CaGR-RAG: Context-aware Query Grouping for Disk-based Vector Search in RAG Systems

Yeonwoo Jeong[1], Kyuli Park[1], Hyunji Cho[1] and Sungyong Park[1]

Department of Computer Science and Engineering, Sogang University

Abstract. Modern 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 disk-based vector search systems, particularly in Retrieval Augmented Generation (RAG) framework. 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 reduces cache efficiency and increases search latency due to excessive disk I/O. To address this, we introduce CaGR-RAG, a context-aware query grouping mechanism that organizes queries based on shared cluster access patterns. Additionally, it incorporates opportunistic cluster prefetching to minimize cache misses during transitions between query groups, further optimizing retrieval performance. Experimental results show that CaGR-RAG reduces 99th percentile tail latency by up to 51.55% while consistently maintaining a higher cache hit ratio than the baseline.

Keywords: Retrieval Augmented Generation · Disk-based Vector Search

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in tasks such as image classification, reasoning, and text generation. However, despite their strengths, LLMs often produce incorrect answers [7] when encountering information that is outside their training data. To mitigate this issue, RAG systems, which reference external vector databases before generating responses, have been widely adopted to complement LLMs.

RAG systems typically load entire vector indexes into memory for fast retrieval. While in-memory vector search offers high-speed retrieval, the size of vector indexes often exceeds available memory capacity [3, 8]. To overcome this limitation, disk-based vector search has emerged as a viable solution [3, 8, 12, 18] in RAG systems. By leveraging high-speed storage (e.g., NVMe SSDs), this approach efficiently manages large-scale vector indexes while balancing retrieval performance and cost.

Disk-based vector search in RAG systems utilizes Approximate Nearest Neighbor (ANN) search, partitioning the vector index into multiple clusters stored on secondary storage. Instead of loading the entire index, the vector database selectively loads only the required clusters during search operations, optimizing both memory usage and retrieval efficiency. Although only the necessary clusters are

loaded into memory, disk I/O remains a major bottleneck in query processing. Several techniques have been proposed to minimize disk I/O during vector searches [2, 8, 12], mainly by improving cache hit rates for frequently accessed clusters.

However, these approaches have a fundamental limitation: they fail to account for the broader query context, limiting their overall effectiveness. Modern embedding models encode text into dense vector representations that capture both semantic meaning and syntactic structures (e.g., question formats, specific phrasings). Consequently, queries with similar structural patterns, even if their content varies, often map to the same regions in vector space. This observation suggests that queries with similar structural patterns tend to access the same clusters, and by grouping them accordingly, cache replacements can be minimized, improving efficiency.

Our preliminary study using the RAG benchmark [15] reveals that queries exhibit non-uniform cluster access patterns across various embedding models. However, no prior research has explicitly explored performance optimization by leveraging query context in disk-based vector search within RAG systems. Existing disk-based vector search approaches process incoming queries sequentially, overlooking the inherent non-uniformity in query distributions. This lack of query grouping based on contextual similarities reduces data locality and limits cache utilization efficiency, ultimately leading to suboptimal retrieval performance.

To improve performance in disk-based vector search for RAG systems, we propose CaGR-RAG, a context-aware query grouping mechanism that leverages query context to enhance cache efficiency. The primary goal of this work is to develop a query reordering algorithm that carefully accounts for non-uniform cluster access patterns. As queries arrive at the vector search engine, CaGR-RAG extracts their associated cluster lists and calculates similarity scores between query pairs using the *Jaccard index*. Based on these scores, CaGR-RAG sorts the queries and dispatches them to the vector database in an optimized order.

However, simply grouping queries presents a challenge: transitioning between query groups may disrupt cluster locality, potentially increasing cache misses. To address this, we designed a data structure that maintains a list of queries for each group and tracks the clusters required by the first query in the next group, enabling proactive prefetching. By leveraging this structure, CaGR-RAG performs opportunistic prefetching of clusters when transitioning between query groups, further improving cache utilization. Our extensive evaluation has confirmed that CaGR-RAG outperforms baseline by up to 51.55% at 99th percentile tail latency while improving overall performance.

2 Background and Motivation

2.1 RAG System

The RAG system operates in two key stages: retrieval and generation. In the retrieval stage, the user query is converted into vector embeddings and passed to

Code 1 Disk based IVF index search program

```python
def Disk_Based_IVF_Index_Search(user_query, nprobe=5):
    qv = model.encode(user_query)     # 1.Encoding user query
    centroids = quantizer.search(qv) # 2.Search first-level index
    centroid_IDs = [1, 3, 5, 7, 9], ev = []
    for centroid in centroid_IDs:
        ev += load(centroid)          # 3.Load vectors
    index.add(ev)                     # 4.Merge vectors into index
    index.search(qv, top-k=n)         # 5.Perform top-k search
```

the vector database, which searches for the closest vectors using distance metrics like cosine similarity or Euclidean distance. Similarity scores are then calculated based on the retrieved vectors, and the most relevant documents are selected in response to the query. During the generation stage, these documents are combined with the original query to generate a detailed response. The retrieval stage is known to contribute up to 41% of the end-to-end latency in RAG systems [13]. In this work, we focus on optimizing the retrieval process to enhance the overall performance of the RAG system.

2.2 Disk-based ANN Search

Performing an exhaustive search that evaluates all vector embeddings ensures high accuracy but can be computationally expensive. To balance search efficiency and computational cost, ANN search is commonly employed.

This technique involves creating a structured index that partitions the vector search space. During the index-building phase, vector embeddings, derived from the raw dataset, are grouped into clusters. Each cluster is represented by its centroid - the high-dimensional point that minimizes the distance to all vectors within the cluster. Rather than searching the entire index, a user query is first mapped to the nearest centroids, significantly reducing the search space. While this reduces the computational load, the entire vector index still needs to be loaded into host memory. If the index's memory footprint exceeds the host memory capacity, the retrieval process may fail due to an out-of-memory error. In such cases, disk-based ANN search provides a viable alternative. Code 1 demonstrates a Python program that performs a search using a disk-based Inverted Vector File (IVF) index [14], a popular method for ANN search.

The disk-based IVF vector search follows a two-level search process. First, the vector database converts the user query into multi-dimensional vectors using a pre-trained embedding model ❶. Before searching within the selected vector space, the first-level index is scanned to identify the centroids to which the query vector is mapped ❷. After retrieving the relevant centroids, the process reads the vector embeddings associated with the selected centroid's cluster from the

disk ❸. The vector embeddings from the selected clusters are then merged into the temporary index ❹. Finally, the vector database performs a similarity search on the merged index ❺.

2.3 Related Work

To date, there has been limited research on enhancing the performance of disk-based vector search within RAG systems through query grouping. Most studies have concentrated on improving performance by refining cache replacement algorithms or optimizing cache utilization for individual queries [1, 2, 8, 12].
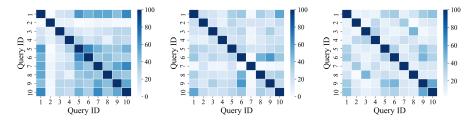
Embedding Vectors Caching. Subramanya et al. introduced DiskANN [8], a graph-based ANN search algorithm designed to handle large-scale vector datasets. DiskANN employs the Vamana algorithm, which stores only a portion of the graph in memory and loads additional edges from disk. To reduce disk accesses further, DiskANN caches frequently accessed graph components associated with a subset of vertices in DRAM.

Chen et al. developed SingleStore-V [2], which integrates vector search capabilities into a relational database, enabling seamless interoperability with traditional SQL operations. However, SingleStore-V caches the entire vector dataset and index in memory, lacking support for selective index loading.

Fu et al. proposed GPTCache [1], an open-source semantic cache for storing and retrieving LLM responses. GPTCache relies on traditional cache eviction policies like Least Recently Used (LRU) or First In, First Out (FIFO) to manage cache size. While GPTCache reduces redundant LLM calls and decreases response times, it assumes the entire vector index is loaded into memory, which does not address memory-constrained environments.

Seemakhupt et al. introduced EdgeRAG [12], which enhances the IVF index algorithm by pruning second-level embeddings to reduce memory footprint and generate them online during vector retrieval. During the index build phase, EdgeRAG profiles embedding generation latencies (e.g., reading latency from disk, adding vectors to the temporal index) for each cluster, storing those that exceed a predefined threshold on disk. Clusters with latencies below the threshold are computed in-flight. EdgeRAG employs cost-aware caching, prioritizing frequently accessed clusters and embeddings with long generation latencies.

Limitations of Prior Works. Despite significant advancements in prior work, approaches that ignore query context have fundamental limitations in optimizing performance for disk-based vector search in RAG systems. Embedding models capture both structural and semantic similarities when converting queries into vector representations, leading to diverse cluster distributions for incoming queries. For example, 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 tend to position similarly structured queries close together within the vector space, even if their topics and content differ. As a result, queries can exhibit varying cluster access patterns. However, existing caching strategies fail to account for this locality, leading to redundant cluster loads, poor cache utilization, and ultimately lower

(a) all-miniLM-L6-v2 [19] (b) gte-modernbert-base [10] (c) multilingual-e5-base [17]

Fig. 1: Cluster accessed pattern per an embedding model.

performance. This highlights the need for query grouping strategies that leverage query context and non-uniform access patterns to enhance performance.
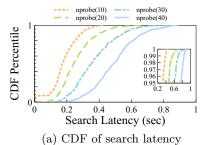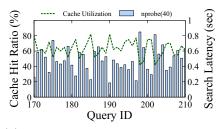
2.4 Motivation

In our preliminary experiments, we investigated the problems of disk-based vector search without query grouping. For the experiment, we constructed an IVF index using hotpotqa dataset [20] from the BEIR benchmark suite [15] and issued queries from the same dataset. We selected FAISS [4] as the vector search library and Apache Kafka [5] as the message queue to handle queries from multiple users. Additionally, we executed the baseline setup in our experimental environment, as outlined in Section 4.1.

Problem 1 - Non-uniform Cluster Access Pattern. Queries from various sources are transformed into multidimensional vector embeddings. Although the content of these queries varies significantly, their embeddings can still map to the same centroids. This occurs because embedding models do not solely capture the explicit topics of queries but instead learn underlying structural and semantic patterns. Additionally, variations in training datasets and neural network architectures influence how queries are distributed in vector space, resulting in non-uniform cluster access patterns.

To analyze cluster access patterns among queries, we conducted a synthetic experiment to measure the similarity between sequentially arriving queries. For this, we adopted three embedding models (all-miniLM-L6-v2, gte-modernbert-base and multilingual-e5-base), which have demonstrated competitive performance in text embedding and retrieval evaluations. We used Jaccard index to measure similarity scores between the queries. Additionally, we set the total number of clusters to 100 and configured nprobe to 10, ensuring that each query vector is searched across 10 clusters.

Figure 1 illustrates the cluster access patterns for each embedding model. Darker colors on the color bar indicate a higher number of shared clusters among queries. Overall, we observe lower similarity between adjacent queries and higher similarity between non-adjacent queries. This pattern is particularly prominent in Figure 1 (a). For instance, Queries 1 and 2 exhibit very low similarity, whereas

(a) CDF of search latency   (b) Cache utilization and search latency

Fig. 2: The cumulative distribution function of search latency per nprobe and the relation between cache hit ratio and search latency when nprobe is set to 40.

Queries 1 and 10 share more than 60% similarity. This observation suggests that grouping queries with similar cluster access patterns enhances cache efficiency.

Problem 2 - Suboptimal Cache Utilization. As explained in Section 2.2, ANN search consists of two-level phases. While both phases typically operate in memory, disk-based ANN search differs. Scenario of baseline is illustrated in Section 3.1. However, the baseline does not account for the non-uniform cluster access patterns of incoming queries and performs vector searches sequentially, leading to an inefficient cache utilization.

We measured vector search latency as the number of clusters increased. For this experiment, we set the total number of cache entries to 50. Figure 2 (a) presents the cumulative distribution function (CDF) of search latency for different nprobe values. As expected, long tail search latency is observed at the highest nprobe value. This occurs because each query accesses a larger number of clusters, and variations in cluster access patterns lead to frequent cache flushing. Consequently, multiple file reads are triggered, increasing read latency and prolonging overall search time. As shown in Figure 2 (b), search latency spikes when the cache hit ratio drops. At Query 198, the cache hit ratio is 42%, while search latency reaches 0.84 seconds. Given that the median latency is 0.48 seconds, this suggests that high cache miss rates contribute significantly to increased tail latency. These findings motivate us to group queries based on cluster access patterns to maximize cache utilization.

3   Design

3.1   Core Idea of CaGR-RAG

We introduce a query grouping scheme that accounts for non-uniform cluster access patterns, along with an opportunistic prefetching strategy. Figure 3 illustrates the vector search steps in both the baseline and CaGR-RAG. The baseline is approach without query grouping and prefetch scheme. In both approaches, each cluster containing vector embeddings is stored on storage at an index build
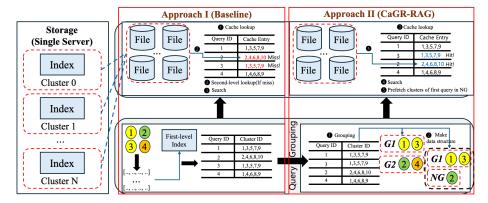
Fig. 3: Illustration of disk-based IVF search for the baseline and CaGR-RAG, respectively. In the illustration, we explain vector search steps of query 2 and 3. We assume that number of total cache entries is 5. Initially, four queries arrive.

phase. As explained in Section 2.2, selective clusters identified through first-level index lookup are retrieved from disk and merged into a temporary vector index.

Both approaches first transform queries into vector embeddings and search the first-level index to extract cluster IDs, starting from the centroids closest to the query vector. Assume baseline performs a similarity search for Query 2. It first scans the cache to retrieve clusters identified in the first level index lookup ❶. However, five cache misses occur because Query 1 has already placed the cache with its clusters. At this point, the baseline loads five clusters from disk, replacing the previous ones ❷. Once all clusters are loaded and merged into the temporary index, it begins the vector search ❸. If subsequent queries do not share highly overlapping clusters, this process repeats.

In contrast, CaGR-RAG groups incoming queries based on the similarity of their accessed clusters ❶. It also generates data structure for each group, including the query set and the first query of the next group ❷. CaGR-RAG sorts the queries with grouping and sends them with data structure to vector database. Since CaGR-RAG forms the query groups, Query 1 and 3 are sequentially searched while the clusters are retrieved without additional disk access ❸. Then the queries are rapidly searched ❹. Moreover, CaGR-RAG leverages data structure and knows which query will be searched following sequence. Using the information, CaGR-RAG triggers prefetching the clusters of following query immediately after the vector search, thereby maximizing cache utilization ❺. As a result, all clusters at Query 2 are loaded from the cache. Our query grouping and opportunistic prefetch scheme when each query group is switched minimizes cache miss, reducing an expensive disk I/O.

3.2 Context-aware Query Grouping Algorithm

CaGR-RAG has two modules: (i) query grouping module and (ii) opportunistic prefetch module. The details of each module are below.

Context-aware Query Grouping Algorithm. To determine cluster similarity per each query, we adopted Jaccard index. It 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 $Q = \{q_1, q_2, \ldots, q_n\}$. Moreover, Each query Q_i has a set of clusters $C(q_i)$. $C(q_i)$ can be represented as Equation 1. Using the query stream and a set of clusters, Jaccard similarity score is calculated as Equation 1.

$$C(q_j) = \{c_1, c_2, \ldots, c_p\} \tag{1}$$

$$J(q_i, q_j) = \frac{|C(q_i) \cap C(q_j)|}{|C(q_i) \cup C(q_j)|} \tag{2}$$

Based on computed Jaccard similarity score between query pairs, we used agglomerative clustering [11] with a predefined Jaccard distance threshold θ to form query groups using using Equation 3. As a result, we create query groups $G = \{G_1, G_2, \ldots, G_n\}$.

$$G_k = \{q_i \in Q \mid J(q_i, q_j) \geq \theta, \forall q_j \in G_k\} \tag{3}$$

In Algorithm 1, step 1 provides a pseudo-code that outlines a workflow of query grouping. For each query Q_i, we assume its associated cluster set $C(q_i)$ has been retrieved. The algorithm then determines whether the query belongs to an existing group. For each group G_i, it compares the current query with all queries in the group and calculates the overlap between cluster sets. If the similarity score exceeds the distance threshold θ, the query is assigned to that group. Otherwise, a new query group is created, and the query is assigned accordingly.

Opportunistic Prefetch Module. Although query grouping is made well, the cache effect might be vanished if query group is switched. For example, when a vector database receives multiple queries as arguments, it lacks prior knowledge of the clusters each query accesses and, consequently, is unaware of query grouping. To prefetch only clusters of next query following query group, we designed a data structure that maintains a list of queries per group, the clusters per each query, and the clusters of the first query in the next group. For this, we defined the first query of next group as $q_F(G_{i+1})$. Then, the clusters of $q_F(G_{i+1})$ is represented as Equation 4. Finally, the data structure can be defined as Equation 5. Based on the data structure, the vector database triggers an opportunistic prefetch clusters of $q_F(G_{i+1})$ and loads the cache, preventing from dropping cache hit due to group switching. The detailed sequences of opportunistic prefetch clusters are explained from step 2 to 4 in Algorithm 1.

$$C(q_F(G_{i+1})) = \{c_1, c_2, \ldots, c_k\} \tag{4}$$

$$D = \{(G_i, \{q_{i1}, \ldots, q_{im}\}, C(G_i), q_F(G_{i+1}), C(q_F(G_{i+1}))) \mid i = 1, \ldots, k\} \tag{5}$$

Algorithm 1 Context-aware Query Grouping and Opportunistic Prefetching

---

Input: Query set $Q = \{q_1, q_2, ..., q_n\}$, cluster set $C(q_i)$, similarity threshold θ, cache state S

Output: Query groups $G = \{G_1, G_2, ..., G_k\}$ and opportunistic prefetching

 1: **Initialize** empty query groups $G \leftarrow \emptyset$
 2:
 3: /* **Step 1: Query Group Representation** */
 4: **for** each query $q_i \in Q$ **do**
 5:      Retrieve cluster set $C(q_i)$
 6:      **for** each existing group $G_j \in G$ **do**
 7:          Compute similarity $J(q_i, q_j)$ for $q_j \in G_j$
 8:          **if** $\max(J(q_i, q_j)) \geq \theta$ **then**
 9:              Assign q_i to G_j
10:              **break**
11:          **end if**
12:      **end for**
13:      **if** q_i is not assigned **then**
14:          Create new group $G_{\text{new}} \leftarrow \{q_i\}$
15:          Add G_{new} to G
16:      **end if**
17: **end for**
18:
19: /* **Step 2: First Query of the Next Group** */
20: **for** each group $G_i \in G$ **do**
21:      Identify the first query: $q_F(G_i) \leftarrow G_i[0]$
22:      Store associated clusters: $C(q_F(G_i))$
23: **end for**
24:
25: /* **Step 3: Data Structure Definition** */
26: Define data structure:
27: $\mathcal{D} \leftarrow \{(G_i, \{q_{i_1}, ..., q_{i_m}\}, C(G_i), q_F(G_{i+1}), C(q_F(G_{i+1}))) \mid i = 1, ..., k\}$
28:
29: /* **Step 4: Prefetching Condition** */
30: **for** each group $G_i \in G$ **do**
31:      **for** each query $q_j \in G_i$ **do**
32:          Perform search using clusters $C(q_j)$
33:          **if** q_j is the last query in G_i and $i < |G|$ **then**
34:              Identify first query of next group: $q_F(G_{i+1})$
35:              Prefetch clusters: $S \leftarrow S \cup C(q_F(G_{i+1}))$
36:          **end if**
37:      **end for**
38: **end for**
39: **return** Updated Query Groups G and Cache State S

---

4 Evaluation

4.1 Experimental Setup

Configurations. All of our experiments were conducted on a single edge server equipped with Ryzen 1950X 16-core 3.4 GHz CPU with 8GB of memory and Samsung 960 SSD with 256GB. We used FAISS [4] version 1.10.0 as vector search engine and Apache Kafka [5] version 3.2.3 as message queue. In the vector search configurations, we selected IVF indexing [14], a widely used algorithm for fast ANN search. Throughout the experiment, we set the total number of clusters to 100 and nprobe to 10. Additionally, we configured the total entries of cache to 40 and Jaccard distance threshold is set to 0.5. All experiments include 1-minute warm-up phase to ensure that cache utilization stabilizes.

Table 1: Details of evaluated datasets

Dataset	Corpus	Number of Records	Embedding Size	Distance Function
nq [9]	4.6 GB	2.68 M	8.3 GB	L2
hotpotqa [20]	11 GB	5.42 M	15.4 GB	L2
fever [16]	7.5 GB	5.23 M	18.5 GB	L2

Query Workloads. To construct the vector index, we used three datasets from BEIR benchmark [15], with details provided in Table 1. Although the total embedding size of each vector index exceeds the DRAM capacity, IVF-based vector search loads only the embeddings corresponding to the nprobe value into memory, reducing memory requirements. Both the corpus and query sets from the BEIR benchmark were converted into multidimensional vector embeddings using the all-miniLM-L6-v2 model [19].

Traffic. In real-world scenarios, users simultaneously send queries to the vector search engine. To maximize resource utilization and improve cache efficiency, the engine batches queries over short intervals [6]. Throughout our experiments, we adjusted the batch size distribution accordingly, randomly generating between 20 and 100 queries per batch.

Comparison Targets and Methods. We evaluated search performance using cache utilization, mean latency, and tail latency. For search latency, we defined the search latency from encoding query to top-k retrieval. As explained in Section 2.2, reading selective clusters from storage is also part of the critical path. To evaluate the effectiveness of our approach, we implemented the cache schemes proposed by EdgeRAG [12] and compared it with our method. In both EdgeRAG and CaGR-RAG, we stored index files for each cluster on storage and profiles the read latency per each cluster during the offline phase.

– **EdgeRAG** is a disk-based IVF index approach that proposes cost-aware cache to consider frequently access clusters and high read latency. We implemented their cache scheme that prioritizes cluster with high generation latency and accessed count.
– **CaGR-RAG** is our framework, which implements context-aware query grouping and opportunistic prefetch scheme whenever query group is switched.
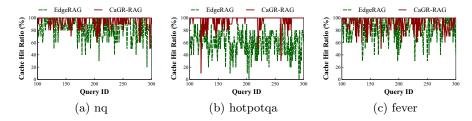
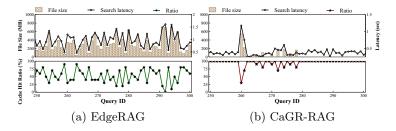Fig. 4: Cache utilization of EdgeRAG and CaGR-RAG under three datasets.

Fig. 5: The relationship between file size reading from disk, search latency, and cache hit ratio in hotpotqa dataset. Note that (a) and (b) use a different scale on the right y-axis.

4.2 Cache Utilization

Figure 4 presents the cache hit ratio for query IDs ranging from 100 to 200, comparing EdgeRAG and CaGR-RAG. Overall, CaGR-RAG achieves a higher cache hit ratio than EdgeRAG. In contrast, CaGR-RAG consistently maintains a high cache hit ratio over 60%. Notably, in Figure 4(b), EdgeRAG exhibits sharp fluctuations in cache hit ratio, occasionally dropping to 0%, whereas CaGR-RAG maintains a stable hit ratio close to 100% with minimal variability. These results stem from CaGR-RAG grouping queries based on cluster similarity, increasing the likelihood of cache hits when processing queries within the same group.

To evaluate the impact of the analyzed cache hit ratio on performance, we selected the hotpotqa dataset, which exhibited the most distinct pattern. Figure 5 presents the file size and search latency for both systems based on the cache hit ratio. In EdgeRAG, when a cluster is evicted from the cache, it is deleted from memory, reflecting its cache management strategy. We analyzed the range from 250 to 300 within the entire query set. In Figure 5 (a), as cache hits decrease, the file size required to fetch clusters from disk increases, leading to longer search times. As explained in Section 2.2, this occurs because queries retrieve clusters from the second-level index on disk.

In contrast, in Figure 5 (b), CaGR-RAG ensures that most queries result in cache hits. At 100% cache hit rate, the search latency remains below 200ms, which is six times lower than the highest latency query (e.g., Query 260). Interestingly, despite having the same cache hit rate, the search latency differs

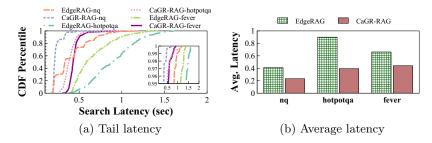(a) Tail latency　　　　　　　　　(b) Average latency

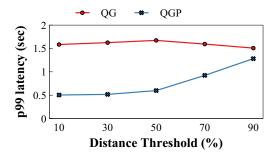Fig. 6: Search latency comparison between EdgeRAG and CaGR-RAG for different datasets.

Fig. 7: 99th percentile tail latency of query grouping (QG) and query grouping and prefetch (QGP) in CaGR-RAG on the hotpotqa dataset with varying Jaccard distance thresholds.

between EdgeRAG and CaGR-RAG. This arises from differences in the cluster file sizes generated during the offline profiling stage. For instance, in the hotpotqa dataset, the largest cluster file is 160MB, while the smallest is 30MB. Currently, clusters are prefetched when switching between query groups. However, performance could be further improved by considering the size of the next file to be read.

4.3 Search Latency

Figure 6 (a) presents CDF of latency and average latency for EdgeRAG and CaGR-RAG across the three datasets. The nested graph at the bottom right zooms in on from 95th to 100th percentile range to highlight the performance difference. At the 99th percentile, EdgeRAG shows high latency of 0.936 seconds for nq, 1.5365 seconds for hotpotqa, and 1.287 seconds for fever. In contrast, CaGR-RAG achieves significantly lower latency of 0.4621, 0.7445, and 0.7584 seconds under the same conditions. Notably, CaGR-RAG reduces the 99th percentile tail latency by up to 51.55% on hotpotqa dataset. Overall, CaGR-RAG achieves lower average latency across all three datasets compared to EdgeRAG in Figure 6 (b).

4.4 Module Effectiveness

We conducted experiments on hotpotqa to evaluate the effectiveness of each module: query grouping (QG) and opportunistic prefetching (QGP). QG represents the approach of only query grouping, while QGP represents the use of both modules. Jaccard distance closer to 100 indicates greater similarity between the two sets. As shown in Figure 7, when distance threshold is 90%, both approaches shows similar 99th percentile latency. In practice, it is rare for query clusters to overlap by 90%. In such cases, the effectiveness of grouping is limited, as each query forms its own group. Although prefetching is performed for each group, frequent cache evictions may reduce the cache hit rate (e.g., Query 260 in Figure 5).

When the cluster similarity ratio per query is 10%, multiple queries are grouped together. However, since multiple groups still exist, QG experiences a significant drop in cache hits whenever the group changes. In contrast, QGP opportunistically prefetches the clusters of the first query in the next group, reducing tail latency by up to 3.1 times compared to QG. This suggests that query grouping alone cannot prevent sudden cache hit drops, while integrating prefetching significantly improves performance.

5 Conclusion

In this work, we introduces CaGR-RAG, a context-aware query grouping and opportunistic prefetch scheme that proactively reorders queries based on their cluster similarity. Our approach significantly improves cache reuse and reduces tail latency compared to the baseline. The proposed query grouping and prefetching scheme is compatible with any cache replacement policy. We hope CaGR-RAG to serve as a solution for accelerating response times in RAG systems.

References

1. Bang, F.: 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. Association for Computational Linguistics, Singapore (Dec 2023). `https://doi.org/10.18653/v1/2023.nlposs-1.24`, `https://aclanthology.org/2023.nlposs-1.24/`
2. Chen, C., Jin, C., Zhang, Y., Podolsky, S., Wu, C., Wang, S.P., Hanson, E., Sun, Z., Walzer, R., Wang, J.: Singlestore-v: An integrated vector database system in singlestore. Proc. VLDB Endow. **17**(12), 3772–3785 (Aug 2024). `https://doi.org/10.14778/3685800.3685805`, `https://doi.org/10.14778/3685800.3685805`
3. Chen, Q., Zhao, B., Wang, H., Li, M., Liu, C., Li, Z., Yang, M., Wang, J.: Spann: highly-efficient billion-scale approximate nearest neighbor search. In: Proceedings of the 35th International Conference on Neural Information Processing Systems. NIPS '21, Curran Associates Inc., Red Hook, NY, USA (2021)
4. Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.E., Lomeli, M., Hosseini, L., Jégou, H.: The faiss library (2024)

5. Garg, N.: Apache Kafka. Packt Publishing (2013)
6. Guo, R., Luan, X., Xiang, L., Yan, X., Yi, X., Luo, J., Cheng, Q., Xu, W., Luo, J., Liu, F., Cao, Z., Qiao, Y., Wang, T., Tang, B., Xie, C.: Manu: a cloud native vector database management system. Proc. VLDB Endow. **15**(12), 3548–3561 (Aug 2022). https://doi.org/10.14778/3554821.3554843, https://doi.org/10.14778/3554821.3554843
7. Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B., et al.: A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. ACM Transactions on Information Systems **43**(2), 1–55 (2025)
8. Jayaram Subramanya, S., Devvrit, F., Simhadri, H.V., Krishnawamy, R., Kadekodi, R.: Diskann: Fast accurate billion-point nearest neighbor search on a single node. Advances in neural information processing Systems **32** (2019)
9. Kwiatkowski, T., Palomaki, J., Redfield, O., Collins, M., Parikh, A., Alberti, C., Epstein, D., Polosukhin, I., Devlin, J., Lee, K., Toutanova, K., Jones, L., Kelcey, M., Chang, M.W., Dai, A.M., Uszkoreit, J., Le, Q., Petrov, S.: Natural questions: A benchmark for question answering research. Transactions of the Association for Computational Linguistics **7**, 452–466 (2019). https://doi.org/10.1162/tacl_a_00276, https://aclanthology.org/Q19-1026/
10. Li, Z., Zhang, X., Zhang, Y., Long, D., Xie, P., Zhang, M.: Towards general text embeddings with multi-stage contrastive learning. arXiv:2308.03281 (2023)
11. Müllner, D.: Modern hierarchical, agglomerative clustering algorithms. arXiv preprint arXiv:1109.2378 (2011)
12. Seemakhupt, K., Liu, S., Khan, S.: Edgerag: Online-indexed rag for edge devices. arXiv preprint arXiv:2412.21023 (2024)
13. Shen, M., Umar, M., Maeng, K., Suh, G.E., Gupta, U.: Towards understanding systems trade-offs in retrieval-augmented generation model inference (2024), https://arxiv.org/abs/2412.11854
14. Sivic, J., Zisserman, A.: Video Google: A text retrieval approach to object matching in videos. In: IEEE International Conference on Computer Vision. vol. 2, pp. 1470–1477 (2003)
15. Thakur, N., Reimers, N., Rücklé, A., Srivastava, A., Gurevych, I.: BEIR: A heterogeneous benchmark for zero-shot evaluation of information retrieval models. In: Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2) (2021), https://openreview.net/forum?id=wCu6T5xFjeJ
16. Thorne, J., Vlachos, A., Christodoulopoulos, C., Mittal, A.: FEVER: a large-scale dataset for fact extraction and VERification. In: NAACL-HLT (2018)
17. Wang, L., Yang, N., Huang, X., Yang, L., Majumder, R., Wei, F.: Multilingual e5 text embeddings: A technical report. arXiv preprint arXiv:2402.05672 (2024)
18. Wang, M., Xu, W., Yi, X., Wu, S., Peng, Z., Ke, X., Gao, Y., Xu, X., Guo, R., Xie, C.: Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. Proc. ACM Manag. Data **2**(1) (Mar 2024). https://doi.org/10.1145/3639269, https://doi.org/10.1145/3639269
19. Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., Zhou, M.: Minilm: deep self-attention distillation for task-agnostic compression of pre-trained transformers. NIPS '20, Curran Associates Inc., Red Hook, NY, USA (2020)
20. Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W.W., Salakhutdinov, R., Manning, C.D.: Hotpotqa: A dataset for diverse, explainable multi-hop question answering (2018), https://arxiv.org/abs/1809.09600