SVFusion: A CPU-GPU Co-Processing Architecture for Large-Scale Real-Time Vector Search

Yuchen Peng[1], Dingyu Yang[1], Zhongle Xie[1], Ji Sun[2], Lidan Shou[1], Ke Chen[1], Gang Chen[1]
[1]Zhejiang University, [2]Huawei Technologies Co., Ltd
{zjupengyc,yangdingyu,xiezl,should,chenk,cg}@zju.edu.cn,sunji11@huawei.com

ABSTRACT

Approximate Nearest Neighbor Search (ANNS) underpins modern applications such as information retrieval and recommendation. With the rapid growth of vector data, efficient indexing for real-time vector search has become rudimentary. Existing CPU-based solutions support updates but suffer from low throughput, while GPU-accelerated systems deliver high performance but face challenges with dynamic updates and limited GPU memory, resulting in a critical performance gap for continuous, large-scale vector search requiring both accuracy and speed. In this paper, we present SVFusion, a CPU-GPU collaborative framework for real-time vector search that bridges sophisticated GPU computation with online updates. SVFusion leverages a hierarchical vector index architecture that employs CPU-GPU co-processing, along with an adaptive two-phase vector caching mechanism to maximize the efficiency of limited GPU memory. It further enhances performance through pipeline-optimized task coordination that overlaps data transfers with computation, and a lightweight deletion handling strategy that supports concurrent operations without blocking ongoing tasks. Empirical results demonstrate that SVFusion achieves significant improvements in query latency and throughput, exhibiting a 20.9× speedup on average compared to baseline methods, while maintaining high recall for large-scale vector datasets under various workloads.

PVLDB Reference Format:
Yuchen Peng[1], Dingyu Yang[1], Zhongle Xie[1], Ji Sun[2], Lidan Shou[1], Ke Chen[1], Gang Chen[1]. SVFusion: A CPU-GPU Co-Processing Architecture for Large-Scale Real-Time Vector Search. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

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

1 INTRODUCTION

The rise of deep learning (DL) and large language models (LLMs) has driven the widespread use of high-dimensional vector embeddings for unstructured data such as text, images, and audio. Efficient retrieval of these embedding vectors is critical for tasks like personalized recommendation [25], web search [59], and LLM-based applications [34]. To meet strict low-latency and high-accuracy requirements under service-level objectives (SLOs), modern vector search systems increasingly employ Approximate Nearest Neighbor Search (ANNS) [36], specifically graph-based solutions [13, 17, 28, 38, 45] that balance accuracy and computational efficiency, in high-dimensional spaces. Building on this, recent work explores GPU-accelerated optimizations, including SONG [61], GANNS [57], and CAGRA [41], leveraging massive parallelism and high memory bandwidth to achieve significant speedups over CPU-based approaches, enabling scalable high-throughput vector search.

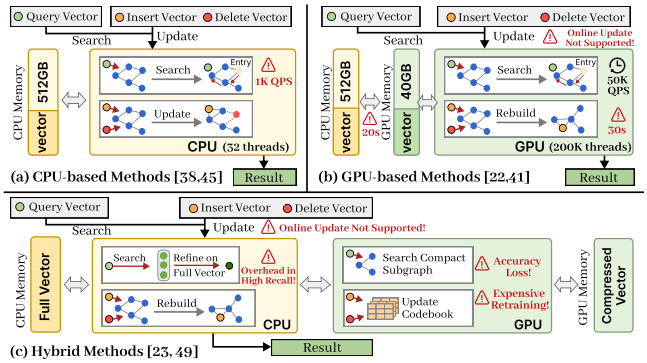


Figure 1: Comparison of existing methods for ANNS.

Despite significant advances in high-performance vector search, most existing systems are designed for static or slowly evolving datasets, making them ill-suited for dynamic environments where high-velocity embeddings must be indexed in real time [9, 10, 30, 35, 42]. As illustrated in Figure 1, large-scale streaming scenarios demand low-latency support for search, insertion, and deletion operations [11]. CPU-based methods [38, 45, 55] offer real-time update capabilities but suffer from high computational overhead and limited memory bandwidth, resulting in poor scalability as datasets grow. Conversely, GPU-based methods [22, 41] achieve high throughput for static indices, but incur significant transfer overhead (up to 20 seconds) when handling large-scale updates, with overall throughput dropping below 500 QPS [32]. Moreover, complete index reconstruction is impractical for high-velocity streaming workloads.

The discrepancy between CPU-based systems (update-friendly but performance-limited) and GPU-based systems (high-performance but static) highlights the need for a unified approach that leverages the strengths of both architectures. Recent hybrid methods [23, 49] attempt to bridge this gap by storing compressed vectors in GPU memory, but they introduce accuracy degradation unsuitable for high-recall requirements and rely on static quantization codebooks that require expensive retraining to accommodate streaming updates [20]. To summarize, there exist two challenges in tackling large-scale streaming ANNS:

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Challenge 1: GPU Memory Capacity Constraints. Modern vector data applications handle large-scale datasets with millions to billions of high-dimensional embeddings [5, 24], often exceeding GPU memory capacity, resulting in poor performance. For example, serving 35M vectors in 768 dimensions [7] requires at least 120GB, far surpassing the typical 40GB capacity of high-end accelerators like NVIDIA A100. This limitation necessitates frequent data movement between GPU and host memory, where the performance benefits of GPU parallelism are severely diminished by costly transfer overheads [47]. While compression techniques [29, 49] can reduce storage requirements, they typically degrade search accuracy or increase latency to maintain comparable results. This motivates the need for efficient CPU-GPU collaborative frameworks that effectively manage data placement.

Challenge 2: Performance Degradation Under Frequent Vector Updates. High-frequency vector updates at scale pose two critical performance issues. First, existing graph-based approaches suffer from limited *update throughput* during frequent insertions and deletions [28, 38], often relying on synchronous delete marking and periodic graph rebuilds, which cause latency spikes and hinder responsiveness. On GPU, this problem is exacerbated by frequent kernel synchronizations required for graph topology modifications [48], along with additional overhead from cross-device data transfers in hybrid CPU-GPU systems [48, 49]. Second, *search accuracy* degrades over time as frequent updates deteriorate graph structure, leading to suboptimal traversal paths [6, 41].

Regarding these challenges, a question arises: can we develop a hybrid approach that avoids data compression while efficiently supporting high-frequency vector updates? In this work, we propose SVFusion, a CPU-GPU cooperative framework for large-scale streaming vector search that bridges GPU computation with real-time index updates. SVFusion is built upon three key innovations: (1) **Hierarchical Vector Index Structure** that organizes vector data and graph topology across CPU and GPU memory tiers to maximize computational efficiency. We design a pipeline-optimized coordination mechanism that assigns the GPU tier to compute frequently accessed vectors for rapid parallel search, while the CPU tier manages dynamic updates and computes less frequently accessed data, maximizing the computational strengths of each processing unit. (2) **Workload-Aware Vector Placement Mechanism** that introduces a novel two-phase strategy to adaptively determine whether to transfer vectors between CPU and GPU, or to perform computations locally on CPU when the required vectors are not available in GPU memory. By evaluating both access patterns and graph structure, this mechanism strikes a better trade-off between data locality and transfer cost, delivering high efficiency and performance stability under dynamic workloads. (3) **Concurrent Update Handling Strategy** that maintains update throughput and search quality during continuous updates. We design a lightweight graph repair mechanism that performs localized reconstruction for vertices that are severely affected by deletions. We further propose a multi-version graph merging algorithm to enable synchronization between background consolidation and concurrent insertions, thus to maximize update efficiency without sacrificing index freshness.

Building on these techniques, SVFusion achieves high-performance real-time vector search with update capabilities across CPU-GPU architectures. In summary, we make the following contributions:

Table 1: Summary of notations.

Notation	Definition
X	the set of N D-dimensional vectors
q	a query vector
k	the number of returned results in ANNS
$G = (V, E)$	a proximity graph with vertex set V and edge set E
X_t	the dataset state at time t
X_t^G	the vector subset residing in GPU memory at time t
M	the GPU memory capacity (number of vectors)
λ_x	the number of future accesses for vector x
$F_{recent}(x, t)$	the recent access count of vector x at time t
$E_{in}(x)$	the number of in-neighbors for vector x
α, β	the weight parameters in the prediction function

- We present *SVFusion*, a novel CPU-GPU cooperative framework for streaming vector search that integrates hierarchical indexing with pipeline-optimized coordination to maximize computational efficiency across heterogeneous memory tiers. (§3).
- We design a workload-aware placement mechanism that efficiently balances data caching and computation decisions, optimizing the performance under dynamic workloads. (§4)
- We propose a concurrent update handling strategy that maintains index quality during continuous updates through lightweight repair and multi-version synchronization techniques. (§5)
- We implement *SVFusion* and conduct a comprehensive evaluation under diverse streaming workloads to verify its effectiveness and efficiency. Results show that our framework achieves up to 9.5× higher search throughput and 71.8× higher insertion throughput while maintaining superior recall rates compared to FreshDiskANN. (§6)

2 PRELIMINARY

We first define the Approximate Nearest Neighbor Search (ANNS) problem and its variant in a streaming workload (SANNS). We then introduce graph-based indexing techniques that serve as the basis for our framework. For clarity, Table 1 summarizes the frequently used notation.

2.1 Approximate Nearest Neighbor Search

Let $X = \{x_1, x_2, \ldots, x_N\} \subset \mathbb{R}^D$ denote a dataset consisting of N vectors, where each data point $x_i \in \mathbb{R}^D$ represents a D-dimensional real-valued vector in $\mathbb{R}^D$. The distance between any two vectors $u, v \in \mathbb{R}^D$ is denoted as $dist(u, v)$ with the Euclidean distance (i.e., the L2 norm). Given a query vector $q \in \mathbb{R}^D$ and a positive integer k ($0 < k < N$), the *nearest neighbor search* (NNS) problem [15] aims to identify a subset $U_{\text{NNS}} \subset X$ containing the k vectors that are closest to q, satisfying the condition that for any $x_i \in U_{\text{NNS}}$ and any $x_j \in X \setminus U_{\text{NNS}}$, it holds that $\text{dist}(x_i, q) \leq \text{dist}(x_j, q)$.

The exact NNS requires computing the distances between a query and all vectors, which is computationally prohibitive for large datasets [52]. To address this issue, most studies [16, 17, 31, 38] focus on approximately finding the nearest neighbors to achieve a balance between search accuracy and efficiency, called *approximate nearest neighbor search* (ANNS). Given an approximation factor $\epsilon > 0$, an ANNS algorithm returns an ordered set of k vectors $U_{\text{ANNS}} = \{x_1, x_2, \ldots, x_k\}$, sorted in ascending order of their distances to q. If x_i^* is the i-th nearest neighbor of q in X, the algorithm satisfies that $\text{dist}(x_i, q) \leq (1 + \epsilon) \cdot \text{dist}(x_i^*, q)$ for all $i = 1, 2, \ldots, k$. The quality of the approximation is typically evaluated using the recall metric [29],

defined as:

$$\text{Recall@}k = \frac{|U_{\text{ANNS}} \cap U_{\text{NNS}}|}{|U_{\text{NNS}}|} \tag{1}$$

where U_{NNS} denotes the ground truth set of the k nearest neighbors. A higher recall corresponds to a smaller ϵ, thus, a higher degree of approximation, reflecting better search accuracy.

2.2 Streaming ANNS

The problem of *streaming approximate nearest neighbor search* (SANNS) extends the traditional ANNS framework to accommodate dynamic datasets that evolve continuously through a sequence of operations. Let $X_0 = \{x_1, x_2, \ldots, x_{N_0}\} \subset \mathbb{R}^D$ denote an initial vector dataset with N_0 vectors, where each vector has a unique identifier. SANNS processes an ordered sequence of operations $O = \{o_1, o_2, ..., o_T\}$ over time, where T is the number of time steps or operations in the stream. At each time step $t \in [1, T]$, exactly one operation o_t is executed on the current dataset state X_{t-1}, resulting in a new state X_t. It supports the following four types of operations:

- **Build**(X_{init}): $X_t = X_{\text{init}}$, initializing the dataset state, where $X_{\text{init}} \subset \mathbb{R}^D$. This operation constructs the initial index structure from a given static dataset and may be used for periodic index rebuilding.
- **Search**(q, k): $X_t = X_{t-1}$. Given a query vector q and a positive integer k, this operation retrieves a set of k vectors from X_{t-1} that approximate the true k-nearest neighbors of q.
- **Insert**(x_t): $X_t = X_{t-1} \cup \{x_t\}$, incorporating a new vector into the dataset. This operation dynamically expands the searchable vector space to accommodate new data.
- **Delete**(x_t): $X_t = X_{t-1} \setminus \{x_t\}$, removing an existing vector $x_t \in X_{t-1}$. This operation contracts the search space by eliminating obsolete or irrelevant vectors.

In contrast to ANNS, which operates on static datasets with fixed ground truth, SANNS faces the challenges of maintaining search accuracy against varying ground truth on continuously evolving datasets, while ensuring efficient dynamic operations. To comprehensively evaluate SANNS methods, two primary metrics are considered. **Accuracy** is quantified using *recall* [21], which measures the approximation quality by comparing the retrieved results against the true nearest neighbors in the current dataset state. **Efficiency** is assessed in terms of *throughput* [45], defined as the number of queries and updates processed per second (QPS).

2.3 Graph-based index

We define a directed graph $G = (V, E)$, where V is the set of vertices with $|V| = N$, and each vertex $v_i \in V$ corresponds to a vector in the dataset. The edge set $E \subseteq V \times V$ encodes pairwise connections between vertices whose associated vectors are close in the underlying vector space. For any vertex $v_i \in V$, we denote its out-neighbor set as $N_{out}(v_i) = \{v_j \in V | (v_i, v_j) \in E\}$ and its in-neighbor set as $N_{in}(v_i) = \{v_j \in V | (v_j, v_i) \in E\}$.

Graph-Based ANNS and SANNS. Graph-based ANNS performs query processing by greedily traversing a proximity graph. Given a query vector q, the search begins from one or more entry points and iteratively explores neighboring nodes with decreasing distance to q until a stopping criterion is met [28]. This approach significantly reduces search complexity compared to exhaustive search while maintaining high recall. To support streaming scenarios (SANNS), the index must handle continuous insertions and deletions. At time t, let $G_t = (V_t, E_t)$ be the current graph. Given an insertion set I_t and deletion set $D_t \subseteq V_t$, the graph is updated to $G_{t+1} = (V_{t+1}, E_{t+1})$, where $V_{t+1} = (V_t \cup I_t) \setminus D_t$, and $E_{t+1} \subseteq V_{t+1} \times V_{t+1}$.

GPU Acceleration for Graph-based ANNS. ANNS can be substantially accelerated using GPUs. In practice, GPU-based methods typically adopt k-nearest neighbor (kNN) graphs with fixed out-degree to fully exploit GPU's massive parallelism, as variable degrees would lead to load imbalance and underutilized computing resources [22, 41]. The key acceleration involves storing critical metadata such as graph structures in GPU's high-bandwidth memory, and utilizing one thread-block per query rather than one thread per query [22] to enable extensive parallelization of core operations. This design achieves substantial performance improvements over CPU-based approaches. Details can be referred to in Appendix A.

3 OVERVIEW

We present SVFusion, a CPU-GPU collaborative framework for SANNS at scale. Our framework addresses the fundamental challenge of supporting real-time vector search through the following core designs. To overcome GPU memory capacity limitations, (1) we introduce a **hierarchical graph-based vector index** that maintains synchronized graph structures across CPU and GPU memory while preserving search performance; (2) we design a **workload-aware vector placement strategy** that manages data placement between CPU and GPU to minimize expensive PCIe transfer overhead. To further improve the performance under high-frequency updates, (3) we propose **lightweight graph repair and multi-version merging mechanisms** that enable high-throughput concurrent updates with minimal accuracy degradation through efficient graph maintenance. As shown in Figure 2a, SVFusion achieves these through three key components:

The **GPU module** accelerates vector search and updates by storing hot vectors and subgraphs in high-bandwidth memory (HBM). We extend CAGRA [41], a state-of-the-art GPU-based index originally designed for static datasets, into a dynamic, update-efficient index with specialized algorithms that support real-time vector search at scales exceeding GPU memory capacity.

The **CPU module** maintains the complete index graph with multi-version replicas and stores all vectors in DRAM[1]. It orchestrates GPU computation and synchronization, and enables concurrent access through version control. This design leverages the large capacity of the main memory to accommodate SANNS operations on massive vector datasets, achieving high-throughput operations with minimal synchronization overhead.

A dedicated **Cache Manager** manipulates data placement between CPU and GPU memory to optimize query performance during hybrid workloads. Unlike traditional caching strategies that

[1]Note that when the vector dataset exceeds available DRAM capacity, SVFusion incorporates a disk-based extension mechanism similar to FreshDiskANN [45], offloading cold or infrequently accessed vectors to disk storage.

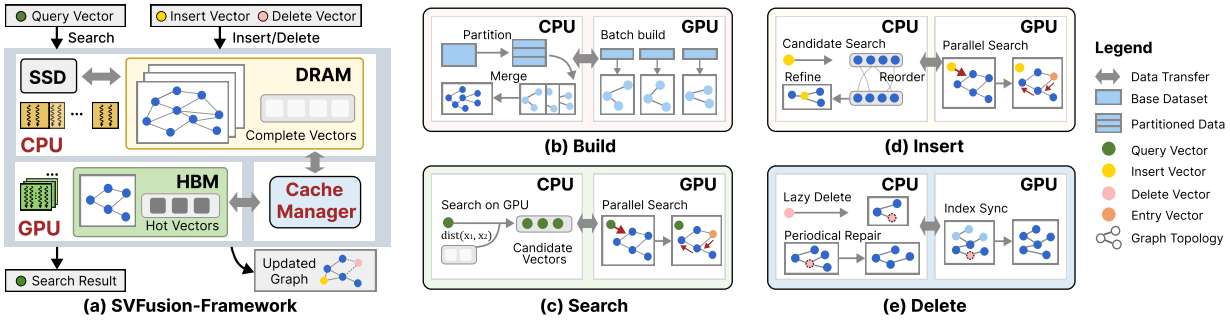


Figure 2: Overview of SVFusion.

rely on temporal locality [40], our cache manager exploits both vector access patterns and graph topology to dynamically load and evict vector entries, maximizing GPU memory residency for vectors accessed during graph traversal.

These components enable flexible memory management across HBM, DRAM, and even disk storage, forming a scalable foundation for streaming vector operations. By decoupling storage and computation, the system supports adaptive data placement and efficient access across memory tiers. This hybrid design overcomes the memory limits of GPU-only systems and the performance bottlenecks of CPU-only systems, offering broad applicability to large-scale, real-time vector search scenarios.

SANNS Workflow. Figure 2b-2e illustrates SVFusion's end-to-end SANNS workflow comprising the following four operations through coordinated CPU-GPU processing:

Build. This operation constructs the index using a **batch-based strategy** tailored for GPU scalability. For datasets that fit in GPU memory, SVFusion follows the CAGRA [41] construction pipeline to fully utilize GPU computation. For larger datasets, it partitions the data, builds subgraphs in parallel for each partition, and merges them into a unified index while preserving inter-partition connectivity. This design enables efficient GPU acceleration while scaling to datasets beyond memory constraints.

Search. Given a query, this operation performs graph traversal starting on the GPU. The **cache manager** checks whether the required vectors or subgraphs are already cached in GPU memory. If present, the search proceeds entirely on GPU; otherwise, SVFusion either transfers the data on demand from CPU or adaptively computes distances on CPU to reduce transfer overhead. Finally, the top-k results are obtained via sorting and returned to the user.

Insertion. This operation integrates new vectors while preserving graph structure and search quality. Each vector first undergoes a GPU-based search to identify candidate neighbors, with on-demand transfers if needed. The CPU then performs **heuristic reordering** to refine neighbor selection and applies **reverse edge insertion** to maintain bidirectional connectivity. After a batch of insertions, updated subgraphs are synchronized from DRAM to HBM to ensure consistency, with large-batch transfers amortizing the overhead. To support fault tolerance and manage memory capacity, SVFusion also periodically persists the graph and vector data to disk.

Deletion. This operation adopts a lightweight deletion mechanism to handle large-scale vector deletion requests. It is based on two complementary strategies: (1) **lazy deletion**, which marks vectors as deleted without immediate structural changes; (2) **asynchronous repair**, which restores graph connectivity through localized topology-aware repair and periodic global consolidation. These processes are designed to be non-blocking, allowing search and insertion operations to proceed concurrently without disruption. Detailed deletion handling mechanisms are described in §5.2.

4 GRAPH-BASED VECTOR SEARCH WITH CPU-GPU CO-PROCESSING

Large-scale dynamic vector search scenarios require graph-based indices to handle massive datasets with continuously evolving index structures and access patterns. While CPU-GPU co-processing offers promising performance gains, effectively managing memory allocation, coordinating task execution, and handling dynamic data placement remains challenging. To this end, SVFusion introduces a hierarchical index architecture, a workload-adaptive caching mechanism, and a pipeline-optimized coordination strategy for maximizing throughput.

4.1 Motivation

Hierarchical Memory Management. GPU memory capacity is limited in large-scale scenarios. While existing approaches compress data on GPU to reduce memory usage [49, 60], this inevitably introduces accuracy degradation. However, a key insight is that real-world access patterns exhibit skewness where certain vectors are accessed more frequently [39]. This observation motivates a hierarchical memory design that caches hot vectors without compression on GPU, achieving both high accuracy and memory efficiency.

Dynamic Cache Management: Access patterns continuously evolve in SANNS, causing previously hot vectors to become cold. Unlike traditional caching approaches [40] that fetch data before computation, vector access in CPU-GPU co-processing determines both where computation occurs and whether data transfer is needed. This motivates a workload-aware cache mechanism to make optimal vector placement decisions under evolving workloads.

Hybrid Processing Coordination: CPU-GPU collaborative ANNS involves multiple computational stages [49] across heterogeneous processors. However, effective coordination remains challenging due to synchronization overhead and resource underutilization. This calls for a pipeline design that minimizes synchronization while enabling concurrent execution.

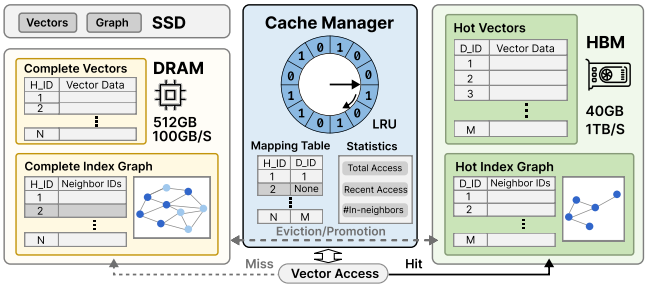


Figure 3: Hierarchical Index Structure in SVFusion.

4.2 Hierarchical Graph-based Vector Index

Figure 3 illustrates our hierarchical index structure that spans CPU and GPU memory tiers. Unlike existing GPU-accelerated ANNS methods that are constrained by limited GPU memory capacity when handling large-scale datasets [41, 61], our design leverages the complementary characteristics of CPU and GPU memory: GPU memory provides high bandwidth (1 TB/s) but limited capacity, while CPU memory offers large capacity (hundreds of GB) but lower bandwidth (100 GB/s). To bridge this bandwidth-capacity gap, our hierarchical design organizes vector data across multiple storage tiers, with a co-processing search algorithm that dynamically leverages both processors to achieve high-throughput ANNS on large-scale datasets.

Hierarchy Index Structure. Our index primarily exploits the CPU–GPU memory to fit datasets exceeding GPU memory limits, coordinated by the cache manager to maintain hot vectors uncompressed on GPU. It remains extensible to persistent storage for extreme-scale datasets that exceed main memory capacity.

Given a dataset of size N, the CPU memory maintains the complete dataset and graph structure $G = (V, E)$ with host-side identifiers ($h\_id \in [1, N]$). The GPU memory caches a selected subset of $M \ll N$ hot vectors with compact device-side identifiers ($d\_id \in [1, M]$). By maintaining hot vectors and their associated subgraphs on GPU, most computations can be served directly from high-bandwidth memory without accuracy loss. For extreme-scale datasets exceeding DRAM capacity, our framework seamlessly extends to incorporate persistent storage, creating a three-tier hierarchy. This design enables scaling from millions to billions of vectors while maintaining consistent performance for varying workloads.

The **cache manager** resides on GPU to coordinate data movement across these storage tiers during SANNS operations. It maintains a mapping table with the mapping function $mapping(h\_id) \rightarrow \{d\_id \mid \text{NONE}\}$, which returns the device identifier for GPU-cached vectors or *NONE* for CPU-resident vectors. This design leverages high-bandwidth GPU memory access for massive parallel vector lookups without the overhead of CPU-GPU synchronization.

CPU-GPU Vector Search. We present SVFusion with the search workflow in Algorithm 1. Unlike prior GPU-only approaches, SVFusion prioritizes GPU execution for cached data and dynamically offloads uncached computations to the CPU or triggers on-demand transfers. The search process begins with parallel query processing (Line 1), where graph traversal is performed independently for each query on GPU (Line 5) until the candidate pool remains unchanged. During neighbor expansion (Lines 7-11), our cache manager checks each neighbor vector's cache status (Line 8) and

Algorithm 1 ANNS using CPU-GPU Co-Processing

Input: graph index G, a batch of p queries $Q = \{q_1, q_2, \ldots, q_p\}$, k for top-k, and candidate pool size $L \geq k$
Output: $K := \cup_{i=1}^{p}\{K_i\}$, where K_i is the set of k-nearest neighbours for $q_i \in Q$

1: **for** each query $q_i \in Q$ **in parallel do**
2: $C_i \leftarrow$ InitCandidatePool(G, L) ▷ *random seed init*
3: **repeat**
4: $x_{curr} \leftarrow C_i$.GetNearest()
5: $X_i \leftarrow$ FetchNeighbors(x_{curr}, G) ▷ *on GPU*
6: $X_i^{GPU}, X_i^{CPU} \leftarrow \emptyset, \emptyset$ ▷ *initialize processing set*
7: **for** each $x \in X_i$ **in parallel do**
8: $d\_id \leftarrow$ mapping(x) **or** WAVP(x)
9: ▷ *vector placement called only on cache miss*
10: $X_i^{GPU} \leftarrow X_i^{GPU} \cup \{x | d\_id \neq \text{NONE}\}$
11: $X_i^{CPU} \leftarrow X_i^{CPU} \cup \{x | d\_id = \text{NONE}\}$
12: $D_i^{CPU} \leftarrow$ ParallelComputeDist(q_i, X_i^{CPU}) ▷ *on CPU*
13: $D_i^{GPU} \leftarrow$ ParallelComputeDist(q_i, X_i^{GPU}) ▷ *on GPU*
14: C_i.Update($D_i^{CPU} \cup D_i^{GPU}, X_i$) ▷ *update candidates*
15: **until** C_i unchanged from previous iteration
16: $K_i \leftarrow k$ nearest candidates to q_i from C_i

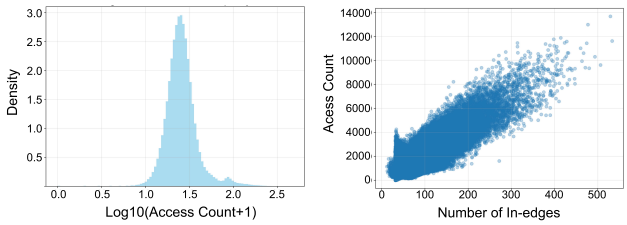
partitions computation on CPU/GPU. On cache hits, vectors are processed directly on GPU using high-bandwidth memory access. For cache misses, we propose a Workload-Aware Vector Placement (WAVP) algorithm that determines whether to promote vectors to GPU or perform computations directly on CPU. This results in partitioning neighbors into X_i^{GPU} and X_i^{CPU} for parallel distance computation on both processors (Lines 10-13). This adaptive strategy balances throughput and memory efficiency, enabling scalable and responsive vector search under GPU memory constraints.

4.3 Workload-Aware Vector Placement

To address the dynamic access patterns of streaming workloads, we propose an adaptive vector placement strategy for optimizing data locality across GPU and CPU memory and efficient computation during ANNS. Rather than static allocation, it prioritizes vectors based on run-time access statistics to maintain high query throughput under varying workload conditions.

Let $X_t = \{x_1, x_2, \ldots, x_N\}$ denote the full set of vectors at timestamp t. Due to GPU memory constraints, only a subset $X_t^G \subseteq X_t$ with $|X_t^G| = M \ll N$ resides in GPU memory, while the remaining vectors are kept in CPU memory. When accessing a vector $x \notin X_t^G$, the **core challenge** of CPU-GPU co-processing is to decide the optimal placement for each vector access: (1) promoting vectors from CPU to GPU memory, or (2) executing distance computations directly on CPU. Always promoting every missed vector incurs excessive data transfer overhead, particularly for infrequently accessed items, while relying solely on CPU computation fails to leverage the parallelism and high bandwidth of the GPU, resulting in suboptimal performance.

To determine the optimal choice between these two strategies for each vector access not resident on GPU, we need to quantify the



(a) Effects of recent access. **(b) Effects of in-neighbors.**

Figure 4: Evaluation of factors for vector access.

trade-off between computational savings and data transfer overhead. We define the following gain function:

$$\text{gain}(x) = \lambda_x \cdot (T_{\text{CPU}} - T_{\text{GPU}}) - T_{\text{transfer}} \tag{2}$$

Here, λ_x represents the number of future accesses for vector x within a specified time window. T_{CPU} and T_{GPU} represent the average time required to compute distances on CPU and GPU, respectively. T_{transfer} is the amortized cost of transferring a vector to GPU memory, typically calculated based on a batch size of 2048 to reduce the communication overhead [8] per vector. When $\text{gain}(x) > 0$, it implies that the expected cumulative speedup from GPU acceleration outweighs the cost of data transfer, justifying the promotion of x to GPU memory.

The key challenge in implementing this gain function lies in obtaining λ_x during ANNS. Since future access frequency is inherently unobservable at decision time, we develop a prediction function F_λ to estimate λ_x based on observable runtime information.

Prediction Function Design. An effective prediction of λ_x requires properties that correlate with future access. As graph-based ANNS relies on graph traversal for vector retrieval, access patterns naturally depend on both temporal factors and graph structure. We therefore analyzed vector access with respect to both properties during ANNS on the MSTuring [3] dataset.

Figure 4a shows the distribution of recent access counts, denoted as $F_{recent}(x,t)$, for vector x at time t. Due to temporal locality in access workloads, $F_{recent}(x,t)$ serves as a strong predictor of future accesses. However, we observe that 50% of vectors account for 95% of accesses, creating a broad "medium-hot" distribution rather than the extreme long-tail pattern typical of traditional caching research [27], making naive LRU caching suboptimal.

Figure 4b reveals that vectors with more in-neighbors, denoted as $E_{in}(x)$, receive consistently higher access counts. This correlation is intuitive: vectors serving as structural hubs are more likely to appear on future search paths, as graph traversal naturally routes through these high-connectivity nodes. Such structural importance suggests we should prioritize caching these high-connectivity vectors in GPU memory to maximize cache hit rates.

Based on these observations, we design a prediction function to estimate future access counts by combining temporal locality $F_{recent}(x,t)$ and structural connectivity $E_{in}(x)$ as key factors. We adopt a linear combination approach for its simplicity and interpretability, though other fitting methods (e.g., polynomial) could also be explored:

$$F_\lambda(x) = \alpha \times F_{recent}(x,t) + \beta \times \log(1 + E_{in}(x)) \tag{3}$$

Algorithm 2 Workload-Aware Vector Placement for Cache Miss

Input: vector x not resident in GPU, vector subset in GPU Memory X_t^G, clock pointer $clock$, reference bits ref
Output: a valid device identifier d_id or NONE

1: **if** $F_\lambda(x) \le \theta$ **then**
2: **return** NONE
3: $F_{min} \leftarrow \min\{F_\lambda(x_i) \mid x_i \in X_t^G, ref[i] = 0\}$
4: **while** true **do**
5: $x_{curr} \leftarrow X_t^G[clock]$
6: **if** $ref[clock] = 0$ **and** $F_\lambda(x_{curr}) = F_{min}$ **then**
7: replace(x_{curr}, x)
8: **return** mapping(x_{curr})
9: **else if** $ref[clock] = 1$ **then**
10: $ref[clock] \leftarrow 0$
11: $clock \leftarrow (clock + 1) \bmod |X_t^G|$

where parameters α and β control the relative importance of temporal and structural features, respectively. In practice, we fix the ratio between α and β and tune their values based on performance evaluation. The cache manager maintains $F_{recent}(x,t)$ using a sliding window mechanism that periodically decays older accesses, and updates $E_{in}(x)$ as the graph topology evolves.

Theoretical Analysis. We provide theoretical justification that our prediction function can effectively guide caching decisions. We model the actual future access count λ_x as:

$$\lambda_x = \lambda_1 \cdot F_{recent}(x,t) + \lambda_2 \cdot \log(1 + E_{in}(x)) + \lambda_3 \tag{4}$$

where $\lambda_1, \lambda_2 > 0$ are weight parameters and λ_3 is a bias term. The logarithmic transformation is used to diminish extremely high values of $E_{in}(x)$. By setting $\alpha = \lambda_1$ and $\beta = \lambda_2$, we have $\lambda_x = F_\lambda(x) + \lambda_3$. This indicates that ranking vectors by $F_\lambda(x)$ is equivalent to ranking by expected future accesses.

Theorem 4.1 (Decision Consistency). *Given the cost ratio of transfer overhead to computation gain, denoted as* $\rho = \frac{T_{transfer}}{T_{CPU} - T_{GPU}}$, *there exists a threshold* $\tau = \rho - \lambda_3$ *such that:*

$$F_\lambda(x) > \tau \iff gain(x) > 0 \tag{5}$$

Proof. To establish the equivalence, we need to show that $F_\lambda(x) > \tau$ if and only if $\text{gain}(x) > 0$. Recall that $\text{gain}(x) > 0$ when the benefit of GPU caching exceeds the transfer cost, which occurs when $\lambda_x > \rho = \frac{T_{\text{transfer}}}{T_{\text{CPU}} - T_{\text{GPU}}}$. Since we have $\lambda_x = F_\lambda(x) + \lambda_3$, the condition $\lambda_x > \rho$ becomes $F_\lambda(x) + \lambda_3 > \rho$, which is equivalent to $F_\lambda(x) > \rho - \lambda_3$. By setting $\tau = \rho - \lambda_3$, we obtain $F_\lambda(x) > \tau \iff \lambda_x > \rho \iff \text{gain}(x) > 0$. □

The WAVP Algorithm. Based on the prediction function, we design the workload-aware vector placement algorithm illustrated in Algorithm 2. Given a vector x not currently in GPU memory, the algorithm returns either the device identifier for caching or *NONE* to indicate CPU-based execution. The placement algorithm consists of two phases: *selective prefetching* and *predictive replacement*.

The *selective prefetch* phase (Lines 1-2) determines whether to promote vector x to GPU memory. We initialize the threshold $\theta = \frac{T_{\text{transfer}}}{T_{\text{CPU}} - T_{\text{GPU}}}$, which corresponds to the minimum frequency required

for GPU promotion to be beneficial. When $F_\lambda(x) \leq \theta$, the system keeps x on CPU for in-place computation.

The *predictive replacement* phase (Lines 3-11) extends a clock-sweep mechanism with prediction-guided victim selection when GPU memory is full. As discussed above, access patterns in ANNS workloads make traditional LRU-based methods prone to thrashing, where specific vectors are evicted but re-fetched in a short interval. We address this by incorporating predicted access frequency into victim selection: among vectors with zero reference bits, we evict the one with the minimum $F_\lambda(x)$ (Line 6). This design strictly controls the eviction decision to prevent vectors with moderate access frequency from being evicted, avoiding redundant data transfer.

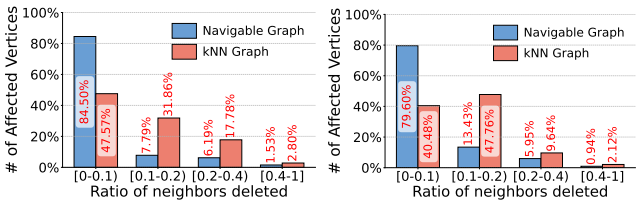
4.4 Pipeline-Optimized Task Coordination

SVFusion maximizes hardware utilization through an efficient task coordination mechanism that orchestrates computations across CPU and GPU resources. In contrast to prior work that relies on sequential execution models or employs fine-grained GPU optimizations [49, 60], SVFusion introduces a pipeline-based design built upon the hierarchical index structure outlined in §4.2. This architecture enables efficient and fine-grained distribution of workload across heterogeneous resources. To further minimize synchronization overhead and improve overall throughput, SVFusion integrates three key coordination optimizations:

Batch-oriented Vector Processing. We employ a batching strategy for both CPU and GPU computations to maximize hardware utilization. Instead of processing retrieved vectors immediately during neighbor expansion, vectors are accumulated into execution-specific batches—X_i^{CPU} and X_i^{GPU} in Algorithm 2—based on their assigned compute device. This deferred execution model improves resource utilization by aggregating sufficient workloads prior to dispatching distance computations, effectively amortizing the overhead associated with GPU kernel launches, SIMD-accelerated CPU operations, and cross-device data transfers.

Concurrent CPU-GPU Execution. SVFusion fully harnesses heterogeneous parallelism across CPU and GPU through two key techniques. First, it employs asynchronous data transfers with separate CUDA streams to overlap computation and communication, enabling both processors to operate concurrently without mutual blocking. Second, GPU memory is partitioned into multiple independent segments, each guarded by lightweight spinlocks. Vectors are mapped to segments via a hash of their identifiers, enabling fine-grained synchronization and minimizing thread contention. This design eliminates the need for global locks, effectively addressing a common bottleneck in massively parallel GPU workloads.

Adaptive Resource Management. SVFusion dynamically balances computational workload between CPU and GPU based on runtime status. As described in §4.3, we use the threshold θ to guide vector placement. The key innovation lies in adaptive threshold tuning: SVFusion continuously monitors system metrics—including dataset size, device utilization, and query latency—to adjust them at runtime. This adaptive strategy ensures efficient resource utilization, mitigates processor-side bottlenecks, and maintains consistent throughput under diverse workload patterns and hardware configurations.



(a) Wikipedia **(b) MSMARCO**

Figure 5: Distribution of the number of deleted neighbors.

5 INDEX UPDATE

5.1 Vector Insertion

SVFusion achieves high-throughput insertion via GPU acceleration and meanwhile maintains the index quality. Traditional CPU-based methods rely on multi-threading but are constrained by small batch sizes, typically limited to around 100 vectors [51] due to the overhead of parallel neighbor search and distance-based reordering. In contrast, our framework leverages the massive parallelism of GPUs to process substantially larger batches (typically 10,000 as analyzed in §6.4) through greedy neighbor search.

We further design a rank-based candidate reordering strategy. Instead of relying on expensive distance computations, we approximate the importance of candidate edges by their ranks. The rank is computed by counting the "detourable routes" [41] for each candidate edge, namely routes that can bypass the edge through two-hop paths via previously selected neighbors. This approach requires only $O(|C| \log |C|)$ complexity to traverse the candidate set C, significantly reducing computational overhead. During reverse edge insertion, we employ fine-grained atomic operations and thread-local buffers to handle concurrent graph updates, achieving better scalability than coarse-grained locking while maintaining graph consistency. The details of the insertion process are presented in Algorithm 5 in the Appendix.

5.2 Vector Deletion

The deletion process removes vertices from the index while preserving index consistency and system availability. To this end, SVFusion decouples deletion into three coordinated steps: (1) logical deletion to mark vertices as inactive without disrupting ongoing queries, (2) localized topology-aware repair to quickly restore essential graph connectivity with minimal overhead, and (3) periodic global consolidation to clean up deleted vertices while preserving index quality.

5.2.1 Logical Deletion. SVFusion employs a **lightweight, lazy deletion strategy** to efficiently support high-frequency removal operations without disrupting system performance. Instead of immediately modifying the graph structure, SVFusion marks deleted vertices using a bitset, treating them as logically removed. These marked vertices are transparently skipped during subsequent insertions, deletions, and query traversals. By deferring the costly operations of structural repair and neighbor updates, this strategy minimizes deletion latency and avoids synchronization bottlenecks.

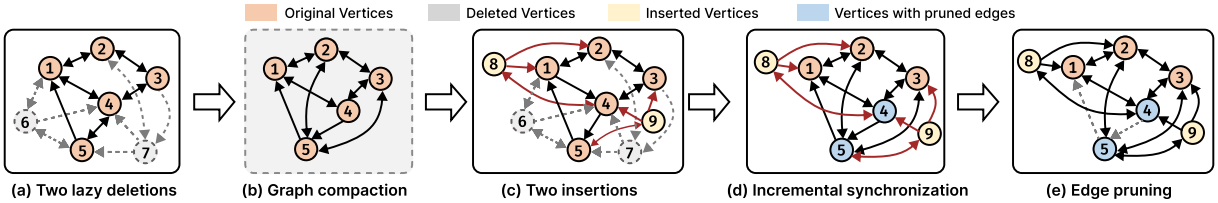


Figure 6: Illustrating the concurrent update pipeline in SVFusion.

5.2.2 Affected Vertices Repairing. While logical deletions provide a lightweight mechanism for removing vectors with minimal latency, their accumulation progressively impairs the structural integrity of the graph and degrades search performance. Previous deletion handling methods [45] are primarily designed for navigable graphs [33], where deletion impacts are typically localized to high-degree hub vertices. These methods defer repair operations until the deletion ratio exceeds a global threshold (e.g., 20%), then perform expensive global graph consolidation. However, kNN graphs exhibit fundamentally different structural properties against this design assumption. Unlike navigable graphs with degree distributions, kNN graphs maintain uniform degree constraints, causing deletion impacts to spread evenly across all vertices rather than concentrating on specific hubs. To quantify this structural difference, we conduct an experimental analysis comparing deletion impact distributions. As shown in Figure 5, after deleting 10% of vertices, kNN graphs exhibit significantly higher neighbor deletion ratios: the proportion of affected vertices with 10-40% deleted neighbors is 3.26× higher, and those with over 40% deleted neighbors are 2.04× higher compared to navigable graphs. Therefore, widespread connectivity degradation severely affects search quality long before global repair thresholds are reached, necessitating continuous incremental repair that targets critical vertices as deletions accumulate.

Localized Topology-Aware Repair. SVFusion addresses the problem of rapid degradation through a localized topology-aware repair strategy. Rather than immediately cleaning up deleted vertices, the approach leverages the deletion bitset to quickly identify severely affected vertices V^L where more than 50% of neighbors have been deleted, and applies lightweight repair only to these critical vertices. Given a vertex $v \in V^L$, in contrast to the expensive consolidation that connects all vertices in $N_{out}(p)$ to v for each deleted neighbor p, our lightweight repair selects at most c vertices from $N_{out}(p)$ to connect to v. The constant c is small (we use $c = 8$), ensuring the number of added edges is $O(cR)$, which is much smaller than the $O(R^2)$ during consolidation, where R is the graph degree. After adding the selected edges in distance order, we apply robust pruning[17] with relaxed filtering conditions to maintain the neighbor count of v constrained by R. This incremental approach preserves graph quality with minimal overhead compared to performing expensive repairs on all affected vertices.

Global Consolidation. When the deletion ratio exceeds a predefined threshold (e.g., 20%), SVFusion globally consolidates all affected neighborhoods by aggregating candidates from outgoing neighbors of the deleted vertices. This consolidation runs asynchronously in the background, allowing concurrent insertions and searches to continue without interruption. To synchronize with foreground operations, the process operates on a graph snapshot, with a graph merging mechanism handling the coordination between different graph versions.

5.3 Concurrent Update

To maximize system throughput, SVFusion employs a multi-version design where background graph consolidation operates on snapshots while foreground insertions continue on the active graph. This decoupling enables concurrent updates and avoids blocking during the time-consuming consolidation process.

Our multi-version mechanism works by creating a new version whenever a background task (e.g., consolidation) starts, duplicating the current graph as a read-only snapshot for task processing. Upon task completion, the results are merged back into the active graph. This background merging creates additional versions for concurrent updates, potentially triggering further merge operations. To prevent unbounded memory growth, we limit concurrent versions and defer new updates when this threshold is reached. Without proper synchronization, however, this merging process can lead to data inconsistency. We therefore propose a lightweight two-phase merging algorithm while maintaining high update throughput.

Incremental Sub-graph Appending. When consolidation begins at time t_0 using snapshot G_{t_0}, ongoing insertions update the active graph to G_{t_1} by t_1. We identify newly inserted vertices $V_{new} = V(G_{t_1}) \setminus V(G_{t_0})$ and directly append the corresponding subgraph $G_{new} = G_{t_1}[V_{new}]$ to the consolidated graph G'_{t_0}.

Reverse Edge Integration. During the interval $[t_0, t_1]$, newly inserted vertices also establish incoming connections to vertices in $V(G_{t_0})$. These reverse edges are essential for maintaining graph connectivity, but would be lost if we only appended the new subgraph in the merging process. To preserve these connections, we maintain an edge update log L that records triplets (v, v_{new}, d), where $v \in V(G_{t_0})$ is an existing vertex, $v_{new} \in V_{new}$ is a newly added vertex, and $d = dist(v, v_{new})$ represents the computed distance.

For each vertex with recorded reverse edges, we add these new connections to its current neighborhood. When this causes a vertex to exceed the degree constraint, we apply α-RNG pruning [17] to select the most valuable connections based on the pre-computed distances. This effectively reduces edge count while maintaining graph connectivity.

Example 1. *Figure 6 illustrates the process of handling concurrent updates. In (a), vertices 6 and 7 are logically deleted at time T_1. During background compaction (b), connections to deleted vertices are repaired. Based on the snapshot in (a), vertices 8 and 9 are inserted at time T_2 shown in (c). To synchronize the two graphs in (b) and (c), the incremental merging (d) appends the new subgraph and integrates reverse edges from the new to existing vertices. Finally, edge pruning*

Table 2: Statistics of experimental datasets.

Datasets	Data Volume	Size (GB)	Dim.	#Query
Wikipedia [7]	35,000,000	101	768	5,000
MSMARCO [4]	101,070,374	293	768	9,376
MSTuring [3]	200,000,000	75	100	10,000

(e) removes two redundant connections to maintain degree constraints while preserving graph connectivity.

6 EVALUATION

In this section, we first introduce the evaluation setup and then present a detailed analysis of the evaluation results.

6.1 Experimental Setup

Hardware Configuration. We conducted our experiments using 16 threads on a machine running Ubuntu 18.04.6 LTS with an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz. The machine has 64 CPU cores, 376 GB of DDR4 DRAM main memory, and an NVIDIA A100 GPU with 40 GB of device memory.

Datasets. We evaluate SVFusion on three widely-used real-world datasets with diverse characteristics: Wikipedia [7], MSMARCO [4], and MSTuring [3]. These datasets, detailed in Table 2, span different dimensions and content, which have been widely used in existing studies to benchmark ANNS systems [12, 44]. Query sets are randomly sampled from each dataset, with ground truth computed via an exhaustive linear scan over the complete dataset.

Workloads. Our evaluation covers diverse streaming workload patterns, each consisting of a collection of vectors and an operation sequence as defined in subsection 2.2. The workloads differ in: (i) the interval between insertions and deletions, and (ii) the spatial correlation among vectors deleted in a single step. To ensure benchmark comparability, we adopt three representative workload patterns from the *2023 Big ANN Challenge* [44]:

- **SlidingWindow**: The dataset is randomly partitioned into $T_{max} = 200$ equal-sized segments. One segment is inserted per step from $T = 1$ to T_{max}. Starting at step $T = T_{max}/2 + 1$, vectors inserted $T_{max}/2$ steps prior are deleted. We begin evaluating search performance at step $T_{max}/2 + 1$ when the index size stabilizes. This workload simulates a vector index that maintains recent data within a fixed time window.
- **ExpirationTime**: The dataset is partitioned into vectors with different lifetimes: short-term (10 steps), long-term (50 steps), and permanent (100 steps), with proportions 10:2:1, respectively. At each step, we insert a $1/T_{max}$ fraction of the dataset, randomly assign lifespan classes, and delete expired vectors. This workload evaluates index performance under varying data lifetimes.
- **Clustered**: The dataset is partitioned into 64 clusters via k-means clustering across 5 rounds. Each round alternates between insertion and deletion phases: first inserting random proportions from each cluster sequentially, then deleting random proportions of active points from each cluster. Since spatially proximate points are inserted and deleted simultaneously, this workload induces extreme distributional shifts in the active point set.

Implementation and Baselines. We implement SVFusion by extending the cuVS 24.02 library [2] following the architecture in Figure 2. Implementation details and the Appendix are available in our GitHub repository[2]. We compare SVFusion with the following baselines:

- **HNSW** [38] is a prominent graph-based ANNS algorithm. Its prevalence in industrial applications and extensive academic research underscores its significance. We use $M = 48$, and $ef\_construction = ef\_search = 128$.
- **FreshDiskANN** [45] is the state-of-the-art graph-based ANNS method that supports dynamic scenarios on memory or SSDs. We configure FreshDiskANN with the same settings as in the original paper [45]. Specifically, we use $R = 64$, $l_b = l_s = 128$, $\alpha = 1.2$ for vector insertion, and $l_d = 128$ for vector search.
- **CAGRA** [41] is NVIDIA's high-performance ANNS algorithm designed for GPUs. It achieves superior performance but encounters GPU memory limitations for large-scale datasets.
- **PilotANN** [23] is a recent hybrid architecture that effectively enables GPU acceleration for CPU-based ANNS, but it does not support dynamic updates.

Existing GPU-based methods either support only static construction [41, 60, 61] or have limited dynamic capabilities constrained by GPU memory [48], with no open-source implementations available. For fair comparison on streaming workloads, we create GPU-accelerated versions of FreshDiskANN and HNSW by offloading distance computations to GPU.

Evaluation Metrics. The design goal of SVFusion is to achieve efficient vector updates while ensuring high search accuracy. Therefore, we focus on three key aspects: **update performance, search performance, and memory efficiency**. For *update performance*, we measure update throughput, representing the number of insertions/deletions processed per second under streaming workloads. For *search performance*, we evaluate both efficiency and accuracy, where efficiency is measured by search throughput, while accuracy is quantified using Recall@k, with k=10 by default. For *memory efficiency*, we measure the **GPU cache effectiveness** by tracking the cache miss rate, defined as the percentage of vector accesses during ANNS that require fetching data from CPU memory rather than the GPU cache. A lower miss rate indicates better cache residency for frequently accessed vectors.

6.2 SANNS Performance

In this subsection, we aim to demonstrate the overall performance of SVFusion and baselines for SANNS. Figure 7 presents results across five workloads on three datasets. We report the recall, search throughput, insert throughput, and GPU miss rate at each timestep. Note that delete throughput is not included because all methods use tombstones to record the deletions at each step, making instantaneous throughput measurements less meaningful. Therefore, Table 3 provides a comprehensive breakdown of total execution time across all operations. Overall, SVFusion achieves superior performance across all operations: up to 9.5× higher search throughput, 71.8× higher insertion throughput, and 9.59× faster deletion than FreshDiskANN, while maintaining the highest recall rates.

Recall Analysis. SVFusion consistently delivers high search quality across all streaming workloads, achieving Recall@10 scores between 91% and 96%, which are comparable to or exceed those of baseline methods. Notably, it outperforms FreshDiskANN with an

[2]https://github.com/zjuDBSystems/svfusion

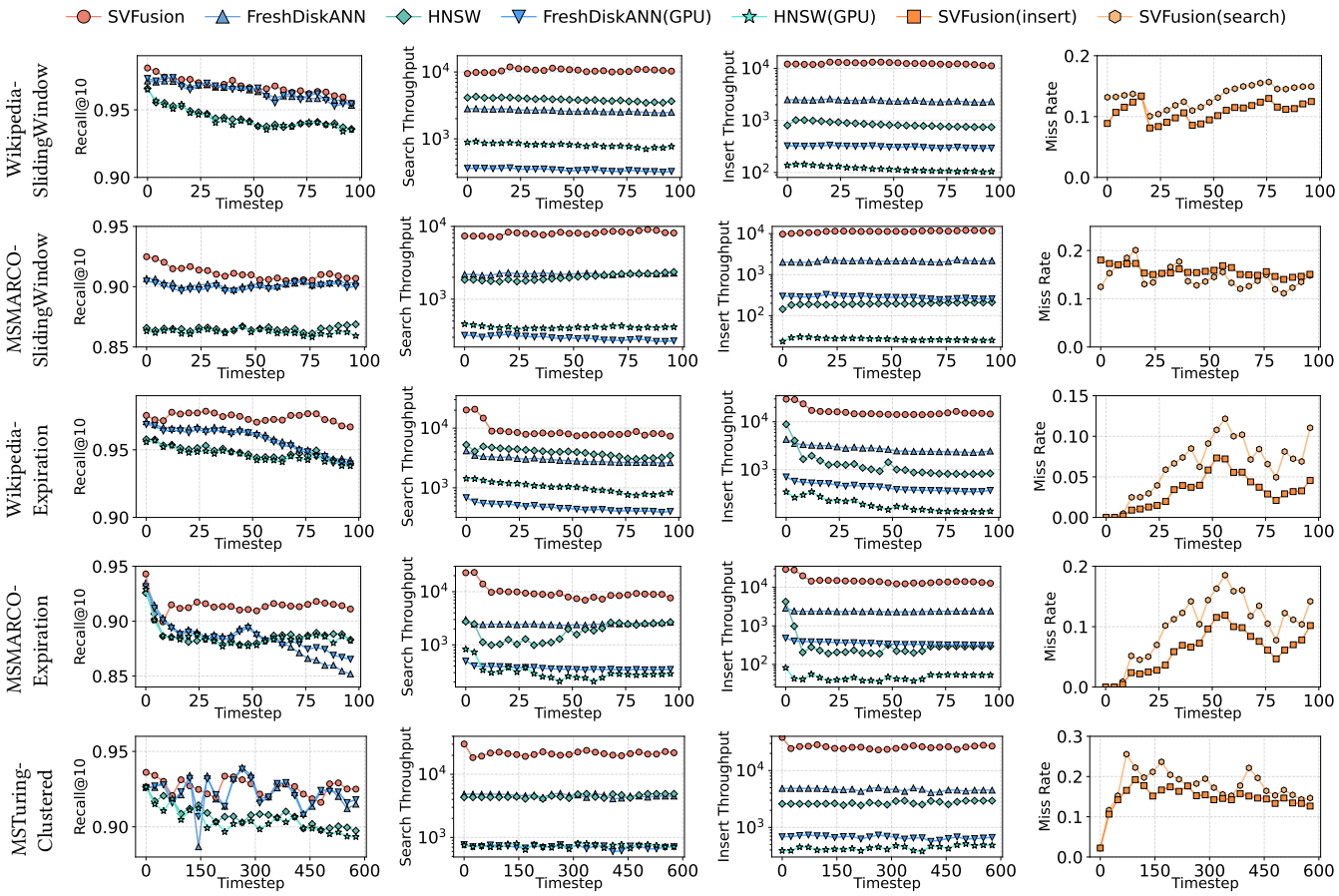


Figure 7: Comparison of recall, search throughput, insert throughput, and miss rate for five various workloads.

average recall improvement of 0.4–3.4%. This improvement stems from our **localized topology-aware repair mechanism**, which continuously detects and fixes critical connectivity issues in the graph topology immediately, preventing the accumulation of structural degradation. In contrast, HNSW exhibits the lowest recall among all methods due to the absence of any repair mechanism during deletions, resulting in persistent structural fragmentation. We also observe that all methods exhibit recall fluctuations due to continuous graph structure updates. These fluctuations are particularly pronounced in the clustered workload, where concentrated regional deletions can severely impact local connectivity.

Throughput Analysis. Figure 7 shows that SVFusion achieves the highest throughput for both search and insertion operations. Specifically, it delivers on average 20.9× higher search throughput and 3.5× higher insertion throughput compared to FreshDiskANN and HNSW. A noteworthy observation is that GPU-accelerated baselines (i.e., FreshdiskANN(GPU) and HNSW(GPU)) perform 5.4-7.2× slower than their CPU counterparts when datasets exceed GPU memory capacity. This is due to the expensive data transfers between the CPU and GPU that outweigh the computational advantages of GPU acceleration. In contrast, SVFusion avoids this pitfall through our adaptive caching algorithm. We also observe that during initial timesteps of workloads that start from an empty dataset (*ExpirationTime* and *Clustered*), search throughput peaks at 40K due to underutilized GPU memory with low miss rates (<1%).

Miss Rate Analysis. We report GPU cache miss rates separately for insertion and search operations to better characterize memory behavior. Although the GPU memory comprises only a small fraction of the overall dataset, SVFusion sustains low and stable miss rates throughout the execution of dynamic workloads. This demonstrates the effectiveness of our **cache placement strategy** in capturing temporal access patterns and avoiding unnecessary evictions, even under frequent updates. Notably, insertion miss rates are on average 19.6% lower than search miss rates. This difference is primarily due to the workload composition: search queries account for only 15–25% of total operations in our experiments, allowing the cache manager to better capture insertion access patterns.

Deletion. As shown in Table 3, the lazy deletion time (tombstone marking) is comparable across all methods. For graph repair efficiency, SVFusion significantly outperforms FreshDiskANN through our localized topology-aware repair strategy. Instead of relying on costly periodic graph reconstructions, SVFusion incrementally addresses connectivity issues. This continuous maintenance prevents structural degradation while reducing both the complexity and frequency of consolidation, resulting in faster deletion processing.

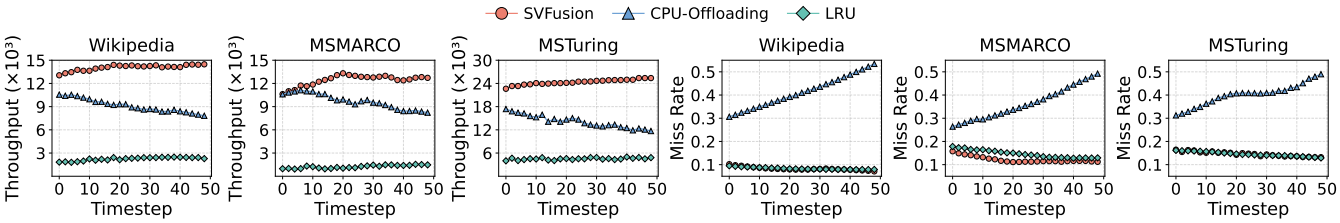


Figure 8: Comparison of different replacement strategies on update throughput and miss rate.

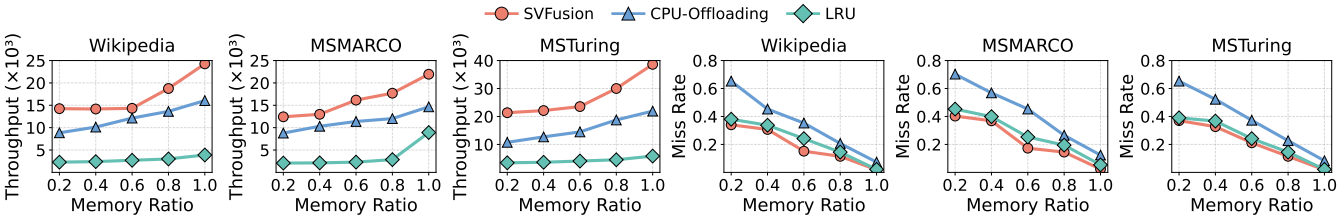


Figure 9: Impact of different GPU memory ratios on update throughput and miss rate.

Table 3: Comparison of recall and running time (in seconds).

Workload	Method	Recall@10	Search	Insert	Delete	
					Mark	Repair
Wikipedia -SW	FreshDiskANN	96.38	98	6291	11	14795
	HNSW	94.41	55	18026	84	–
	SVFusion	**96.74**	**29**	**895**	**8**	**1207**
MSMARCO -SW	FreshDiskANN	90.21	215	7058	16	18214
	HNSW	86.49	162	77666	86	–
	SVFusion	**91.12**	**56**	**1336**	**11**	**862**
Wikipedia -Exp	FreshDiskANN	95.89	172	11142	**21**	2023
	HNSW	94.71	88	27537	149	–
	SVFusion	**97.39**	**58**	**1891**	24	**1082**
MSMARCO -Exp	FreshDiskANN	88.39	379	12572	27	2463
	HNSW	88.38	392	117274	152	–
	SVFusion	**91.39**	**102**	**2065**	**24**	**752**
MSTuring-Clustered	FreshDiskANN	92.19	845	19122	41	21165
	HNSW	90.75	859	94776	188	–
	SVFusion	**92.74**	**167**	**3234**	**29**	**2183**

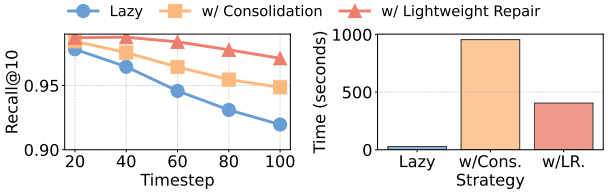


Figure 10: Effect of deletion strategies.

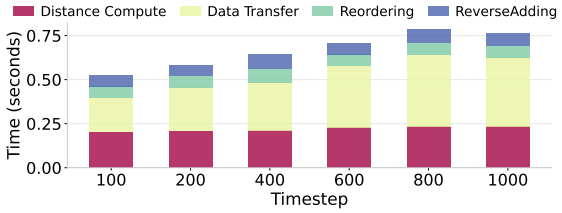


Figure 11: Performance breakdown of insertions.

6.3 Component Analysis

Effectiveness of Adaptive Vector Caching. We conduct two complementary experiments to evaluate our CPU-GPU vector caching strategy from two perspectives.

Replacement Strategies. We compare our workload-aware placement strategy against two baselines under the *SlidingWindow* workload described in §4.3: *LRU replacement* and *CPU offloading*. Figure 8 demonstrates that our method achieves the highest throughput while maintaining a steadily decreasing miss rate. Specifically, our approach outperforms CPU offloading by 1.9× and LRU replacement by 7.2× on average. This significant performance gap reveals a critical insight: while frequent replacement achieves lower miss rates, it introduces substantial CPU-GPU transfer overhead that degrades overall throughput. In contrast, our adaptive policy strikes a better trade-off between data locality and transfer cost, delivering both high efficiency and performance stability.

Memory Constraints. To evaluate the robustness of our caching strategy under varying memory constraints, we fix the dataset size and vary the available GPU memory from 20% to 100%. As shown in Figure 9, our method consistently outperforms both baselines across all settings, achieving 1.7× and 5.6× higher average throughput than CPU offloading and LRU, respectively. Importantly, even as GPU memory decreases, resulting in higher miss rates, our throughput remains stable. This indicates that our design effectively overlaps CPU-side computation with data transfer, masking latency, and sustaining high performance under limited GPU capacity.

Effect of Deletion Strategies. We compare three approaches: lazy deletion alone, with global consolidation, and our method combining both with localized repair. As shown in Figure 10, our method achieves 5.2% and 2.3% higher recall than the two methods, respectively, while reducing overhead by 57.6% compared to global consolidation alone. This confirms our localized repair strategy effectively preserves index quality with minor performance impacts.

Overhead Breakdown. To achieve a deeper understanding of our framework's performance characteristics, we break down the time cost of each insertion operation into its key components. As illustrated in Figure 11, *data transfer* dominates at 45.4% of the total time, followed by *distance computing* (33.6%), *heuristic reordering* (10.3%), and *reverse adding* (10.6%). This breakdown reveals two important observations. First, despite leveraging GPU acceleration,

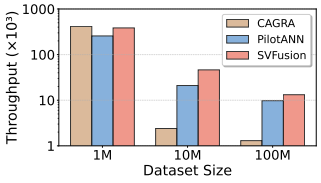


Figure 12: Comparison with GPU-based methods.

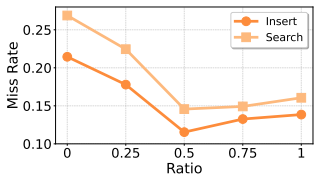


Figure 13: Effect of prediction parameters.

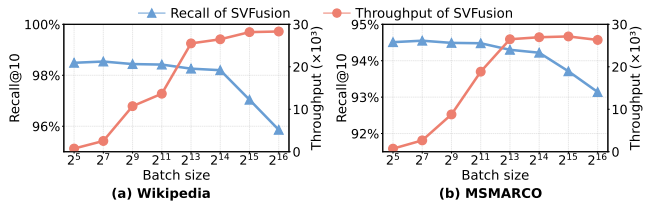


Figure 14: Impact of varying batch size of updates.

data movement between CPU and GPU remains the primary performance bottleneck, especially as the index grows and more vectors are evicted to CPU memory. Second, distance computation time remains relatively stable across timesteps, highlighting the effectiveness of our pipelined CPU-GPU parallelism in hiding computation latency. Overall, this analysis underscores the importance of optimizing data locality and transfer efficiency in hybrid-memory vector search systems and motivates our design of adaptive caching and asynchronous execution pipelines.

Effectiveness of CPU-GPU vector search. Figure 12 illustrates the CPU-GPU search performance on MSMARCO across different dataset scales under the Recall@10=0.9 constraint. For datasets that fit within GPU memory, SVFusion achieves comparable performance to CAGRA with slightly lower throughput due to the overhead of maintaining CPU-side data structures. When the dataset exceeds GPU memory, CAGRA experiences significant performance degradation due to inefficient CPU-GPU data transfers through unified virtual memory (UVM). Compared to PilotANN, SVFusion achieves 1.5×-2.2× speedup across all scales due to PilotANN's two-stage design requiring CPU-side refinement. This demonstrates that our CPU-GPU co-processing approach can efficiently handle large-scale vector search without data compression.

6.4 Parameter Sensitivity

Prediction Parameters. We evaluate how the parameters in our prediction function affect miss rates by varying the ratio $\frac{\alpha}{\alpha+\beta}$, where 0 represents prediction solely on graph topology and 1 indicates prediction based on recent accesses. Figure 13 shows that higher weights for recent accesses generally reduce miss rates, with optimal performance achieved at a ratio around 0.6.

Batch Size. We further investigate how batch size influences SVFusion's update throughput and search accuracy on Wikipedia and MSMARCO, as shown in Figure 14. As the batch size increases from 2^5 to 2^{16}, update throughput steadily improves, primarily due to enhanced GPU utilization and reduced per-operation overhead. However, when the batch size exceeds 2^{13} (8,192), Recall@10 begins to decline, indicating that large batches delay the updates of graph connectivity and degrade search quality. To balance this trade-off, we adopt a default batch size of 10,000, which offers high update throughput while maintaining stable search accuracy.

7 RELATED WORK

Vector Indexes. ANNS is a classical problem with a large body of research work. Partition-based methods leverage tree structures [37, 58], and quantization techniques [14, 19, 20, 29] to organize vectors into spatial partitions but suffer from the curse of dimensionality. Hash-based approaches [18, 26, 50, 62] project high-dimensional vectors into lower-dimensional hash buckets to accelerate similarity search. Graph-based methods [13, 17, 28, 43, 56, 63] construct proximity graphs where nodes represent vectors and edges connect similar vectors, enabling efficient navigation during search. However, most techniques target static datasets and perform poorly with frequent updates.

Streaming Vector Search. Recent research has focused on enabling dynamic updates in vector search systems to support streaming data scenarios. FreshDiskANN [45] maintains new vectors in memory before periodically merging them into a disk-resident graph. SPFresh [55] implements a cluster-based approach with lightweight incremental rebalancing (LIRE) that adjusts partitions as data distribution changes. Other approaches [48, 53, 54] focus on maintaining graph connectivity or parallel acceleration for streaming updates. Despite these advances, existing systems face significant limitations: they either support only limited dataset sizes, handle specific operations (insertions), or exhibit degraded search quality in high-dimensional spaces.

GPU-accelerated ANNS. Leveraging GPUs for vector search has been explored extensively to address the computational demands of large-scale applications. Pioneer works like Faiss [31] and Raft [1] first migrated vector retrieval from CPU to GPU. SONG [61] reformulated graph-based search to better utilize GPU parallelism, while GANNS [57] and CAGRA [41] have further improved GPU-based algorithms with specialized kernel design. For system design optimization, [23, 32, 46, 60] converged on integrating heterogeneous CPU-GPU scheduling, distributing data between GPU and CPU, and establishing periodic synchronization mechanisms. However, these methods are designed for static datasets where indexes remain fixed during querying. The recent RTAMS-GANNS [48] supports real-time insertions on GPUs but cannot scale to large datasets. Thus, efficient dynamic index update on GPUs, particularly for graph-based methods at scale, remains an open challenge.

8 CONCLUSION

We presented SVFusion, a CPU-GPU collaborative framework for real-time updates for large-scale vector search. It introduces a hierarchical vector index spanning memory tiers, enabling efficient data distribution across heterogeneous hardware. We employ an adaptive caching mechanism that optimizes data placement by considering both access patterns and graph structure relationships. SVFusion further incorporates a lightweight deletion strategy that enables concurrent operations without blocking, critical for dynamic workloads. Experimental results demonstrate that SVFusion achieves up to 72× higher throughput while maintaining superior recall compared to state-of-the-art methods across various streaming workloads.

REFERENCES

[1] 2022. Rapidsai/raft: RAFT contains fundamental widely-used algorithms and primitives for data science, Graph and machine learning. https://github.com/rapidsai/raft.

[2] 2024. cuVS: Vector Search and Clustering on the GPU. https://github.com/rapidsai/cuvs/tree/v24.12.00.

[3] 2024. Microsoft Turing-ANNS-1B. https://learning2hash.github.io/publications/microsoftturinganns1B/.

[4] 2024. microsoft/ms_marco. https://huggingface.co/datasets/microsoft/ms_marco.

[5] 2024. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/.

[6] 2024. updating an index. https://github.com/rapidsai/cuvs/issues/295.

[7] 2024. wikimedia/wikipedia. https://huggingface.co/datasets/wikimedia/wikipedia.

[8] 2025. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[9] 2025. Personalized help from Gemini. https://gemini.google/overview/personalization.

[10] 2025. The Platform for Building Stateful Agents. https://www.letta.com/.

[11] 2025. ScaNN for AlloyDB: The first PostgreSQL vector search index that works well from millions to billion of vectors. https://cloud.google.com/blog/products/databases/how-scann-for-alloydb-vector-search-compares-to-pgvector-hnsw.

[12] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

[13] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-based similarity search for scalable data science. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1548–1559.

[14] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*. 37, 6 (2014), 1247–1260.

[15] Kenneth L. Clarkson. 1994. An Algorithm for Approximate Closest-Point Queries. In *Proceedings of the Symposium on Computational Geometry*. 160–164.

[16] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell. (TPAMI)* 44, 8 (2022), 4139–4150.

[17] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment* 12, 5 (2019), 461–474.

[18] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 541–552.

[19] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2025. Practical and asymptotically optimal quantization of high-dimensional vectors in euclidean space for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[20] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3 (2024), 167.

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

[22] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. 2023. GGNN: Graph-Based GPU Nearest Neighbor Search. *IEEE Trans. Big Data* 9, 1 (2023), 267–279.

[23] Yuntao Gui, Peiqi Yin, Xiao Yan, Chaorui Zhang, Weixi Zhang, and James Cheng. 2025. PilotANN: Memory-Bounded GPU Acceleration for Vector Search. *CoRR* abs/2503.21206 (2025).

[24] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: a cloud native vector database management system. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3548–3561.

[25] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based Retrieval in Facebook Search. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 2553–2561.

[26] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 9, 1 (2015), 1–12.

[27] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. 1998. Optimal Histograms with Quality Guarantees. In *PVLDB*. 275–286.

[28] 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).

[29] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*. 33, 1 (2011), 117–128.

[30] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *CoRR* abs/2404.12457 (2024).

[31] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[32] Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, Jyothi Vedurada, et al. 2024. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *arXiv preprint arXiv:2401.11324* (2024).

[33] Jon M Kleinberg. 2000. Navigation in a small world. *Nature* 406, 6798 (2000), 845–845.

[34] 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 *Advances in Neural Information Processing Systems*.

[35] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. 2018. The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform. In *Proceedings of the International Middleware Conference*. 9–16.

[36] Ting Liu, Andrew Moore, Ke Yang, and Alexander Gray. 2004. An investigation of practical approximate nearest neighbor algorithms. *NIPS* 17 (2004).

[37] Ruiyao Ma, Yifan Zhu, Baihua Zheng, Lu Chen, Congcong Ge, and Yunjun Gao. 2024. GTI: Graph-Based Tree Index with Logarithm Updates for Nearest Neighbor Search in High-Dimensional Spaces. *Proceedings of the VLDB Endowment* 18, 4 (2024), 986–999.

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

[39] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. 2025. Quake: Adaptive Indexing for Vector Search. *arXiv preprint arXiv:2506.03437* (2025).

[40] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 297–306.

[41] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2024. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. In *IEEE International Conference on Data Engineering (ICDE)*. 4236–4247.

[42] Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. 2023. MemGPT: Towards LLMs as Operating Systems. *CoRR* abs/2310.08560 (2023).

[43] Y Peng, B Choi, TN Chan, J Yang, and J Xu. 2023. Efficient approximate nearest neighbor search in multi-dimensional databases. *Proceedings of the International Conference on Management of Data (SIGMOD)*. (2023).

[44] 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).

[45] 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. *arXiv preprint arXiv:2105.09613* (2021).

[46] Rafael Souza, André Fernandes, Thiago SFX Teixeira, George Teodoro, and Renato Ferreira. 2021. Online multimedia retrieval on CPU–GPU platforms with adaptive work partition. *J. Parallel and Distrib. Comput.* 148 (2021), 31–45.

[47] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around.. *SIGMOD Rec.* 53, 2 (2024), 21–37.

[48] Yiping Sun, Yang Shi, and Jiaolong Du. 2024. A Real-Time Adaptive Multi-Stream GPU System for Online Approximate Nearest Neighborhood Search. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*. 4906–4913.

[49] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. 2025. Towards High-throughput and Low-latency Billion-scale Vector Search via CPU/GPU Collaborative Filtering and Re-ranking. In *USENIX Conference on File and Storage Technologies (FAST)*. 171–185.

[50] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2023. DB-LSH 2.0: Locality-sensitive hashing with query-based dynamic bucketing. *IEEE Transactions on Knowledge and Data Engineering* 36, 3 (2023), 1000–1015.

[51] Nitish Upreti, Krishnan Sundaram, Hari Sudan Sundar, Samer Boshra, Balachandar Perumalswamy, Shivam Atri, Martin Chisholm, Revti Raman Singh, Greg Yang, Subramanyam Pattipaka, et al. 2025. Cost-Effective, Low Latency Vector Search with Azure Cosmos DB. *arXiv preprint arXiv:2505.05885* (2025).

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

[53] Wentao Xiao, Yueyang Zhan, Rui Xi, Mengshu Hou, and Jianming Liao. 2024. Enhancing HNSW Index for Real-Time Updates: Addressing Unreachable Points and Performance Degradation. *arXiv preprint arXiv:2407.07871* (2024).

[54] Haike Xu, Magdalen Dobson Manohar, Philip A. Bernstein, Badrish Chandramouli, Richard Wen, and Harsha Vardhan Simhadri. 2025. In-Place Updates of a Graph Index for Streaming Approximate Nearest Neighbor Search. *CoRR* abs/2502.13826 (2025).

[55] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 545–561.

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

[57] 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 *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 552–564.

[58] Yuxiang Zeng, Yongxin Tong, and Lei Chen. 2023. LiteHST: A Tree Embedding based Method for Similarity Search. *Proc. ACM Manag. Data* 1, 1 (2023), 35:1–35:26.

[59] Jianjin Zhang, Zheng Liu, Weihao Han, Shitao Xiao, Ruicheng Zheng, Yingxia Shao, Hao Sun, Hanqing Zhu, Premkumar Srinivasan, Weiwei Deng, Qi Zhang, and Xing Xie. 2022. Uni-Retriever: Towards Learning the Unified Embedding Based Retriever in Bing Sponsored Search. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 4493–4501.

[60] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 23–40.

[61] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *IEEE International Conference on Data Engineering (ICDE)*. 1033–1044.

[62] Bolong Zheng, Zhao Xi, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S Jensen. 2020. PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search. *Proceedings of the VLDB Endowment* 13, 5 (2020), 643–655.

[63] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. 2024. SeRF: segment graph for range-filtering approximate nearest neighbor search. *Proceedings of the ACM on Management of Data (SIGMOD)* 2, 1 (2024), 1–26.

Algorithm 3 Best-First Search on Graph-based Index

Input: graph index $G(V, E)$, query q, entry points EP, candidate size ef
Output: C contains ef nearest neighbors

1: priority queue $C \leftarrow EP$
2: **repeat**
3: $u \leftarrow$ the first unchecked node in C
4: **for** unvisited $v \in \{v|(u, v) \in E\}$ **do**
5: mark v as visited
6: $d \leftarrow$ euclidean(q, v)
7: C.insert(v) w.r.t. d
8: C.resize(ef)
9: **until** C has no unchecked node

A ANNS IN GRAPH-BASED INDEX

In this appendix we provide pseudocode for the Best-First Search (BFS) on graph-based index, and the GPU-based ANNS in CAGRA [41], a state-of-the-art implementation.

Algorithm 3 illustrates how BFS operates on a graph-based index such as DiskANN [28]. When searching for the k nearest neighbors for a large batch of queries, CPU-based approaches can only process queries in parallel using individual threads, constrained by the limited number of CPU cores. In contrast, GPU-based algorithms store both the graph structure and vectors in GPU memory, enabling the utilization of thousands of thread blocks to perform distance computations and graph traversals simultaneously.

Algorithm 4 shows the search process in CAGRA. The algorithm maintains a candidate pool C_i (corresponding to CAGRA's internal top-L list) for each query q_i, which stores the current best candidates with their distances. The search begins with random initialization (Lines 1-2), where InitialPoints(G) randomly samples nodes from the graph index and adds them to the candidate pool after distance computation. The main search loop then iteratively performs graph traversal and candidate refinement: at each iteration, the algorithm selects the nearest unvisited candidate x_{curr} from C_i, fetches its neighbors X_i from the graph structure, computes distances between the query and all neighbors in parallel on GPU, and updates the candidate pool with promising new candidates (Lines 4-9).

A key optimization in CAGRA is that distance calculations are performed conditionally - distances are only computed for nodes that haven't been evaluated before, avoiding redundant computations across iterations. The algorithm converges when the candidate pool stabilizes, meaning no better candidates are found in recent iterations, and finally returns the top-k nearest neighbors (Line 10).

B IMPLEMENTATION DETAILS

We implement SVFusion by extending the cuVS 24.12 library [2] with CPU-GPU cooperative mechanisms for streaming vector search. Specifically, (i) We extensively refactor the CAGRA [41] codebase—modifying tens of thousands of lines—to support dynamic updates and CPU-GPU co-processing over large-scale graph-based vector indexes, enabling efficient vector insertion and deletion through bitset-based filtering. (ii) We develop a cache manager that maintains bidirectional mappings between CPU memory and GPU memory for efficient data movement. (iii) We design a multi-partition strategy that partitions the cache space into independent regions, each managed by separate locks to minimize contention and maximize system throughput. This partitioning scheme enables concurrent cache operations across multiple threads while maintaining data consistency.

Algorithm 4 GPU-based ANNS in CAGRA

Input: graph index G, a batch of p queries $Q = \{q_1, q_2, \ldots, q_p\}$, k for top-k, and candidate pool size $L \geq k$
Output: $K := \cup_{i=1}^{p}\{K_i\}$, where K_i is the set of k-nearest neighbours for $q_i \in Q$

1: **for** each query $q_i \in Q$ **in parallel do**
2: $C_i \leftarrow$ InitializeCandidatePool(L) ▷ *Initialize top-L list*
3: Add InitialPoints(G) to C_i ▷ *Random pickup initial points*
4: **while** not converged **do**
5: $x_{curr} \leftarrow C_i$.GetNearest() ▷ *Get top-1 candidates*
6: $X_i \leftarrow$ FetchNeighbors(x_{curr}, G) ▷ *Graph traversal*
7: ParallelComputeDist(q_i, X_i) ▷ *Distance computation*
8: C_i.Update(X_i) ▷ *Merge with top-L list*
9: *converged* $\leftarrow$ CheckConvergence(C_i)
10: $K_i \leftarrow k$ nearest candidates to q_i from C_i

Algorithm 5 Vector Insertion in SVFusion

Input: graph index G, new vectors $V = \{v_1, v_2, \ldots, v_n\}$, degree d, batch size B
Output: updated graph index G' with inserted vectors

1: **for** each batch $V_b \subseteq V$ of size B **do**
2: $N_b \leftarrow$ GPUParallelSearch(V_b, G, $2d$) ▷ *GPU search for 2d candidates*
3: **for** each vector $v_i \in V_b$ **in parallel do**
4: *detour_counts* $\leftarrow$ CountDetourable($N_b[i]$)
5: ▷ *Rank-based reordering*
6: *edges*$[i] \leftarrow$ SelectByRank($N_b[i]$, *detour_counts*)
7: **for** each vector $v_i \in V_b$ **in parallel do**
8: *reverse_edges*$[i] \leftarrow$ ReverseAdd(*edges*$[i]$, G)
9: ▷ *Atomic updates*
10: Interleave(*edges*$[i]$, *reverse_edges*$[i]$, $G[v_i]$)
11: BatchUpdate(G, V_b) ▷ *Sync CPU-GPU graph state*

Algorithm 6 Global Consolidation

Input: graph index $G(V, E)$ with $|V| = n$, set of vertices to be deleted L_D
Output: consolidated graph on nodes V' where $V' = V \setminus L_D$

1: **for** each $p \in V \setminus L_D$ s.t. $N_{out}(p) \cap L_D \neq \emptyset$ **do**
2: $\mathcal{D} \leftarrow N_{out}(p) \cap L_D$
3: $C \leftarrow N_{out}(p) \setminus \mathcal{D}$ ▷ *initialize candidate list*
4: **for** each $v \in \mathcal{D}$ **do**
5: $C \leftarrow C \cup (N_{out}(v) \setminus \mathcal{D})$
6: $N_{out}(p) \leftarrow$ RobustPrune(p, C, α, R)