AETHER: EFFICIENT SERVING OF LARGE-SCALE VECTOR SEARCH WITH SHARDED INDEXES

Jiaheng Lu [1] **Zhenning Yang** [1] **Jeremy Flics** [1] **Mosharaf Chowdhury** [1]

ABSTRACT

Retrieval-Augmented Generation (RAG) is a popular technique for improving the reliability of Large Language Models (LLMs) by reducing hallucination. Implementing this effectively often requires repeatedly searching through large vector databases. At scale, these vector searches become a substantial computational bottleneck for RAG-enabled LLM inference. In this project we introduce Aether, a prototype system that enhances search efficiency across massive sharded datasets through scheduling optimizations and dynamic file management. Aether is a highly adaptable system that is designed to scale across multi-tier memory systems, such as CXL-enabled clusters. Through comprehensive evaluations across diverse workloads and under varying memory constraints, we demonstrate that Aether significantly outperforms baseline approaches. Leveraging asynchronous I/O, our scheduling optimizations achieve an average throughput improvement of 19.7% over their synchronous counterpart. Additionally, dynamic index file management using a novel LFU+ caching policy outperforms traditional LRU by 14.5% in serving throughput.

1 INTRODUCTION

Large language models (LLMs)-based applications, such as chatbots and AI-driven conversational agents (Caldarini et al., 2022; Bozkurt, 2023), have become increasingly prevalent. However, a critical issue with these models is their tendency to produce "hallucinated" content— inaccurate statements that may sound convincing, but is not grounded in facts (Wang et al., 2023; Ye et al., 2023; Tonmoy et al., 2024; Xu et al., 2024). This becomes problematic when users rely on such generated information as a source of knowledge for decision making. The primary reasons behind this phenomenon include outdated training data and inevitable information loss during the model training process. Keeping the training data continuously updated while maintaining data quality, is logistically challenging and practically infeasible. On the other hand, even with the most up-to-date data, the LLMs will still require continuous fine-tuning to be able to incorporate new information. This too is not practical. Unavoidably, even state of the art models such as GPT-4 (OpenAI, 2024) and Llama (Touvron et al., 2023a;b), have a finite number of parameters. As such, they alone will never be able to encapsulate the entirety of world knowledge. (Minaee et al., 2024; Villalobos et al., 2022).

[1]Department of Computer Science and Engineering, University of Michigan, Ann Arbor, Michigan, USA. Correspondence to: Jiaheng Lu <jhlu@umich.edu>, Zhenning Yang <znyang@umich.edu>, Jeremy Flics <jflics@umich.edu>, Mosharaf Chowdhury <mosharaf@umich.edu>.

1.1 Retrieval-Augmented Generation

To mitigate these issues, retrieval-augmented generation (RAG) frameworks have emerged (Borgeaud et al., 2022; Lewis et al., 2020; Jiang et al., 2023; Huang & Huang, 2024; Gao et al., 2024) as a powerful solution. RAG enhances LLMs by integrating external data retrieval into the text generation process, thereby grounding responses with verifiable information. Drawing a parallel to human memory systems, comparing the long-term memory, which holds basic and general knowledge, to the parameters of LLMs (Sumers et al., 2024). Actively recalling previous experiences allows humans to make more informed decisions based on relevant past events. Unlike humans, LLMs lack such capability on their own. They do not have the mechanism to draw from specific past interactions; instead, their responses are generated based solely on the datasets with which they were trained on. As we delve into the mechanics of RAG, the concept of vector similarity search plays a crucial role, especially in how information is indexed and retrieved effectively.

1.2 Vector Similarity Search

Embeddings in deep learning are numerical representations of various data types, including text, images, and other modalities, or they can be a combination of multiple modalities, allowing DL models to process and analyze complex and diverse information effectively. Vector similarity search is essential, leveraging embeddings that capture the semantic essence of data and enable meaningful comparisons through

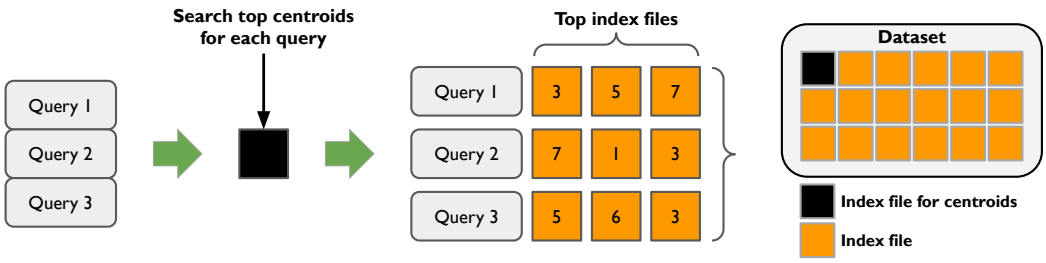

Figure 1. Vector search on sharded indexes

methods like k-nearest neighbors (KNN).

KNN is an exhaustive search method where the similarity between items is determined by finding the k^{th} closest vectors in the database to a given query vector. The closeness is often measured using Euclidean distance or cosine similarity, with the latter being preferred in text-related applications for its focus on vector orientation. However, as vector databases grow in size and dimensionality, KNN becomes computationally intensive. One way to manage this computational demand is to sacrifice exhaustiveness for faster-running algorithms. This computation-accuracy trade off has been well studied by, so called approximate nearest neighbor (ANN) algorithms (Andoni et al., 2018). A typical approach is to preprocess the entire dataset and generate an index that is optimized for faster searching. To do this, we used a sharded Inverted File Indexing (IVF) as described in Section 1.3. Other popular ANN techniques are discussed in Section 6.

Aside from approximation algorithms, more classical systems optimizations can be applied to vector similarity search. Techniques like intelligent batching (Zhang et al., 2024), careful execution planning (Jang et al., 2023), and maximizing concurrency (Wang et al., 2021) are all well studied. These techniques are particularly suited for enhancing vector search processes.

1.3 Motivation

The motivation behind enhancing vector search capabilities stems from the demand to scale up these systems to handle vast external knowledge bases efficiently. The primary challenge in vector search lies in its extensive memory requirements. Ideally, to achieve rapid vector search responses, index files should reside in fast memory such as dynamic random-access memory (DRAM). Existing solutions like Faiss (Douze et al., 2024; Johnson et al., 2017) and Rummy (Zhang et al., 2024), utilizing GPUs compute power to accelerate vector serach and vector query processing. However, in the context of RAG-enabled LLMs, where GPUs are predominantly occupied by model parameters, alternative solutions are necessary.

Our proposed system, Aether, aims to efficiently handle vector search across very large datasets. Traditionally, two existing methods are used when dealing with large datasets that cannot be fully loaded into fast memory. One method involves using memory-mapped files to load only the significantly smaller index graph, while the data remains on disk. However, this approach often results in slow response times due to frequent disk reads and is not entirely index agnostic.

The other method involves sharding (Douze, 2020) the dataset and indexing each shard individually, as shown in Figure 1. We first apply clustering algorithms, such as k-means, to partition the large dataset into smaller clusters and their corresponding centroids. We then index these smaller clusters and centroids. When a query is received, we initially search the centroid index to identify the top centroids relevant to the query. These centroids direct us to specific cluster index files for a more detailed search. We iterate through a list of top index files, performing vector searches sequentially. Finally, we merge and sort all retrieved data points by their distances to the query, selecting the top-k results to return to the user. This method is index agnostic and requires only a subset of index files at runtime, enhancing memory efficiency.

Our system optimizes the serving of large-scale vector searches for the sharded indexes, and specifically tailored for applications like RAG-enabled LLMs, where it is crucial that vector search and query processing do not compete for limited GPU resources. We observed significant improvements in both intra- and inter-batch response latency through careful search planning and dynamic index file management. Our key contributions in this project include:

- **Development of a Prototype System:** We developed a prototype system designed for serving vector search across large datasets using sharded indexes. The code is open-source and available on `https://github.com/JhengLu/GENAI_CXL_PLUS/tree/main` and `https://github.com/zyang37/VectorSearch_ShardIndex`.
- **Optimization of Throughput:** We enhanced intra- and inter-batch throughput through asynchronous in-

dex loading and dynamic index file management. A solution that significantly outperform naive planning, and can be generalized to multi-tier memory systems like CXL-enabled cluster to further enhanced scalability.

- **Comprehensive System Evaluation:** We evaluated our system across various workloads and under different constraints. This included testing with batches that had varying percentages of out-of-distribution (OOD) queries and operating in environments with different memory budgets.

2 SOLUTION

We proposed two main optimization strategies: intra-optimization in Section 2.1, which focuses on single batch, and inter-optimization in Section 2.2, which improves latency for continuous serving. Lastly, we outline the overall system design in Section 2.3 and we discuss each system component in detail, explaining their functions, interactions, and the roles they play in ensuring the overall system performance.

2.1 Intra-Batch Planning

The centroid index serves as the initial access point for our search, guiding us to the top centroid results which then direct us to specific index files for a more detailed, fine-grained search. In practice, we employ batch vector search, and the retrieval results are naturally structured like the search plan 1 (S1), as shown in Figure 2, where we have top indexes correspond to each query. Following this, we can immediately proceed with the fine-grained vector search,

Figure 2. Search plan 1 (S1)

However, this naive search plan often results in slow response times as shown in Figure 3. On one hand, batch vector search enhances speed by utilizing parallel processing to handle multiple queries simultaneously. It also reduces overhead by minimizing the frequency of data loading. Consequently, batch processing proves more efficient, allowing for the simultaneous and accelerated processing of multiple queries compared to sequential processing. On the other hand, search plan 1 introduces redundant loading of index files. For instance, using Figure 2 as an example, during the processing of query 1, we load index file 3 and perform a vector search. Once completed, we proceed to the next

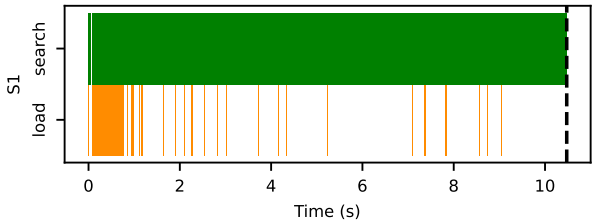

Figure 3. S1 latency breakdown for vector search for a single batch of 10,000 queries, assuming sufficient system memory for all required index files. The bottom bars represent the time spent loading each index file, while the top bars indicate the time spent searching an index file.

index file. While we accumulate retrieval results for query 1, memory constraints might force the eviction of some index files. Suppose index file 3 is evicted following a Least Recently Used (LRU) policy, only to be required again for query 2, having to re-load from the disk. This frequent swapping, especially under tight memory constraints, exacerbates further decreases the vector search performance.

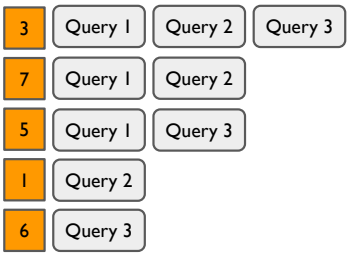

Figure 4. Search plan 2 (S2)

To address both single query search and the redundancy in index file loading, a revised approach involves restructuring the search plan to batch queries that share common index files, as illustrated in Figure 4. Before initiating the search, we first identify opportunities for batching. For example, if index file 3 is required for queries 1, 2, and 3, we batch these queries together. This strategy reintroduces batch vector processing, capitalizing on opportunities to process multiple queries concurrently. Additionally, organizing index files in the outer loop of the search process eliminates the need for redundant loading. In the context of intra-batch, following search plan 2 (S2) ensures that each top index file is loaded and searched only once. Implementing this revised plan has significantly improved response times compared to the original naive search plan.

So far, our search and loading operations have been conducted sequentially as blocking operations, shown at Figure 5. We've identified further opportunities for optimization by leveraging asynchronous I/O. This approach allows searching and loading to occur simultaneously without blocking each other. For example, during a vector search, the I/O system is typically underutilized. Meanwhile, the

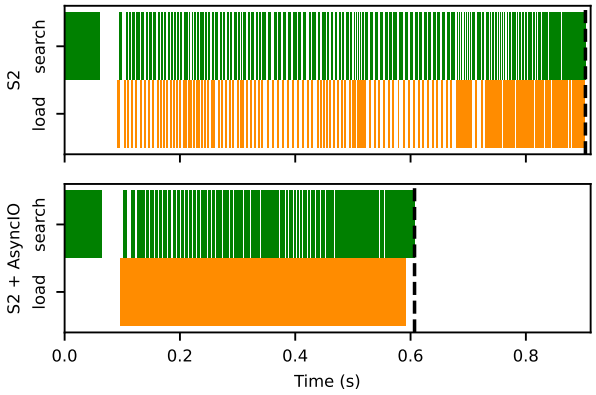

Figure 5. Latency breakdowns for S2 (top) and S2 with asynchronous I/O (bottom).

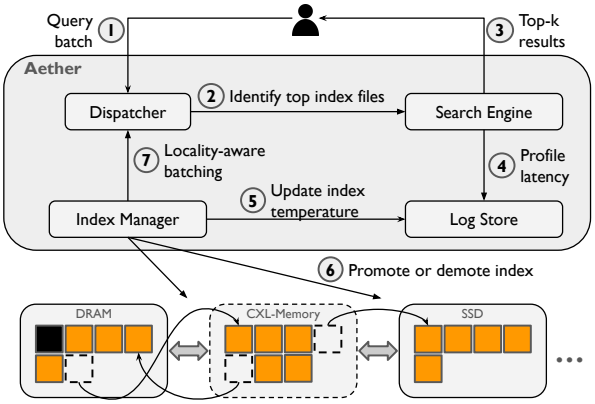

Figure 6. Aether architecture: serving large-scale vector search utilizing multi-tier memory, such as CXL-enabled clusters.

next index file required according to the search plan can be pre-loaded. If the loading of an index file completes before the current search finishes, the next search can begin immediately. If the loading is still in progress when the search completes, only a minimal delay is necessary before initiating the next search. This overlapping of tasks significantly reduces idle time or "bubbles" in the search pipeline.

Additionally, we've noticed that reordering the search plan can further minimize these bubbles. By prioritizing index files associated with larger query batches, we not only extend the search duration—allowing more time for index file pre-fetching—but also set the stage for more efficient processing of smaller batches later on. That means when we eventually deal with index files linked to only a few queries, these can occur almost back-to-back. The pre-fetching done during the longer searches ensures that there is little to no waiting time for loading the next index files, achieving better latency (Figure 5).

2.2 Inter-Batch Memory Management

In the context of serving vector searches, there is an opportunity to dynamically adjust the placement of index files based on their "temperature", reflecting their current usage rate. Initially, during the intra-batch phase, there is a period of I/O idleness at the very beginning, as the system waits for the vector search of the centroid index to complete. This delay is necessary to determine the top index files for subsequent searches.

Furthermore, moving into the inter-batch phase, the system does not always have a predefined search plan, allowing the index manager to dynamically promote or demote index files based on their temperature. We've implemented two key policies for managing this: the traditional Least Recently Used (LRU) and a modified version of Least Frequently Used (LFU), which we term LFU+. Unlike traditional LFU, which simply counts accesses, LFU+ adjusts the count by the number of queries processed per index file. This modification better represents the actual importance of index files, as a single large batch search on one index file might be more significant than several single-query searches on another.

This dynamic placement of index files has led to an small enhancement in our intra-batch optimization strategy. When reordering index searches, the system now prioritizes index files that are already loaded in DRAM (locality-aware), followed by those associated with larger query batches, allowing us to initiate searches as quickly as possible.

2.3 System Design

Aether is a framework for efficiently serving vector search queries over large datasets. as shown in Figure 6. At a high level, Aether efficiently batches requests, and searches through sharded index files to find the top-k results for each query. Users submit requests to the dispatcher which figures out which index files are needed for the search, and batches the requests accordingly. Next, ANN searches are conducted on a per-index file basis and returned to the user. To do this efficiently, frequently used index files are loaded into DRAM by the index manager. Throughout this entire process, profile data is logged both to inform index file popularity rankings and to support experimental measurement.

A number of optimizations are made to improve the scalability and performance of this process. First, as an offline preprocessing step the corpus is sharded into index files using k-means clustering. Then when a batch of queries comes in, the most relevant index files are identified based on the distance between the query vectors and the index file centroids. Next, the dispatcher generates search plan for the current batch. Whenever an index file is accessed, the index manager updates its internal popularity rankings and loads/evicts indexes accordingly.

2.3.1 Dispatcher

The Dispatcher handles incoming query batches and generating efficient search plans. It starts by taking in a query batch and conducting a vector search over the centroid index to identify the top index files for each query. It then identifies batching opportunities among queries within the same batch, organizing the search plan such that the index file is in the outer loop, and queries sharing the same index file are batched together. The search order for index files is strategically reordered: priority is given first to index files already present in DRAM, and second to those with a larger batch size, in order to minimize interruptions in the search process. Ultimately, the Dispatcher outputs a search plan for the current batch to the search engine, ensuring optimal resource use and low response latency.

2.3.2 Search Engine

The Search Engine executes the search plan generated by the Dispatcher. It first verifies the availability of the required index file in DRAM. If the file is not already in DRAM, the engine attempts to load it, pausing the search until the file is fully loaded. If the file is already in DRAM or once it is loaded, the engine proceeds with the vector search. Throughout the search process, the engine keeps track of all partial results and continuously updates the result matrix, which has the shape of $(NumQuery, k)$. This involves concatenating partial results and sorting the retrieved data points by their distance to the query, ensuring that only the top-k results are retained. Once the process is complete, the Search Engine returns the final sorted, topk results.

2.3.3 Index Manager

The Index Manager oversees index file operations, implemented as a separate thread from the main process for asynchronous handling. It operates primarily based on the search plan provided for each batch, which details the specific index files needed for current queries, thus facilitating precise intra-batch optimization. When a search plan is active, the Index Manager adheres strictly to it to manage index files accordingly. However, in periods of inactivity with no incoming requests, the Index Manager shifts its focus to managing the index files based on their popularity rankings. This involves maintaining an in-memory cache of index files and adjusting their presence in DRAM based on a defined policy such as Least Recently Used (LRU), Least Frequently Used (LFU), or enhanced versions of these policies (LRU+ or LFU+). This dynamic approach allows the Index Manager to optimize resource use effectively, ensuring that the system is prepared for new queries as they arrive.

2.3.4 Log Store

The Log Store is a component designed to capture and store performance logs specifically related to the loading and storing actions of index files. This data serves two main purposes: it provides a quantitative basis for experimental measurement, allowing for the assessment and improvement of the system's performance, and it acts as a source of information for the Index Manager. By analyzing these logs, the Index Manager can refine the index file ranking, optimizing the way index files are placed based on actual usage and performance metrics. This ensures that the system's memory management adapts dynamically to the changing demands and patterns observed during operation.

3 EVALUATION

We conducted a thorough evaluation of our system using synthetic datasets. The evaluation involved a dataset with 1,000 clusters, implying the creation of 1,000 distinct shards, each indexed using the Inverted File (IVF) method. Each cluster comprised 10,000 embeddings, with each embedding having a dimensionality of 128. To generate these embeddings, we used a Gaussian distribution, assigning different mean values to each cluster to minimize overlap among them. Furthermore, we employed a smaller standard deviation in the distribution to further reduce the overlap, ensuring that each cluster's embeddings were distinct and clearly separable from those of other clusters.

3.1 Experiment Setup

The evaluation is structured around a two-tier memory setup, consisting of DRAM and disk storage. When an index file is required for a query, the system first checks if it is available in DRAM. If the file is not in DRAM, it is loaded from disk into DRAM before vector search. If the file is already in DRAM, vector search is executed immediately.

We simulate memory constraints by defining the maximum number of index files that DRAM can hold. When capacity is exceeded, files are swapped in and out based on either the LRU or LFU+ eviction policies, depending on the specified system configuration. This approach allows us to evaluate the system's performance under various memory constraints.

Users can specify the number of queries in each batch. We also define the percentage of these queries that are generated from a different Gaussian distribution, identifying these as out-of-distribution (OOD) queries. This setup allows us to test the system's robustness in handling anomalies queries in a batch during vector searches. Query batches are generated according to user specifications, including the top-k results to return and the "nprobe" parameter, which indicates the number of top index files to examine during the search.

Table 1. Latency for generating S2

Batch size	Planning latency (s)
10	0.000079
100	0.00031
1000	0.0024
10,000	0.023
100,000	0.23
1000,000	2.41

Inter-batch evaluation involves defining the number of requests, with each request containing a randomly determined percentage of OOD queries. Users select an eviction policy—either LRU or LFU+—to manage the memory effectively.

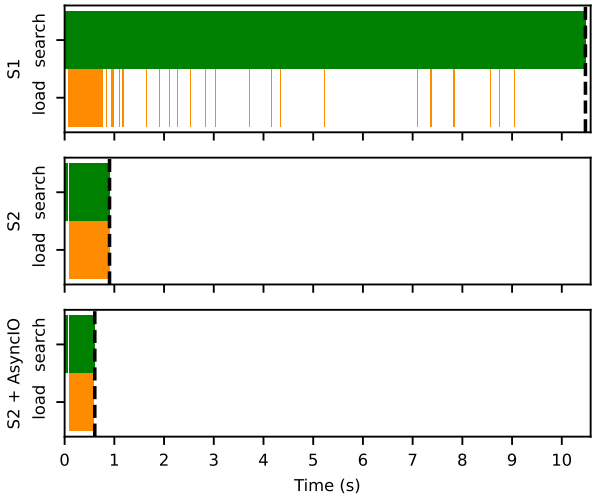

Figure 7. Latency breakdowns for S1 (top), S2 (middle) and S2 with asynchronous I/O (bottom). Single query batch with 10,000 queries, returning top 10 data points per query, with the "nprobe" parameter set to 10.

3.2 Search Plan Generation Latency

In the evaluation section of our study, we observed that vector search results are inherently structured similarly to S1. Therefore, following the centroid search, we immediately have access to S1 without the need for additional time-consuming processes such as those required for generating S2. This allows us to initiate the fine-grained search almost instantaneously. To this end, we have profiled the time required to generate S2 (Table 1), which includes not only the time needed to reverse the structure of S1 but also the time necessary to reorder the results based on batch size and the locality of the index files.

3.3 Tight Memory Budgets

We assessed the intra-batch throughput (queries per second) under various memory budgets, with the results depicted in Figure 8. We find that the performance of S2 is largely independent of the memory budgets, provided that the system can accommodate at least one index file necessary for the current search. This feature is particularly beneficial in environments with limited memory, as it ensures efficient utilization of available resources. Once a search involving a specific index file is completed, that index file can be safely evicted from memory, as it is no longer required for the remainder of the current batch. This strategy helps in managing memory more effectively by freeing up space that can be used for loading other necessary index files.

Additionally, S2's integration with asynchronous I/O (Async IO) can take advantage of higher memory capacities by preloading more index files, although benefits plateau once memory availability exceeds the maximum required for active searches. On average, S2 with Async IO is 19.7% faster than S2. In contrast, S1 consistently underperforms due to its excessive redundant index file loading and inefficient processing of single queries.

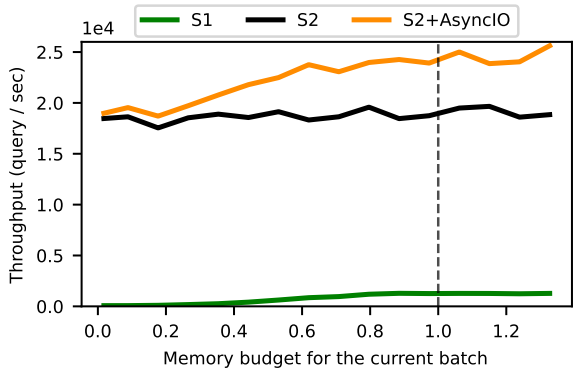

Figure 8. Throughputs for S1, S2, and S2 with Async I/O under varying memory budgets. The vertical dotted line indicates a 100% memory budget for the current batch. For example, if executing search plan requires 100 index files for the current batch, a 100% memory budget means the system has sufficient memory to hold all 100 index files without needing to swap. This is why the throughputs for all search plans plateau at this point.

3.4 Out-of-Distribution Query-batch

While S2's performance is generally independent of memory constraints, it becomes sensitive when handling batches that contain Out-of-Distribution (OOD) queries as shown in Figure 9. OOD queries often require a significantly different subset of index files compared to typical queries, which drastically reduces the batching opportunities. This leads to an increase in single query searches, which are less efficient. Although implementing S2 with asynchronous I/O (Async IO) does achieve a slightly higher throughput, the inefficient single query searches still dominate overall performance, with only 2% higher throughput compared to normal S2.

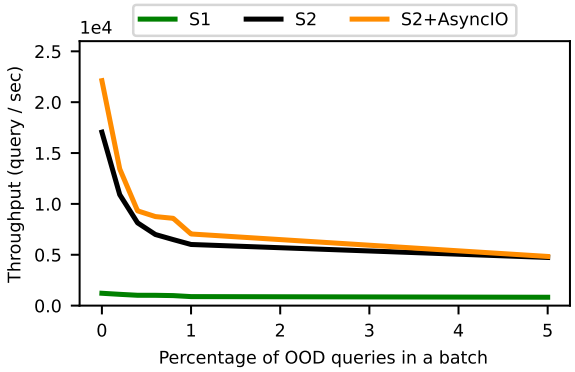

Figure 9. Throughputs for S1, S2, and S2 with Async I/O under diverse workloads containing OOD queries.

3.5 Serving Throughput

In assessing continuous vector search serving performance, we evaluated two caching policies, Least Recently Used (LRU) and the enhanced Least Frequently Used (LFU+), as depicted in Figure 10. Our test involved running through 100 requests, each containing 10,000 queries, into which we randomly injected OOD queries to simulate anomaly within realistic workloads. Our findings indicate that LFU+, through dynamic index file promotion and demotion, processed all 100 requests 14.5% faster than LRU.

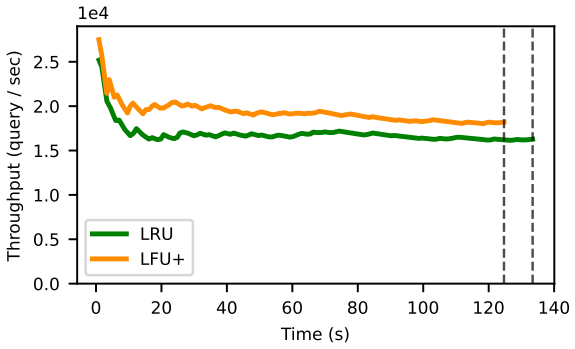

Figure 10. Throughputs for processing 100 requests using two different policies, LRU and LFU+, in S2 with Async I/O.

4 Limitations

While Aether breaks ground on scalable vector searching systems, it has some limitations that, if addressed, could substantially improve its generality and performance. The first of these is a more robust experimental setup. Specifically, due to resource constraints, it was challenging to model realistic workloads- especially those coming from multiple distributions of data. As evidenced by Figure 9 the performance of S2 is substantially impacted by out of distribution queries. This was hard for us to measure because it is truly a problem that appears at a larger scale than our hardware allowed. In our case, generating synthetic data from multiple distributions almost immediately led to each query batch using every index file, thus making our caching strategy a moot point. In addition, we assume clusters are approximately the same size which might not be the case when it comes to large realistic datasets.

5 Future Work

Beyond just addressing its limitations, Aether raises many questions that could guide future research projects.

Gaussian Mixtures Model. Given appropriate scale, we have some ideas for future work that could ameliorate the performance degradation brought forth by a heterogeneous query distribution. One approach would be to try to learn the underlying distributions from which the queries are coming. For example a Gaussian Mixtures Model (GMM) (Reynolds et al., 2009) could be used to cluster incoming queries into several distributions. Then for each distribution one Aether process can be run. This should minimize cache pollution due to out-of-distribution queries.

Other Indexing Methods. Currently Aether uses IVF for vector search. However, this framework should be agnostic to ANN algorithm. One way to improve the generality of Aether would be to test it with different vector search strategies such as HNSW (Malkov & Yashunin, 2018).

RAG LLM Integration. Since one of the main applications of vector search is RAG-enabled LLMs, it would be also interesting to run some end-to-end experiments on serving RAGs using Aether to serve vector search. Latency, throughput, and output accuracy are all relevant metrics to track.

Integration with Multi-Tier Memory Systems. Another area that deserves more attention from future work is optimizing Aether in a multi-tier memory setting either locally, or over a network. In this case, it is possible that some tiers would require different promotion and demotion schemes. Since the goal of this work is to support vector search at arbitrary scale, it would be interesting to integrate this serving system with memory hierarchies that are used in and across data centers. For example, supporting vector search at global scale would require integration with a Content Delivery Network (CDN) (Vakali & Pallis, 2003). Alternatively, we have done some work benchmarking CXL memory which brings its own challenges and opportunities.

Index File Migration in CXL-enabled Clusters. In Aether, our primary focus lies in index file migration between DRAM and Disk. Looking ahead, we aim to extend this capability to include migration between DRAM and CXL-attached remote memory. This enhancement promises to accelerate vector search speed compared to swapping between

disk and DRAM. Notably, local DRAM indexing achieves a search speed nearly three times faster than indexing solely on CXL-attached memory, making Aether's intelligent index file placement valuable in multi-tiered memory systems. For details on the relationship between search speed and index file location, as well as our progress on multi-tiered memory system migration, refer to the appendix A.

QoS-aware Vector Search Serving. In this paper, Aether introduces an innovative intra-batch planning method that associates each index file with queries, investigating both sequential and concurrent search and loading strategies. While the latter boosts overall speed, it may encounter bandwidth constraints. Moreover, ensuring Quality of Service (QoS) in vector search serving based on user priorities is essential. For details on bandwidth impact on query throughput and our ongoing work on QoS-aware bandwidth control for query processes, see appendix B.

6 Related Work

Many ANN algorithms are used to improve vector search performance. These algorithms typically use one of three techniques: clustering, hashing, and building searchable graphs. IVF, as described in Section 1.3 is an example of a clustering technique. The following is a brief survey of popular ANN techniques that fall into the other categories.

Locality-Sensitive Hashing (LSH) (Datar et al., 2004) segments vectors and repeatedly hashes them. Then, two vectors are considered similar provided that these hashes collide at least once. In this way, vectors can be compared based on their segments which reduces search time to sublinear. Hierarchical Navigable Small Worlds (HNSW) (Malkov & Yashunin, 2018) embeds data points as nodes on several graphs of increasing complexity that can be searched in a hierarchical manner. The idea here is similar to that of a probabilistic skip list. Each layer in the hierarchy can be greedily searched until a local minimum is reached, until a vector within acceptable distance from the query vector is found.

Modern large vector datasets (Jégou et al., 2021) contain billions of vectors. Storing these massive datasets in DRAM proves to be highly costly due to their extensive memory footprint, spanning thousands of GB. To improve scalability, a hierarchical memory system can be used. One way this can be achieved is by using a large pool of slower but cheaper memory (Maruf et al., 2023).

Utilizing Compute Express Link (CXL) enhances server memory capacity and bandwidth, essential for managing extensive vector datasets in cloud environments. Additionally, reusing DDR4 memory from decommissioned servers is currently supported by CXL, which is gaining substantial attention across industry and academic sectors. Pond (Li et al., 2023) and TPP (Maruf et al., 2023) both adopt the CXL standard to optimize memory usage. TPP introduces an OS-level, application-transparent page placement mechanism for CXL-enabled memory, while Pond develops a CXL-based full-stack memory pool for cloud deployment, incorporating a prediction model for latency and resource management at the datacenter scale.

7 Conclusion

In this project, we developed a prototype system called Aether, optimized for vector search across large datasets with sharded indexes. This system significantly outperforms naive planning by utilizing asynchronous index loading and dynamic file management, adaptable to multi-tier memory systems like CXL-enabled clusters for enhanced scalability. Comprehensive evaluations demonstrated that S2 with asynchronous I/O is 19.7% faster than its synchronous version, and our LFU+ caching policy outperforms LRU by 14.5%. These improvements highlight the system's robustness and efficiency across various workloads and memory conditions, making it a valuable asset for vector search applications.

References

Andoni, A., Indyk, P., and Razenshteyn, I. Approximate nearest neighbor search in high dimensions, 2018.

Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., van den Driessche, G., Lespiau, J.-B., Damoc, B., Clark, A., de Las Casas, D., Guy, A., Menick, J., Ring, R., Hennigan, T., Huang, S., Maggiore, L., Jones, C., Cassirer, A., Brock, A., Paganini, M., Irving, G., Vinyals, O., Osindero, S., Simonyan, K., Rae, J. W., Elsen, E., and Sifre, L. Improving language models by retrieving from trillions of tokens, 2022.

Bozkurt, A. Unleashing the potential of generative ai, conversational agents and chatbots in educational praxis: A systematic review and bibliometric analysis of genai in education. *Open Praxis*, Nov 2023. doi: 10.55982/openpraxis.15.4.609.

Caldarini, G., Jaf, S., and McGarry, K. A literature survey of recent advances in chatbots. *Information*, 13(1), 2022. ISSN 2078-2489. doi: 10.3390/info13010041. URL https://www.mdpi.com/2078-2489/13/1/41.

Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pp. 253–262, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138857. doi: 10.

1145/997817.997857. URL https://doi.org/10.1145/997817.997857.

Douze, M. Indexing 1t vectors, 2020. URL https://github.com/facebookresearch/faiss/wiki/Indexing-1T-vectors.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. The faiss library, 2024.

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., and Wang, H. Retrieval-augmented generation for large language models: A survey, 2024.

Huang, Y. and Huang, J. A survey on retrieval-augmented text generation for large language models, 2024.

Jang, I., Yang, Z., Zhang, Z., Jin, X., and Chowdhury, M. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23. ACM, October 2023. doi: 10.1145/3600006.3613152. URL http://dx.doi.org/10.1145/3600006.3613152.

Jiang, Z., Xu, F. F., Gao, L., Sun, Z., Liu, Q., Dwivedi-Yu, J., Yang, Y., Callan, J., and Neubig, G. Active retrieval augmented generation, 2023.

Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with gpus, 2017.

Jégou, H., Tavenard, R., Douze, M., and Amsaleg, L. Searching in one billion vectors: re-rank with source coding. In *ICASSP*, 2021.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. Retrieval-augmented generation for knowledge-intensive nlp tasks. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

Li, H., Berger, D. S., Novakovic, S., Hsu, L., Ernst, D., Zardoshti, P., Shah, M., Rajadnya, S., Lee, S., Agarwal, I., Hill, M. D., Fontoura, M., and Bianchini, R. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*, 2023.

Malkov, Y. A. and Yashunin, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, 2018.

Maruf, H. A., Wang, H., Dhanotia, A., Weiner, J., Agarwal, N., Bhattacharya, P., Petersen, C., Chowdhury, M., Kanaujia, S., and Chauhan, P. TPP: Transparent page placement for CXL-enabled tiered memory. In *ASPLOS*, 2023.

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., and Gao, J. Large language models: A survey, 2024.

OpenAI, e. a. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Reynolds, D. A. et al. Gaussian mixture models. *Encyclopedia of biometrics*, 741(659-663), 2009.

Sumers, T. R., Yao, S., Narasimhan, K., and Griffiths, T. L. Cognitive architectures for language agents, 2024.

Tonmoy, S. M. T. I., Zaman, S. M. M., Jain, V., Rani, A., Rawte, V., Chadha, A., and Das, A. A comprehensive survey of hallucination mitigation techniques in large language models, 2024.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023a.

Touvron, H., Martin, L., and Kevin Stone, e. a. Llama 2: Open foundation and fine-tuned chat models, 2023b.

Vakali, A. and Pallis, G. Content delivery networks: status and trends. *IEEE Internet Computing*, 7(6):68–74, 2003. doi: 10.1109/MIC.2003.1250586.

Villalobos, P., Sevilla, J., Besiroglu, T., Heim, L., Ho, A., and Hobbhahn, M. Machine learning model sizes and the parameter gap, 2022.

Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., Yu, K., Yuan, Y., Zou, Y., Long, J., Cai, Y., Li, Z., Zhang, Z., Mo, Y., Gu, J., Jiang, R., Wei, Y., and Xie, C. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pp. 2614–2627, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457550. URL https://doi.org/10.1145/3448016.3457550.

Wang, X., Yan, Y., Huang, L., Zheng, X., and Huang, X. Hallucination detection for generative large language

models by Bayesian sequential estimation. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 15361–15371, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.949. URL https://aclanthology.org/2023.emnlp-main.949.

Xu, Z., Jain, S., and Kankanhalli, M. Hallucination is inevitable: An innate limitation of large language models, 2024.

Ye, H., Liu, T., Zhang, A., Hua, W., and Jia, W. Cognitive mirage: A review of hallucinations in large language models, 2023.

Zhang, Z., Liu, F., Huang, G., Liu, X., and Jin, X. Fast vector query processing for large datasets beyond GPU memory with reordered pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 23–40, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/zhang-zili-pipelining.

A INDEX FILE MIGRATION BETWEEN CXL AND LOCAL DRAM

The distribution of index files on CXL and local DRAM can significantly impact latency as shown in Figure 11, using Faiss' exact KNN search method, the speed of placing all index files on the local DRAM can approach three times that of placing all index files on the CXL attached memory. Therefore, optimizing index file allocation and enabling the promotion and demotion of index files in multi-tiered memory systems is crucial.

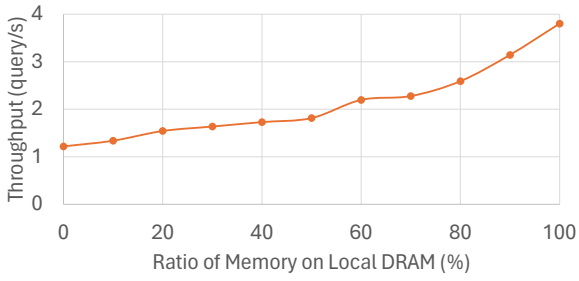

Figure 11. Ratio of Memory on Local DRAM Influence on Throughput

Linux handles page migration using the `migrate_pages()` function, primarily designed to migrate only LRU pages. This migration is triggered when a page accessed by the CPU is not located on the local NUMA node, leading to NUMA hinting faults. The LRU mechanism in Linux comprises two lists: the `active_list` and the `inactive_list`. Pages are initially placed in the `inactive_list` and, if accessed frequently, are promoted to the `active_list` to shield them from reclamation. Conversely, pages are demoted back to the `inactive_list` if the `active_list` becomes overpopulated.

To manually migrate the entire index file, we plan to use the `activate_page()` function, which transfers a page from the `inactive_list` to the `active_list`. Additionally, the `/proc/<pid>/numa_maps` file, which provides information on virtual memory address ranges and page numbers, will be instrumental in managing all page data for specific index files.

When indexing with Faiss, the index files are initially loaded and stored in memory as anonymous pages and later stored as file pages, though infrequently accessed. It is also important to consider demoting duplicate file pages to remote memory to enhance system performance. We have already had the tool to demote memory for specific processes, which modifies the cgroup to do that.

B QOS AWARE BANDWIDTH CONTROL FOR VECTOR SEARCH

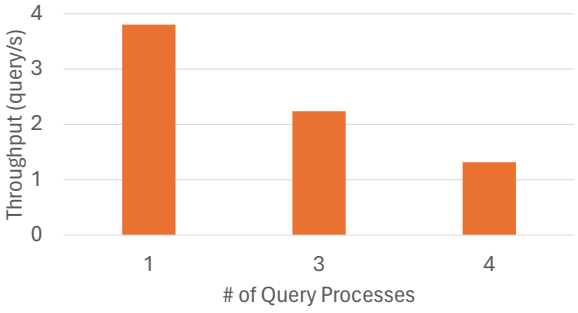

Figure 12. Number of Query Process Influence on Throughput

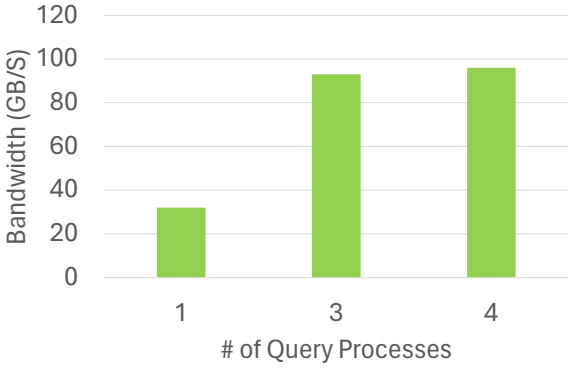

Figure 13. Bandwidth of Multiple Query Processes

In this paper, Aether introduces a novel intra-batch planning approach that categorizes each index file with queries. We

explore two strategies: sequentially searching and loading queries, and concurrently loading and searching multiple index files. The latter can significantly enhance overall speed. However, simultaneous multiple query processes can lead to bandwidth limitations. In Figure 13, running four concurrent Faiss' exact KNN search processes increases bandwidth usage to 96 GB/s, nearing the machine's maximum capacity of 100 GB/s. This causes each process's throughput to decrease to one-third of its individual performance, as shown in Figure 12, despite no memory or core contention, all query processes utilize local DRAM. Specifically, running a single query process allows for a bandwidth of 32 GB/s, whereas running four processes concurrently reduces each process's bandwidth to 24 GB/s, leading to the decrease in query throughput.

To manage this, we developed a bandwidth control tool using OS APIs (e.g., cgroup), which is available open-source at https://github.com/JhengLu/GENAI_CXL_PLUS/tree/main. In modern cloud environments, where users may have varying priorities, it is crucial not just to batch queries by arrival time but also according to user priorities. Our approach wants to enable simultaneous query execution while managing bandwidth allocation through our tool, decreasing the bandwidth allocated to lower-priority queries to favor higher-priority ones.