CALL: Context-Aware Low-Latency Retrieval in Disk-Based Vector Databases

Yeonwoo Jeong[1], Hyunji Cho[1], Kyuli Park[1], Youngjae Kim[1], Sungyong Park[1†]
[1]*Sogang University*, Seoul, Republic of Korea
{akssus12, llawliet37, kyuripark, youkim, parksy}@sogang.ac.kr

***Abstract*—Embedding models capture both semantic and syntactic structures of queries, often mapping different queries to similar regions in vector space. This results in non-uniform cluster access patterns in modern disk-based vector databases. While existing approaches optimize individual queries, they overlook the impact of cluster access patterns, failing to account for the locality effects of queries that access similar clusters. This oversight increases cache miss penalty. To minimize the cache miss penalty, we propose CALL, a context-aware query grouping mechanism that organizes queries based on shared cluster access patterns. Additionally, CALL incorporates a group-aware prefetching method to minimize cache misses during transitions between query groups and latency-aware cluster loading. Experimental results show that CALL reduces the 99th percentile tail latency by up to 33% while consistently maintaining a higher cache hit ratio, substantially reducing search latency.**

***Index Terms*—Disk-based Vector Search, Low Latency**

I. INTRODUCTION

Large Language Models (LLMs) often generate responses that include incorrect information when prompted with queries beyond their training data [1]. To mitigate this limitation, Retrieval-Augmented Generation (RAG) applications enhance user prompts by integrating relevant domain-specific documents retrieved from a vector database [2]–[4]. In these systems, unstructured documents are embedded into high-dimensional vectors and indexed, allowing the prompt's embedding to be compared against them using similarity metrics (e.g., inner product or cosine similarity) for grounded response generation. Recently, RAG applications are increasingly deployed as backend cloud services, handling continuous streams of user queries in diverse domains such as real-time clinical guidance [5], [6] and interactive assistant agents [7], [8]. These *stream-based RAG applications* frequently encounter concurrent queries, as illustrated in Figure 1, which require low-latency for each response. In stream-based RAG applications, timely document retrieval is crucial for supporting fast LLM inference, making low-latency vector search a critical challenge.

In practice, vector databases commonly load entire vector index into memory. While the in-memory vector search offers high-speed retrieval, the size of vector

†S. Park is the corresponding author.

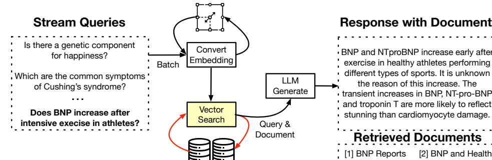


Fig. 1: A workflow of stream-based RAG application. Red arrows indicate vector search in the RAG pipeline.

indexes often exceeds available memory capacity [9]–[11]. For example, constructing one billion floating-point vectors in 96 dimensions requires over 350 GB of memory, which often surpasses the memory capacity of a typical single server [10]. To address this memory constraint, two alternative solutions have been explored: distributed vector search and disk-based vector search. Distributed vector search [12] partitions the index across multiple servers and parallelizes the search, allowing the system to scale with increasing index size. While effective, this approach incurs significant infrastructure costs, as larger indexes require additional memory-heavy index servers. In contrast, disk-based vector search [9]–[11] stores the entire vector index on high-speed secondary storage such as NVMe SSDs. This approach eliminates the need for memory-intensive servers and offers a cost-effective alternative by leveraging a single machine equipped with fast storage. These systems partition the vector index into multiple clusters and retrieve only the relevant clusters on demand at runtime.

Although partial clusters are loaded into memory at runtime, I/O remains a key bottleneck in disk-based vector databases [10]. Empirical studies reveal that I/O accounts for over 90% of the total search latency, while computation constitutes less than 10% [9]. To minimize I/O overhead, modern vector databases incorporate application-level cache mechanisms [9], [11], [13]. For instance, FAISS [14], a widely used vector database, employs a cluster cache to retain frequently accessed cluster files, which are partitioned subsets of the full vector index. Before accessing storage, FAISS checks the cluster cache. If the required cluster is cached, it avoids disk access, otherwise it retrieves the cluster from the disk, incurring latency. To mitigate cache miss penalties, prior works have proposed several techniques

such as frequency-based caching [11], [13]. However, these approaches often overlook deeper system-level bottlenecks that arise under multi-user query workloads.

Our preliminary studies using the RAG benchmark datasets [15] reveal that query access patterns exhibit non-uniform cluster locality. (Section III). In stream-based RAG applications, where queries are processed in batches, this non-uniform locality presents a valuable opportunity to optimize cluster cache utilization. However, prior work generally treats incoming queries as independent, missing opportunities for scheduling across batched queries. In contrast, our work focuses on the streaming scenario, where batch-level query scheduling can significantly reduce overall latency.

We identify three key problems that limit the performance benefits of in-batch query scheduling.

Non-Uniform Cluster Access Patterns. Modern embedding models encode text into dense vector representations that capture both semantic meaning and syntactic structures (e.g., question formats and specific phrasings). As a result, even queries with different content within the same batch may be mapped to similar regions in the vector space. However, existing approaches process incoming queries in order and fail to consider contextual similarities among queries within the batch. This lack of awareness leads to suboptimal cache hit rate. Reordering queries that access a high proportion of the same cluster files can significantly improve cluster locality in vector databases.

Prefetch Invisibility. While such grouping increases cache hit rates within each group, it introduces a new challenge at group boundaries. Transitions between groups with disjoint cluster access patterns can lead to abrupt cache evictions. Because the vector database is unaware of the group structure, it cannot prefetch the necessary clusters in advance. As a result, some clusters are evicted prematurely, reducing cache efficiency and increasing tail latency.

Imbalanced Cluster Loading. Even with well-formed query groups and effective prefetching, some clusters inevitably need to be fetched from disk at runtime. This introduces a system-level challenge that can reduce parallel efficiency during vector search. Cluster files vary significantly in size due to uneven document distribution or inconsistent vector density across shards [16]. Modern vector search engines such as Milvus [12] and Qdrant [17] typically assign homogeneous tasks (e.g., loading documents or generating embeddings) to a shared job queue processed in parallel by multiple worker threads. Similarly, cluster loading tasks enter a shared queue and are dispatched to threads in round-robin order. When large cluster files are added to the queue without considering their size, they become stragglers, occupying a thread for a long time, which causes severe load imbalance among worker threads.

Our Approach. This paper proposes CALL, context-aware low latency retrieval in disk-based vector databases, which jointly addresses the three aforementioned problems: non-uniform cluster access patterns, prefetch invisibility, and imbalanced cluster loading.

First, we propose a context-aware grouping method that analyzes cluster access patterns of batched queries and partitions them into virtual groups based on similarity. While content or keyword-based query grouping has proven effective in web caching and recommendation systems, these approaches are fundamentally unsuitable for vector databases. This is mainly because queries in vector databases are represented as dense, high-dimensional embeddings, which inherently lack explicit semantic tokens or features suitable for keyword-based analysis. Furthermore, relying solely on vector similarity does not guarantee optimal caching behavior, as semantically similar queries may still access disjoint sets of clusters. Instead, our approach shifts caching granularity from individual queries to groups of queries sharing similar cluster access patterns.

Second, we introduce a group transition-hinted prefetching algorithm, which uses group-level metadata to prefetch clusters ahead of group transitions, reducing cache miss penalties. It attaches lightweight metadata that marks group boundaries, enabling the system to anticipate cache needs and reduce cache evictions during group switches.

Finally, we present a load-weighted greedy packing algorithm which prioritizes large cluster files and distributes them evenly across available worker threads. This approach mitigates the long-tail task execution that can reduce overall efficiency in parallel cluster loading.

In summary, our key contributions are as follows:

- We reveal that queries processed in batches exhibit non-uniform cluster access patterns, offering opportunities to improve cache hit rate through query reordering.
- We identify why disk-based vector databases using cluster caches fail to improve search performance despite employing cache replacement policies.
- We propose CALL, a system designed to integrate seamlessly with various caching mechanisms employed in modern vector databases.
- Our extensive evaluation across various datasets demonstrates that CALL outperforms baseline methods by up to 33% in 99th percentile tail latency and achieves up to an 84% improvement in end-to-end search latency, all without introducing additional overhead for grouping.

II. Background and Related Works

A. Vector Database

Vector databases are specialized systems designed to store, index, and search large-scale, high-dimensional

embeddings. The core functionality of a vector database consists of two main stages: indexing and retrieval.

Indexing. Rather than storing all vectors in a flat array, Approximate Nearest Neighbor (ANN) indexing techniques organize vector embeddings to prune the search space. ANN methods are typically categorized into graph-based and clustering-based algorithms.

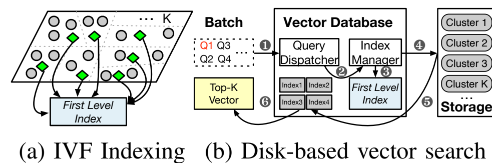
- Graph-based indexing builds a navigable proximity graph over the vector space. Each node (e.g., vector) is connected to a subset of other nodes based on local similarity, forming a small-world topology. The graph encodes neighborhood relationships that can be exploited for efficient traversal during search. This offers high search accuracy and fast convergence but typically requires high memory overhead.
- Clustering-based indexing partitions the vector space into disjoint regions, using unsupervised clustering algorithms like k-means [18]. This allows the search to focus on a small number of candidate regions, reducing computation and disk I/O. This approach is particularly effective for disk-based deployments due to its compact index representation. In this paper, we employ the clustering-based indexing method to implement disk-based ANN vector search.

Retrieval. Once the index has been constructed, vector search is performed by encoding the user query into a dense embedding using the same model used during indexing. The query vector is traversed through the index to find a subset of candidate vectors likely to be similar. Similarity scores between the query vector and the candidate vectors are computed using distance metrics such as Euclidean distance. The system then returns the top-k vectors with the highest similarity scores.

B. Disk-based ANN Vector Search

A disk-based ANN vector search mechanism is proposed to address the memory limitations of in-memory vector search, where the entire index often cannot fit into DRAM. To overcome this, this technique offloads large-scale vector index to high-speed storage and fetches on-demand indexes needed for ANN search [9], [10]. Disk-based ANN vector search commonly adopts clustering-based indexing algorithms [11], [14].

Figure 2 (a) shows a workflow of Inverted File (IVF [19]) indexing, one of the widely used clustering-based algorithms. The entire vector space is first partitioned into k regions through k-means clustering. During training, k-means iteratively refines a set of centroids by alternating between two steps: assigning each vector to its nearest centroid and updating each centroid as the mean of its assigned vectors. This process repeats until convergence, producing a set of representative centroids that partition the space into disjoint clusters. The K clusters with the disjoint vector space are stored on disk as vector arrays. These centroids are stored in a first level index, which resides entirely in memory. At query time, the vector database uses the first level index to perform an initial comparison, narrowing the search scope.



(a) IVF Indexing (b) Disk-based vector search

Fig. 2: IVF indexing and disk-based vector search workflow. In (a), circles in gray represent vectors, while green diamonds denote centroids.

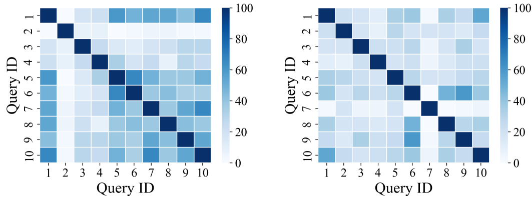
Figure 2 (b) shows a disk-based vector search using IVF indexing. It follows a two-level search process. Upon receiving batched queries (e.g., Q1, Q2, Q3, and Q4), each query is encoded into a multi-dimensional vector ❶. The query dispatcher sends the query vectors to index manager ❷. Then, the index manager scans the first level index to identify the centroids closest to each query vector ❸. The vector database exposes a parameter, *nprobe*, which controls the number of clusters scanned among the clusters. In this example, four out of k clusters are selected for Q1's vector search. Based on the found centroids, the index manager requests four clusters from the storage ❹ and constructs a partial index ❺. The partial index, built from the selected clusters, enables top-k search by restricting computation to a small subset of the full vector index. Finally, Q1 is searched over the indexes, after which the vector database ranks the results by distance, returning the k nearest vectors ❻.

C. Related Works

To date, there have been several works to enhance the performance in disk-based ANN vector search under constrained memory conditions. Most existing work focuses on optimizing cache hit rate at the individual query level by applying traditional cache replacement policies [9], [11], [13].

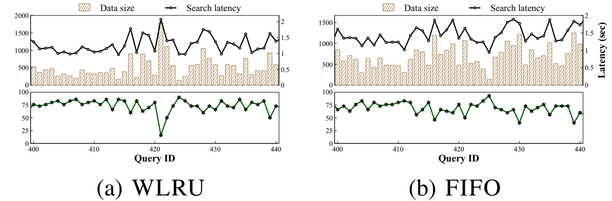
Caching Management adopts a multi-tiered storage architecture, where frequently accessed clusters are retained in memory, while infrequently accessed clusters are stored in storage. DiskANN [9] introduces a disk-based ANN search algorithm optimized for large-scale vector datasets. DiskANN keeps a partial search graph in memory and lazily loads additional nodes and edges from SSD. To cut random I/O, it caches frequently accessed graph data for a subset of high-access vertices in DRAM.

EdgeRAG [11] is a disk-based vector search framework that indexes only the clusters relevant to each query at runtime. At the indexing stage, it profiles per-cluster embedding latency (e.g., disk reads and temporary index construction). Using these metrics, it applies a cost-



(a) all-miniLM-L6-v2 [20] (b) gte-modernbert-base [21]

Fig. 3: Cluster access patterns per embedding model.



(a) WLRU (b) FIFO

Fig. 4: Time-series cache hit ratio, cumulative read size of the clusters, and end-to-end search latency under two cache replacement policies.

aware LRU policy that prioritizes clusters that are either frequently accessed or slow to load.

GPTCache [13] presents an application-level semantic cache store for entire LLM responses. GPTCache caches entire LLM responses to reuse prior results when semantically similar queries arrive. It checks the embedding similarity between the incoming request and cached responses, returning the responses if the embedding similarity exceeds the predefined threshold. Rather than caching full responses, our approach caches shared clusters, allowing partial indexes to be reused across semantically similar queries.

These works have aimed to improve cache efficiency at the individual query level, but they overlook the varying cluster access patterns among queries within a batch. Consequently, they fail to exploit inter-query cluster locality, leading to redundant cluster loads, poor cache hit rate, and ultimately lower search performance in batched query scenarios.

III. Motivations

A. Non-Uniform Cluster Access Patterns

To analyze cluster access patterns, we conducted a synthetic experiment to measure the similarity in the batched queries. For this, we used hotpotqa dataset [22] from BEIR benchmark suite [15], using its query set as input. hotpotqa is a large-scale multi-hop question answering dataset, widely used to evaluate the retrieval performance of LLM-based QA tasks, as each query requires combining evidence from multiple documents. Since our system processes queries in batches, a burst of arrivals leads to larger batch sizes, which reflects practical scenarios where multiple queries arrive concurrently. To enable query bursts occurring within short time windows, we generated query traffic following a Weibull distribution and selected FAISS [14] as the vector search engine.

We employ all-miniLM-L6-v2 and gte-modernbert-base to account for variation in embedding dimensionality and learning objectives, which are 384 and 768 dimensions, respectively. MiniLM is trained via knowledge distillation for efficient general-purpose semantic matching, whereas GTE is instruction-tuned to better align with retrieval-oriented tasks. This selection allows us to evaluate our system across models with different capacities, alignment strategies, and dimensionalities.

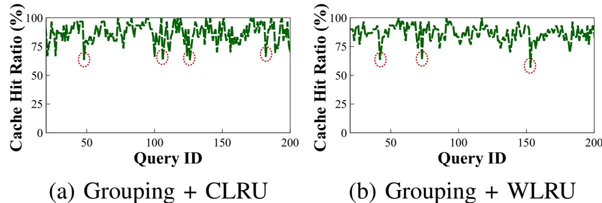
To quantify similarity, we use the Jaccard index [23], which measures the overlap between the set of all clusters accessed by queries in a batch. The Jaccard index computes the overlap between the sets of clusters per query and represents it as a ratio. We set the total number of clusters to 100 and the query fan-out to 10, ensuring that each query accessed 10 clusters.

Figure 3 depicts the cluster access patterns for each embedding model. The heatmaps show the degree of cluster overlap across a sequence of queries, where darker shades represent higher similarity. We randomly sampled 10 queries from the evaluation set to construct the heatmap, where both the X and Y axes represent query indices in the sampled sequence (i.e., query execution order). Notably, the similarity between adjacent queries is often low, while certain non-adjacent query pairs exhibit high overlap, indicating non-uniform cluster patterns.

For instance, Query 1 and 2 exhibit minimal overlap, whereas Query 1 and 10 share over 60% of their accessed clusters. Similarly, Figure 3 (b) shows that Query 6, 8, and 9 exhibit an overlap, each sharing more than 50% of their accessed clusters. These observations stem from the behavior of the embedding model. The embedding models capture both structural and semantic similarities when converting queries into vector representations, leading to diverse cluster distributions across the queries. Many queries follow similar syntactic patterns (e.g., ”What year did Einstein win the Nobel Prize?” or ”What year was Google founded?”). Since embedding models learn these recurring structures, they often map structurally similar queries to nearby regions in the vector space, even when their topics differ. *This finding implies that reordering batch queries based on dynamic cluster access patterns can significantly enhance cache hit rate.*

B. Analysis of Replacement Effects on Cluster Cache

Cache replacement policies ignoring the non-uniform cluster patterns fall short of maximizing cache efficiency. To verify this, we applied two cache replacement policies in the same scenario. One is WLRU that retains only the top 10 most frequently accessed clusters within each



(a) Grouping + CLRU (b) Grouping + WLRU

Fig. 5: Time-series cache hit ratio trend applying query grouping and prefetch schemes. Red dotted circles indicate cache hit ratio drop by query group transition.

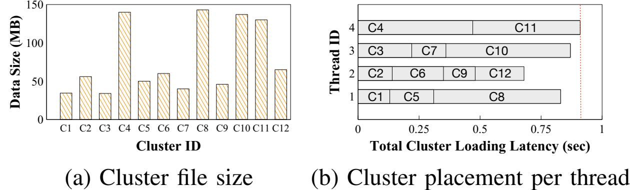
1-minute window. The other is FIFO, which evicts the cluster that has remained in the cache the longest.

Figure 4 presents cumulative read size of the clusters, and search latency per cache replacement policy based on the cache hit ratio. Across all policies, we observe distinct performance characteristics.

As shown in Figure 4 (a), WLRU exhibits significant performance degradation in certain scenarios. For example, at Query 421, the cache hit rate drops to 20%. The query incurs a latency spike of 2.2 seconds while reading 1.8 GB of cluster files from storage, approximately 9 times higher than when the hit rate is 80%. This demonstrates the window boundary problem: clusters just outside the current window are evicted, even if they remain highly relevant to upcoming queries. In addition, FIFO policy shows higher cache hit rate fluctuation. Cache hit ratios frequently drop into about 60%, and the corresponding latencies exhibit more frequent and pronounced spikes, occasionally approaching 2 seconds. The results indicate that the existing caching policy fails to capture access patterns across queries, leading to unstable and highly fluctuated cache hit rates.

C. Prefetch Invisibility

While grouping queries based on similar cluster access patterns appears to be a logical approach to enhance cache efficiency, it does not always yield higher cache hit rates. To investigate the cause of inconsistent cache hit rates despite query grouping, we performed an experiment where queries were grouped based on Jaccard similarity. Details of our query grouping are described in Section IV-B. Additionally, we extend both CLRU and WLRU with a prefetching scheme. For example, CLRU prefetches clusters with the highest loading costs every minute, while WLRU prefetches the most frequently accessed clusters, each with a fixed prefetching degree of 20. Despite these efforts to prefetch clusters, we consistently observed noticeable drops in cache hit rates at group boundaries, as illustrated in Figure 5. This sudden cache miss occurs because the vector database lacks visibility into query group boundaries, making it unable to prefetch clusters for the next group. We refer to this limitation as *prefetch invisibility*, which restricts the effectiveness of grouping strategies.



(a) Cluster file size (b) Cluster placement per thread

Fig. 6: Cluster size and thread-level loading latency.

D. Imbalanced Cluster Loading

To load missing clusters from storage quickly, the system employs multiple cluster loaders that perform parallel disk access. However, our analysis of FAISS, a widely-used vector database, reveals that it assigns cluster loading tasks to worker threads using round-robin scheduling policy, without accounting for the differences in cluster file sizes. Assigning cluster loading tasks by cluster ID, without accounting for differences in file sizes, can cause load imbalance across threads. This scheduling causes load imbalance when cluster sizes vary. To illustrate this problem, we sorted all 100 clusters by size and selected the top 4 largest and bottom 8 smallest clusters. Figure 6 (a) shows file size distribution across 12 clusters. For instance, the largest cluster C11 exceeds 140 MB, while C2 or C9 are under 40 MB. Figure 6 (b) visualizes how these clusters are assigned to threads under round-robin scheduling. Consequently, Thread 4, assigned the large cluster C11, exhibits a prolonged loading time compared to others. While most threads complete their tasks within 0.5 to 0.75 seconds, the entire loading process is stalled until Thread 4 finishes at 0.91 seconds. This straggler effect nullifies the benefits of parallelism, as the system must wait for the slowest thread to complete. This observation indicates that even with well-formed query groups and effective prefetching, performance gains can be significantly undermined if cluster loading is not carefully scheduled to balance workload across threads.

IV. Design

In this section, we begin by presenting an overview of CALL. We then provide a detailed explanation of how query grouping is systematically integrated into disk-based vector search to achieve scalable and low-latency when serving vector indexes that exceed host memory capacity. Additionally, we present a high-level overview of how CALL operates.

A. Overview of CALL

Figure 7 presents an overview of CALL, highlighting its three core modules – context-aware grouping module, group-aware prefetch module, and latency-aware cluster load module. CALL is performed after the indexing phase is completed. As described in Section II-A, the embedding model transforms documents into multi-dimensional vectors, which are partitioned into clusters and stored

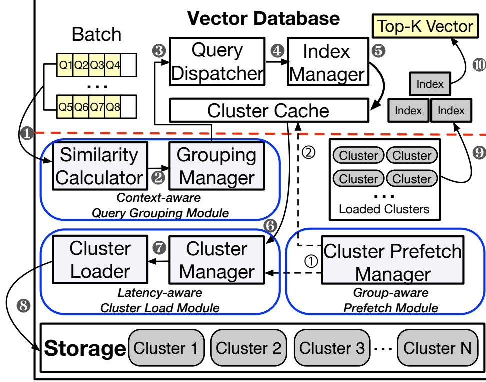


Fig. 7: An overview of CALL. It consists of three modules, outlined by blue lines. Black arrows indicate synchronous paths, and dotted arrows denote asynchronous ones. Order is marked by numbered circles.

as separate files. After completing the indexing phase, CALL processes batched queries using three modules. **Context-aware Grouping Module** receives 8 queries (e.g., Q1 $\sim$ Q8) as a batch and probes first level index to identify the set of clusters for each query ❶. Then, *similarity calculator* computes pairwise cluster similarity scores and groups queries with overlapping cluster access patterns ❷. The queries in the batch are reordered based on similarity scores and forwarded to vector database ❸. The reordering of queries within the batch is described in Figure 8. Upon receiving the queries, query dispatcher sends them to *index manager* ❹. It first attempts to load required clusters from *cluster cache* ❺. For missing clusters, **Latency-aware Cluster Load Module** is invoked to fetch candidate clusters from storage ❻. Subsequently, *cluster manager* provides metadata including cluster size, storage path to *cluster loader* ❼, which then performs I/O to read the corresponding files from the storage ❽. Hereafter, partial indexes are constructed using both cached and clusters from the storage ❾. Finally, top-k search is performed across the partial indexes, after which results are sorted and the k-nearest vectors are returned ❿. **Group-aware Prefetch Module** monitors the execution order within each query group. Upon completion of the last query in the current group, *cluster prefetch manager* proactively fetches clusters required by the first query in the subsequent group ①. Then it loads the clusters into the cluster cache ②.

B. Context-aware Grouping Module

To determine cluster similarity across the queries, we adopted Jaccard index. Cluster similarity calculator computes the similarity with the proportion of the size of the intersection and the union of two sets. For example, a query stream is defined as $Q = \{q_1, q_2, \ldots, q_n\}$, where each query q_i has a set of clusters $C(q_i)$. The Jaccard similarity is then computed using their respective cluster sets, as shown in Equation 1.

$$J(q_i, q_j) = \frac{|C(q_i) \cap C(q_j)|}{|C(q_i) \cup C(q_j)|}, C(q_j) = \{c_1, c_2, \ldots, c_p\} \tag{1}$$

$$G_k = \{q_i \in Q \mid J(q_i, q_j) \geq \theta, \forall q_j \in G_k\} \tag{2}$$

Based on the computed similarity scores, we applied agglomerative clustering, a hierarchical clustering method that groups similar items based on pairwise distances [24] with a predefined Jaccard distance threshold θ to form query groups using Equation 2. If the similarity score is within the distance threshold θ, the query is assigned to the corresponding group. Otherwise, a new query group is created, and the query is assigned accordingly. By applying the threshold-based similarity scoring to all queries, group manager constructs a set of query groups $G = \{G_1, G_2, \ldots, G_n\}$. CALL performs grouping once for each incoming batch of queries. While the current batch is being executed, newly arriving queries are accumulated and processed as the next batch.

Due to variability in user activity and external events, query arrival rates often exhibit sudden bursts. Hence, it is critical not only to improve cache efficiency via query grouping but also to ensure that the grouping process scales well with traffic and incurs minimal overhead.

Naive Jaccard index computation compare similarities between cluster sets $C(q_i)$ and $C(q_j)$ for all query pairs, where each cluster set is stored as a hash table. Although hash tables offer average $O(1)$ lookup time, they rely heavily on key comparisons and conditional branching to verify an overlap of clusters. During Jaccard similarity computation, repeated set membership checks (e.g., if cluster in set) introduce numerous conditional branches. Since the access patterns across cluster sets are irregular and input-dependent, these branches are hard to predict, resulting in frequent branch mispredictions. To avoid the branch-heavy control flow, we introduce a vectorized Jaccard computation method.

$$\mathbf{v}_i = [b_1, b_2, \ldots, b_K], \quad b_k = \begin{cases} 1 & \text{if } c_k \in C(q_i) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$\mathbf{I}_{i,j} = \mathbf{v}_i \cdot \mathbf{v}_j^\top, \quad |\mathbf{v}_i \cup \mathbf{v}_j| = \|\mathbf{v}_i\|_1 + \|\mathbf{v}_j\|_1 - \mathbf{I}_{i,j} \tag{4}$$

$$J(q_i, q_j) = \frac{\mathbf{I}_{i,j}}{|\mathbf{v}_i \cup \mathbf{v}_j|} \tag{5}$$

For Q_i, its vectorized cluster set is shown as Equation 3. This maps the entire cluster space into a contiguous memory block. Through a vectorized data structure, pairwise intersections and union cardinalities can be computed using matrix-based bitwise operations as shown in Equation 4. This eliminates conditional branching by replacing set membership checks with SIMD-friendly instructions. As a result, the Jaccard similarity can be

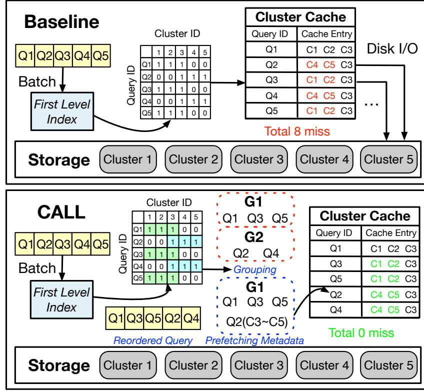


Fig. 8: High-level scenarios of disk-based vector search for the baseline and CALL. The number of total cache entries is 3. Initially, five queries are sent as a batch.

computed efficiently without control-flow divergence, as shown in Equation 5. Its branch-less formulation significantly accelerates query grouping.

C. Group-aware Prefetch Module

Query grouping is performed once per batch, and the assigned groups remain fixed throughout processing. However, despite this grouping, the query dispatcher cannot detect where the group boundaries lie within a batch. This limitation can delay cluster loading during group transitions, leading to cache misses. To overcome this limitation, we introduce a lightweight prefetching metadata P, defined in Equation 6. Here, FQ denotes the first query of the next group G_{i+1}, while $FQSET$ represents the set of clusters of FQ. This metadata is attached to each group during the grouping phase and forwarded to the query dispatcher module.

$$P = \{\text{FQ} : Q_{i+1}(q_f),\ \text{FQSET} : C(Q_{i+1}(q_f))\} \tag{6}$$

Figure 8 illustrates the vector search steps in baseline and CALL. We assume that cluster cache loads cluster files of first query Q1. Since the baseline does not account for cluster similarities across the queries, it searches them in sequential order. Before building the vector index, it checks the cache, but because each query accesses different clusters, a sequence of cache misses occurs. This results in eight consecutive cache misses, each triggering disk I/O during the search.

In contrast, CALL performs grouping on batched queries and generates metadata that encodes group information, including the cluster IDs required by the first query of the subsequent group. Based on the grouping, the queries within the batch are reordered to maximize cache hit ratios, resulting in consecutive cache hits. Specifically, CALL asynchronously preloads cluster files (e.g., C4, C5 for Q2) immediately after completing the vector search for Q5, using the prefetching metadata.

Algorithm 1 Load-Weighted Greedy Packing Algorithm

Input : $M = \{m_1, m_2, ..., m_N\}$, Size($c_i$), thread count T
Output : Load order $L = [c_{i_1}, c_{i_2}, ..., c_{i_N}]$

1: **procedure** LoadWeightedGreedyPacking
2: Sort M in descending order by Size(m_i)
3: Initialize empty thread groups $G_1, \dots, G_{\lceil N/T \rceil}$
4: **for** each cluster m in sorted M **do**
5: Find the earliest thread group G_j such that $|G_j| < T$
6: Assign m to thread group G_j
7: **end for**
8: Concatenate $L \leftarrow G_1 \| G_2 \| \dots \| G_{\lceil N/T \rceil}$
9: **return** L
10: **end procedure**

D. Latency-aware Cluster Load Module

In disk-based vector databases, the size of each cluster file varies significantly due to non-uniform vectors, resulting in uneven vector densities across clusters. This imbalance becomes problematic when loading multiple clusters in parallel. A commonly used scheduling method, such as the round-robin strategy adopted by FAISS, evenly assigns cluster files to threads based on cluster count. However, this approach can lead to severe load imbalance because a thread processing a large cluster file becomes a straggler, dominating the total loading time and reducing parallelism efficiency. To address this straggler problem, we introduce a load-weighted greedy packing algorithm that prioritizes the assignment of larger cluster files to earlier thread groups. As shown in Equation 7, M denotes the set of missing clusters, where each cluster m_i has an associated size s_i.

$$M = \{m_1, m_2, \dots, m_N\}, s_i = \text{Size}(m_i), L_j = \sum_{m_i \in G_j} s_i \tag{7}$$

Then, the total load L_j of thread group G_j is computed as the sum of cluster sizes assigned to that group. Our goal is to balance L_j across all groups to minimize total cluster loading latency and avoid stragglers. Algorithm 1 demonstrates how to balance the total load L_j.

V. Evaluation

A. Experimental Setup

Implementation. We implemented CALL based on FAISS version 1.10.0 as vector search engine. To enable disk-based ANN search, we employed IVF indexing algorithm provided in FAISS library. During all experiments, we set the total number of clusters to 100, nprobe to 30, and the number of threads to 8. The cluster cache was configured to hold up to 50 entries, and the Jaccard distance threshold θ was set to 0.3. A detailed analysis of θ's impact on search latency is presented in Section V-E. This means that each query creates a partial index using 30 cluster files. Given that the total number of cluster cache entries is 50, up to 50 clusters can reside in the cache. Upon a cache miss,

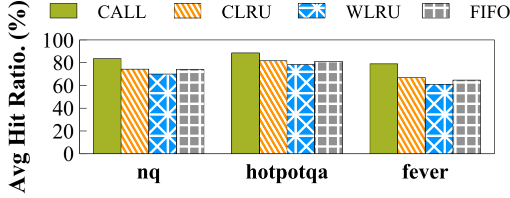


Fig. 9: Average cache hit ratio across datasets under three cache replacement policies.

the cluster cache evicts the least recently used entries first, removing as many entries as the number of missing clusters. Each experiment included a 1-minute warm-up phase to allow the cache to stabilize. Our code is available at *https://github.com/akssus12/CALL*.

Platform. Our experiments were conducted on a server equipped with Intel(R) i7-8700K 3.70GHz CPU, 16 GB DDR4 DRAM, and a 256GB Samsung 960 NVMe SSD.

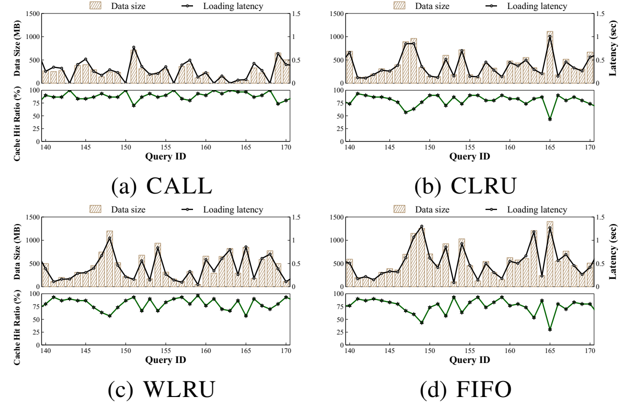
TABLE I: Details of evaluated datasets.

Dataset	Records	Embedding Size	Peak Memory Usage	Fit in Memory
nq [25]	2.68 M	8.3 GB	14 GB	O
hotpotqa [22]	5.42 M	15.4 GB	25.5 GB	X
fever [26]	5.23 M	18.5 GB	26.4 GB	X

Datasets. To construct the vector index, we used three public datasets – nq [25], hotpotqa [22], and fever [26] from BEIR benchmark [15]. Details of each dataset are provided in Table I. Despite the total embedding size being smaller than system memory capacity (i.e., nq and hotpotqa), we observed peak memory usage during index construction when loading multiple cluster files concurrently. This is because each cluster file is first loaded into the kernel page cache and subsequently copied into user space memory. As a result, both the kernel and user space temporarily hold redundant copies of the same data, doubling memory consumption, thereby exceeding available memory capacity. Both the corpus and query sets from the same dataset are converted into vector embeddings using all-miniLM-L6-v2 model, which has 384 dimensional dense vector space.

Traffic. In real-world scenarios, users often issue multiple queries concurrently, leading to bursty traffic patterns in vector database systems. To emulate such behavior, we generate a scalable and bursty query workload using a traffic generator modeled by the Weibull distribution. During the normal phase, the system issues queries at a steady rate of 100 queries per second. At each time interval, there is a 10% probability of entering a burst phase, during which the query rate surges up to three times the normal rate to simulate sudden traffic surges.

Evaluation Metrics and Comparison Targets. We used four evaluation metrics: cache hit ratio, search latency, grouping time, and cluster loading latency. The search latency is measured from exploring the cluster cache to vector retrieval. To evaluate the effectiveness of the proposed design, we implemented three cache replacement schemes and evaluated them with CALL.



(a) CALL (b) CLRU (c) WLRU (d) FIFO

Fig. 10: Time-series cache hit ratio, cumulative read size of the clusters, and cluster loading latency under three cache replacement policies.

- CLRU prioritizes cache retention based on two metrics that considers both the access frequency of clusters and their associated loading latency [11].
- WLRU retains only the top 10 most frequently accessed clusters within each 1-minute window.
- FIFO evicts the cluster that has resided in the cluster cache the longest.

B. Overall Performance

Cache Hit. Figure 9 shows that CALL consistently achieves higher average cache hit ratios compared to all baselines across the evaluated datasets. Among them, fever dataset exhibits the most significant gap. CALL reaches a 92% hit ratio, while WLRU and FIFO fall to 75% and 60%, respectively.

To further analyze the enhancement of cache hit ratio, we fixed the dataset fever and measured time-series cache hit ratio, cumulative read size of clusters, and the cluster loading latency. As shown in Figure 10, CALL maintains a stable cache hit ratio throughout the query stream. As a result, the amount of data read from disk remains low, and the corresponding cluster loading latency consistently stays below 0.5 seconds. In contrast, WLRU and FIFO show sharp drops in cache hit ratio, triggering frequent disk reads and causing loading latency to spike above 1.5 seconds. This improvement is mainly due to query grouping. By reordering queries with similar cluster access patterns, clusters are reused efficiently within each group, improving cache hit ratio. Additionally, group-aware prefetching helps prevent cache drops during group transitions by loading the next group's required clusters in advance. Even with fewer clusters loaded from disk, CALL avoids stragglers by balancing cluster file sizes across threads, further reducing loading latency.

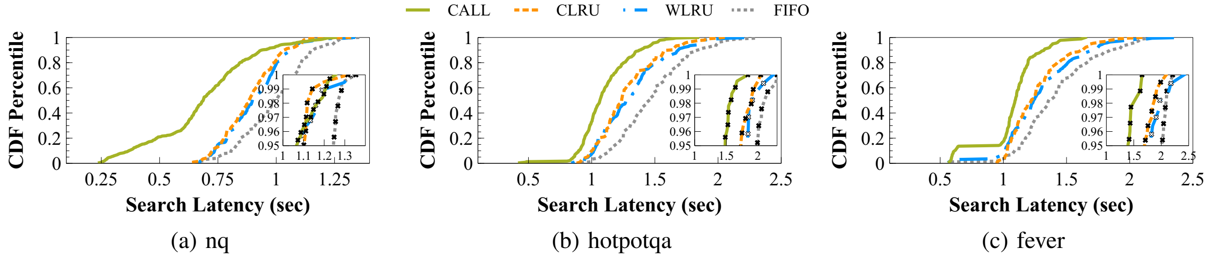


Fig. 11: Tail search latency across datasets under three cache replacement policies.

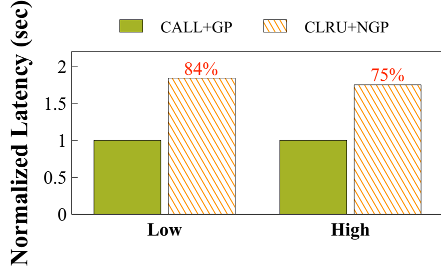


Fig. 12: Normalized end-to-end search latency.

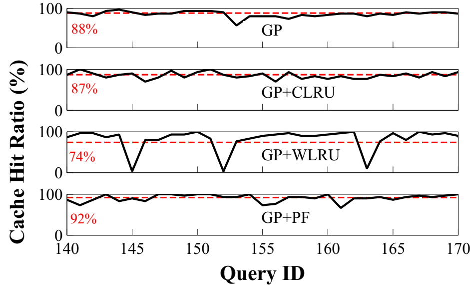


Fig. 13: Effectiveness of group-aware prefetch. Red dotted line indicates an average cache hit ratio. GP denotes context-aware grouping module, and PF denotes group-aware prefetch module.

Tail Latency. Figure 11 compares the cumulative distribution functions (CDF) of end-to-end search latency across the three datasets. Similarly, tail latency remains low across all datasets and nearly all time intervals. In fever dataset, as shown in Figure 11 (c), CALL reduces 99th percentile latency by up to 33%, 32%, and 21% compared to WLRU, FIFO, and CLRU, respectively. In Figure 11 (a), CLRU slightly reduces search latency from the 95th to 98th percentile. In addition to CLRU, both WLRU and FIFO exhibit similar performance at the 96th percentile. This is because nq dataset has a relatively small embedding size that fits comfortably in memory, as shown in Table I. In such settings, cluster reuse is high, and the working set of active clusters remains stable over time. However, these fail to handle extreme tail cases, such as the 100th percentile. In the following sections, we analyze the contribution of each module in CALL to reducing search latency.

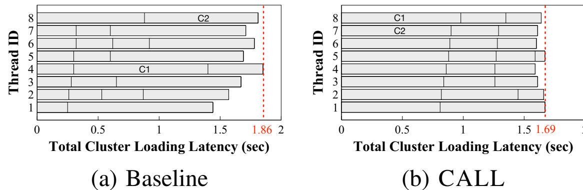
TABLE II: Normalized CPU branch miss rate. CALL buffers queries for 3 seconds under low traffic and 10 seconds under high traffic conditions.

Number of Queries	Grouping Time (sec)		Branch Miss Rate	
	Bitmap	Hash Table	Bitmap	Hash Table
1000(Low)	0.58	3.57	1	1.27
1500	0.85	6.46	1	1.32
2000	1.07	11.55	1	1.38
2500(High)	1.47	18.81	1	1.47

C. Module Effectiveness

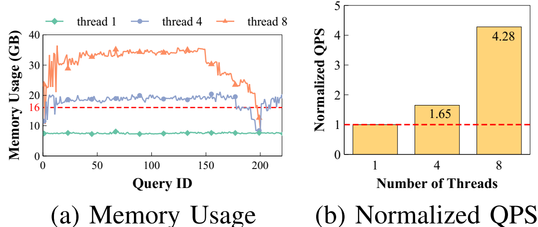
Effectiveness of Context-aware Grouping. To ensure that query grouping introduces minimal overhead, we compared the runtime and CPU efficiency of bitmap-based and hash table-based Jaccard computation. As shown in Table II, as the number of queries increases from 1000 to 2500, the grouping time of the bitmap-based method increases slightly, whereas the hash table-based method grows drastically up to 18.81 seconds. This efficiency stems from the substantially lower branch miss rate achieved by the bitmap-based approach, reducing miss rate by up to 47%. As a result, CALL performs query grouping with negligible delay. Figure 12 further confirms this by showing that CALL completes full query processing up to 84% faster than baseline, even under high traffic. This improvement stems not only from the scalability of our grouping method, but also from its ability to significantly increase cache hit rates, which directly translates to reduced disk I/O.

Effectiveness of Group-aware Prefetch. Figure 13 highlights the effectiveness of group-aware prefetching. Even with grouping alone, CALL achieves an average cache hit rate of 88%, though cache drops still occur at group boundaries. By sending prefetching metadata with batched queries, our prefetch module ensures the first query of every group hits the cache, achieving the 100% cache hit and raising the overall hit rate to 92%. In contrast, CLRU and WLRU, lacking group awareness, rely solely on frequency-based metrics and fail to prefetch effectively. Notably, WLRU occasionally misses all clusters, indicating a complete prefetch failure. This issue arises because WLRU preloads clusters that



(a) Baseline (b) CALL

Fig. 14: Total cluster loading latency across multiple worker threads under varying cluster size distributions. Red dotted line indicates longest execution time.



(a) Memory Usage (b) Normalized QPS

Fig. 15: Impact of the number of search threads on memory usage and query throughput. Red dotted line in (a) indicates total system memory capacity.

were frequently accessed within a fixed time window (e.g., 1 minute), without considering the actual cluster needs of the incoming query group. Due to this coarse-grained frequency tracking and lack of group-level visibility, WLRU often prefetches irrelevant clusters, especially at group boundaries.

Effectiveness of Latency-aware Cluster Load. Figure 14 illustrates the execution timeline of each thread during parallel cluster loading. Under the baseline, cluster loading suffers from load imbalance, where thread 4 becomes a straggler due to large cluster C1, leading to a total latency of 1.86 seconds. In contrast, CALL assigns large clusters across threads using a greedy packing strategy, achieving balanced execution and reducing total cluster loading latency to 1.69 seconds, a 10% improvement over the baseline.

D. Memory Usage Analysis

Figure 15 illustrates trade-off between query throughput and memory usage as the number of search threads increases. While adding threads boosts throughput, which is 1.65 times higher with 4 threads and 4.28 times higher with 8, leading to excessive memory consumption due to redundant cluster loading. This results in out-of-memory failures when usage exceeds the 16 GB limit. In contrast, CALL maintains a compact footprint under 8GB by maximizing cache reuse through single-threaded, context-aware grouping. This design achieves low-latency vector search without memory shortage, making it well-suited for resource-constrained environments.

TABLE III: 95th, 99th, mean percentile, average latency and cache hit ratio for each Jaccard distance threshold θ, represented in milliseconds.

Threshold	Mean	95th	99th	Average	Hit Ratio
0.1	0.795	1.199	1.411	0.834	85.8%
0.3	0.756	1.021	1.204	0.772	87.4%
0.5	0.913	1.253	1.438	0.951	77.4%
0.7	0.777	1.078	1.292	0.805	84.2%
0.9	0.793	1.062	1.253	0.814	84.4%

TABLE IV: Time taken for each phase of CALL, represented in milliseconds. Each grey row indicates an additional module of CALL.

Module	Q1	Q2	Q3	Q4
Cluster load balance	0.001	0.002	0.002	0.001
Cluster prefetch	0.097	0.149	0.261	0.151
Vector search	0.498	0.466	0.53	0.428
Total duration	0.596	0.617	0.793	0.58

E. Sensitivity of Jaccard Threshold

Jaccard distance threshold θ is a parameter used to measure how much clusters overlap between queries. The threshold closer to 1 indicates a higher degree of overlap, meaning the clusters for two queries are more similar. We used hotpotqa dataset and followed default experimental setup described in Section V-A. As shown in Table III, when the threshold is set to 0.3, all evaluation metrics achieve their best performance. This is attributed to the well-grouped queries under a moderate threshold (e.g., 0.3), which reduces group switching frequency compared to both lower and higher thresholds.

F. Overhead Analysis

CALL introduces two key modules: a group-aware prefetch module and a latency-aware cluster load module. The group-aware prefetch module asynchronously preloads missing cluster files for the first query in the next group. Meanwhile, the latency-aware cluster load module balances cluster loads for distributing them evenly across available worker threads. To analyze the latency overhead incurred by these additional modules, we fixed the dataset to hotpotqa and randomly selected four queries that triggered cluster prefetch.

Table IV presents the measured execution time of each module involved in query processing. Overall, the latency-aware cluster module accounts for less than 1% of the total duration. On the other hand, the cluster prefetch module accounts for an average of 20% of the total duration. CALL initially blocks the vector search while waiting for all prefetched candidate clusters. This means that as the number of missing clusters increases, the search process stalls entirely until all clusters are available,

which increases the search latency. In future work, we plan to revise the cache lookup behavior so that CALL proceeds immediately with the clusters already cached (e.g., clusters that loaded into the cache through prefetch module) and reads the missing ones synchronously. This is expected to improve the responsiveness by allowing vector search to start earlier, although it may slightly reduce the cache hit rate.

VI. Conclusion and Future Work

Disk-based vector search systems suffer from fundamental limitations that arise from ignoring contextual relationships between queries, leading to suboptimal cache behavior and significant latency penalties. Through empirical analysis, we reveal that non-uniform cluster access patterns, cache misses during group transitions, and thread-level load imbalance are the key contributors to increasing search latency. To address this issue, the proposed CALL in this study effectively schedules the queries within a batch by reordering them to maximize cluster cache utilization. Our extensive evaluation reveals that CALL significantly reduces both average search latency and tail latency compared to existing approaches across various real-world workloads.

Our approach, although designed for disk-based vector databases, has potential to extend beyond memory-constrained environments. By leveraging its query grouping and prefetching strategies, CALL can be applied to diverse cases such as disaggregated memory platform [27], [28] and edge-cloud infrastructure [29]. We hope that CALL will serve as a solution for providing low-latency retrieval in disk-based vector databases.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2024-00453929) (RS-2024-00416666) and Development of National Mental Health Trend Monitoring and Management Platform by the Korean government (MSIT) (RS-2022-KH131189).

References

[1] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025.

[2] S. Rosenthal, A. Sil, R. Florian, and S. Roukos, "Clapnq: Cohesive long-form answers from passages in natural questions for rag systems," *Transactions of the Association for Computational Linguistics*, vol. 13, pp. 53–72, 2025.

[3] Z. Wei, D. Huang, J. Zhang, C. Song, S. Zhang, J. Zhang, Z. Li, K. Jiang, R. Li, and Q. Duan, "Garag: A general adaptive question-answering system based on rag," in *Proceedings of the 2024 International Conference on Cloud Computing and Big Data*, pp. 442–447, 2024.

[4] D. Chen, Y. Liu, M. Zhou, Y. Zhao, H. Wang, S. Wang, X. Chen, T. F. Bissyandé, J. Klein, and L. Li, "Llm for mobile: An initial roadmap," *ACM Transactions on Software Engineering and Methodology*, 2024.

[5] J. Wu, J. Zhu, Y. Qi, J. Chen, M. Xu, F. Menolascina, and V. Grau, "Medical graph rag: Towards safe medical large language model via graph retrieval-augmented generation," 2024.

[6] M. Sankaradas, R. K. Rajendran, and S. T. Chakradhar, "Streamingrag: Real-time contextual retrieval and generation framework," *arXiv:2501.14101*, 2025.

[7] A. T. Neumann, Y. Yin, S. Sowe, S. Decker, and M. Jarke, "An llm-driven chatbot in higher education for databases and information systems," *IEEE Transactions on Education*, 2024.

[8] Y. Ang, Y. Bao, Q. Huang, A. K. Tung, and Z. Huang, "Tsgassist: An interactive assistant harnessing llms and rag for time series generation recommendations and benchmarking," *Proceedings of the VLDB Endowment*, vol. 17, no. 12, pp. 4309–4312, 2024.

[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] 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, Mar. 2024.

[11] K. Seemakhupt, S. Liu, and S. Khan, "Edgerag: Online-indexed rag for edge devices," *arXiv:2412.21023*, 2024.

[12] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, *et al.*, "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 2614–2627, 2021.

[13] F. Bang, "GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings," in *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pp. 212–218, Dec. 2023.

[14] Faiss, "A library for efficient similarity search." https://github.com/facebookresearch/faiss, 2025. Accessed: 2025-07-02.

[15] E. Kamalloo, N. Thakur, C. Lassance, X. Ma, J.-H. Yang, and J. Lin, "Resources for brewing beir: Reproducible reference models and statistical analyses," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, p. 1431–1440, 2024.

[16] H. J. Romain Tavenard, Laurent Amsaleg, "Balancing clusters to reduce response time variability in large scale image search," 2010.

[17] Qdrant, "High performance vector search at scale." https://github.com/qdrant/qdrant, 2025. Accessed: 2025-07-02.

[18] S. Na, L. Xumin, and G. Yong, "Research on k-means clustering algorithm: An improved k-means clustering algorithm," in *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*, pp. 63–67, 2010.

[19] J. Sivic and A. Zisserman, "Video Google: A text retrieval approach to object matching in videos," in *IEEE International Conference on Computer Vision*, vol. 2, pp. 1470–1477, 2003.

[20] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," *Advances in neural information processing systems*, vol. 33, pp. 5776–5788, 2020.

[21] Z. Li, X. Zhang, Y. Zhang, D. Long, P. Xie, and M. Zhang, "Towards general text embeddings with multi-stage contrastive learning," *arXiv:2308.03281*, 2023.

[22] Hotpotqa, "Hotpotqa: A dataset for diverse, explainable multi-hop question answering." https://github.com/hotpotqa/hotpot, 2025. Accessed: 2025-07-02.

[23] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," in *Proceedings of the international multiconference of engineers and computer scientists*, vol. 1, pp. 380–384, 2013.

[24] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *arXiv:1109.2378*, 2011.

[25] NQ, "Natural questions (nq) contains real user questions issued to google search, and answers found from wikipedia by annotators.." https://github.com/google-research-datasets/natural-questions, 2025. Accessed: 2025-07-02.

[26] J. Thorne, A. Vlachos, C. Christodoulopoulos, and A. Mittal, "FEVER: a large-scale dataset for fact extraction and VERification," in *NAACL-HLT*, 2018.

[27] Y. Liu, F. Fang, and C. Qian, "Efficient vector search on disaggregated memory with d-hnsw," *arXiv:2505.11783*, 2025.

[28] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "{CXL-ANNS}:{Software-Hardware} collaborative memory disaggregation and computation for {Billion-Scale} approximate nearest neighbor search," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 585–600, 2023.

[29] Z. Yao, Z. Tang, J. Lou, P. Shen, and W. Jia, "Velo: A vector database-assisted cloud-edge collaborative llm qos optimization framework," in *2024 IEEE International Conference on Web Services (ICWS)*, pp. 865–876, 2024.