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

Yuchen Peng
Zhejiang University†
Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
zjupengyc@zju.edu.cn

Dingyu Yang*
Zhejiang University†
Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
yangdingyu@zju.edu.cn

Zhongle Xie
Zhejiang University†
Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
xiezl@zju.edu.cn

Ji Sun
Huawei Technologies Co., Ltd
sunji11@huawei.com

Lidan Shou*
Zhejiang University†
Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
should@zju.edu.cn

Ke Chen
Zhejiang University†
Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
chenk@zju.edu.cn

Gang Chen
Zhejiang University†
Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
cg@zju.edu.cn

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 GPU-CPU-disk 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 a workload-aware vector caching mechanism to maximize the efficiency of limited GPU memory. It further enhances performance through real-time coordination with CUDA multi-stream optimization and adaptive resource management, along with concurrency control that ensures data consistency under interleaved queries and updates. Empirical results demonstrate that SVFusion achieves significant improvements in query latency and throughput, exhibiting a 20.9× higher throughput on average and 1.3× to 50.7× lower latency compared to baseline methods, while maintaining high recall for large-scale datasets under various streaming workloads.

PVLDB Reference Format:
Yuchen Peng, Dingyu Yang, Zhongle Xie, Ji Sun, Lidan Shou, Ke Chen, and Gang Chen. SVFusion: A CPU-GPU Co-Processing Architecture for Large-Scale Real-Time Vector Search. PVLDB, 19(5): 1074 - 1087, 2026.
doi:10.14778/3796195.3796216

*Corresponding authors: Dingyu Yang, Lidan Shou
†The State Key Laboratory of Blockchain and Data Security

Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796216

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 [29], web search [67], and LLM-based applications [38]. 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) [40], specifically graph-based solutions [14, 20, 32, 42, 53] that balance accuracy and computational efficiency in high-dimensional spaces. Building on this, recent work explores GPU-accelerated optimizations, including SONG [69], GANNS [65], and CAGRA [47], leveraging massive parallelism and high memory bandwidth to achieve significant speedups over CPU-based approaches, enabling scalable high-throughput vector search.

Meanwhile, the explosive growth of continuously generated embeddings has led to a fundamental shift in vector search from offline indexing to streaming processing [10, 11, 34, 39, 48]. In real-time use cases like emergency response [45], retrieval-augmented generation (RAG) systems continuously ingest live data while retrieval relies upon previous results, requiring index freshness to maintain coherent LLM responses. E-commerce platforms such as JD.com [39] serve millions of independent search queries that must reflect inventory changes. In live-video analytics and surveillance, streams yield vast numbers of embedding vectors per second, which must be both continuously indexed and queried for tasks such as

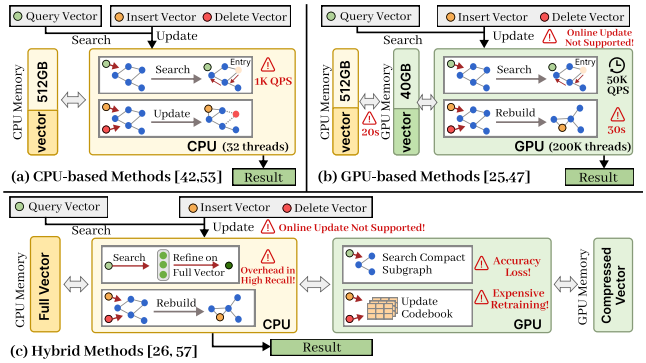


Figure 1: Comparison of existing methods for ANNS.

object tracking and anomaly detection [51]. These applications demand not only real-time data synchronization under continuous updates but also higher processing throughput and lower latency.

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. As illustrated in Figure 1, large-scale streaming scenarios demand support for search, insertion, and deletion operations [12]. CPU-based methods [42, 53, 63] 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 [25, 47] achieve high throughput for static indices, but incur significant transfer overhead when handling large-scale updates [36]. 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-throughput but static) motivates a unified hybrid approach. Recent methods [26, 57] attempt to bridge this gap by storing compressed vectors in GPU memory, but they introduce accuracy degradation unsuitable for high-recall requirements and requiring expensive retraining to handle streaming updates [24]. To summarize, there exist two challenges in tackling large-scale streaming ANNS:

Challenge 1: GPU Memory Capacity Constraints. Modern vector data applications handle large-scale datasets with millions to billions of high-dimensional embeddings [6, 27], often exceeding GPU memory capacity, resulting in poor performance. For example, serving 35M vectors in 768 dimensions [8] 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 [55]. While compression techniques [33, 57] 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 [32, 42], 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 [56], along with additional overhead from cross-device data transfers in hybrid CPU-GPU systems [56, 57]. Second, *search accuracy* degrades over time as frequent updates deteriorate graph structure, leading to suboptimal traversal paths [7, 47].

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 acceleration with real-time updates to achieve high throughput while maintaining low latency. SVFusion is built upon three key innovations: (1) **Hierarchical Vector Index Structure** that organizes vector data and graph topology across GPU and CPU memory and disk to maximize computational efficiency. We design a real-time coordination mechanism with CUDA multi-stream optimization and adaptive resource management to achieve both high throughput and low latency. (2) **Workload-Aware Vector Placement Mechanism** that adaptively determines 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) **Concurrency Control for Real-time Updates** that enables safe and high-performance concurrent queries and updates. We provide a consistency guarantee combining fine-grained locking with cross-tier synchronization, and a multi-version mechanism to ensure consistency and superior performance.

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

- We present *SVFusion*, a novel GPU-CPU-disk cooperative framework for streaming vector search that integrates hierarchical indexing with real-time 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 performance under dynamic workloads. (§4)
- We propose a concurrency control protocol that maintains data consistency through fine-grained locking 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, 71.8× higher insertion throughput, and 1.3× to 50.7× lower latency while maintaining superior recall rates compared to baseline methods. (§6)

2 PRELIMINARY

We first define the Approximate Nearest Neighbor Search (ANNS) problem and its streaming variant (SANNS), followed by an overview

Table 1: Summary of notations.

Notation	Definition
X	the set of N D-dimensional vectors
q	a query vector
$G = (V, E)$	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

of the graph-based indexing techniques. 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 [17] 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 $\text{dist}(x_i, q) \leq \text{dist}(x_j, q)$.

Exact nearest neighbor search is computationally prohibitive for large datasets [60], leading most studies [19, 20, 35, 42] to adopt *approximate nearest neighbor search* (ANNS) for a balanced trade-off between accuracy and efficiency. 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 $i = 1, 2, \ldots, k$.

2.2 Streaming ANNS

The *streaming approximate nearest neighbor search* (SANNS) problem extends traditional ANNS to handle continuously evolving datasets with dynamic insertions and deletions. SANNS processes an ordered sequence of operations $O = \{o_1, o_2, \ldots, o_T\}$ 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 vector 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 $X_{\text{init}} \subset \mathbb{R}^D$, with periodic rebuilding optionally.
- **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 on static datasets with fixed ground truth, SANNS faces the challenges of preserving accuracy on evolving datasets while ensuring efficient dynamic operations.

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 directional connections between vertices: $(v_i, v_j) \in E$ means vertex v_i links to nearby vertex v_j, with edges directed from each vertex toward its selected neighbors. For any vertex $v_i \in V$, we denote its out-neighbor set as $N_{out}(v_i) = \{v_j \in V \mid (v_i, v_j) \in E\}$ and its in-neighbor set as $N_{in}(v_i) = \{v_j \in V \mid (v_j, v_i) \in E\}$.

Graph-Based ANNS and SANNS. Graph-based ANNS performs query processing by greedily traversing a proximity graph. Proximity graphs are primarily classified into two categories: *k-nearest neighbor graphs (KNNG)* where each vertex maintains exactly k outgoing edges to its approximate nearest neighbors [18], and *relative neighborhood graphs (RNG)* that optimize connectivity by removing redundant edges while maintaining variable degrees [20, 32]. Given a query vector q, vector search begins from some entry points, which can be either randomly chosen [32, 43, 47] or supplied by a separate algorithm [23, 31]. Then the procedure iteratively explores neighboring nodes with decreasing distance to q until a stopping criterion is met [32]. To support streaming scenarios (SANNS), the index must handle continuous updates. 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 KNNG with fixed out-degree to fully exploit GPU's massive parallelism, as variable degrees would lead to load imbalance and underutilized computing resources [25, 47]. The key acceleration involves storing critical metadata such as graph structures in high-bandwidth memory, and utilizing one thread-block per query rather than one thread per query [25] to enable extensive parallelization of core operations. This design achieves substantial performance improvements over CPU-based approaches.

3 OVERVIEW

We present SVFusion, a GPU–CPU–disk collaborative framework for SANNS. Our framework addresses the fundamental challenge of supporting real-time vector search and updates. To overcome GPU memory capacity limitations, (1) we introduce a **hierarchical graph-based vector index** that maintains synchronized graph structures across GPU, CPU, and disk tiers, enabling seamless transitions between memory layers as dataset size grows; (2) we design a **workload-aware vector placement strategy** that dynamically manages data residency and caching across GPU, CPU, and disk to minimize PCIe and I/O overheads. To further improve the performance under high-frequency updates, (3) we propose a **real-time coordination mechanism** with **concurrency control** that ensures consistency across memory tiers while enabling high-throughput, low-latency concurrent operations. 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 [47], a state-of-the-art GPU-based index originally designed for static datasets, into a dynamic, update-efficient

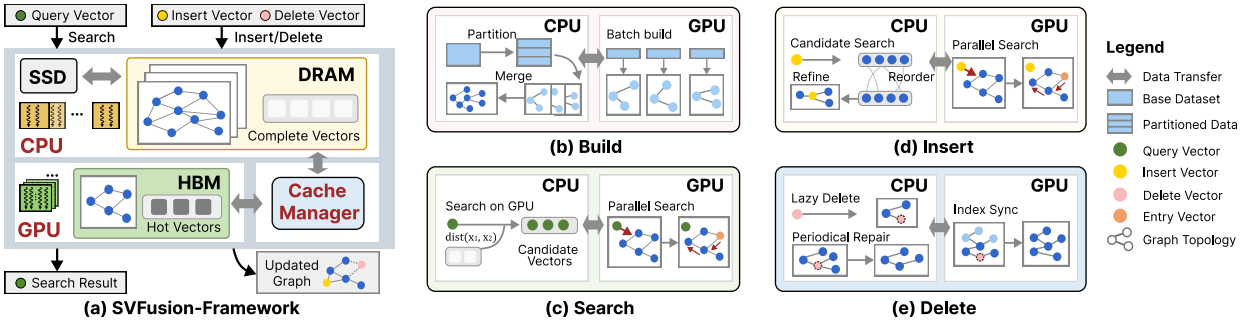


Figure 2: Overview of SVFusion.

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 or offloads cold vectors to disk when CPU memory is exhausted. It orchestrates CPU computation and synchronization, and enables concurrent access control.

A dedicated **Cache Manager** manipulates data placement across GPU, CPU, and disk tiers to optimize query performance under hybrid workloads. Unlike traditional caching strategies that rely on temporal locality [46], 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.

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

Build (§4.2). The index is constructed using a GPU-parallel strategy inspired by CAGRA [47], performing neighbor search and graph linking to fully leverage GPU parallelism and memory bandwidth. For large datasets, it partitions the data, builds subgraphs for each partition on GPU, and merges them into a unified index on CPU while preserving inter-partition connectivity. When the complete graph exceeds CPU memory, this operation involves merging subgraphs within a bounded memory window that avoids loading the entire graph into memory.

Search (§4.2 & 4.3). Given a query, this operation performs graph traversal starting on GPU. The **cache manager** checks whether required vectors or subgraphs are already cached in GPU memory. If present, the search proceeds entirely on GPU; otherwise, SVFusion either transfers the data from CPU on demand 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 (§5.1). 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.

Deletion (§5.2). 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.

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

To handle massive vector datasets with dynamic access patterns, SVFusion introduces a hierarchical index architecture, a workload-adaptive caching mechanism, and a real-time coordination strategy, enabling high-throughput, low-latency streaming vector search.

4.1 Motivation

Hierarchical Memory Management. Due to the limited capacity of GPU memory, billion-scale vector datasets cannot be fully accommodated. While compressing vectors [57, 68] reduces memory footprint, it often degrades search accuracy. Observing that real-world access patterns are highly skewed [44], we adopt a hierarchical memory design, caching hot vectors uncompressed on GPU to enable scalable, high-accuracy search across massive datasets.

Dynamic Cache Management: In streaming ANNS, access patterns evolve continuously, making previously hot vectors cold. Unlike traditional caching approaches [37, 46] 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 optimally adapt vector placement under evolving workloads.

4.2 Hierarchical Graph-based Vector Index

Figure 3 illustrates our hierarchical index structure spanning CPU and GPU memory tiers. Unlike existing GPU-accelerated ANNS methods constrained by limited GPU memory capacity when handling large-scale datasets [47, 69], 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 at scale.

Hierarchy Index Structure. Our index adopts a three-tier GPU–CPU–disk architecture, enabling efficient caching, seamless data movement, and scalability beyond GPU and main memory limits. Given a dataset of size N, the CPU memory maintains the graph

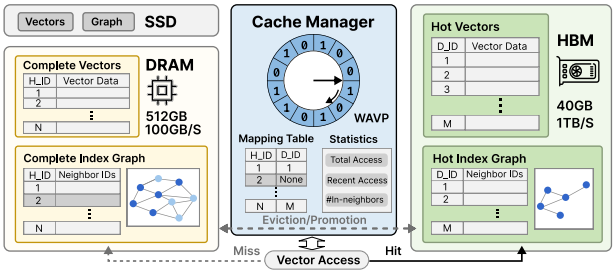


Figure 3: Hierarchical Index Structure in SVFusion.

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 large-scale datasets exceeding DRAM capacity, our framework seamlessly extends to incorporate persistent storage, creating a three-tier hierarchy. Vectors and graph structures are stored on disk in the same format as in CPU memory, with frequently accessed portions cached in CPU and GPU tiers through on-demand loading. 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.

Index Construction. During index construction, the dataset is partitioned based on GPU memory capacity. When extended to persistent storage with large-scale datasets, SVFusion loads data of each partition from disk to CPU, transfers it to GPU for local subgraph construction, and writes the completed subgraph back to disk. The CPU then merges these subgraphs by computing cross-partition edges within a bounded memory window, loading vectors on demand from disk. This approach ensures memory usage regardless of dataset scale, with the final graph persisted to disk while frequently accessed portions are cached in CPU-GPU tiers.

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 parallel search process begins with randomly initializing each query's candidate pool (Lines 1-2). Then the graph traversal is performed on GPU (Line 5) until the candidate pool remains unchanged. Random entry points provide GPU-friendly parallelism and avoid bias seed maintenance under dynamic updates. During neighbor expansion (Lines 7-11), our cache manager checks each neighbor vector's cache status (Line 8) and partitions computation on CPU or 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

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 kNNs for q_i

1: **for** each query $q_i \in Q$ **in parallel do**
2: $C_i \leftarrow$ InitCandidatePool(G, L) ▷ *random initialization*
3: **repeat**
4: $x_{curr} \leftarrow C_i$.GetNearest(q_i)
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

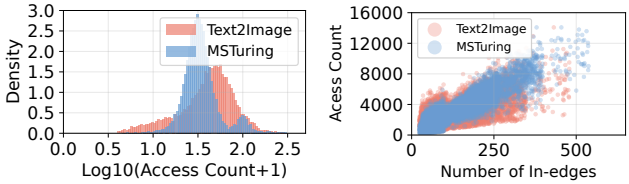
perform computations directly on CPU. This results in partitioning neighbors into X_i^{GPU} and X_i^{CPU} for parallel distance computation on both processors (Lines 10-13). This adaptive strategy balances throughput and memory efficiency, enabling scalable and responsive vector search under GPU memory constraints. Note that maintaining data copies across CPU and GPU introduces consistency challenges during index updates. We address this through immediate bitset synchronization for logical deletions and batched graph updates from CPU to GPU, detailed in Section 5.3.

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 and efficient computation during ANNS.

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. This fundamentally differs from traditional cache replacement policies, such as LRFU [37], which are optimized to maximize cache hit ratios. In CPU–GPU co-processing, promoting every missed vector to GPU cache might lead to excessive data transfers, while computing all queries on CPU underutilizes GPU parallelism. Our caching strategy instead jointly optimizes transfer cost and computational efficiency, deciding adaptively whether to cache a vector on GPU or to compute directly on CPU.

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



(a) Distribution of Recent Access. (b) Distribution of In-neighbors.

Figure 4: Characterization of Vector Access Patterns.

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

$$\text{gain}(x) = \lambda_x \cdot (T_{\text{CPU}} - T_{\text{GPU}}) - T_{\text{transfer}} \tag{1}$$

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 per vector [9]. 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, access patterns naturally depend on both temporal factors and graph structure. We therefore analyzed vector access with respect to both properties during ANNS across MSTuring [4] and Text2Image [50] with different dimensions and data types. Figure 4 illustrates the key characteristics of vector access patterns. Recent access frequency $F_{recent}(x,t)$ captures temporal locality but exhibits a medium-hot distribution rather than a long-tail. In contrast, in-neighbor degree $E_{in}(x)$ shows a strong structural correlation with access frequency, reflecting that searches often traverse high-connectivity hubs. However, such structural importance cannot be exploited by existing cache strategies such as LRFU [37] that bases its replacement decision on recency and frequency. To effectively cache these high-connectivity vectors in the graph, our method explicitly incorporates graph structural connectivity into the caching decision process.

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:

$$F_\lambda(x) = \alpha \times F_{recent}(x,t) + \beta \times \log(1 + E_{in}(x)) \tag{2}$$

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.

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: $clock \leftarrow (clock + 1) \bmod |X_t^G|$

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:

$$\lambda_x = \lambda_1 \cdot F_{recent}(x,t) + \lambda_2 \cdot \log(1 + E_{in}(x)) + \lambda_3 \tag{3}$$

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 $\lambda_x = F_\lambda(x) + \lambda_3$. Recall that caching is beneficial when $\text{gain}(x) > 0$. It holds when $\lambda_x > \rho$, where $\rho = \frac{T_{\text{transfer}}}{T_{\text{CPU}} - T_{\text{GPU}}}$ is the cost ratio of transfer overhead to computation gain. Since we have $\lambda_x = F_\lambda(x) + \lambda_3$, the condition 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 $F_\lambda(x) > \tau \iff \lambda_x > \rho \iff \text{gain}(x) > 0$. This shows our approach reduces cache decisions to simple threshold comparisons.

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 $F_\lambda(x) \le \theta$, the system keeps x on CPU for in-place computation. In the *predictive replacement* phase (Lines 3-11), we extend the clock-sweep policy with a prediction-guided eviction strategy: when GPU memory is full, the system evicts the vector with zero reference bits and the lowest predicted frequency $F_\lambda(x)$, effectively mitigating thrashing under dynamic ANNS workloads. This design strictly controls eviction decisions to avoid unnecessary vector replacements and redundant data transfers.

To scale to large-scale datasets, SVFusion extends its memory hierarchy to include disk storage, forming a cascading lookup pipeline across GPU, CPU, and disk. Upon cache misses, vectors are asynchronously prefetched from lower tiers, guided by a hash-based directory that tracks data residency. When CPU memory is saturated,

vectors with the lowest F_λ scores are demoted to disk, ensuring efficient multi-tier data flow and consistent performance at scale.

4.4 Real-time Coordination for SANNS

To support both high-throughput and low-latency workloads, we design a real-time coordination mechanism that dynamically adapts GPU execution. While large batches improve throughput, latency-sensitive queries are processed immediately via a low-latency execution path combining multi-stream coordination [56] and multi-CTA execution [47]. An adaptive resource manager further adjusts GPU utilization under fluctuating streaming loads, and cold-start handling ensures stable performance from system initialization.

Low-latency Processing. To achieve low latency for small-batch queries in latency-oriented scenarios, we leverage two techniques to utilize GPU resources. First, we integrate a CUDA multi-stream coordination mechanism, comprising multiple search streams and a dedicated update stream. This design enables concurrent operations: search streams process queries of varying batch sizes in parallel, while the update stream handles insertions and deletions asynchronously with a consistency guarantee detailed in §5.3. Second, inspired by CAGRA's multi-CTA design [47], the system enables intra-query parallelism where multiple GPU thread blocks collaborate for a single query.

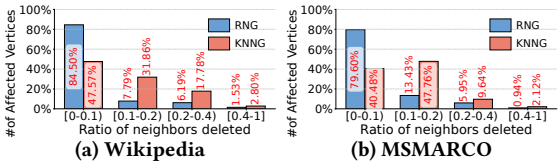
Adaptive Resource Management. SVFusion dynamically allocates GPU resources to adapt to varying workloads. To support concurrent multi-stream execution, we partition GPU memory into independent segments guarded by lightweight spinlocks, with vectors mapped to segments via identifier hashing. Streams with higher workload overhead are automatically allocated more GPU memory and compute resources. Additionally, the caching threshold θ is continuously adjusted based on observed miss rates and the distribution of estimated future access counts $F_\lambda(x)$. When miss rates increase along with higher average F_λ, θ is increased to prevent cache thrashing by being more selective about replacements. All adaptations operate automatically without manual tuning.

Cold Start Handling. To avoid startup slowdowns when the system begins processing streaming operations, SVFusion performs a three-phase initialization: (1) **GPU resource priming** pre-allocates vector storage in GPU global memory and creates CUDA streams for asynchronous execution; (2) **data structure initialization** prepares the cache manager's mapping table and begins to access tracking metadata; and (3) **GPU cache warm-up** preloads vectors based on predicted access counts. For large datasets, SVFusion caches top-ranked high in-degree vectors predicted by Equation 2 to minimize GPU cache misses.

5 INDEX UPDATE

5.1 Vector Insertion

The insertion process consists of two phases: (1) it performs a search operation to find candidate neighbors for new vectors, and (2) then establishes edges from the new vector to its selected neighbors and updates these neighbors with reverse edges. To achieve high throughput, our framework leverages GPU's massive parallelism to enable large batches of insertions concurrently, compared to CPU-based methods that are limited to around hundreds of vectors due to thread pool constraints [59]. For low-latency requirements, we employ the techniques described in §4.4, such as multi-stream and multi-CTA, to maintain high GPU utilization.



(a) Wikipedia (b) MSMARCO

Figure 5: Distribution of Deleted Neighbor Counts.

We further design a rank-based candidate reordering strategy. For each candidate at position i in the candidate list with sequential order, we count how many previously selected neighbors (at position $j < i$) contain this candidate in their neighbor lists—each occurrence represents a detourable path that can bypass the direct edge. We then sort candidates by detourable path counts in ascending order and select the top candidates up to the graph degree. This avoids expensive distance computations with only $O(|C| \log |C|)$ complexity to traverse the candidate set C. 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 preserving consistency.

5.2 Vector Deletion

SVFusion handles vector deletions through a three-stage process: (1) logical marking of inactive vertices to avoid query disruption, (2) localized topology-aware repair for rapid connectivity recovery, and (3) periodic global consolidation asynchronously.

5.2.1 Logical Deletion. SVFusion employs a lightweight, lazy deletion strategy to efficiently support high-frequency removal operations. Instead of immediately modifying the graph structure, SVFusion marks deleted vertices as logically removed using a bitset. These marked vertices are transparently skipped during subsequent insertions, deletions, and queries.

5.2.2 Affected Vertex Repair. While logical deletions enable low-latency removals, their accumulation gradually fragments the graph structure, degrading connectivity and search efficiency. Previous deletion handling methods [53] are primarily designed for RNG [20], where deletions primarily affect high-degree hubs and global consolidation is deferred until a deletion threshold (e.g., 20%) is reached. However, KNNG enforces uniform degree constraints, causing deletion impacts to spread evenly across all vertices. Our analysis (Figure 5) shows that after deleting 10% of vertices, KNNG has 3.26× more vertices with 10-40% deleted neighbors and 2.04× more with over 40% deleted neighbors. This pervasive connectivity loss degrades search quality long before global repair thresholds are reached, necessitating continuous, incremental repairs focused on the most affected 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

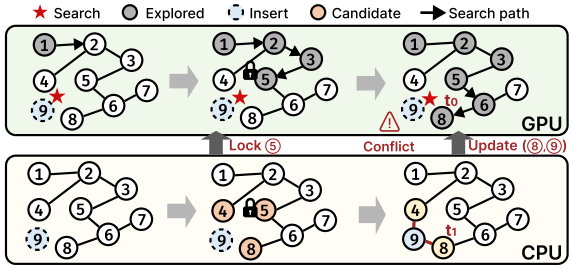


Figure 6: Illustration of the Concurrency Control Workflow.

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 $N_{out}(p)$. 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. This incremental approach preserves graph quality with minimal overhead compared to performing expensive repairs on all affected vertices.

Global Consolidation. When the deletion ratio surpasses a predefined threshold (e.g., 20%), SVFusion performs a global consolidation of all affected neighborhoods by aggregating candidates from the outgoing neighbors of deleted vertices. This consolidation executes asynchronously in the background on the latest graph snapshot, allowing insertions and queries to proceed concurrently without disruption. To ensure consistency with sequential operations, the process leverages a graph snapshot and a merging mechanism to coordinate updates across different graph versions.

5.3 Concurrency Control

SVFusion employs a two-level concurrency control protocol to enable safe and efficient SANNS operations across hierarchical memory tiers. We first establish *consistency guarantees* through fine-grained locking and synchronized data structures, ensuring searches observe consistent data despite interleaved updates. Building on this foundation, a *multi-version mechanism* decouples expensive background consolidation from serial operations via snapshot isolation, enabling concurrent execution without blocking.

Consistency Guarantees. SVFusion provides a consistent view of the index across GPU, CPU, and disk tiers, ensuring that all queries and updates observe a coherent state of the hierarchical index. Consistency is maintained through fine-grained local synchronization and coordinated cross-tier updates over three shared structures: the CPU-resident graph, the multi-tier deletion bitset, and the CPU–GPU mapping table.

Local synchronization. Within each memory tier, fine-grained locks coordinate concurrent operations. Searches traverse neighbor lists under read locks, while insertions acquire read locks for candidate search and upgrade to write locks only on affected neighbor lists during edge establishment. Deletions update the shared bitset under exclusive lock while marking the affected version, allowing other operations to skip logically deleted vertices.

Cross-tier coordination. To maintain consistency during updates, SVFusion adopts a two-phase synchronization protocol between CPU and GPU memory. For deletions, bitsets are atomically updated across tiers under exclusive lock to ensure consistent visibility. For topology updates (insertions, repairs, or consolidations), modified neighbor lists are first committed on the CPU under write lock with an incremented version, while corresponding GPU cache entries retain their versions. These updates are then asynchronously propagated to the GPU in batches, with version identifiers updated upon transfer completion. During this transient phase, GPU queries encountering version conflicts transparently fall back to CPU memory to ensure consistent results. This lightweight version control ensures asynchronous updates never overwrite concurrently modified regions, preserving correctness while enabling high-throughput, non-blocking propagation across memory tiers.

Figure 6 illustrates this with concurrent search and insertion. As the search traverses the GPU graph, an insertion for vertex 9 arrives and locks vertex 5's neighbor list during candidate selection (middle). After edges are established on CPU with version t_1, the lock is released. When the search detects a version mismatch at affected vertices, it falls back to CPU to ensure data consistency.

Multi-version Mechanism. To decouple expensive background consolidation from online vector operations, SVFusion adopts a multi-version concurrency mechanism that maintains a small set of consistent graph versions across memory tiers. Each version provides a complete, immutable snapshot of the graph topology for concurrent background processing, while new operations proceed on the active version under consistency guarantees.

A new graph version is created whenever a background task (e.g., consolidation or large-scale repair) begins. The system duplicates the current graph metadata as a read-only snapshot G_{t_0} for task execution. Insertions and deletions continue on the active graph G_{t_1}. Upon completion, the background task produces an updated graph G'_{t_0}, which is merged back into G_{t_1} to form a unified version. To avoid unbounded version accumulation, SVFusion enforces a bounded-version policy, deferring new version creation once the limit is reached and scheduling consolidation serially.

During merging, we first identify new vertices inserted after the snapshot was taken, and append the corresponding induced subgraph to the consolidated result. This incremental approach ensures that recently inserted vertices are retained in the merged structure without reprocessing the entire graph. To maintain bidirectional connectivity, we record reverse edge updates that occur between existing and newly inserted vertices during the consolidation window $[t_0, t_1]$. Before committing the new version, SVFusion atomically applies these updates by augmenting each affected vertex neighborhood with the corresponding reverse edges, ensuring both structural completeness and connectivity preservation.

6 EVALUATION

6.1 Experimental Setup

Hardware Configuration. The experiments are conducted on a dual-socket server running Ubuntu 18.04.6 LTS, equipped with two Intel Xeon Gold 5218 CPUs (32 physical cores, 64 threads, forming two NUMA nodes), 376 GB DDR4 DRAM (12 channels, 256 GB/s), an NVIDIA A100 GPU with 40 GB device memory (PCIe 3.0), and Intel D3-S4510 960 GB enterprise SSDs for persistent storage.

Datasets. We evaluate SVFusion on five widely used real-world datasets with diverse dimensions and data types, which have been extensively utilized [13, 44, 52]. Table 2 summarizes the detailed

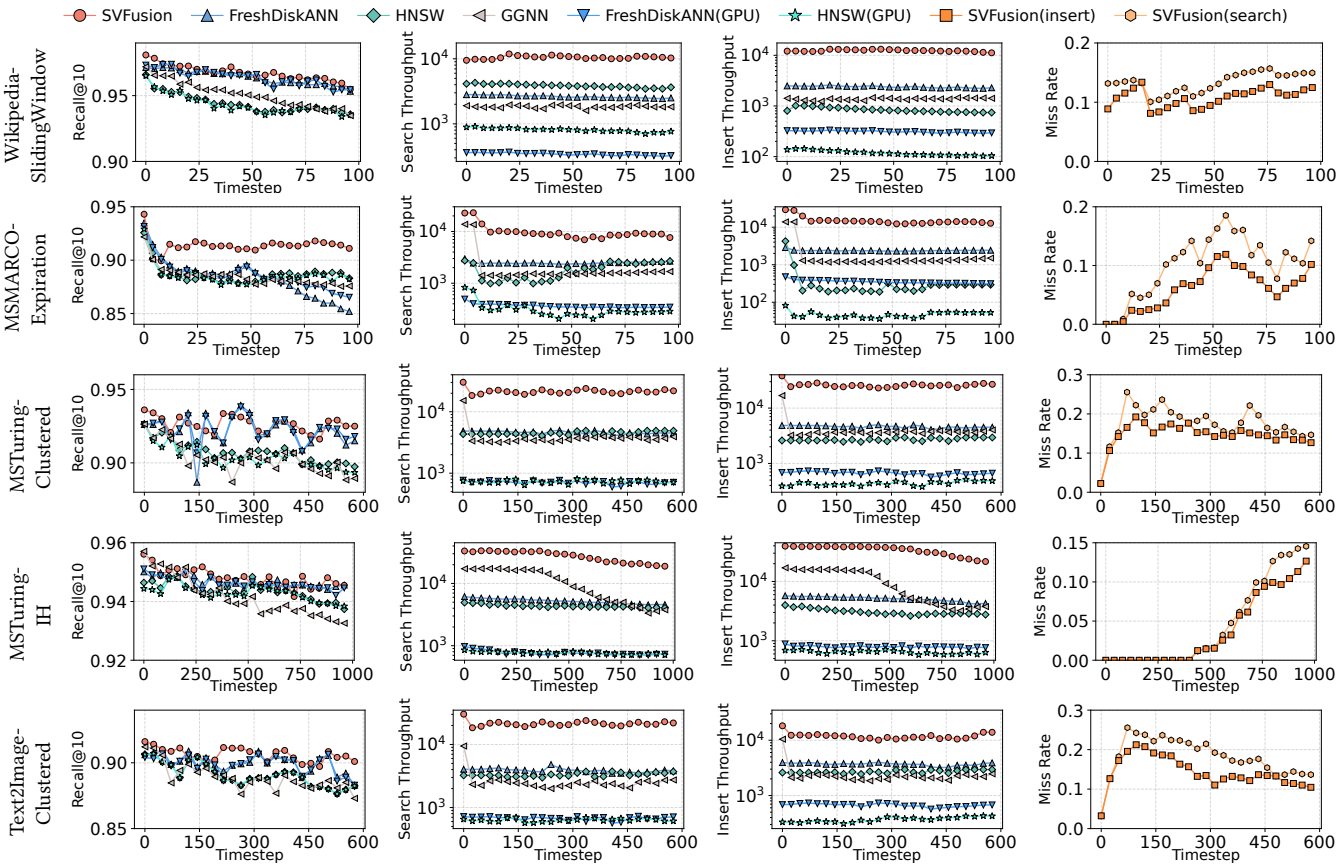


Figure 7: Comparison of recall, search throughput, insert throughput, and miss rate for five various workloads.

Table 2: Statistics of experimental datasets.

Datasets	N	Size (GB)	D	N_q	Type
Wikipedia [8]	35,000,000	101	768	5,000	Text
MSMARCO [5]	101,070,374	293	768	9,376	Text
MSTuring [4]	200,000,000	75	100	10,000	Text
Deep1B [16]	1,000,000,000	358	96	10,000	Image
Text2Image [50]	100,000,000	75	200	10,000	Image

properties of each dataset. Queries are randomly sampled from each dataset, with ground truth computed via exhaustive linear scan.

Workloads. Our evaluation covers diverse streaming workload patterns, each consisting of a collection of vectors and an operation sequence as defined in §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 simulate realistic streaming scenarios, we continuously consume search, insertion, and deletion vectors from a Kafka source at varying QPS rates. We adopt three representative workload patterns from the *2023 Big ANN Challenge* [52] and *MSTuring-IH* generated in [44]:

- **SlidingWindow**: The dataset is evenly divided into $T_{max} = 200$ segments. One segment is inserted per step, and from step $T = T_{max}/2 + 1$, the segment inserted $T_{max}/2$ steps earlier is deleted. Evaluation begins at $T_{max}/2 + 1$. This workload simulates a sliding window that retains only the most recent data.
- **ExpirationTime**: Vectors are assigned different lifetimes—short-term (10 steps), long-term (50 steps), and permanent (100 steps)—in a 10:2:1 ratio. At each step, $1/T_{max}$ of the dataset is inserted, with expired vectors removed accordingly. This setup evaluates index behavior under heterogeneous data lifespans.
- **Clustered**: The dataset is partitioned into 64 clusters using k-means across 5 rounds. Each round alternates between insertion and deletion phases: randomly inserting and then deleting vectors from each cluster. As spatially adjacent points are modified together, this workload creates strong distributional shifts.
- **MSTuring-IH**: Starting with 20 million vectors, the dataset grows to 200 million over 1,000 operations with a 90% insert and 10% search ratio, testing scalability and query stability under large-scale incremental growth.

Implementation and Baselines. We implement SVFusion by extending the cuVS 24.02 library [3] following the architecture in Figure 2. We compare SVFusion with the following baselines:

- **HNSW** [42] is a representative graph-based ANNS method, extensively deployed in industrial systems and studied in academia. We use $M = 48$, and $ef\_construction = ef\_search = 128$.
- **FreshDiskANN** [53] is the state-of-the-art graph-based ANNS method that supports dynamic scenarios on memory or SSDs. We configure FreshDiskANN with the settings in [2]: $R = 64, l_b = l_s = 128, \alpha = 1.2$ for insertion, and $l_d = 128$ for search.
- **CAGRA** [47] is NVIDIA's high-performance ANNS algorithm designed for GPUs. It achieves superior performance but encounters GPU memory limitations for large-scale datasets.

• **PilotANN** [26] is a recent hybrid architecture that effectively enables GPU acceleration, but does not support dynamic updates.
• **GGNN** [25] is a GPU-based ANNS method that constructs hierarchical graph structures similar to HNSW, but is designed for static datasets that fit within GPU memory capacity.

Existing GPU-based methods either support only static construction [47, 68, 69] or have limited dynamic capabilities constrained by GPU memory [56]. For fair comparison on streaming workloads, we create GPU-accelerated versions of FreshDiskANN and HNSW by offloading distance computations to GPU, and extend GGNN to support updates with GPU cache management.

Evaluation Metrics. We focus on four key metrics: (1) **Throughput**. It measures the number of search queries and update operations (insertions/deletions) processed per second under streaming workloads; (2) **Latency**. It denotes the end-to-end time from when a request is submitted until the final result is returned; (3) **Recall**. Recall@k denotes the recall for each search query, with k=10 by default; (4) **GPU cache miss rate.** It is 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 Overall Performance

To evaluate the overall performance of SVFusion, Figure 7 presents recall, search throughput, insert throughput, and GPU miss rate across seven workloads on five datasets. These workloads involve high update volumes, with performance targets of high throughput and bounded p99 latency within 900 ms [39]. Delete throughput is not included since all methods use logical markers for deferred deletions, making instantaneous rates incomparable. Overall, SVFusion consistently outperforms all baselines, achieving up to 9.5× higher search throughput and 71.8× higher insertion throughput than FreshDiskANN, while maintaining the highest recall throughout.

Recall Analysis. SVFusion maintains consistently high search quality across all streaming workloads, achieving Recall@10 scores between 91% and 96%, comparable to or surpassing all baselines. It outperforms FreshDiskANN with an average recall improvement of 0.4–3.4%, attributed to its **localized topology-aware repair mechanism** that proactively detects and restores critical graph connectivity, mitigating structural degradation over time. In contrast, HNSW yields the lowest recall due to its lack of a deletion repair strategy, leading to persistent graph fragmentation. All methods exhibit recall fluctuations under continuous structural updates.

Throughput Analysis. Figure 7 demonstrates that SVFusion achieves the highest throughput for both search and insertion operations, delivering on average 20.9× higher search throughput and 3.5× higher insertion throughput than FreshDiskANN and HNSW. Notably, GPU-accelerated baselines, such as GGNN run 4.5–7.8× slower than SVFusion once the dataset exceeds GPU capacity, as frequent CPU–GPU transfers negate the benefits of GPU acceleration. During early timesteps of workloads initialized from empty datasets (e.g., *ExpirationTime*), search throughput peaks at 40K due to fully utilized GPU computation and minimal cache misses (<1%).

Miss Rate Analysis. We also report GPU cache miss rates for insertion and search operations to better characterize multi-tier memory behavior. Despite GPU memory representing only a small fraction of the dataset, SVFusion maintains consistently low miss rates across dynamic workloads, demonstrating the effectiveness of its cache placement strategy in capturing temporal and graph structural access patterns and minimizing redundant evictions.

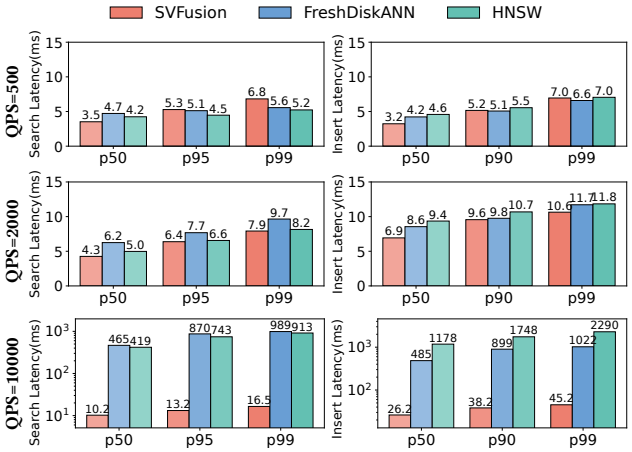


Figure 8: Comparison of p50, p95 and p99 query latencies.

Latency Analysis. To comprehensively evaluate SVFusion's latency performance under varying streaming intensities, we scale the request rate from low (QPS=500) to high (QPS=10,000) on the MSTuring dataset. Figure 8 reports the p50, p95, and p99 latencies for both search and insertion operations. At low request rates, SVFusion shows comparable performance with a moderate 15–30% increase in p99 latency due to CPU–GPU synchronization overhead that cannot be effectively amortized under limited query concurrency. Despite this effect, all observed latencies remain below 10 ms, indicating minimal real impact on responsiveness. Under moderate throughput, SVFusion demonstrates strong latency stability, achieving p50 search latency of 4.3 ms and p99 of 7.9 ms, representing 32% and 18% reductions compared to FreshDiskANN and HNSW, respectively. This improvement results from the multi-stream coordination mechanism, which overlaps computation and data transfer to mitigate blocking across concurrent operations. At high request rates, the advantage becomes even more pronounced: while baseline methods experience severe queue buildup with p99 latencies exceeding 900 ms (45–50.7× slower), SVFusion sustains tail latencies of 16.5 ms for search and 45.3 ms for insertion. These results highlight SVFusion's scalable coordination design, achieving both high throughput and consistently low latency under heavy load.

6.3 Component Analysis

Effectiveness of Adaptive Vector Caching. We conduct two complementary experiments to evaluate our caching strategy with four alternatives: *w/o WAVP*, which always computes distances on CPU for cache misses without data transfer; *LRU/LFU/LRFU*, which replace our WAVP mechanism with traditional caching strategies.

Replacement Strategies. We first evaluate the effectiveness of our workload-aware vector placement (WAVP) strategy under the *SlidingWindow* workload. Figure 9 demonstrates that SVFusion with WAVP achieves the highest throughput and lowest latency, outperforming all baseline methods by up to 7.2× in throughput

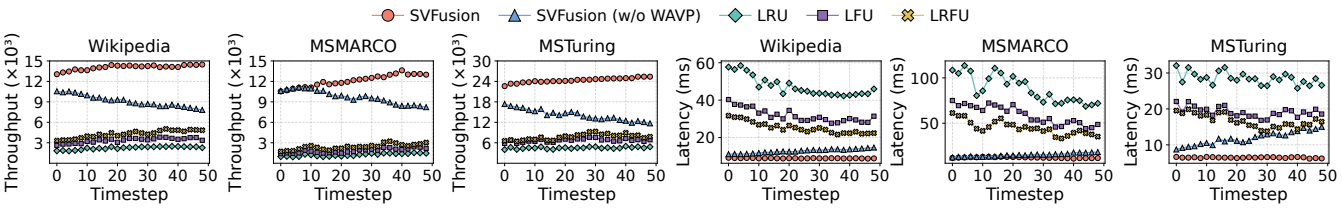


Figure 9: Comparison of different replacement strategies on update throughput and latency.

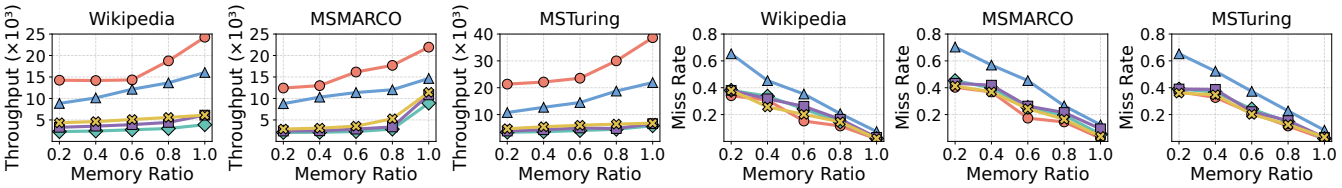
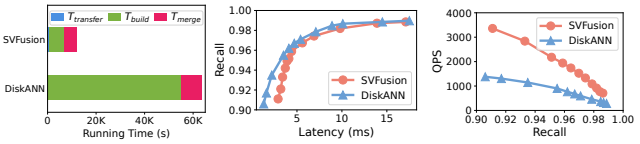


Figure 10: Impact of different GPU memory ratios on update throughput and miss rate.



(a) Construction time **(b) Recall-Latency** **(c) Recall-Throughput**

Figure 11: Effect of GPU-CPU-disk framework

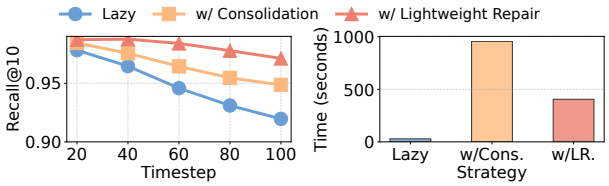


Figure 12: Effect of deletion strategies.

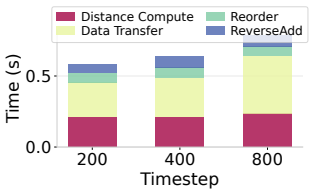


Figure 13: Perf. Breakdown.

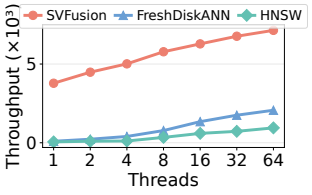


Figure 14: Effect of #Threads.

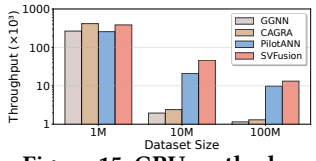


Figure 15: GPU methods.

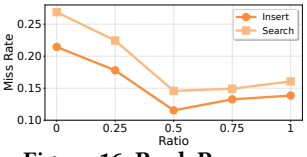


Figure 16: Pred. Param.

and 5.1× in latency reduction. These gains arise because traditional cache policies overlook workload dynamics, causing excessive vector replacements and costly CPU–GPU data transfers. By adaptively modeling temporal and structural access patterns, WAVP minimizes unnecessary migrations and maximizes GPU utilization, achieving high efficiency and stable performance across varying workloads.

Memory Constraints. To evaluate the robustness of our caching strategy under varying GPU memory, we use the same dataset while scaling available GPU memory from 20% to 100%. As shown in Figure 10, our method consistently outperforms all baselines across all settings, achieving 1.7×, 3.3×, 4.1×, and 5.6× higher average throughput than SVFusion (w/o WAVP), LRFU, LFU, and LRU, respectively. Notably, even under reduced GPU capacity, which increases cache miss rates, SVFusion maintains stable throughput. This demonstrates that our design effectively overlaps CPU computation with data transfers, hiding memory latency and sustaining high performance under GPU memory constraints.

Effectiveness of Disk tier. We evaluate the disk-extended SVFusion framework on the Deep1B dataset [16]. As shown in Figure 11a, SVFusion achieves a 5.26× faster index construction compared with DiskANN, primarily attributed to a 9.1× acceleration in GPU-based subgraph building. However, the disk merging stage remains the primary bottleneck. Figure 11b and 11c present the throughput and latency performance across different recall levels. At high recall targets, SVFusion achieves latencies 0.7–1.6% higher than those of DiskANN. This minor degradation stems from the increased data partitioning necessitated by limited GPU memory, which slightly weakens cross-partition connectivity. However, SVFusion outperforms in throughput by 2.3× at comparable recall levels, highlighting the effectiveness of our hierarchical index design.

Effect of Deletion Strategies. We compare three approaches: lazy deletion alone, lazy deletion with global consolidation, and our method combining both with localized repair. As shown in Figure 12, 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.

Performance Breakdown and Scalability. We break down the time cost of insertion of SVFusion into key components (Figure 13): *data transfer* (45.4%), *distance computation* (33.6%), *heuristic reordering* (10.3%), and *reverse add* (10.6%). This highlights two insights: (1) despite GPU acceleration, CPU–GPU data movement remains the dominant bottleneck as index size grows; (2) distance computation remains stable, demonstrating the effectiveness of our pipelined CPU–GPU parallelism. We also evaluate scalability by varying thread counts from 1 to 64 (Figure 14). While all methods show throughput improvements with more threads, the gains diminish beyond 16 threads, with an average of only 1.24×.

Table 3: Impact of consistency guarantee.

QPS	Recall@1				p99 Latency (ms)			
	500	2K	5K	10K	500	2K	5K	10K
w/ sync	0.96	0.96	0.96	0.96	4.3	6.7	10.8	33.2
w/o sync	0.96	0.74	0.51	0.18	4.3	4.8	5.5	7.4

Effectiveness of CPU-GPU vector search. Figure 15 illustrates the CPU-GPU search performance on MSMARCO across different dataset scales under Recall@10=0.9. 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 and GGNN experience significant performance degradation due to inefficient CPU-GPU data transfers through unified virtual memory. Compared to PilotANN, SVFusion achieves 1.5×-2.2× speedup across all scales due to PilotANN's CPU-side refinement. This demonstrates that SVFusion can efficiently handle large-scale vector search without data compression.

Effectiveness of Consistency Guarantee. To validate our concurrency control protocol, we design a stress test where each batch contains an interleaved mix of 50% insertions and 50% search, with batch sizes of 10 and request rates ranging from 500 to 10K. Recall@1 is used as the primary metric, as a correct system should always return the newly inserted vector as the nearest neighbor. As shown in Table 3, SVFusion with synchronization maintains stable Recall@1 at 0.96 across all request rates, confirming strong read-after-write consistency. In contrast, the *no-synchronization* variant suffers severe degradation, dropping from 0.96 at 500 QPS to 0.18 at 10K QPS (an 81% decline), as searches increasingly observe partially updated or inconsistent graph states. Although enforcing synchronization increases p99 latency from 7.4ms to 33.2ms, this overhead is bounded and justified, as it guarantees correctness under extreme streaming workloads. These results demonstrate that our fine-grained, multi-tier coordination protocol effectively balances consistency and performance in high-throughput SANNS.

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 16 shows that higher weights for recent accesses generally reduce miss rates, with optimal performance achieved at a ratio around 0.6.

Batch Size. We analyze the effect of batch size on SVFusion's performance in Figure 17. Larger batches substantially increase throughput but degrade recall beyond 2^{13} due to delayed graph updates. Latency also grows exponentially, exceeding one second at 2^{11}, reflecting the classic latency–throughput trade-off. SVFusion supports adaptive batch sizing to dynamically balance throughput and query latency for streaming ANNS workloads.

7 RELATED WORK

Vector Indexes. ANNS is a classical problem with a large body of research work. Partition-based methods [15, 22, 24, 28, 33, 41, 44, 66] use tree structures and quantization to organize vectors into spatial

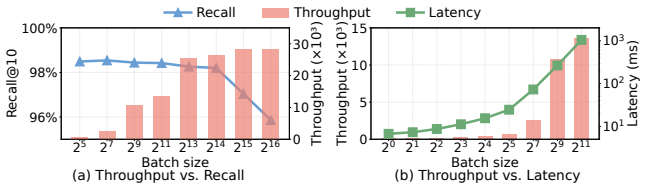


Figure 17: Impact of varying batch size of updates.

partitions but suffer from the curse of dimensionality. Hash-based approaches [21, 30, 58, 70] project high-dimensional vectors into lower-dimensional hash buckets to accelerate similarity search. Graph-based methods [14, 20, 32, 49, 64, 71] construct proximity graphs to enable efficient vector 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 scenarios. FreshDiskANN [53] maintains new vectors in memory before periodically merging them into a disk-resident graph. SPFresh [63] implements a cluster-based approach with lightweight incremental rebalancing that adjusts partitions as data distribution changes. Other approaches [56, 61, 62] focus on maintaining graph connectivity or parallel acceleration. 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 extensively explored to address computational demands at scale. Faiss [35] and Raft [1] pioneered CPU-to-GPU migration, while SONG [69], GANNS [65], and CAGRA [47] improved GPU parallelism through algorithmic and kernel optimizations. For system design optimization, [25, 26, 36, 54, 68] 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.

8 CONCLUSION

We presented SVFusion, a GPU-CPU-disk collaborative framework for large-scale and real-time 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 structures. SVFusion further incorporates a concurrency control protocol that ensures data consistency while maintaining high throughput and low latency. Experimental results demonstrate that SVFusion achieves up to 71.8× higher throughput and 1.3× to 50.7× lower latency compared to state-of-the-art methods across various streaming workloads.

ACKNOWLEDGMENTS

This work was supported by the "Pioneer" R&D Program of Zhejiang (2025C01001), CCF-Huawei Populus Grove Fund (Grant No. CCF-HuaweiDB202408), the National Nature Science Foundation of China (62572434), the Fundamental Research Funds for the Central Universities (226-2024-00145).

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. Accessed: 2026-01-08.

[2] 2023. NeurIPS 2023 Streaming Challenge and Beyond. https://github.com/harsha-simhadri/big-ann-benchmarks/tree/main/neurips23/streaming. Accessed: 2026-01-08.

[3] 2024. cuVS: Vector Search and Clustering on the GPU. https://github.com/rapidsai/cuvs/tree/v24.12.00. Accessed: 2026-01-08.

[4] 2024. Microsoft Turing-ANNS-1B. https://big-ann-benchmarks.com/neurips21.html. Accessed: 2026-01-08.

[5] 2024. microsoft/ms_marco. https://huggingface.co/datasets/microsoft/ms_marco. Accessed: 2026-01-08.

[6] 2024. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates. Accessed: 2026-01-08.

[7] 2024. updating an index. https://github.com/rapidsai/cuvs/issues/295. Accessed: 2026-01-08.

[8] 2024. wikimedia/wikipedia. https://huggingface.co/datasets/wikimedia/wikipedia. Accessed: 2026-01-08.

[9] 2025. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. Accessed: 2026-01-08.

[10] 2025. Personalized help from Gemini. https://gemini.google/overview/personalization. Accessed: 2026-01-08.

[11] 2025. The Platform for Building Stateful Agents. https://www.letta.com. Accessed: 2026-01-08.

[12] 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. Accessed: 2026-01-08.

[13] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *Information Systems* 87 (2020), 101374.

[14] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. ELPIS: Graph-Based Similarity Search for Scalable Data Science. *PVLDB* 16, 6 (2023), 1548–1559.

[15] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *TPAMI* 37, 6 (2014), 1247–1260.

[16] Artem Babenko and Victor Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *CVPR*. 2055–2063.

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

[18] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures. In *WWW*. 577–586.

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

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

[21] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*. 541–552.

[22] 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. *SIGMOD* 3, 3 (2025), 1–26.

[23] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *SIGMOD* 1, 2 (2023), 27.

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

[25] 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.

[26] 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).

[27] 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. *PVLDB* 15, 12 (Aug. 2022), 3548–3561.

[28] Guoyu Hu, Shaofeng Cai, Tien Tuan Anh Dinh, Zhongle Xie, Cong Yue, Gang Chen, and Beng Chin Ooi. 2025. HAKES: Scalable Vector Database for Embedding Search Service. *PVLDB* 18, 9 (2025), 3049–3062.

[29] 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 *KDD*. 2553–2561.

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

[31] Masajiro Iwasaki. 2016. Pruned Bi-directed K-nearest Neighbor Graph for Proximity Search. In *International Conference on Similarity Search and Applications*, Vol. 9939. 20–33.

[32] 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. *NeurIPS* 32 (2019).

[33] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *TPAMI* 33, 1 (2011), 117–128.

[34] 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).

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

[36] 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).

[37] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE transactions on Computers* 50, 12 (2001), 1352–1361.

[38] 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 *NIPS*.

[39] 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.

[40] Ting Liu, Andrew Moore, Ke Yang, and Alexander Gray. 2004. An Investigation of Practical Approximate Nearest Neighbor Algorithms. *NIPS* 17 (2004).

[41] 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. *PVLDB* 18, 4 (2024), 986–999.

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

[43] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *SIGPLAN*. 270–285.

[44] 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. In *OSDI*. Article 9, 17 pages.

[45] NVIDIA. 2025. Streaming Data to RAG. https://build.nvidia.com/nvidia/streaming-data-to-rag/blueprintcard. Accessed: 2026-01-08.

[46] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *SIGMOD*. 297–306.

[47] 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 *ICDE*. 4236–4247.

[48] 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).

[49] Y Peng, B Choi, TN Chan, J Yang, and J Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *SIGMOD* (2023), 27.

[50] Yandex Research. 2021. Benchmarks for Billion-Scale Similarity Search. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search. Accessed: 2026-01-08.

[51] Chenghao Rong, Jessie Hui Wang, Juncai Liu, Jilong Wang, Fenghua Li, and Xiaolei Huang. 2021. Scheduling Massive Camera Streams to Optimize Large-Scale Live Video Analytics. *IEEE/ACM Transactions on Networking* 30, 2 (2021), 867–880.

[52] 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).

[53] 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).

[54] 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.

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

[56] Yiping Sun, Yang Shi, and Jiaolong Du. 2024. A Real-Time Adaptive Multi-Stream GPU System for Online Approximate Nearest Neighborhood Search. In *CIKM*. 4906–4913.

[57] 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 *FAST*. 171–185.

[58] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2023. DB-LSH 2.0: Locality-sensitive hashing with query-based dynamic bucketing. *TKDE* 36, 3 (2023), 1000–1015.

[59] 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).

[60] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *PVLDB* 14, 11 (2021), 1964–1978.

[61] 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).

[62] 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).

[63] 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 *SOSP*. 545–561.

[64] Shuo Yang, Jiadong Xie, Yingfan Liu, Jeffrey Xu Yu, Xiyue Gao, Qianru Wang, Yanguo Peng, and Jiangtao Cui. 2025. Revisiting the Index Construction of Proximity Graph-Based Approximate Nearest Neighbor Search. *PVLDB* 18, 6 (2025), 1825–1838.

[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 *ICDE*. IEEE, 552–564.

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

[67] 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 *KDD*. 4493–4501.

[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 *NSDI*. 23–40.

[69] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *ICDE*. 1033–1044.

[70] 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. *PVLDB* 13, 5 (2020), 643–655.

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