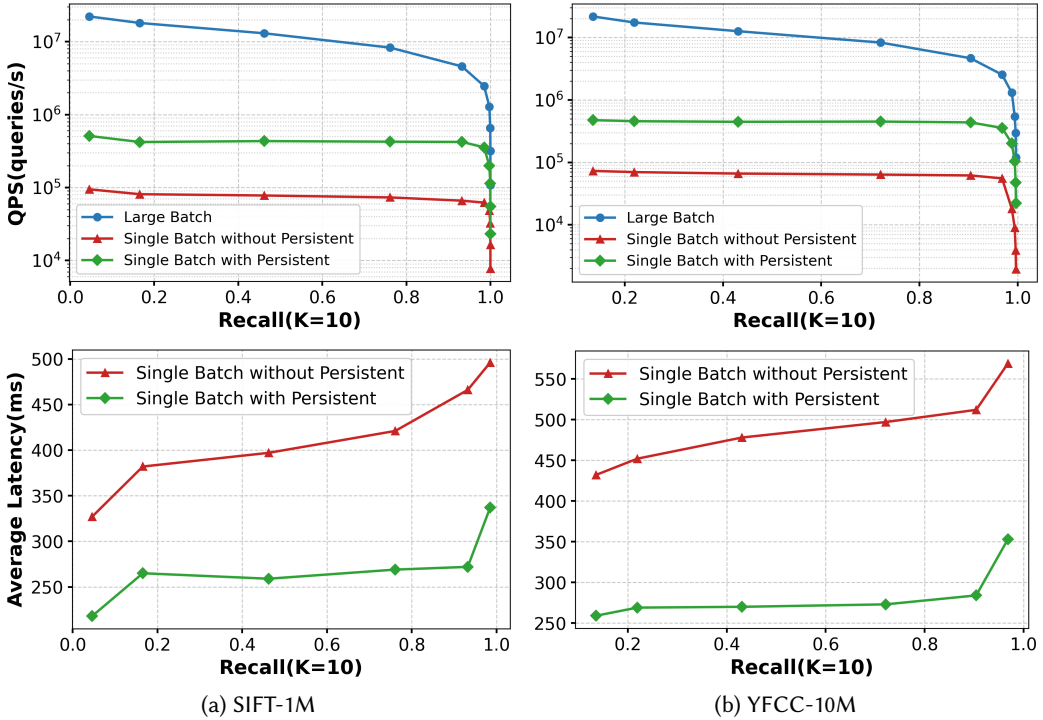


Fig. 14. Persistent kernel results on SIFT and YFCC datasets.

How does the choice of specificity threshold T affect the search performance? To analyze the impact of the specificity threshold T on our algorithm, we vary T and observe its influence on performance, as shown in Fig. 15. The parameter T controls the transition between IVF-BFS-only search ($T = \infty$) and IVF-Graph-only search ($T = 0$). At $T = 0$, the graph search results in worse performance

compared to BFS due to its constant overhead, as analyzed in § 4.1.2. At moderate thresholds, such as $T = 2000$ or $T = 5000$, the algorithm achieves a balanced trade-off, providing both competitive QPS and high recall. As T increases further, BFS becomes less efficient due to its exhaustive nature. This demonstrates that the threshold T is a key parameter for tuning to get the best performance.

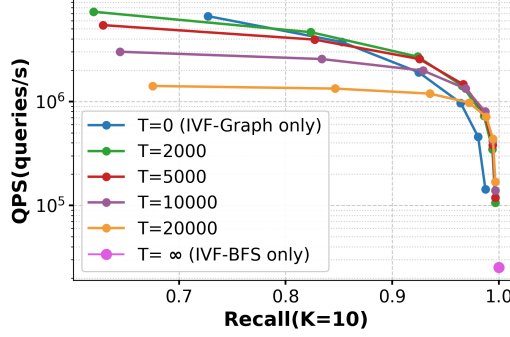


Fig. 15. VecFlow varying the specificity threshold T on YFCC dataset.

How does VecFlow perform on different GPUs? Fig. 17 shows the comparison of VecFlow over different types of GPUs. On GH200, VecFlow achieves 8.3M, 4M, and 300K QPS on the DEEP, YFCC, and WIKI datasets, respectively, which are 1.66x, 1.6x, and 2x faster than the results on A100. This is expected because H200 GPU has a memory bandwidth of 4.8TB/s, which is 2.4x higher than A100.

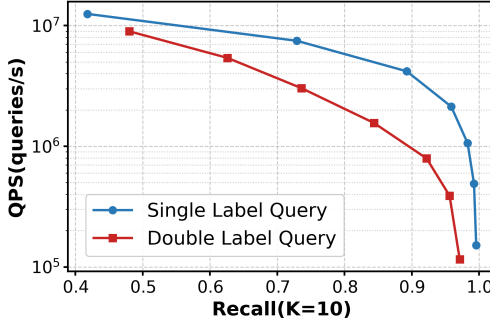


Fig. 16. Performance of single label and double label query on YFCC dataset.

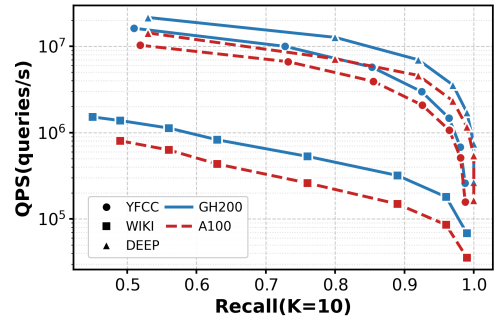


Fig. 17. Performance of VecFlow on NVidia GH200 GPU.

How does VecFlow perform on real-world production workloads? We evaluated VecFlow on a real-world recommendation dataset containing 4 million vectors ($D=64$) with over 10,000 labels showing a long-tailed distribution. Using a specificity threshold of 2000, VecFlow processed 110 high-specificity labels through IVF-Graph and 10,000 low-specificity labels with IVF-BFS. At 92% recall, VecFlow achieves 6.51 million QPS, while baseline approaches struggle with accuracy – CAGRA with post-processing reaches only 53% recall even at reduced throughput (175K QPS), and CAGRA with inline filtering achieves only 55% recall at 135K QPS. This demonstrates VecFlow's ability to maintain both high throughput and high accuracy in real production environments, where filtered-ANNS is critical for personalized recommendations.

Does VecFlow add high index construction time? Fig. 18 compares the index construction times for CAGRA, FilteredVamana, StitchedVamana, and VecFlow. CAGRA demonstrates exceptional index build speed, primarily due to its efficient GPU-based implementation. In contrast, FilteredVamana

and StitchedVamana, which rely on CPU computation, show considerably slower performance. StitchedVamana, in particular, is further hindered by the additional overhead of constructing and stitching separate indices. For our method, VecFlow first collects the data IDs associated with each label. For IVF-Graph, VecFlow aggregates the vectors belonging to each label $X_l = \{X_i | i \in C_l\}$ with C_l which contains data point indices and use multiple CUDA streams to construct different label graphs concurrently while sharing the same memory pool. After construction, VecFlow releases the vectors per label since VecFlow's final index shares only one global vector storage. For IVF-BFS, VecFlow has a highly optimized kernel that arranges the data in an interleaved format without requiring computational operations like ANN-based graph indexing, making this component's processing time negligible due to GPU's high memory bandwidth. This approach enables VecFlow to achieve significantly faster build times than both FilteredVamana and StitchedVamana, except on the YFCC-10M dataset with 200K labels, where building separate graph indices takes longer than FilteredVamana's single-index approach. However, VecFlow still lags behind CAGRA's single-index construction speed. Enhancing its index construction efficiency remains an important area for future research, particularly for large-scale applications requiring frequent index updates.

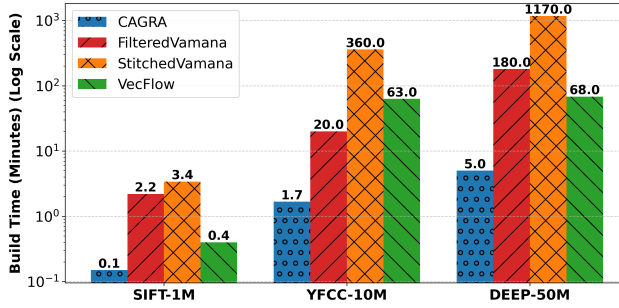


Fig. 18. Index construction time comparison.

6 Conclusion

Modern AI-based applications require ANNS with filtered search. However, indexing and search algorithms with filters on GPUs must be re-designed to achieve high performance potential. We present a new GPU-based indexing and search algorithm called VecFlow, which applies a hierarchical and divide-and-conquer design of filtered-ANNS for GPUs. Furthermore, VecFlow designs redundancy-bypassing IVF-Graph and interleaved-scan based IVF-BFS to achieve high compute and memory efficiency while attaining high accuracy. Evaluation on both public and semi-synthetic datasets show that VecFlow outperforms existing solutions by two orders of magnitude in QPS and establishes the new state-of-the-art for ANNS with filters.

Acknowledgments

We sincerely appreciate the insightful feedback from the anonymous reviewers. This research was supported by the National Science Foundation (NSF) under Grant No. 2441601. The work utilized the DeltaAI system at the National Center for Supercomputing Applications (NCSA) through allocation CIS240055 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. The Delta advanced computing resource is a collaborative effort between the University of Illinois Urbana-Champaign and NCSA, supported by the NSF (award OAC 2005572) and the State of Illinois. This work also utilized the Illinois Campus Cluster and NCSA NFI Hydro cluster, both supported by the University of Illinois Urbana-Champaign and the University of Illinois System.

References

- [1] Accessed: 04-13-2025. pgvector: Open-source vector similarity search for Postgres. <https://github.com/pgvector/pgvector>.
- [2] RAPIDS AI. 2025. cuVS. <https://github.com/rapidsai/cuvs>. Accessed: 2025-01-18.
- [3] Mohammad Aliannejadi, Hamed Zamani, Fabio Crestani, and W. Bruce Croft. 2018. Target Apps Selection: Towards a Unified Search Framework for Mobile Devices. In *SIGIR 2018*. 215–224.
- [4] Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. 2018. Approximate Nearest Neighbor Search in High Dimensions. *arXiv preprint arXiv:1806.09823* (2018).
- [5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD 1990*. 322–331.
- [6] Ben Landrum and Magdalen Dobson Manohar and Mazin Karjekar and Laxman Dhulipala. 2024. IVF2: Fusing Classic and Spatial Inverted Indices for Fast Filtered ANNS. https://big-ann-benchmarks.com/neurips23_slides/IVF_2_filter_Ben.pdf.
- [7] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517.
- [8] Philip A Bernstein, Siddharth Gollapudi, Suryansh Gupta, Ravishankar Krishnaswamy, Sepideh Mahabadi, Sandeep Silwal, Gopal R Srinivasa, Varun Suriyanarayana, Jakub Tarnawski, Haiyang Xu, et al. [n. d.]. Graph-based algorithms for nearest neighbor search with multiple filters. ([n. d.]).
- [9] Big-ANN. [n. d.]. NeurIPS’23 Competition Track: Big-ANN. <https://big-ann-benchmarks.com/neurips23.html>. Accessed: 2024.
- [10] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3772–3785. <https://doi.org/10.14778/3685800.3685805>
- [11] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A library for fast approximate nearest neighbor search*. <https://github.com/Microsoft/SPTAG>
- [12] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, Qiang Wang, and Wei Zhao. 2019. Vector and line quantization for billion-scale similarity search on GPUs. *Future Gener. Comput. Syst.* 99 (2019), 295–307.
- [13] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. <https://doi.org/10.1016/j.jalgor.2003.12.001>
- [14] Mostafa Dehghani, Hamed Zamani, Aliaksei Severyn, Jaap Kamps, and W. Bruce Croft. 2017. Neural Ranking Models with Weak Supervision. In *SIGIR 2017*. 65–74.
- [15] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web (Hyderabad, India) (WWW ’11)*. Association for Computing Machinery, New York, NY, USA, 577–586. <https://doi.org/10.1145/1963405.1963487>
- [16] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. *arXiv preprint arXiv:2401.08281* (2024). <https://doi.org/10.48550/arXiv.2401.08281>
- [17] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. 2020. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal* 29 (2020), 33–59.
- [18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. In *VLDB’19*.
- [19] Sumit Ganguly, Phillip B. Gibbons, Yossi Matias, and Avi Silberschatz. 1996. Bifocal sampling for skew-resistant join size estimation. *SIGMOD Rec.* 25, 2 (June 1996), 271–281. <https://doi.org/10.1145/235968.233340>
- [20] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR 2013*.
- [21] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB’99*. 518–529.
- [22] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. 2023. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023*. 3406–3416.
- [23] Google. 2022. Go beyond the search box: Introducing multisearch. <https://blog.google/products/search/multisearch/>. Accessed: 2025.
- [24] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. 2023. GGNN: Graph-Based GPU Nearest Neighbor Search. *IEEE Transactions on Big Data* 9, 1 (2023), 267–279. <https://doi.org/10.1109/TBDATA.2022.3161156>

- [25] Jiafeng Guo, Yixing Fan, Qingyao Ai, and W. Bruce Croft. 2016. A Deep Relevance Matching Model for Ad-hoc Retrieval. In *CIKM 2016*. 55–64.
- [26] Neha Gupta. 2021. Introduction to hardware accelerator systems for artificial intelligence and machine learning. In *Advances in Computers*. Vol. 122. Elsevier, 1–21.
- [27] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *CIKM '13*. 2333–2338.
- [28] HuggingFace. 2025. WikiANN dataset. <https://huggingface.co/2024anonymous/wiki-ann>.
- [29] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).
- [30] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. In Product Quantization for Nearest Neighbor Search. *TPAMI* 2011.
- [31] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *CoRR* abs/1702.08734 (2017). arXiv:1702.08734 <http://arxiv.org/abs/1702.08734>
- [32] Yannis Kalantidis and Yannis S. Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR 2014*. 2329–2336.
- [33] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 5110–5121.
- [34] Victor Lempitsky. 2012. The Inverted Multi-index. In *CVPR '12*. 3069–3076.
- [35] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (Vancouver, BC, Canada) (NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 793, 16 pages.
- [36] Junnan Li, Dongxu Li, Caiming Xiong, and Steven C. H. Hoi. 2022. BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA (Proceedings of Machine Learning Research, Vol. 162)*. PMLR, 12888–12900.
- [37] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data – Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2020), 1475–1488. <https://doi.org/10.1109/TKDE.2019.2909204>
- [38] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019).
- [39] LongChain. [n. d.]. LongChain: Build context-aware reasoning applications. <https://github.com/langchain-ai/langchain>. Accessed: 2025.
- [40] Yury A. Malkov and D. A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR* arXiv preprint abs/1603.09320 (2016).
- [41] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013*. 3111–3119.
- [42] Milvus-io. 2022. Milvus-docs: Conduct a hybrid search. <https://github.com/milvus-io/milvus-docs/blob/v2.1.x/site/en/userGuide/search/hybridsearch.md>. Accessed: 2025.
- [43] Bhaskar Mitra, Fernando Diaz, and Nick Craswell. 2017. Learning to Match using Local and Distributed Representations of Text for Web Search. In *WWW 2017*.
- [44] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *TPAMI* 2014 36, 11 (2014), 2227–2240.
- [45] Mohammad Norouzi and David J. Fleet. 2013. Cartesian K-Means. In *CVPR 2013*.
- [46] Hiroyuki Ootomo, Akira Naruse, Corey J. Nolet, Ray Wang, Tamas B. Fehér, and Y. Wang. 2023. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. *2024 IEEE 40th International Conference on Data Engineering (ICDE) (2023)*, 4236–4247.
- [47] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *VLDB J.* 33, 5 (2024), 1591–1615.
- [48] 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* 2, 3 (2024), 120.
- [49] Inc. Pinecone Systems. 2024. Overview. <https://docs.pinecone.io/docs/overview>. Accessed: 2025.
- [50] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models

- From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 8748–8763.
- [51] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.
 - [52] Harshit Sharma and Anmol Sharma. 2024. A Comprehensive Overview of GPU Accelerated Databases. *arXiv preprint arXiv:2406.13831* (2024).
 - [53] Harsha Simhadri. 2025. Big ANN Benchmarks. <https://github.com/harsha-simhadri/big-ann-benchmarks>. Accessed: 2025-01-18.
 - [54] Harsha Vardhan Simhadri, Martin Aumüller, Amir Ingber, Matthijs Douze, George Williams, Magdalen Dobson Manohar, Dmitry Baranchuk, Edo Liberty, Frank Liu, Ben Landrum, et al. 2024. Results of the Big ANN: NeurIPS'23 competition. *arXiv preprint arXiv:2409.17424* (2024).
 - [55] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *CoRR* abs/2105.09613 (2021).
 - [56] 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.
 - [57] Karthik V., Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. 2024. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *arXiv:2401.11324 [cs.DC]* <https://arxiv.org/abs/2401.11324>
 - [58] Christophe Van Gysel, Maarten de Rijke, and Evangelos Kanoulas. 2016. Learning Latent Vector Spaces for Product Search. In *CIKM '16*. 165–174.
 - [59] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. 2023. An Efficient and Robust Framework for Approximate Nearest Neighbor Search with Attribute Constraint. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
 - [60] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631* (2021).
 - [61] Weaviate. 2022. Weaviate Documentation: Filters. <https://weaviate.io/developers/weaviate/current/graphql-references/filters.html>. Accessed: 2025.
 - [62] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3152–3165. <https://doi.org/10.14778/3415478.3415541>
 - [63] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (Portland, OR, USA) (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2241–2253. <https://doi.org/10.1145/3318464.3386131>
 - [64] Lei Yu, Karl Moritz Hermann, Phil Blunsom, and Stephen Pulman. 2014. Deep Learning for Answer Sentence Selection. *CoRR* abs/1412.1632 (2014).
 - [65] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated Proximity Graph Approximate Nearest Neighbor Search and Construction. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 552–564.
 - [66] Hamed Zamani, Bhaskar Mitra, Xia Song, Nick Craswell, and Saurabh Tiwary. 2018. Neural Ranking Models with Multiple Document Fields. In *WSDM '18*.
 - [67] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 377–395. <https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi>
 - [68] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Vector Query Processing for Large Datasets Beyond {GPU} Memory with Reordered Pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 23–40.
 - [69] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1033–1044.

Received January 2025; revised April 2025; accepted May 2025