CCD-Level and Load-Aware Thread Orchestration for In-Memory Vector ANNS on Multi-Core CPUs

Yuchen Huang*†‡, Baiteng Ma*†‡, Yiping Sun‡, Yang Shi‡, Xiao Chen‡
Xiaocheng Zhong‡, Zhiyong Wang‡, Yao Hu‡, Chuliang Weng†
†*East China Normal University*, ‡*Xiaohongshu Inc (RedNote)*
†{ychuang, btma}@stu.ecnu.edu.cn, ‡{sunyiping, shiyang1, chenxiao4}@xiaohongshu.com,
‡{zhongxiaocheng, sunzhenghuai, xiahou}@xiaohongshu.com, †clweng@dase.ecnu.edu.cn

arXiv:2605.10090v1 [cs.IR] 11 May 2026

***Abstract*—Vector approximate nearest neighbor search (ANNS) underpins search engines, recommendation systems, and advertising services. Recent advances in ANNS indexes make CPU a cost-effective choice for serving million-scale, in-memory vector search, yet per-core throughput remains constrained by memory-access latency of vector reading and the compute intensity of distance evaluations in production deployments. With the growing scale of the business and advances in hardware, modern CCD-based multi-core CPUs have been widely deployed for high throughput in our services. However, we find that simply increasing core counts does not yield optimal performance scaling.**

To improve the efficiency of more cores from the CCD-based architecture, we analyze the distributions of real-world requests in our production environments. We observe high access locality in vector search in our online services and low cache utilization, resulting from overlooking the multi-chiplet nature of CCD-based CPUs. Hence, we propose a workload- and hardware-aware thread orchestration framework at CCD-level that (i) provides a uniform interface for both inter-query parallel HNSW search and intra-query parallel IVF search, (ii) achieves cache-friendly and workload-adaptive mapping of task dispatching, and (iii) employs CCD-aware task stealing to address load imbalance.

Applied to real production workloads from search, recommendation, and advertising services of Xiaohongshu (RedNote), our approach delivers up to 3.7× higher throughput and 30–90% reductions in P50 and P999 latency. In detail, compared with the original framework, the cache-miss ratio decreases by 6–30%, and the total CPU stall is reduced by 20–80%.

***Index Terms*—Multi-core architecture, Vector search, Thread orchestration.**

I. Introduction

Multimodal unstructured data (e.g., images, documents, and videos) are usually embedded into high-dimensional vectors for efficient retrieval [1]–[3]. At RedNote [4], our search engine, recommendation system, and advertising services deploy thousands of million-scale vector tables with Approximate nearest neighbor search (ANNS). Moreover, with hundreds of millions of monthly active users (MAU) [5], we must sustain tens of millions of queries per second (QPS) under millisecond-level latency across these tables. While GPUs offer fast index construction via massive parallelism and high memory bandwidth [6], [7], in-memory vector indexes (e.g., HNSW, Vamana, and IVF [8]–[10]) on CPUs are cost-effective choices for low-latency online serving to meet the actual performance requirements.

*These authors contributed equally to this work.

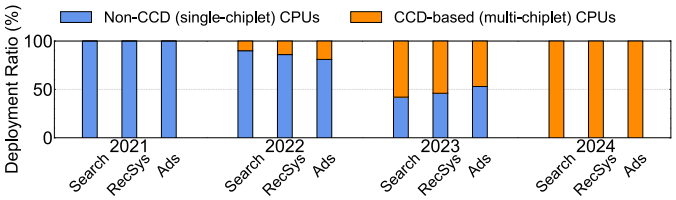


Fig. 1: ***The proportions of non-CCD and CCD-based multi-core CPUs deployed on services of RedNote in recent years.***

Driven by the need to push throughput under strict service-level agreements (SLA) of response latency and recall rate, our services are urgent for CPU cores. At the same time, as monolithic CPU scaling currently hits reticle-size, yield, and cost limits, vendors expand core counts of these modern CPUs through Core-Complex-Dies (CCD) architectures [11], [12]. Different from the scaling of cores by adding more sockets and processors as NUMA (Non-Uniform Memory Access) nodes (i.e., inter-socket scaling), CCD-based CPUs scale cores in a single socket (i.e., intra-socket scaling).

With a steady year-on-year increase in the share of these CCD-based CPU architectures in the server market [13], [14], we have started to adopt these CPUs for vector search in online services recently. Across the services of RedNote, including search engine, recommendation system, and advertising service, the ratio of CCD-based CPUs in production (shown in Figure 1) has steadily increased and has now nearly displaced the previously used single-chiplet CPUs.

Unfortunately, simply adding more CPU cores with hardware upgrades does not yield optimal performance scaling in real workloads. To figure out the reasons, we analyze the real-world workload characteristics of vector search in our industrial production environments and the hardware traits of modern CPU chiplets. We observe: (1) within sequential short time windows, queries hitting the same vector table tend to probe overlapping subsets of vectors, exhibiting strong memory locality; (2) different CCDs have the independent last level cache, therefore improper CCD–request dispatches can incur chiplet-level cache pollution and break affinity, intensifying memory contention; and (3) search overhead differs significantly across vector tables, while the naïve way to solve this load imbalance can incur a low ratio of cache reusing from cross-chiplet stealing.

To unleash the potential of CCD-based multi-core CPUs,

we design a CCD-level and load-aware thread orchestration framework for in-memory vector ANNS.

First, aiming at the graph-based HNSW index [8] and the clustering-based inverted-file (IVF) index [10], which are widely used in our systems and other industrial online services [15]–[17], we define a uniform task submission interface. It can easily replace the previous framework to enable search parallelization for both inter-query HNSW and intra-query IVF. Second, we design a task dispatcher, achieving a cache-friendly mapping between search tasks and chiplets, and a workload monitor to adapt the mapping to dynamic requests. It balances the skew memory traffic of multiple HNSW tables or clusters in IVF, minimizing the L3 cache pollution across requests. Third, we develop a CCD-aware stealing strategy to address the load imbalance of search tasks, which reduces cross-chiplet steals to minimize cache waste.

Prior studies have well optimized in-memory ANNS across multiple aspects—including search process [18]–[21], distance computation [22], [23], parameter tuning [22], [24], and quantization compression [25], [26]. However, to the best of our knowledge, no prior work has targeted thread orchestration optimizations for in-memory ANNS to improve search performance on modern multi-core CPU architecture and real-world industrial workloads. In this paper, we focus on the perspective of thread orchestration in vector search, achieving 1.4×-3.7× throughput and reducing latency by up to 90%. Our main contributions and key techniques can be summarized as follows:

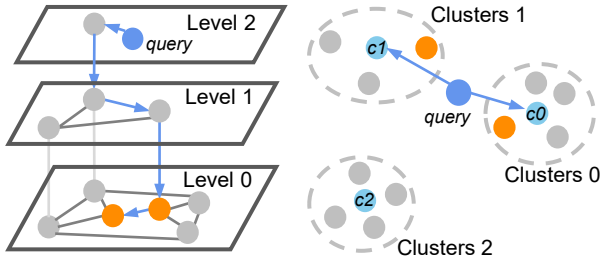
- We share our deployment patterns of vector search comprehensively and describe the characteristics of real-world workloads in our online industrial services.
- We design a CCD-level and load-aware thread orchestration framework with a drop-in and compatible interface to integrate with inter-query HNSW and intra-query IVF.
- We present how to balance dynamic memory traffic across search tasks in task dispatching and mapping adaptation, in order to achieve minimized cache pollution.
- We propose to make CCD-aware stealing strategy according to the hardware topology when balancing the skew overhead of different search tasks.
- We conduct real-world evaluations to show the efficiency, including end-to-end performance improvements and detailed gains (e.g., cache hit/miss, CPU stall).

II. Background

We introduce structures and search processes of two vector indexes widely adopted in industrial practice—graph-based HNSW and clustering-based IVF. We then discuss how these indexes are organized and operated in our online production services, including deployment choices with scenarios. Finally, we present the micro architecture of modern multi-core CPUs (i.e., CCD-based multi-chiplet CPU).

A. Index Structure and Search

In this paper, we focus on the HNSW and IVF, because they are the most popular indexes for vector search and have been deployed at large scale in our production services.



(a) ***Process of HNSW.*** (b) ***Process of IVF.***

Fig. 2: ***Structures and search processes of HNSW and IVF.***

1) Graph-based HNSW: In HNSW, each vector draws a maximum level from a geometric distribution and is inserted into all levels below. Points link up to M neighbors to nearby ones on the same level; higher levels are sparser shortcuts, while the bottom (*Level 0*) is dense and holds all vectors.

As shown in Figure 2a, starting from an entry point on the top level (i.e., *Level 2*), the query performs greedy descent: at each level, it repeatedly moves to the nearest points, then drops one level. After arriving at *Level 0*, a best-first search explores a candidate queue of size *efSearch*, maintaining the current top-k. The queue stops updating when all remaining candidates cannot find nearer neighbors, and the final top-k is returned.

2) Clustering-based IVF: In IVF, vectors are partitioned into *nlist* clusters via k-means; each vector is assigned to its nearest centroid and stored in the corresponding inverted list.

As shown in Figure 2b, for a query, we first compute its distances to all centroids (i.e., *c0, c1, c2*) and select the *nprobe* closest clusters (i.e., *Cluster 1 and 0*). Then, it scans the lists of nearest clusters: computing distances between query and candidates from these lists, and ranking candidates to obtain the final top-k.

B. Deployment Choice

Graph-based HNSW and cluster-based IVF offer complementary trade-offs between construction overhead and hardware requirements of low-latency search.

HNSW index can minimize per-query distance evaluations, so millisecond-level latency is commonly achievable for the search process on a single core. Its drawback is costly construction and updating. Building HNSW can take several minutes to hours at 1–100M scale and updating can take milliseconds, which makes frequent full rebuilds and inserts inefficient for workloads requiring freshness [27].

By contrast, IVF performs a coarse k-means partition followed by list assignment and can be (re)built within seconds to a few minutes and offer microsecond-level inserts, which suits high-freshness deployments [28]. However, query processing requires scanning many candidates across several probed lists. Achieving millisecond latency per query therefore relies on the intra-query parallelism of scanning lists and evaluating distances across multiple threads/cores (e.g., FAISS via per-list scanners and multi-threaded OpenMP [29]).

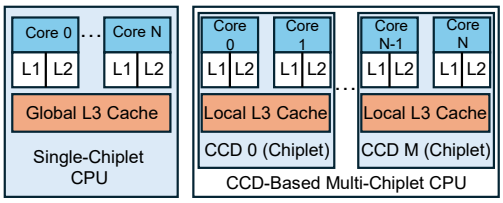


Fig. 3: ***Single-chiplet CPU vs. CCD-based multi-core CPU.***

Hence, in practice, for services whose serving patterns favor graph indexes, we co-locate multiple HNSW indexes on a single node and execute each table's queries on a single core to maximize system throughput. For services suited to IVF, we parallelize each query of indexes across multiple CPU cores on a node, distributing scans over the probed lists. We detail these two design choices in subsection III-A.

C. CCD-based CPU Architecture

From the perspective of hardware development, as process scaling slows, server CPUs increase core counts through multi-chiplet designs rather than monolithic dies. Recent AMD EPYC processors (e.g., Rome, Milan, Genoa, and Turin) [30], [31] and Intel's next-generation Sapphire Rapids [32] all adopt multi-chiplet architectures. And multi-core CPUs of this type have been widely applied in our services.

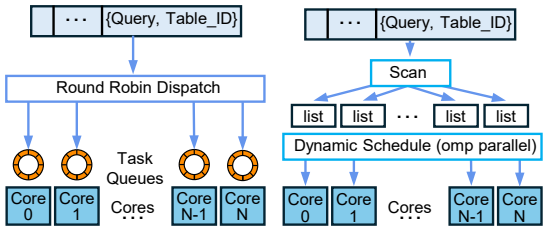
As shown in Figure 3, taking the AMD EPYC series as an example, multiple Core Complex Dies (CCDs) are integrated into one CCD-based multi-core CPU. Each CCD groups several cores that share a local last-level (i.e., level-3) cache (LLC/L3). The key shift from legacy single-chiplet CPUs is that the L3 cache is no longer implemented as one large, globally shared structure; instead, each chiplet (i.e., CCD) integrates its own local L3 cache, and these caches are only directly visible and low-latency to the cores within that chiplet [33]. Note that CCD-based CPU scales the count of cores as a whole processor within a single socket, which is different from scaling cores by adding more sockets to equip more processors.

III. Motivation

A. Prior Thread Orchestrations in Services

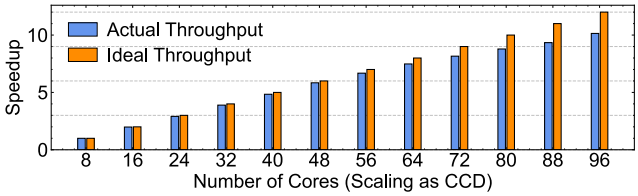
In this subsection, we describe prior patterns of task dispatch in our online production services, including inter-query parallelism of HNSW and intra-query parallelism of IVF.

Inter-query dispatch of HNSW. Since a single search based on HNSW typically completes within milliseconds on one core, we co-locate multiple HNSW tables on a node and drive throughput via inter-query parallelism. As shown in Figure 4a, incoming requests—each carrying *Query* (i.e., query_vector and top-k) and *Table_ID*—are enqueued by a lightweight dispatcher to per-core ring task queues. It uses a round-robin policy that cyclically assigns the next request to the next core's task queue, already providing O(1) scheduling overhead and avoiding centralized locks and long queues on hot cores.

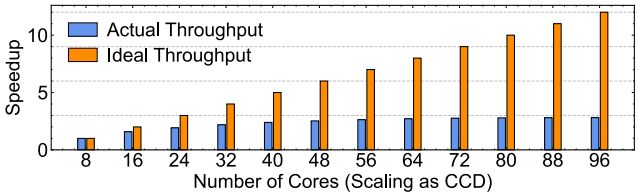


(a) ***Dispatch of HNSW.*** (b) ***Dispatch of IVF.***

Fig. 4: ***Orchestration of HNSW and IVF.***



(a) ***Performance scaling trend of the HNSW tables.***



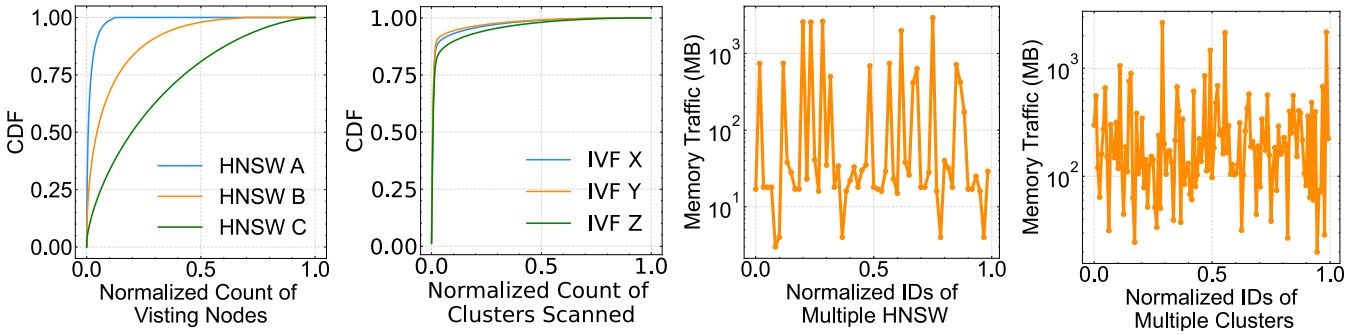
(b) ***Performance scaling trend of the IVF tables.***

Fig. 5: ***Scaling trends on the CCD-based multi-core CPU.*** Recall of HNSW and IVF can reach 99% and 95%, respectively, when requests' top-k is varying between top-100-500. These are referred to workloads from our online services.

Intra-query dispatch of IVF. For services with frequent updates and strict freshness, we adopt IVF, whose build/insert is fast but whose query is more compute-heavy than graph-based HNSW. Though the inter-query parallelism can achieve the optimal throughput for IVF indexes in theory because of less multi-thread communications than intra-query model, it can violate the latency SLAs due to the dispatch that only one core for each query. Hence, to keep millisecond latency, a single query is usually parallelized across multiple cores in our online services. As shown in Figure 4b, the query first identifies the closest posting lists; it then scans *nprobe* lists in parallel (e.g., *#pragma omp for schedule(dynamic)* in FAISS). In detail, it assigns work in each of all lists that threads pull from a shared task pool as they finish a prior one, whose pull-based policy balances load approximately and utilizes all cores for intra-query parallelism.

B. Inefficiency of Scaling Cores

Based on one of the platforms (the AMD 4th Gen EPYC 96-core processor) described in subsection VIII-A, we evaluate the scaling trends with real-world traces of our production services, including inter-query dispatch of 60 different HNSW indexes (each has 1-10 million vectors) co-located in one of our online nodes and intra-query dispatch of 15 IVF indexes



(a) ***Search access frequency of multiple HNSW tables.*** (b) ***Search access frequency of clusters on the IVF tables.*** (c) ***Memory traffic of multiple HNSW tables.*** (d) ***Memory traffic of clusters on the IVF table.***

Fig. 6: ***Distribution of search access frequency and memory access traffic.***

(each table contains 10 thousand to 15 million vectors from one node of our frequently-updating services). The results of these two cases are shown in Figure 5.

We continually increase the number of CPU cores available to the vector search workload to scale overall throughput. However, integrating these two patterns of task dispatching with the CCD-based multi-core CPU fails to deliver the anticipated improvements in throughput with scaling of cores.

Figure 5a shows throughput scaling under naïve inter-query dispatch across multiple HNSW tables as the core count increases in increments of CCD-level granularity. We observe $\sim$2$\times$ to $\sim$9.9$\times$ speedup when scaling from 16 to 96 cores. However, a substantial gap remains between the actual throughput measured and ideal throughput, with the 96-core configuration achieving only $\sim$82% of the ideal throughput (i.e., the count of CCDs $\times$ one CCD's throughput).

Figure 5b reports the throughput scaling of naïve intra-query dispatch in FAISS across 15 IVF tables on a single node. The trend echoes the efficient scaling of multi-table HNSW, but with poorer overall scaling. With fewer than 32 cores, there are still clear gains, whereas as the number of cores and threads increases, the incremental benefit becomes negligible. Across configurations with 2–12 CCDs, the speedup is only about 1.6$\times$ to 2.8$\times$.

These suggest that on modern CCD-based CPU architectures, adding more cores isn't being utilized effectively in vector search. There is a substantial performance gap, which also represents an opportunity for optimization.

C. Characterizing Workloads

The subsection II-A presents the search workflows of both the graph-based HNSW index and the clustering-based IVF index. In HNSW, the search proceeds by following the outgoing edges of each visited node to choose the next candidate; this iterative process involves a large amount of random memory access, making this stage highly memory-intensive on latency. Similarly, in IVF, when scanning each selected cluster (list), the query must be compared against every vector in that cluster to compute distances. This requires traversing all selected lists and imposes substantial pressure on memory bandwidth.

Hence, to alleviate memory pressure on ANNS—both latency and bandwidth—and better harness the performance of multi-core CPUs, we next analyze and discuss the memory-access workload characteristics of in-memory ANNS in our production setting, as shown in Figure 6, Figure 7 and Figure 8. It is collected from online workload logs within a short time window (about one minute), and the data of vectors is the same as previous ones in subsection III-B.

First, the search access frequency from the single-table queries exhibits pronounced locality across both index families. Using six representative online tables (three HNSW and three IVF), we gather statistics—for each query trace within the short window—the cumulative distribution of accesses over (a) nodes visited in the HNSW graph and (b) clusters scanned in the IVF index. As shown in Figure 6a and Figure 6b, the CDFs rise steeply near the origin: a small fraction of nodes/clusters (lists) accounts for a large fraction of accesses during search. This heavy-tailed, Zipf-like pattern appears consistently across all six tables (HNSW A/B/C and IVF X/Y/Z), indicating that query traffic usually repeatedly touches a concentrated hotspot set rather than spreading uniformly. In short, both graph-based node visiting in HNSW and clustering-based list scans in IVF display strong search-access locality in production workloads.

Second, we quantify how memory traffic (i.e., the total volume of bytes accessed—reads, writes) is distributed across many co-located HNSW tables on a single node (as shown in Figure 6c) and clusters within the IVF table (as shown in Figure 6d). For HNSW, we analyze the online node that hosts $\sim$60 independent tables; for IVF, we analyze one 2M-level dataset partitioned into 8192 clusters and break memory traffic down by its clusters (lists) scanned as a representative example. The per-table/per-cluster traffic is plotted on a log scale (MB) against normalized IDs. Both panels reveal extreme skew: a small subset of HNSW tables and a small subset of IVF clusters account for disproportionately large traffic, with per-item volumes varying by one to several orders of magnitude (from tens to thousands of MB). In other words, memory accesses concentrate on "hot" tables/lists rather than spreading evenly. Additionally, the memory traffic of each

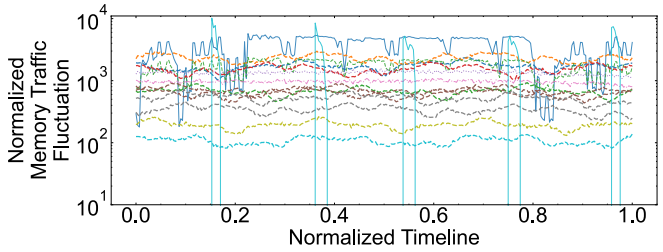


Fig. 7: ***Dynamic fluctuation of memory traffic along the time window.*** These are sampled requests from 15 vector tables.

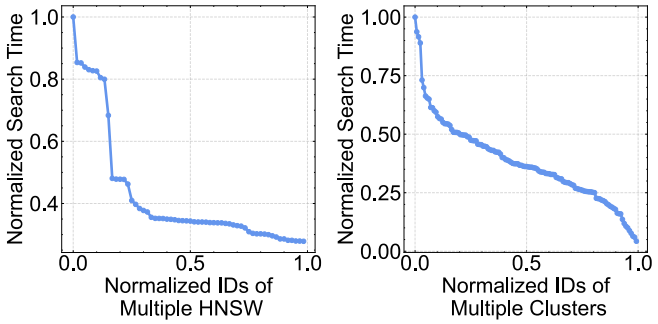
table is dynamically changing as the requests' fluctuation. As shown in Figure 7, we present the fluctuations of tables' normalized memory traffic with a fixed sampling window (e.g., minute-level segment sampled here). Skew memory traffic of each HNSW table or cluster of IVF would change with the dynamic throughput, which means the "hot" and "cold" roles would frequently transmit as the sliding of time windows.

Third, we measure the per-item processing time on the same traces in a single-core run to minimize confounding factors and obtain the actual average overhead of a single search. As shown in Figure 8, these search/scan items are sorted by their mean latency. Figure 8a details the per-table search time for the 60 co-located HNSW tables, and Figure 8b shows the per-cluster scan time within the single IVF table. Both curves display a steep head and a long tail: a small subset of tables/lists incurs disproportionately high latency, while many others are comparatively light. The spread spans multiples of the median, indicating pronounced variance across items rather than uniform cost. Such skew would still directly translate into multi-core load imbalance, though we submit search tasks uniformly under naïve task dispatches (e.g., round-robin submission for multiple HNSW tables or static list chunks for scans of IVF index).

D. Motivation and Challenges

Motivation. *Effective use of the L3 cache can provide an opportunity to alleviate memory pressure for both inter-query scaling of multi-table HNSW and intra-query scaling of single-table IVF.* On CCD-based multi-core CPUs, the L3 cache has expanded dramatically alongside core counts. The two generations of AMD CCD-architecture CPUs deployed in our production environment both provide 4 MB of L3 cache per core—up to 32/16 MB per CCD. Meanwhile, a small subset of HNSW nodes and IVF clusters accounts for most query touches. In practice, consecutive queries within short windows tend to revisit the same nodes/lists, creating a recurrent "hot set." If this hot set is kept resident in the per-CCD local L3 cache, node-chasing in HNSW and dense scans in IVF can serve from L3 rather than DRAM, cutting latency and bandwidth demand.

Challenges. In multi-chiplet CPUs, turning locality into performance improvement in industrial environments requires (i) *compatibility*—a drop-in orchestration framework, (ii) *CCD-friendly task dispatching with dynamic adaptation*, and (iii) *load balance with awareness of hardware topology*.



(a) ***Search time of multiple HNSW tables on a node.*** (b) ***Search time of multiple clusters on the IVF table.***

Fig. 8: ***Distribution of search overhead.***

Compatibility and ease of adoption. It needs a minimally invasive solution. We require a unified framework in thread orchestration that exposes uniform task-submission and can keep the hardware details largely transparent while supporting inter-query HNSW and intra-query IVF without index rewrites.

CCD-friendly task dispatching with dynamic adaptation. The L3 cache (i.e., LLC) is independent across CCDs, so naïve spreading of queries can incur cache pollution from imbalanced memory traffic. We need to map tasks to CCDs suitably, balancing dynamic hot-cold memory traffic on multiple CCDs.

Load balance with awareness of hardware topology. Since the search overhead across HNSW tables and cluster lists in IVF is skew. The framework must react online—balancing cores for jobs and using locality-aware stealing—to meet both cache reuse and high utilization.

IV. Design Overview

Our goal is to turn multi-core CPUs with per-CCD cache locality into end-to-end performance improvements while keeping cores effectively utilized and source codes of indexes unchanged. As shown in Figure 9, we introduce a drop-in orchestration framework between the vector indexes (inter-query HNSW and intra-query IVF) and the CCD-based processor. The framework comprises three cooperating components: a highly compatible *CCD-Level Task Submission*, a *Task Dispatcher & Workload Monitor* that performs *Mapping & Dynamic Adapting*, and a *Thread Orchestration* runtime with *CCD-Affinity Task Stealing*. The interface can replace a conventional thread orchestration framework used in our online services previously with uniform task submission and minimal alterations.

CCD-level task submission. Based on this abstraction of our framework, codes of various indexes can emit different search task implementations to our framework via a uniform API: `submit(search_functor, query, Mapping_ID)`. Here, *search_functor* is a pluggable callable that can directly hook into (i) an inter-query HNSW table search or (ii) an IVF cluster-list flat scan, and returns a partial top-k. The *query* object encapsulates request metadata such as the desired k value of the nearest neighbors' count, the raw query vector, optional filters from the original index implementations, and

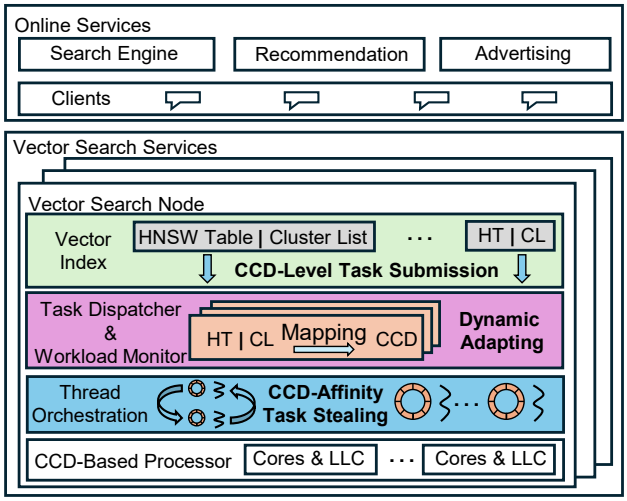


Fig. 9: ***Overview of our framework.***

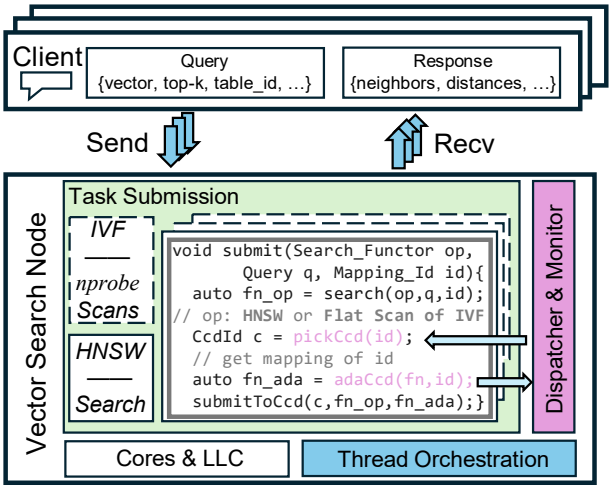


Fig. 10: ***Workflow of CCD-level task submission.***

the source client information. We unify identifiers by abstracting both inter-query HNSW tables and clusters of intra-query IVF tables with *Mapping_ID*. The dispatcher uses this *Mapping_ID* to package search tasks and routes tasks to CCDs while keeping the index code unchanged and the submission interface minimal.

Task dispatcher and workload monitor for adapting mapping. This module provides routing of each submitted task to cores as the CCD level by maintaining a mapping from *Mapping_ID* to task queues. The dispatcher builds this mapping to meet two goals: (i) keep a HNSW table or an IVF cluster's working set stably resident in one CCD's LLC, and (ii) avoid concentrating multiple high-traffic keys on the same CCD, which would over-concentrate memory traffic and pollute the cache (frequently evicting the LLC footprint of HNSW tables or cluster lists). The Workload Monitor continuously measures per–table/cluster request frequency with bytes touched, and then adapts the mapping online: it remaps and spreads coincident hot keys across different CCDs adaptively, and mixes hot and cold keys evenly within a CCD.

CCD-aware task stealing in thread orchestration. Workers are pinned one-per-core and pull from local task deques. To address the load imbalance across search, we develop a CCD-affinity task stealing strategy with hierarchy. It prefers same-CCD victims to retain LLC hits and only enables cross-CCD steals under whole-CCD idleness and sustained imbalance. The runtime supports both execution styles—independent HNSW queries and parallel IVF list scans with private per-thread top-k merged on completion—thus providing compatibility across workloads while meeting locality and utilization goals. Moreover, the processors' hardware topology is maintained inside this orchestration layer (e.g., the exact number of CCDs, the *core count per CCD*, and the *core-CCD mapping*).

V. CCD-Level Task Submission

A. Uniform Workflow of CCD-level Task Submission

To minimize changes from the framework migration and ensure compatibility across different index types and parallelization modes (inter-query and intra-query), we expose a single submission interface that both index types (HNSW and IVF) reuse: `submit(. . .)`. Here, `search_functor` is a pluggable callable that binds to the concrete search logic (either an inter-query HNSW traversal or an IVF list scan of intra-query IVF) but remains opaque to the runtime. The `query` object encapsulates all request metadata, including raw vector, desired top-k, optional filters, and client/session information. Then, the framework can package it and trace the task end-to-end without introducing index-specific implementation details.

We unify identifiers by treating an HNSW table and a cluster list of an IVF table uniformly as `Mapping_ID` (`table_id` of HNSW and `table-cluster_id` of IVF). CCD placement is decoupled from the detailed implementation of different search processes: the *Task Dispatcher* resolves the mapping between `Mapping_ID` and the CCD where tasks are submitted to (i.e., `pickCcd(id)`, as shown in Figure 10), and enqueues the task to the selected cores of CCD's run queues. Before submission, this layer also registers the `adaCcd(fn_op,id)` to emit a record to the *Dispatcher & Monitor*, enabling online workload adaptation and periodic refinement of the mapping between CCDs and `Mapping_ID`. This closes the feedback loop without changing the submission surface of our previous services or the search implementations of vector indexes.

B. Inter-query and Intra-query Integration

Integration with inter-query search of HNSW. Each HNSW request is packaged as a *single* task that carries the `query` with the client information. The dispatcher assigns this task to one CCD according to its `Mapping_ID` (i.e., its table ID). The bound `search_functor` executes the table search on one core of that CCD and gets the final top-k neighbors for the request; the result is returned directly to the client. Inter-query parallelism arises from many such tasks across multiple tables concurrently scheduled on different cores.

Integration with intra-query search of IVF. For each IVF table, the request first identifies the $nprobe$ nearest clusters.

The framework then *decomposes* the query into multiple per-list *scan tasks*, each sharing the same `query` but carrying a different `table-cluster_id` as the `Mapping_ID`. These scan tasks are dispatched to multiple cores across cores and CCDs. Each scan task returns the local optimum top-k neighbors from its cluster. Then, after all the local optimum top-k are returned, it performs a k-way merge, and returns the aggregated top-k to the client. This per-list scan of IVF shares the same `submit(. . .)` abstraction and delegates CCD selection, monitoring, and adaptation to the dispatcher, with the per-table HNSW search, yielding a uniform workflow for inter-query HNSW and intra-query IVF.

VI. Task Dispatcher and Workload Monitor

In this section, we detail the task dispatcher and workload monitor: how to map HNSW tables and clusters of IVF tables to CCDs, and modify the dynamic adaptation that updates the mapping according to the real-time memory traffic.

A. CCD Mapping of Task Dispatcher

Motivation from traffic skew. On a single online node, when multiple HNSW tables are co-located or when all IVF tables are decomposed into clusters, the memory traffic (total bytes accessed by reads/writes during task execution) is highly skewed, as shown earlier in Figure 6c and Figure 6d. Some hot HNSW tables and hot clusters in IVF account for disproportionately large volumes, while other tables (or clusters) are relatively cold with fewer requests and lighter memory traffic. This skew implies that cache residency—and therefore end-to-end performance—is dominated by where these tables' and clusters' tasks are dispatched.

Per-CCD local cache affinity. On CCD-based multi-core CPUs, each CCD owns a private local L3 cache shared only by the cores within that CCD. Consequently, cache affinity is created at the CCD granularity: tasks issued for the same hot table/cluster benefit when they are dispatched to the same CCD, whose L3 cache can retain the corresponding frequently-accessing memory zone.

Hot–hot vs. hot–cold and the implication for CCD mapping. Given the strong memory traffic skew, the mapping decision made at task submission (i.e., `pickCcd(T|C_Id id)`) should be load-aware to be cache friendly. If the mapping places two *hot* tables/clusters on the same CCD, then all subsequent search tasks carrying those IDs are routed to that CCD; their bursts interleave in a single per-CCD L3 cache, causing reciprocal evictions and cache pollution and, in turn, lower hit rates and worse performance (i.e., *hot–hot co-location*, as shown in the left of Figure 11). Conversely, mapping a hot item together with a *cold* one on a CCD (i.e., *hot–cold co-location*, as shown in the right of Figure 11) spreads memory traffic: the cold item contributes fewer queries in the short time window and little L3 cache demand, allowing the hot item's working set to remain resident and improving in-cache retention.

Design rule for mapping. Hence, `pickCcd(id)` consults the current load-aware mapping in our task dispatcher with two priorities: (i) maintain *stickiness* so repeated submissions for the same T|C_id (i.e., `Mapping_ID`) return to the same CCD to exploit memory locality; (ii) *avoid* hot–hot placements on the same CCD and *prefer* hot–cold pairings to decrease cache pollution from frequent eviction.

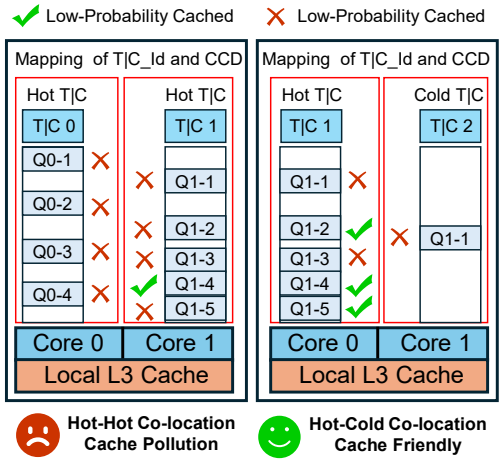


Fig. 11: ***Cache affinity of hot-hot and hot-cold dispatch.*** T|C 0 and 1 are two hot HNSW tables (or clusters in IVF) with more queries and heavier memory traffic. T|C 2 is a cold HNSW table (or cluster in IVF) with fewer queries and lighter memory traffic. Qx-y is the yth query on T|C x sequentially.

B. Dynamic Adaptation of Workload Monitor

However, neither HNSW nor IVF exposes a primitive that directly reflects the memory traffic during searches. Moreover, in an online service, this quantity fluctuates with dynamic requests and temporal locality. To address these issues, we first define an online, low-overhead estimation of per-query memory traffic. Second, we use these estimates to generate balanced hot-cold co-location to be cache friendly. Third, we develop a snapshot swap for stability, to adapt the mapping between CCDs and tables (or clusters) over time. The submission path registers `adaCcd(fn, id)` into search; upon completion, it notifies the workload monitor with measured counters (e.g., touched nodes / scanned vectors), so the monitor can update per-table (or cluster) statistics and evolve the CCD mapping.

Approximate estimation of memory traffic. Let D be dimensionality, s_v bytes per element (e.g., 4 for FP32), and $B_v = D \cdot s_v$ the vector payload size. Let s_{id} be the ID width (e.g., usually 4 bytes for UINT32). For HNSW, a query with search width ef_{search} touches at most N nodes (the runtime returns the exact count); each touch reads the vector and its adjacency. We approximate

$$T_{\mathrm{HNSW}} \approx N \cdot \left(B_v + M \cdot s_{\mathrm{id}}\right) + \delta_{\mathrm{meta}}, \qquad (1)$$

where M is the per-node out-degree. And in practice, δ_{meta} covers small write/reads for maintaining top-k heap and visited list (typically $< 1\%$) in the search process of HNSW, so it usually can be ignored.

For IVF, after routing to $nprobe$ cluster lists for intra-query dispatching, we attribute traffic *per probed list* rather than at

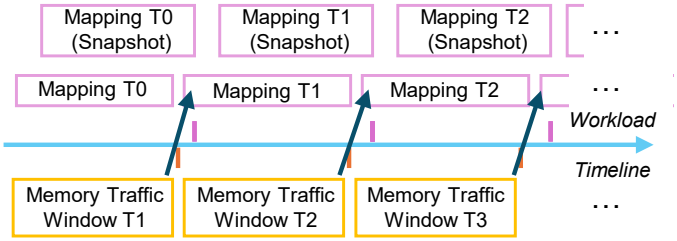


Fig. 12: ***Mapping adaptation with snapshot.***

the whole-query level. For list $\mathcal{L}_i$ with $S_i = |\mathcal{L}_i|$ scanned vectors, the traffic is

$$T_{\text{IVF}}(\mathcal{L}_i) \;\approx\; S_i \cdot B_v \;. \tag{2}$$

Balancing CCD–item mapping with hot-cold co-location. Given items (i.e., HNSW tables or clusters' lists in IVF) memory-traffic estimates $\hat{T}_1, \ldots, \hat{T}_n$ and m CCDs, map items to CCDs so that (i) hot items are paired with cold items on the same CCD to avoid hot–hot contention, and (ii) total traffic per CCD is balanced. Let $\mu = \frac{\sum_j \hat{T}_j}{m}$ be the target per-CCD load. After sorting items (tables or clusters) by traffic, we always place on the currently least-loaded CCD, and greedily pair the largest remaining item with the smallest remaining item so the placement approaches μ while encouraging hot–cold co-location.

Algorithm 1 Balanced Hot–Cold Pairing for Mapping

Require: traffic array $\hat{T}[1..n]$, number of CCDs m

1: $\mu \leftarrow \frac{\sum_j \hat{T}[j]}{m}$ ▷ target per-CCD load
2: sort items by $\hat{T}$ (desc) into $A[1..n]$ ▷ ordering by heat
3: initialize memory traffic of workloads $L[1...m] \leftarrow 0$; $i \leftarrow 1$; $j \leftarrow n$ ▷ two-ended sweep for hot and cold memory traffic
4: **while** $i \le j$ **do**
5: $r^\star \leftarrow \textsc{Get}\min_r L[r]$ ▷ get ssleast-loaded CCD
6: $t \leftarrow A[i]$; $i \leftarrow i+1$
7: cap $\leftarrow \max\{0,\ \mu - L[r^\star] - \hat{T}[t]\}$ ▷ residual to target
8: **if** $i \le j$ **and** $\hat{T}[A[j]] \le$ cap **then**
9: $p \leftarrow A[j]$; $j \leftarrow j-1$
10: place $\{t, p\}$ on $r^\star$; $L[r^\star] \leftarrow L[r^\star] + \hat{T}[t] + \hat{T}[p]$
11: **else**
12: place t on $r^\star$; $L[r^\star] \leftarrow L[r^\star] + \hat{T}[t]$
13: **end if**
14: **end while**
15: **return** Mapping {items to CCD}

Details are shown in Algorithm 1. Lines *1–3* compute the target per-CCD load μ and order items by estimated traffic. Lines *4–14* perform a two-ended sweep: always place on the least-loaded CCD (line *5*), take the hottest remaining item (line *6*), and steer each placement toward μ via the residual capacity (line *7*). If the coldest item fits, it is paired with the hot item (lines *8–10*) to promote hot–cold co-location; otherwise, the hot item is placed alone (lines *11–13*). Lines *14–15* finish the loop and return the mapping. This greedy pairing, combined with least-load placement, balances traffic across CCDs while reducing hot–hot co-location.

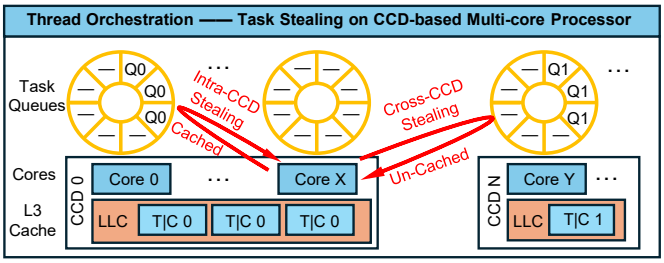


Fig. 13: ***Cache affinity of intra-CCD and cross-CCD task stealing.*** T|C 0 and 1 in LLC mean that memory zones related to HNSW table 0 and 1 (or clusters in IVF) are cached in the L3 cache of a CCD chiplet. Q0 and Q1 represent the query on table (or cluster) 1 and 0. If Core X steals Q1 from Core Y, the memory accessing would be in the un-cached state. While if it steals Q0 from Core 0, the data can be cached.

Windowed re-mapping with snapshot swap. As shown in Figure 12, every adaptation interval (e.g., we typically use minute-level intervals in e-commerce services, and we adjust the interval based on how frequently the traffic has changed in the past for the given business.), the workload monitor aggregates events to get memory traffic of all HNSW or IVF lists in the sliding window. As the time window goes by, the workload monitor builds a *next-map* in the background while the snapshot of *current-map*, used by the task dispatcher, serves traffic. Once the new mapping is ready, it publishes an new epoched snapshot for the task dispatcher: new submissions use the new mapping immediately, while in-flight tasks finish on the current-map. When all tasks of the old epoch retire, the next-map atomically replaces the current snapshot. This snapshot-based handover yields stable latency during reconfiguration while continuously adapting to shifting hot-cold workloads.

VII. Thread Orchestration for CCD-Based Cores

A. Motivation and Consideration

The heavy-tailed distribution of per-item search costs renders naive round-robin scheduling ineffective, causing significant multi-core load imbalance. We address this via a classic decentralized scheduling approach: each worker, with its threads pinned to cores, maintains a private ring-based task queue. Dynamic load balancing is then achieved through work-stealing, where idle workers procure tasks from their busy peers.

However, on CCD-based processors, the L3 cache is private per CCD, which makes stealing locality a matter. As illustrated in Figure 13, *intra-CCD* stealing continues execution under the same LLC that already holds the hot working set (e.g., frequently revisited HNSW nodes or IVF lists), preserving cache residency. *Cross-CCD* stealing migrates the task onto

a different LLC; the working set is typically cold in the uncached state there and must be re-warmed, increasing memory traffic and lowering hit rate. Hence, we need to balance skewed work while prioritizing cache affinity.

B. Topology-aware Stealing

We prefer intra-CCD task stealing and conduct cross-CCD stealing only when necessary. The thread orchestration module materializes the CPU topology once at initialization. For each core i, we record two disjoint neighbor sets: $\mathcal{S}_{\text{in}}(i)$ (cores on the same CCD) and $\mathcal{S}_{\text{cross}}(i)$ (cores on other CCDs). A thread is pinned to each core and follows a simple hierarchy at runtime: (1) pop locally; (2) steal within $\mathcal{S}_{\text{in}}(i)$; (3) only then downgrade to attempt $\mathcal{S}_{\text{cross}}(i)$. The workloop of cores is shown in Algorithm 2.

Algorithm 2 CCD-Topology Friendly Task Stealing Workloop

Require: local deque Q_i, local CPU core i, intra-CCD and cross-CCD core sets of local CPU core $\mathcal{S}_{\text{in}}(i)$, $\mathcal{S}_{\text{cross}}(i)$

```
while true do
    t ← PopLocal(Q_i)
    if t ≠ null then
        Execute(t)                          ▷ own-queue first to execute
        continue
    end if
    t ← TrySteal(S_in(i))
    if t ≠ null then
        Execute(t)
                ▷ intra-CCD steal preserves cache residency
        continue
    end if
    t ← TrySteal(S_cross(i))
    if t ≠ null then
        Execute(t)                          ▷ cross-CCD as the last choice
        continue
    end if                                  ▷ Ready to receive the next task
end while
```

VIII. Evaluation

In this section, we first describe the experiment setup, including environmental configurations, workloads from real-world production, and baselines we compared. Then, we present the detailed evaluations to answer the following questions:

- How does the current framework we proposed affect end-to-end saturated performance on CCD-based modern CPUs, with respect to throughput and latency, compared to previous versions?
- To what extent are the end-to-end benefits attributable to improvements in hitting rates of L3 cache, reductions in CPU stall time, and enhanced core utilization?
- How does the proposed framework perform under real-world industrial online workloads compared with the prior thread orchestration mode?

AMD 4th Gen EPYC 96-core CPU		
CPU	Core	96
	CCD	12
	Memory	576GB DDR5
CCD	Core	8
	Total L3	32MB
Core	L1 Cache	32KB
	L2 Cache	1MB
	L3 Cache	4MB
	Frequency	3.5GHz

AMD 2nd Gen EPYC 48-core CPU		
CPU	Core	48
	CCD	12
	Memory	512GB DDR4
CCD	Core	4
	Total L3	16MB
Core	L1 Cache	32KB
	L2 Cache	0.5MB
	L3 Cache	4MB
	Frequency	2.6GHz

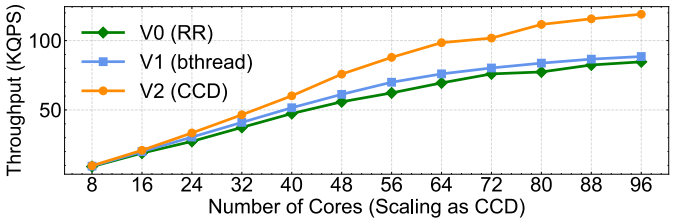
TABLE I: ***Hardware configuration of our platforms.***

A. Experiment Setup

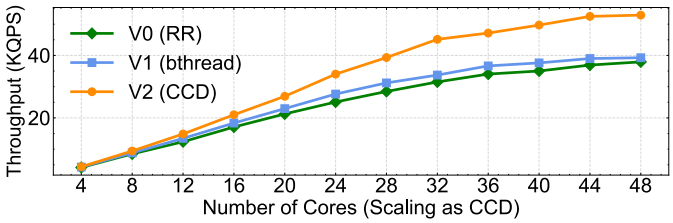
Hardware and software configuration. We use two configurations for evaluation, each equipped with an AMD multi-core CPU based on the CCD architecture (from two different generations, including AMD 4th Gen Genoa 96-core 9654 processor and AMD 2nd Gen Rome 48-core 7K62 processor). The detailed information is shown in Table I. The operating systems are CentOS 8 with a Linux kernel 5.10. All implementations are written in C++ and are compiled with GCC 11.4 using the -O3 optimization flag. And our HNSW and IVF indexes are implemented with reference to hnswlib [8] and FAISS [10]. The distance evaluation of both HNSW and IVF is accelerated by AVX instructions.

Workloads. All datasets and queries are collected from our production recommendation and advertising services. For inter-query HNSW deployment, we adopt 60 HNSW tables from a serving node, and each vector table contains 1M to 10M rows. For intra-query IVF deployment, we adopt 15 tables from a serving node, and each vector table contains 10K to 15M rows. The vector dimensionality of both the datasets and the queries ranges from 64 to 256. And the k values of top-k are 100 to 500, and the recall rates reach above 90% during search, which are commonly used in our services. Similarity is measured using the L2 distance. For HNSW, we set *M* = 32 (i.e., number of neighbors per node) and *efConstruction* = 500 (i.e., size of the candidate list during construction). For IVF, we set *nlist* (i.e., number of clusters) = 128 to 8192 according to the total rows of the table, for ensuring the building time within a minute-level to meet the freshness requirements in our production services. Each *efSearch* of HNSW tables is set to keep requests' recall rates reaching 99%. And each *nprobe* of IVF tables is set to keep requests' recall rates reaching 95%. The set of queries is collected from logs of services, which contains about 1 million requests of HNSW tables and about 1.1 million requests of IVF tables.

Baselines. We refer to the naïve round-robin orchestration framework as **V0 (RR)**. We denote as **V1 (bthread)** the thread scheduling framework that does not account for the CCD topology and augments scheduling with work stealing (following the bthread [34] stealing implementation), which has been widely deployed in our production workloads. We refer to the optimal, CCD-level and load-aware thread orchestration framework presented in this paper as **V2 (CCD)**. For HNSW, we compare the performance differences among V0, V1, and V2. For IVF, we use the intra-query parallel implementation

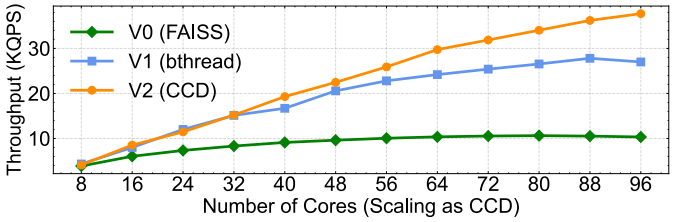


(a) ***Saturated throughput of HNSW on the 4th Gen 96-core CPU.***

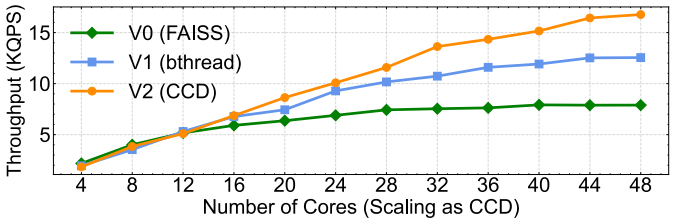


(b) ***Saturated throughput of HNSW on the 2nd Gen 48-core CPU.***

Fig. 14: ***Saturated throughput of HNSW as scaling CCDs.***



(a) ***Saturated throughput of IVF on the 4th Gen 96-core CPU.***



(b) ***Saturated throughput of IVF on the 2nd Gen 48-core CPU.***

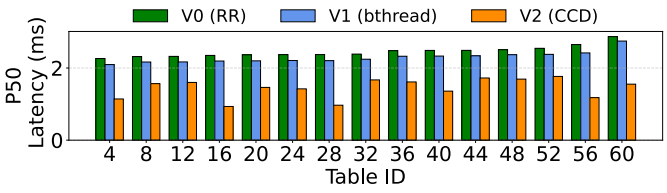
Fig. 15: ***Saturated throughput of IVF as scaling CCDs.***

of FAISS (i.e., the default OpenMP-based implementation) as comparison **V0 (FAISS)**, which submits all scans of lists within each query on all CPU cores for intra-query parallelism.

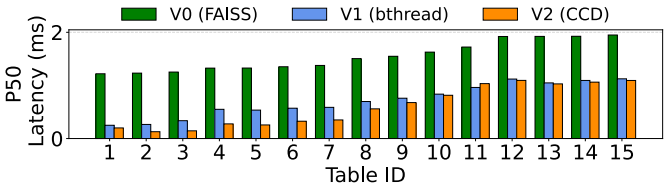
B. Overall Performance

In this subsection, we use real production workloads of both HNSW and IVF to conduct offline stress testing and evaluate performance under saturated load. We report aggregate throughput at saturation and tabulate the median (P50) latency and the tail (P999) latency, which together characterize the service's upper bound of admissible request load and quality of service (QoS).

First, we report saturated throughput on two generations of CCD-based multi-core CPUs, including the 4th Gen 96-core processor and the 2nd Gen 48-core processor. The core count

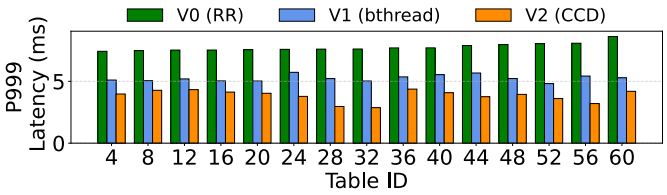


(a) ***Comparison of P50 search latency from HNSW tables.***

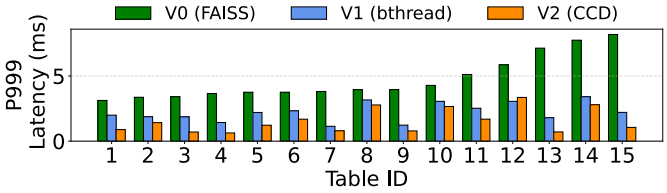


(b) ***Comparison of P50 search latency from IVF tables.***

Fig. 16: ***Comparisons of P50 search latency.***



(a) ***Comparison of P999 search latency from HNSW tables.***

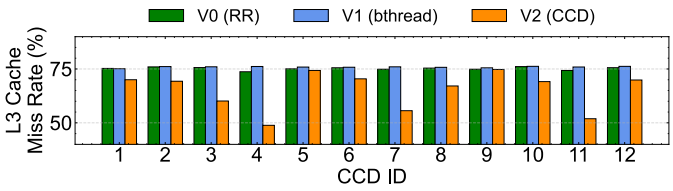


(b) ***Comparison of P999 search latency from IVF tables.***

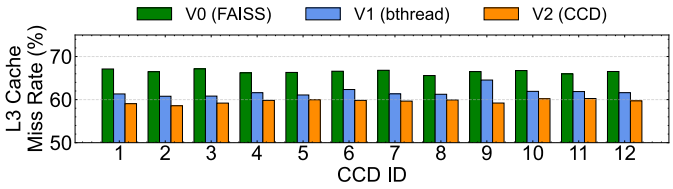
Fig. 17: ***Comparisons of P999 search latency.***

is scaled one CCD at a time (i.e., at each step, we enable all cores within a CCD). As shown in Figure 14 and Figure 15, the CCD-aware design we proposed (V2) consistently delivers the highest throughput and scales better than the bthread-based V1 and the baseline V0. For HNSW (Fig. 14), throughput rises nearly linearly as CCDs are added, reaching over 100 KQPS at 96 cores and ∼50 KQPS at 48 cores, with V2 maintaining a clear margin over V1/V0 across the range. For IVF (Figure 15), V2 likewise shows superior scaling, attaining 35 KQPS at 96 cores versus 25 KQPS for V1 and 10 KQPS for V0. These results indicate that our CCD-level and load-aware orchestration framework improves the maximum throughput and the scalability of performance on the CCD-based architecture.

Second, we report the median P50 search latency and the tail P999 search latency, which are evaluated with 96 cores of the 4th Gen EPYC processor. Results of median latency are shown in Figure 16. And results of tail latency are shown in Figure 17. For the HNSW case, we display 15 out of the 60



(a) ***Cache miss rates of HNSW tables.***



(b) ***Cache miss rates of IVF tables.***

Fig. 18: ***Comparisons of L3 cache miss rates.***

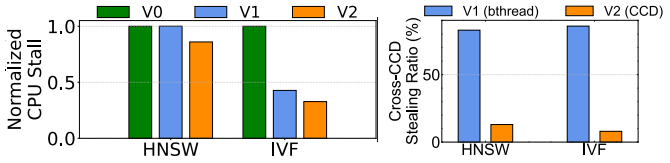
tables (ranked by performance of V0, selecting one table out of every four). Due to limited space here, although we only show some of HNSW tables, the overall trend across all 60 tables is the same. For the IVF case, we display all 15 tables.

Figure 16 and Figure 17 show that the CCD-level thread orchestration framework, V2 (CCD), achieves the lowest latency across both median (P50) and tail (P999) latency for inter-query HNSW search and intra-query IVF search in our services' workloads. Relative to the baseline V0 (FAISS), V1 (bthread) introduces proactive task stealing, which balances work more evenly across workers and reduces queueing hot spots. This improves utilization and lowers both median latency and tail latency, compared with V0 (i.e., round-robin dispatching in HNSW and OpenMP in FAISS-IVF). In addition to naive task stealing, our CCD-level and load-aware V2 framework further reduces latency by improving cache efficiency (e.g., cache-friendly request dispatching and reduced cross-chiplet task stealing), so that more query processing happens in the CPU caches rather than in slower memory. As a result, V2 we proposed in this paper consistently outperforms V1 and V0 on both HNSW and IVF setups in terms of P50 and P999 search latency.

C. Detailed Study

In this subsection, we present the detailed report from three aspects, including overall hit/miss rates of L3 cache (Figure 18), CPU stall, and the behavior of task stealing (Figure 19).

First, we study the L3 cache behavior as shown in Figure 18. Overall, on both HNSW (Figure 18a) and IVF (Figure 18b), our V2 framework markedly reduces the fraction of L3–cache misses. The results are collected with *AMD uProf* on the 4th Gen 96-core, 12-CCD processor while running end-to-end search workloads. For HNSW, V2 delivers consistently lower miss rates than both V1 (bthread) and the V0 baseline (round-robin) across all 12 CCDs. V1 shows similar cache miss rates without optimizations addressing the CCD-based architecture. For IVF, V1 already slightly improves upon the OpenMP-based FAISS baseline, while V2, benefiting from hot–cold colocation in the mapping policy and a CCD-preferential work-stealing strategy, achieves the lowest L3 cache miss rate.



(a) ***Decrease of CPU stall during search.*** (b) ***Decrease of cross-CCD task stealing during search.***

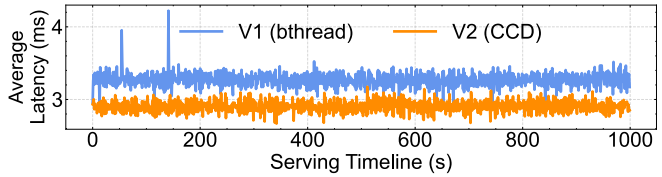
Fig. 19: ***Comparisons of CPU stall and cross-CCD stealing of both HNSW and IVF.***

Second, we report the CPU stall. As shown in Figure 19a, we also record CPU stall under HNSW- and IVF-based search loads. Here, *CPU stall* denotes cycles in which the core cannot retire instructions—typically because it is waiting on long-latency data and instruction fetches, (e.g., cache misses spilling to memory), or CPU contention (e.g., thread switches), thereby reducing effective utilization. The lower stall implies better CPU use, whereas higher stall leads to latency spikes and throughput losses. On HNSW, V0 and V1 show comparable stall, while V2 achieves a noticeably lower stall thanks to fewer long-latency memory accesses arising from improved cache affinity. On IVF, V1 benefits from task-stealing–based scheduling and already reduces stall relative to the OpenMP baseline, and V2 further increases cache hit likelihood on top of that, yielding the lowest stall overall.

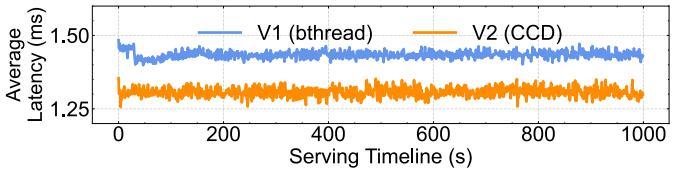
Third, we record the behavior of task stealing, when each CPU core's own task queue is empty. Results are shown in Figure 19b. While work-stealing is crucial for load balancing, unrestricted stealing can lead to significant performance degradation due to local L3 cache misses. Specifically, when a core in one CCD steals a task from another CCD, the subsequent execution of that task is likely to access data previously residing in the remote CCD's L3 cache and waste the cache space. We propose to prioritize stealing tasks from cores within the same CCD when a core becomes idle. This prioritization is achieved by leveraging CPU hardware topology information to define the stealing targets' vicinity. For HNSW, the cross-CCD stealing ratio is reduced from $\sim 75\%$ in V1 to below 10% in V2. For IVF, the ratio is similarly suppressed from above 80% in V1 without CCD-aware to a mere $\sim 5\%$ in current V2.

D. Serving Performance

In the production-style pressure-limited setting, we compare two serving frameworks (V1 and V2) over a continuous 1000s run. For every second, we aggregate all requests that arrived in that second and record the average response latency. These per-second averages form the time series in Figure 20, with HNSW and IVF backends shown in Figure 20a and Figure 20b. Across both index types, V2 (CCD) delivers consistently lower average latency and tighter dispersion than V1 (bthread). On HNSW, V1 exhibits intermittent spikes,



(a) ***Average response time of requests on HNSW tables.***



(b) ***Average response time of requests on IVF tables.***

Fig. 20: ***Comparison of average response time as a timeline.***

whereas V2 remains flat and stable, suggesting better Qos. On IVF, both traces are steady, yet V2 persistently tracks below V1. Overall, under realistic admission control, CCD-level and load-aware V2 provides more stable service and lower average response time. In addition, we set the time window as 10 s for the dynamic remapping of tasks here. It can be seen that the remapping strategy is a low-overhead adjustment and harmless for the services.

IX. Related Works

Research development in vector search. Early systems relied on locality-sensitive hashing (LSH) and tree-based indexes (e.g., KD-tree) [35], [36], whose search efficiency degrades in high dimensions. Subsequently, partitioned (clustering) indexes (e.g., IVF and balanced k-means trees, BKT) [10], [37] and graph-based indexes (e.g., NSG, HNSW, and Vamana) deliver superior recall–latency trade-offs and become mainstreams for deployments [8], [9], [38]. Meanwhile, quantization [26] advances from PQ and SQ toward more effective schemes such as the recent RaBitQ [25], which shrink index footprints and enable shorter-code distance estimation. We are introducing these advanced quantization methods into services. We argue that, with memory touches cut from quantization and the scaling of L3 cache from CCD increased, the effectiveness of our framework and the future finer-grained task–thread orchestration would amplify per-CCD cache benefits.

Industrial development in vector search. Recent industrial practice converges on DB-centric, cloud-native designs, purpose-built or vector-augmented DBMS (e.g., Milvus, Pinecone, SingleStore-V) [16], [39]–[41]. Many existing challenges and open problems are well discussed and summarized in [42], [43]. Then, production systems in large vendors, such as BlendHouse, GaussDB-Vector, and Azure Cosmos DB, present how to integrate vector search into their well-developed databases [44]–[46]. VSAG describes their framework's optimizations about graph-based ANNS comprehensively [22]. After SPANN, DiskANN, and Starling are proposed [47]–[49], [50] pgvector and Cosmos DB show the integration of persistent vector search and relational and NoSQL databases [45]. TigerVector integrates vector search and graph query within the native graph database, TigerGraph [51]. From another special aspect that is different from these remarkable works, we characterize the in-memory vector search of industrial services at RedNote, and propose our thread orchestration to optimize it according to hardware features of CCD-based processors.

Research about optimization on CCD-based multi-core CPUs. Modern multi-core CPUs partition cores and L3 cache across dies (e.g., CCD on AMD EPYC), introducing non-sharing L3 cache even within a socket. AMD's industry retrospectives detail the motivations and design choices behind chipletization (Infinity Fabric topology, CCD organization) and how advanced 2D/3D modular packaging co-evolves with software stacks [30], [52]. Recent system studies reveal that hardware-oblivious software design can severely underutilize increased cores and cache capacity. And [53] demonstrates that chiplet-aware placement of multiple workers (instantiations) can speed up OLAP workload up to 7 $\times$ by deploying query engines with CCD-local granularity. In the inference of recommendation, CCD-based CPUs are widely used as well. [12] and [54] study how to break bottlenecks in the deep learning model's embedding stage on CPUs by improving data reuse in caches of CCD-based architectures. These works motivate our co-design to study the real-world workloads of vector search, and develop this thread framework for services of vector search with better performance than the previous thread orchestration framework on CCD-based multi-core CPUs.

X. Conclusion

Driven by the requirements of large-scale online ANN services with tight latency SLA at RedNote, we analyzed why merely adding cores on modern CCD-based CPUs fails to translate hardware upgrading into throughput: short-window skew, chiplet-isolated L3 cache, and per-table/cluster imbalance jointly degrade cache residency and utilization. We presented a CCD-level and load-aware orchestration framework that unifies inter-query (HNSW) and intra-query (IVF) parallelism behind a drop-in submission interface in our services. It couples cache-friendly task-to-CCD mapping with online adaptation, and employs topology work stealing to balance load without sacrificing locality. Evaluated by real-world workloads and deployed in production, our approach delivers higher saturation throughput, reduces P50 and P999 latency to achieve better QoS stability. These results demonstrate that our co-design thread orchestration with CPU topology of the CCD architecture is an effective and practical lever for accelerating in-memory vector search in industrial services.

Acknowledgment

We sincerely thank anonymous reviewers for their valuable feedback. This work was conducted by Yuchen Huang and Baiteng Ma during their internship at RedNote Engine Architecture Department. Yuchen Huang and Chuliang Weng are supported by the National Natural Science Foundation of China (Grant No.62272171). Chuliang Weng is the corresponding author.

AI-Generated Content Acknowledgement

We gratefully acknowledge the assistance of AI models (e.g., Google's Gemini model) in this work. AI tools supported our coding process by suggesting implementations and refactoring routines, which improved reliability and readability. We also used AI tools to proofread and correct grammar across the entire manuscript, helping us ensure clearer, more consistent expression without altering the scientific content. We sincerely thank their developing teams for enabling these capabilities. All the assistance of generative AI tools is under the authors' supervision and final review.

References

[1] R. Castro Fernandez, E. Mansour, A. A. Qahtan, A. Elmagarmid, I. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang, "Seeping Semantics: Linking Datasets Using Word Embeddings for Data Discovery," *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 989–1000, 2018.

[2] G. Chen, R. Sun, Y. Jiang, T. Li, Y. Dai, Q. Shi, X. Qin, J. Fu, P. Chen, R. Huang, N. Li, Q. Zhang, J. Liang, H. Li, and K. Gai, "A Cold-start Recommendation System at Kuaishou Designed from the Short-video Perspective," *Companion Proceedings of the ACM on Web Conference 2025*, p. 124–132, 2025.

[3] Y. Zhang, S. Liu, and J. Wang, "Are There Fundamental Limitations in Supporting Vector Data Management in Relational Databases? A Case Study of PostgreSQL," *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 3640–3653, 2024.

[4] Xiaohongshu Inc (RedNote). https://www.xiaohongshu.com/.

[5] Number of monthly active users of Xiaohongshu app. https://www.statista.com/statistics/1327421/china-xiaohongshu-monthly-active-users/.

[6] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang, "CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs," *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 4236–4247, 2024.

[7] NVIDIA cuVS. https://developer.nvidia.com/cuvs.

[8] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020.

[9] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[10] Facebook faiss. https://github.com/facebookresearch/faiss.

[11] AMD EPYC™ 9004 Series Architecture Overview. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/white-papers/58015-epyc-9004-tg-architecture-overview.pdf .

[12] K. Nair, A.-C. Pandey, S. Karabannavar, M. Arunachalam, J. Kalamatianos, V. Agrawal, S. Gupta, A. Sirasao, E. Delaye, S. Reinhardt, R. Vivekanandham, R. Wittig, V. Kathail, P. Gopalakrishnan, S. Pareek, R. Jain, M. T. Kandemir, J.-L. Lin, G. G. Akbulut, and C. R. Das, "Parallelization Strategies for DLRM Embedding Bag Operator on AMD CPUs," *IEEE Micro*, vol. 44, no. 6, pp. 44–51, 2024.

[13] AMD Data Center Server Market Share Hits New High. Https://hothardware.com/news/amd-server-revenue-market-share-hits-new-high.

[14] AWS and AMD. Https://aws.amazon.com/cn/ec2/amd/.

[15] Aliyun. https://www.alibabacloud.com/en/product/lindorm.

[16] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, "Milvus: A Purpose-Built Vector Data Management System," *Proc. ACM Manag. Data*, p. 2614–2627, 2021.

[17] Y. Su, Y. Sun, M. Zhang, and J. Wang, "Vexless: A Serverless Vector Data Management System Using Cloud Functions," *Proc. ACM Manag. Data*, vol. 2, no. 3, May 2024.

[18] Z. Peng, M. Zhang, K. Li, R. Jin, and B. Ren, "iQAN: Fast and Accurate Vector Search with Efficient Intra-Query Parallelism on Multi-Core Architectures," *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, p. 313–328, 2023.

[19] C. Li, M. Zhang, D. G. Andersen, and Y. He, "Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination," *Proc. ACM Manag. Data*, p. 2539–2554, 2020.

[20] M. Chatzakis, Y. Papakonstantinou, and T. Palpanas, "DARTH: Declarative Recall Through Early Termination for Approximate Nearest Neighbor Search," 2025, https://arxiv.org/abs/2505.19001.

[21] Y. Huang, X. Fan, S. Yan, and C. Weng, "Neos: A NVMe-GPUs Direct Vector Service Buffer in User Space," *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 3767–3781, 2024.

[22] X. Zhong, H. Li, J. Jin, M. Yang, D. Chu, X. Wang, Z. Shen, W. Jia, G. Gu, Y. Xie, X. Lin, H. T. Shen, J. Song, and P. Cheng, "VSAG: An Optimized Search Framework for Graph-Based Approximate Nearest Neighbor Search," *Proc. VLDB Endow.*, vol. 18, no. 12, p. 5017–5030, Sep. 2025.

[23] Faiss-Metric. https://github.com/facebookresearch/faiss/wiki/MetricType-and-distances.

[24] T. Yang, W. Hu, W. Peng, Y. Li, J. Li, G. Wang, and X. Liu, "VDTuner: Automated Performance Tuning for Vector Data Management Systems," *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 4357–4369, 2024.

[25] J. Gao and C. Long, "RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search," *Proc. ACM Manag. Data*, vol. 2, no. 3, May 2024.

[26] H. Jégou, M. Douze, and C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[27] Recall/Build time (s) of HNSW. https://ann-benchmarks.com/hnswlib.html.

[28] Recall/Build time (s) of Faiss-IVF. https://ann-benchmarks.com/faiss-ivf.html.

[29] OpenMP. https://www.openmp.org/.

[30] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, "Pioneering chiplet technology and design for the AMD EPYC™ and Ryzen™ processor families," *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, p. 57–70, 2021.

[31] R. Bhargava and K. Troester, "AMD Next-Generation "Zen 4" Core and 4th Gen AMD EPYC Server CPUs," *IEEE Micro*, vol. 44, no. 3, pp. 8–17, 2024.

[32] Intel Unveils Sapphire Rapids: Next-Generation Server CPUs. https://fuse.wikichip.org/news/6119/intel-unveils-sapphire-rapids-next-generation-server-cpus/.

[33] J. L. Phillip Allen Lane, "The AMD Rome Memory Barrier," 2022, https://arxiv.org/abs/2211.11867.

[34] bthread. Https://github.com/apache/brpc/tree/master/src/bthread.

[35] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," *Proc. VLDB*, pp. 518–529, 1999.

[36] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[37] (2019) SPTAG: Space Partition Tree And Graph (BKT/KDT). https://github.com/microsoft/SPTAG.

[38] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph," *Proc. PVLDB*, vol. 12, no. 5, pp. 461–474, 2019.

[39] R. Guo, X. Luan, L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu, Z. Cao, Y. Qiao, T. Wang, B. Tang, and C. Xie, "Manu: a cloud native vector database management system," *Proc. VLDB Endow.*, vol. 15, no. 12, p. 3548–3561, Aug. 2022.

[40] (2022) Pinecone. https://www.pinecone.io/.

[41] C. Chen, C. Jin, Y. Zhang, S. Podolsky, C. Wu, S.-P. Wang, E. Hanson, Z. Sun, R. Walzer, and J. Wang, "SingleStore-V: An Integrated Vector Database System in SingleStore," *Proc. VLDB Endow.*, vol. 17, no. 12, p. 3772–3785, Aug. 2024.

[42] J. J. Pan, J. Wang, and G. Li, "Vector Database Management Techniques and Systems," *Proc. ACM Manag. Data*, p. 597–604, 2024.

[43] J. Wang, E. Hanson, G. Li, Y. Papakonstantinou, H. Simhadri, and C. Xie, "Vector Databases: What's Really New and What's Next?

(VLDB 2024 Panel)," *Proc. VLDB Endow.*, vol. 17, no. 12, p. 4505–4506, Aug. 2024.

[44] Z. Niu, X. Tian, X. Peng, and X. Chen, "BlendHouse: A Cloud-Native Vector Database System in ByteHouse," *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pp. 4332–4345, 2025.

[45] J. Sun, G. Li, J. Pan, J. Wang, Y. Xie, R. Liu, and W. Nie, "GaussDB-Vector: A Large-Scale Persistent Real-Time Vector Database for LLM Applications," *Proc. VLDB Endow.*, vol. 18, no. 12, p. 4951–4963, Sep. 2025.

[46] N. Upreti, H. V. Simhadri, H. S. Sundar, K. Sundaram, S. Boshra, B. Perumalswamy, S. Atri, M. Chisholm, R. R. Singh, G. Yang, T. Hass, N. Dudhey, S. Pattipaka, M. Hildebrand, M. Manohar, J. Moffitt, H. Xu, N. Datha, S. Gupta, R. Krishnaswamy, P. Gupta, A. Sahu, H. Varada, S. Barthwal, R. Mor, J. Codella, S. Cooper, K. Pilch, S. Moreno, A. Kataria, S. Kulkarni, N. Deshpande, A. Sagare, D. Billa, Z. Fu, and V. Vishal, "Cost-Effective, Low Latency Vector Search with Azure Cosmos DB," *Proc. VLDB Endow.*, vol. 18, no. 12, p. 5166–5183, Sep. 2025.

[47] S. J. Subramanya, Devvrit, R. Kadekodi, R. Krishnaswamy, and H. V. Simhadri, "DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node," *NeurIPS*, 2019.

[48] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search," *NeurIPS*, 2021.

[49] M. Wang, W. Xu, X. Yi, S. Wu, Z. Peng, X. Ke, Y. Gao, X. Xu, R. Guo, and C. Xie, "Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment," *Proc. ACM Manag. Data*, vol. 2, no. 1, Mar. 2024.

[50] J. Shim, J. Oh, H. Roh, J. Do, and S.-W. Lee, "Turbocharging Vector Databases Using Modern SSDs," *Proc. VLDB Endow.*, vol. 18, no. 11, pp. 4710–4722, 2025.

[51] S. Liu, Z. Zeng, L. Chen, A. Ainihaer, A. Ramasami, S. Chen, Y. Xu, M. Wu, and J. Wang, "TigerVector: Supporting Vector Search in Graph Databases for Advanced RAGs," *Proc. ACM Manag. Data*, p. 553–565, 2025.

[52] A. Smith, G. H. Loh, M. J. Schulte, M. Ignatowski, S. Naffziger, M. Mantor, M. Fowler, N. Kalyanasundharam, V. Alla, N. Malaya, J. L. Greathouse, E. Chapman, and R. Swaminathan, "Realizing the AMD Exascale Heterogeneous Processor Vision," *Proc. 51st ACM/IEEE Int'l Symp. on Computer Architecture (ISCA), Industry Track*, pp. 876–889, 2024.

[53] A. Fogli, B. Zhao, P. Pietzuch, M. Bandle, and J. Giceva, "OLAP on Modern Chiplet-Based Processors," *Proc. VLDB Endow.*, vol. 17, no. 11, p. 3428–3441, Jul. 2024.

[54] R. Jain, T. Chou, O. Kayiran, J. Kalamatianos, G. H. Loh, M. T. Kandemir, and C. R. Das, "Load and MLP-Aware Thread Orchestration for Recommendation Systems Inference on CPUs," *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 589–603, 2025.