Disk-Based Shared KV Cache Management for Fast Inference in Multi-Instance LLM RAG Systems

Hyungwoo Lee
Sogang University
Seoul, Republic of Korea
azwie@sogang.ac.kr

Kihyun Kim
Sogang University
Seoul, Republic of Korea
kihyun@sogang.ac.kr

Jinwoo Kim
Sogang University
Seoul, Republic of Korea
jinwookim@sogang.ac.kr

Jungmin so
Sogang University
Seoul, Republic of Korea
jso1@sogang.ac.kr

Myung-Hoon Cha
ETRI
Daejeon, Republic of Korea
mhcha@etri.re.kr

Hong-Yeon Kim
ETRI
Daejeon, Republic of Korea
hykim@etri.re.kr

James J. Kim
Soteria Inc.
Seoul, Republic of Korea
jkim@soteria-sysm.com

Youngjae Kim†
Sogang University
Seoul, Republic of Korea
younkim@sogang.ac.kr

***Abstract*—Recent large language models (LLMs) face increasing inference latency as input context length and model size grow. Retrieval-augmented generation (RAG) exacerbates this by significantly increasing input tokens, leading to higher computational overhead during the prefill stage and prolonged time-to-first-token (TTFT). To address this, the paper proposes using a disk-based key-value (KV) cache to reduce the prefill computational burden, thereby shortening TTFT. We also introduce a disk-based shared KV cache management system, called Shared RAG-DCache, for multi-instance LLM RAG service environments. This system leverages the locality of documents related to user queries in RAG and queueing delays in LLM inference services to proactively generate and store disk KV caches for query-related documents, sharing them across multiple LLM instances to enhance inference performance. In experiments on a single host with 2 GPUs and 1 CPU, Shared RAG-DCache achieved a 15–71% increase in throughput and up to a 12–65% reduction in latency, depending on the resource configuration.**

***Index Terms*—LLM, KV-Cache, RAG, Vector DB, TTFT**

I. Introduction

Large Language Models (LLMs) have demonstrated exceptional performance across various tasks, and their increasing scale continues to deliver progressively more powerful capabilities. Models with billions or even trillions of parameters have significantly advanced the state-of-the-art in natural language processing, enabling remarkable abilities in understanding context, generating coherent text, and generalizing across diverse linguistic scenarios. Nevertheless, despite their extensive capacity, LLMs frequently encounter difficulties when tasked with providing accurate responses involving the most recent or specialized internal corporate data, as such information typically falls outside the scope of their static pre-training datasets. This limitation arises because these models do not inherently possess mechanisms to dynamically update knowledge post-training, which significantly restricts their applicability in scenarios requiring up-to-date or domain-specific insights.

To address this limitation, Retrieval-Augmented Generation (RAG) [1] has gained attention. RAG improves LLM prompts by retrieving external documents related to the user query, thereby increasing the accuracy of responses regarding up-to-date information or domain-specific knowledge [1]–[3]. However, incorporating external context documents into the prompt significantly increases its length, leading to longer Time-To-First-Token (TTFT) [4] and reduced throughput. This phenomenon arises primarily from the increased computational complexity during the prefill phase of LLM inference, where the model computes attention scores and generates the corresponding key-value (KV) matrices for all tokens in the expanded prompt. Specifically, the complexity of KV cache generation scales approximately as $(O(L \cdot N^2 \cdot D))$ [5], [6], where L represents the number of Transformer layers, N denotes the total token length of the input (including both original prompts and added contexts), and D corresponds to the dimensionality of the hidden representations.

During this process, each token's embedding is transformed into query, key, and value vectors, after which self-attention calculations occur between these vectors, producing the attention scores and resulting value vectors. These calculated key-value pairs, which constitute the KV cache, must be computed for every token within the input prompt during the prefill stage, imposing substantial computational overhead. Particularly, as the input length (N) grows due to appended retrieved documents, the self-attention operations require quadratic complexity $(O(L \cdot N^2 \cdot D))$, substantially escalating the computation demand and slowing inference speed. This increased complexity becomes especially pronounced with larger LLM models, whose parameter sizes further amplify the computational cost.

Nevertheless, in RAG systems, query locality is observed, where a subset of external documents tends to be frequently referenced across multiple user queries. To quantify this, an analysis of question-and-answer datasets such as SQuAD [7], HotpotQA [8] and TriviaQA [9] found that, on average, processing 50% of queries requires accessing only a small fraction—between 3.1% (for SQuAD) and 31.39% (for TriviaQA)—of the total unique documents retrieved. This indicates that precomputed KV caches for these commonly accessed

†Y. Kim is the corresponding author.

documents could significantly reduce redundant computations during inference.

Related Work: Leveraging this query locality, existing studies have explored pre-computing and caching KV states for RAG. RAGCache [10] employs multi-level GPU/CPU memory caching, but memory capacity limits large-scale document caching. TurboRAG [11] utilizes offline disk-based precomputation, achieving significant TTFT reduction by addressing accuracy issues with mask and positional embeddings, yet it lacks explicit multi-instance/host KV cache sharing. More recent systems like CacheBlend [12] and Cache-Craft [13], often vLLM-based and single-host oriented, focus on combining or managing chunk-level KV caches with selective recomputation to maintain quality. CacheBlend emphasizes efficient combination of KV caches from multiple text chunks, while Cache-Craft adds hierarchical storage (GPU, CPU, SSD) and advanced preloading. However, these approaches do not primarily target proactive, shared disk-based KV caching tailored for dynamic multi-instance/host RAG environments with queue-aware prefetching.

In this paper, we propose a disk-based KV cache management system composed of two solutions: *RAG-DCache* and *Shared RAG-DCache*. RAG-DCache precomputes and stores the KV cache for frequently retrieved document chunks within a disk-resident vector database. During inference, these precomputed KV caches are reused directly, eliminating the costly recomputation of the full document context. Shared RAG-DCache extends this concept to multi-instance inference environments, enabling multiple LLM instances to share a common KV cache stored on disk, thus further enhancing inference performance by proactively generating and distributing KV caches across instances during query waiting periods.

The proposed system consists of three main components. The *KV Cache Manager* handles offline precomputation and manages disk-based KV caches integrated within the vector database. In multi-instance settings, the *KV Cache Generator* proactively prefetches and generates KV caches during query wait times, utilizing idle resources. Finally, the *Prompt Generator (RAG Processor)* constructs LLM prompts by combining retrieved KV caches with user queries, bypassing redundant computations. And our system has the following distinctions from existing research:

- Supports multi-instance/multi-host LLM service environments. Through a centralized Shared KV Cache Manager, KV caches stored on disk can be efficiently shared and reused among multiple LLM instances.
- Leverages query queue time by proactively prefetching and generating KV caches for waiting queries using idle CPU or GPU. This mechanism utilizes waiting time, reducing response latency when the query is processed.
- Presents an optimal resource allocation strategy in a multi-instance environment. It showed dedicating KV cache generation to the CPU while utilizing all GPUs for LLM inference achieves the performance gains.

In experiments on a server equipped with dual GPUs and a single CPU, using the SQuAD dataset, employing RAG-DCache (the single-instance component) reduced TTFT by approximately 10–20%, with throughput increasing with model and batch size. Furthermore, our multi-instance solution, Shared RAG-DCache, achieved more improvements, increasing throughput by up to 71% and reducing latency by up to 65%.

II. Background and Motivation

A. KV Cache Utilization in LLM Inference

Transformer-based LLMs [5] generate text in an autoregressive manner by producing one token at a time. To generate each token, the model processes the entire prompt and then, during the decode phase, reuses the previously generated tokens as input to predict the next token [14]–[18]. Recomputing all tokens at every generation step would be inefficient, so the model stores the Key and Value matrices from previous steps in GPU memory as a cache, which is then reused for subsequent token predictions. This KV cache is critical for avoiding redundant computations and helps reduce the overall complexity of the decode phase. For example, optimization libraries like DeepSpeed-Inference [19] incorporate KV caching to enhance inference efficiency in large Transformer models.

B. RAG Prompt Composition

In a RAG system, external documents are retrieved using a vector database. These documents are divided into manageable chunks, then converted into vector embeddings using an embedding model. These embeddings, together with their document IDs and original texts, are stored in the vector database [20], [21]. When a query is received, it is similarly embedded and matched against this vector store to retrieve the top-k most relevant document chunks. The final LLM prompt is then composed by concatenating the retrieved document texts with the user query, typically structured as: "Document: retrieved texts + Query: user question + Answer:". However, as the number of tokens in the prompt lengthens due to added context, the computational complexity during the prefill phase increases significantly, leading to higher TTFT and reduced throughput.

To address this issue, we proposes utilizing external knowledge documents not only as retrieval sources but also as precomputed KV caches. Specifically, if the KV tensors of the external knowledge documents are precomputed and stored, they can be directly reused whenever the same document is included in future LLM prompts. By leveraging disk-based caching of these precomputed KV tensors, the costly prefill computations associated with document processing are significantly reduced, resulting in shorter TTFT and improved inference performance. However, the effectiveness of this caching mechanism fundamentally depends on query locality—the frequency with which particular documents are repeatedly referenced across multiple queries. Hence, understanding and exploiting query locality becomes crucial for optimizing system performance, which we discuss further in the following section.

C. Locality of Documents Retrieved by Queries in RAG

To verify the locality between queries and their retrieved documents when using RAG, we used popular question-answering datasets, SQuAD, HotpotQA and TriviaQA. We first

constructed a FAISS vector database by embedding documents (using all-MiniLM-L6-v2 [22] embedding model) from each dataset. Then, for each query, we measured the similarity between the query embeddings and the document embeddings using an inner-product-based similarity metric, retrieving the top-1 most similar document to query. And using the [query, top-1 document] data, we calculated the proportion of documents required to process the queries.

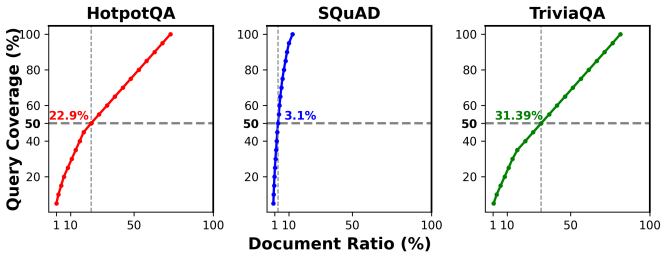


Fig. 1: CDF of query related documents.

The results (Fig. 1) show that only 22.9%, 3.1%, and 31.4% of the most frequently retrieved documents account for 50% of all queries in each dataset. This result suggests that caching a small subset of frequently accessed documents can effectively serve a large portion of queries, highlighting the efficiency of using document caching in RAG systems.

D. Shared KV Cache for Multi-instance LLM Inference

LLM-based inference services generally operate multiple model instances in parallel to handle numerous real-time user requests [14]. For example, on a server with two GPUs, two LLM instances can be run to process two queries simultaneously, or one GPU can be allocated to a different task. In such multi-instance environments, each instance performs inference independently, so an instance cannot inherently access the KV cache computed by another instance. Therefore, to maximize the benefits of caching, a structure is needed that allows instances to exchange cache data via shared memory or storage (e.g., disk). We focus on a disk-based KV cache sharing approach to cope with the increasing number of LLM parameters and the growing size of input tokens.

Moreover, as requests per second increase with rising service loads, requests that exceed the capacity of an individual instance incur queue wait time [23]. In a single-instance LLM environment, only one request can be processed at a time, so subsequent requests must inevitably wait. However, in a multi-instance environment, there is a possibility of utilizing free resources or other devices to prepare tasks that are waiting in the queue. For instance, while one GPU is decoding a current query, it may be possible to leverage another idle GPU or a CPU to carry out preliminary work for the next query, thereby reducing response latency. The proposed Shared RAG-DCache implements this idea by prefetching necessary document KV caches for requests that wait in the inference service queue beyond a certain threshold. As a result, when the request eventually reaches an LLM instance, it can focus solely on the decoding step.

Figure 2 shows the average response time from when a client issues a query until the response is received—divided into queue wait time, LLM processing time, and network time (communication time between client and LLM server)—under varying query rates per second, using a single Llama-3.2-1B [24] model on one GPU. For detailed experimental settings, refer to Table I in Section 4. At 1–2 queries per second, LLM processing time makes up over 70% of the total time. However, once the query arrival rate exceeds the LLM processing time, the queue wait time increases exponentially. This suggests that under heavy loads, it is beneficial to utilize the waiting period for prefetch operations. Therefore, in a multi-instance environment, it is desirable to share caches among multiple instances to avoid redundant computation, and prepare caches in advance during wait times, allowing the GPU to focus on pure inference tasks. Shared RAG-DCache is the system designed to meet these needs.

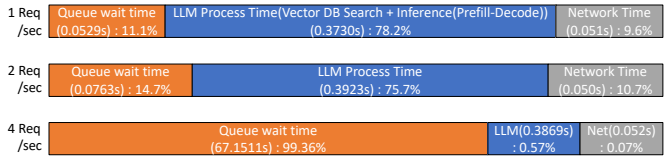


Fig. 2: Inference Latency Based on Requests Per Second in a Multi-Instance LLM(with RAG) Service Environment.

III. Design and Implementation

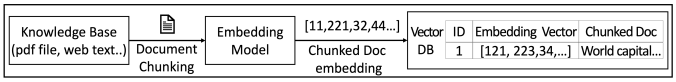
In this section, we detail the architecture and operation of our proposed RAG-DCache system—a disk-based KV cache for single-instance LLM inference—as well as its extension to a multi-instance environment, called Shared RAG-DCache.

A. Disk-based KV Cache Structure and Operation

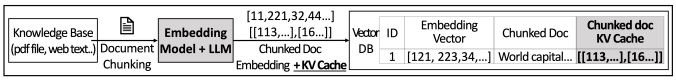
The RAG-DCache system extends the traditional Retrieval-Augmented Generation pipeline by adding a disk-resident Key–Value cache storage and associated management modules. The basic idea is to precompute the KV cache for each document chunk in the vector database and store these caches on disk so that they can be reloaded during inference instead of recomputed from the original text.

Figure 3 contrasts a standard vector database with our augmented version. In a conventional RAG setup, the vector database stores embeddings of document chunks for similarity search. In our approach RAG-DCache, we augment each stored document chunk with its precomputed "chunked-document KV cache," pairing it with the document's ID and embedding in the database. Because document changes rarely once the vector DB is built, we can leverage idle hardware to generate these KV caches offline and persist them to disk. It is also possible to generate a KV cache during inference for a newly encountered document and then add it to the DB for future reuse. By caching each document's transformer key-value pairs in advance, the LLM can skip directly to using this cached representation when that document is retrieved for a query, rather than processing the raw text each time.

Fig. 4 illustrates the design and operation of RAG-DCache, which incorporates a KV Cache–linked vector DB for LLM inference using RAG. The main components of disk-based KV Cache are as follows:



(a) General vector DB



(b) Vector DB with additional utilization of KV Cache

Fig. 3: Vector DB creation process and tuple structure for RAG.

- **KV Cache Manager**: This module is responsible for creating and managing the stored caches. It generates the KV cache for document chunks using the LLM and stores the resulting key-value tensors in the vector database. It also handles retrieval of these caches from disk upon request. To minimize disk I/O latency, the KV Cache Manager employs an in-memory (CPU RAM) cache to hold frequently or recently used KV entries, leveraging faster memory access and reducing repeated disk reads. In essence, it acts as the interface between slow persistent storage and the rest of the system, optimizing cache generation and lookup.
- **RAG Processor**: The RAG Processor orchestrates the RAG inference workflow. Upon receiving a user query, it performs a similarity search on the vector database to fetch the relevant document ID. It then requests the corresponding KV caches for the document from the KV Cache Manager, and composes the final LLM prompt by combining the query with the retrieved KV caches.
- **Integrated Vector Database**: This is an extended vector store that holds not only each document chunk's embedding and original text, but also the precomputed KV caches. Each entry in the vector DB effectively becomes a tuple of the form (embedding, document ID, text, KV cache). The inclusion of the KV cache alongside the embedding means that after retrieval, the system immediately has access to the document's encoded representation for the LLM. Since documents are largely static, these caches can be generated offline and remain valid unless the document content changes. For any new documents added to the corpus, on-demand cache generation can be performed and the caches appended to the database, keeping the cache store up-to-date. This integrated DB design ensures that the vector index serves a dual purpose: it provides nearest-neighbor search for relevant documents and acts as a lookup table for their cached LLM representations.

The end-to-end operation of RAG-DCache proceeds as follows (refer to the numbered steps in Fig. 4):

❶ Offline Cache Preparation: Initially, use existing documents to pre-generate the KV caches and build a KV-augmented vector database. The KV Cache Manager takes each document chunk and computes its KV cache using the LLM model, then stores this cache in the vector DB alongside the document's embedding and ID. This step can be done offline or in the background, populating the disk cache before queries arrive. By the end of this step, the system has a disk-based cache of key-value pairs ready for many documents in the corpus.

❷, ❸ Query Retrieval: When a user query comes in, the RAG Processor embeds the query and searches the vector database for the most relevant document ID. This yields the ID of the document that will be used to augment the query.

❺, ❻ KV Cache Retrieval: For each document ID obtained in step ❷, ❸, the RAG Processor requests the corresponding KV cache from the KV Cache Manager. The KV Cache Manager checks its memory cache for the entry; if present, it returns it immediately. If not, it loads the KV cache from disk into memory and returns it to the RAG Processor.

❼, ❽ Prompt Composition: Meanwhile, the user's query text is converted into the appropriate embedding or token IDs for the LLM model if not already done during retrieval. Once the KV cache for the document is in hand, the RAG Processor constructs the final LLM input prompt. It does this by inserting the retrieved KV cache data into the model's context as "past key-values" and appending the user query tokens as the current input. Essentially, the LLM is tricked into believing it has already processed the retrieved documents, because their resulting key-value pairs are provided, and now it only needs to attend to the user's query. In practice, this means setting the model's internal key-value state to the cached values and providing the query tokens as the next sequence to process.

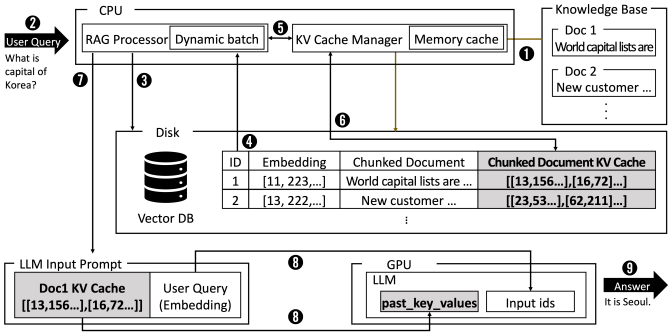


Fig. 4: Design and Operation of RAG-DCache.

❾ LLM Inference (Prefill + Decode): The LLM, now armed with the combined prompt (document KV Cache + query embeddings), proceeds to generate an answer. It first goes through its prefill phase and then the decode phase to produce output tokens. Because of RAG-DCache, the prefill phase is dramatically accelerated: instead of recomputing key-value pairs for the document's text on the GPU, the model directly uses the precomputed keys and values. It only needs to encode the user query itself and then can immediately attend to the cached document representations when predicting the answer. After the prefill, the decode stage proceeds as usual to generate the response token by token.

By leveraging pre-stored caches in this manner, RAG-DCache reduces the TTFT and overall computational load on the GPUs. In a baseline RAG system without caching, the LLM must process the full text of the retrieved documents for every single query, leading to significant repeated work in the prefill stage. Our approach avoids this repetition. There is an overhead for loading the KV cache from disk (when a cache is not already in memory), but as long as efficient storage (fast

SSD) and caching strategies are used, the sum of "(disk load time) + (cached prefill time)" is typically much less than the original prefill time required to encode the documents from scratch. In other words, even accounting for disk I/O, the TTFT with RAG-DCache is lower than without it, provided the caches are effectively utilized. This will be quantitatively demonstrated in our evaluation. Note that when multiple documents are retrieved (k >1), we do not calculate cross-attention between the documents we only calculate KV values between the user query and the retrieved documents. This may lead to accuracy degradation as shown in the evaluation.(Fig. 8)

To address this issue, instead of precomputing each document's KV Caches and storing them in the vector DB, we changed our approach so that during inference—when RAG retrieves documents—the KV Caches for the top-k documents are generated and stored together with the vector DB. We also made sure that the generated KV Caches are stored and managed within the vector DB along with the documents' IDs, allowing us to handle any top-k scenario. For example, if top-k = 3 and the retrieved document IDs are 1, 2, and 3, we calculate the attention for those three documents together to generate their KV Caches, then add that combination of document IDs to the vector DB so the KV Cache Manager can easily locate them. This approach applied to Shared RAG-DCache, which will be described next, leverages idle time during the inference process to precompute KV caches, regardless of the number of retrieved documents. These precomputed KV caches are then stored on disk, shared across instances, and reused to improve efficiency and maintain accuracy.

B. Multi-Instance Structure of RAG-DCache

While RAG-DCache improves single-instance performance, modern LLM services often deploy multiple LLM instances (across one or more GPUs or nodes) to handle high query throughput. In such environments, caches computed in one instance could be beneficial to others. We therefore introduce Shared RAG-DCache, an extension of RAG-DCache for multi-instance LLM service environments. Shared RAG-DCache enables cache sharing and cache prefetching across multiple parallel LLM inference processes. The goal is to exploit both the locality of document usage across different queries and the idle time that queries spend waiting in a queue due to high load to proactively generate and distribute KV caches.

Fig. 5 shows the architecture of Shared RAG-DCache, which builds on the single-instance design with additional components for multi-instance. The main components are as follows:

- **KV Cache Generator**: This is a new background module that proactively creates KV caches for queued queries. It continuously monitors the central query request queue and identifies queries that have been waiting longer than a predetermined threshold. For a query that is stuck in the queue (indicating the system is busy and the query will not be served immediately), the KV Cache Generator takes action: it immediately computes an embedding for the query, uses it to search the vector database for top-k relevant documents, and then computes the KV caches for those documents

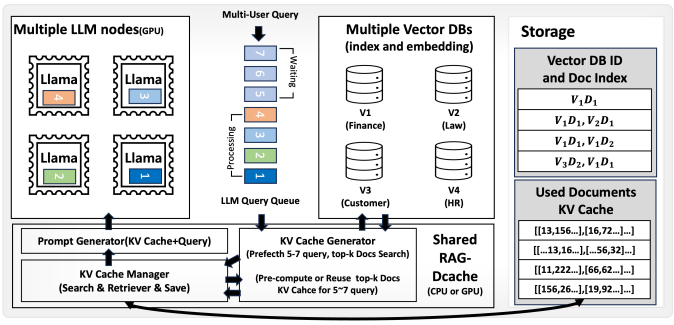


Fig. 5: Shared RAG-DCache Architecture.

on the fly. Essentially, it performs steps 2–5 of the RAG-DCache workflow ahead of time for queries that are still waiting. The KV generation uses the same LLM model that will ultimately answer the query, but it can be executed on any available device – for example, on an idle GPU if one exists, or on the CPU if all GPUs are busy – since this is done asynchronously. Once generated, the new KV cache is stored to disk via the KV Cache Manager and indexed by document ID so that any LLM instance can retrieve it later. If a KV cache for a particular document was already created previously, the generator will detect this and avoid redundant computation. In that case, the existing cache can be reused directly. This component effectively prefetches document caches during the query's waiting time, leveraging otherwise idle compute resources to reduce future work.

- **Shared KV Cache Manager**: In a multi-instance setup, instead of each LLM instance having its own independent KV Cache Manager, we deploy a centralized KV Cache Manager service. This service coordinates the storage and sharing of KV caches among all LLM instances. It receives newly generated caches from the KV Cache Generator and inserts them into the global disk-based cache. When an LLM instance needs a KV cache for a document, it queries this shared manager rather than a local disk, and the manager supplies the data to the instance over the network or inter-process channel. The Shared KV Cache Manager thus acts as a cache server, ensuring that all LLM instances have a consistent view of available KV entries and that once a document's cache is generated by any one instance or the generator, it can be used by all. Like the single-instance manager, it also implements a memory caching layer (using CPU RAM) with an eviction policy (e.g., LRU) to keep frequently accessed caches readily available. This is especially important in multi-instance scenarios to avoid repeatedly hitting the disk if multiple instances request the same cache around the same time.
- **Prompt Generator (per-instance)**: Each LLM instance is equipped with a Prompt Generator module. This component is conceptually similar to the RAG Processor's prompt composition step in the single-instance case, but tailored for a multi-instance context. When a query is assigned to a specific LLM instance for processing, that instance's Prompt Generator will request any needed KV caches from the Shared KV Cache Manager. It then combines the retrieved

KV cache(s) with the query text to form the final input prompt for its local LLM, identical to how it was described in the single-instance workflow. With the KV cache preloaded into the model's context, the LLM instance can skip directly to decoding the answer, greatly reducing the latency for that query. Essentially, the Prompt Generator ensures each instance makes full use of the globally cached data: it injects the shared KV into the model and thereby avoids that instance doing any redundant prefill computation for the documents.

With these components, Shared RAG-DCache transforms a multi-instance deployment into a cooperative caching system. Fig. 6 illustrates the Shared RAG-DCache operation sequence:

❶ ❷ Queue Monitoring: The system monitors the central queue of incoming queries continuously. If a query's wait time exceeds a configured threshold (indicating prolonged queuing due to heavy load), that query is flagged for cache prefetching. The threshold can be tuned – any query waiting longer is considered a good candidate to start processing early, since it likely will wait that long anyway.

❸ Document Pre-search: For each flagged query, the KV Cache Generator immediately kicks in. It takes the query, computes its embedding, and performs a vector DB lookup to fetch the top-k most similar documents. This step is analogous to the retrieval step normally done by an LLM instance, but here it happens in parallel, on an idle thread/CPU or a free GPU, while the query is still in queue. By doing this in advance, we obtain the set of documents we anticipate the query will need, without delaying the query's actual service time.

❹ KV Cache Preparation: Next, for each of the k retrieved documents, the system prepares the KV cache. If the shared cache already contains a KV entry for a document, the KV Cache Generator will simply load that cache – possibly from disk to memory – immediately. If a cache is missing, the generator will perform a prefill computation for that document using the LLM model to create the KV cache. Once generated, the new KV cache is stored into the shared vector DB on disk via the Shared KV Cache Manager, making it available system-wide. This step effectively precomputes the heavy part of the LLM's work for the document while the query is still waiting in line. It's done for all top-k documents so that the query's entire retrieved context is cached ahead of time.

❺ LLM Inference: Eventually, the queued query reaches the front of the queue and is assigned to an LLM instance for execution. At this point, thanks to the previous steps, the KV caches for its relevant documents have likely already been generated and stored. The assigned instance's Prompt Generator fetches those caches from the Shared KV Cache Manager. The Prompt Generator then constructs the LLM prompt, combining the query text with the retrieved KV caches. Now the LLM can immediately begin decoding the answer, since the time-consuming document encoding work was done earlier. Effectively, the query's waiting time is utilized for precomputation, leading to significantly faster responses once the query is actively processed.

Through this mechanism, Shared RAG-DCache ensures that no two instances ever duplicate the same KV computation, and that the query wait times in a busy service are put to productive use. Since all KV caches reside in a shared disk-based store, any cache generated by one instance or by the prefetcher is immediately available to all other instances. This not only cuts down latency for the individual query that triggered the prefetch, but also improves throughput overall: multiple LLM instances can pull from the same cache repository, benefitting from each other's work. The design takes advantage of the locality in user queries and the nature of queued request handling to significantly reduce redundant computation across the system.

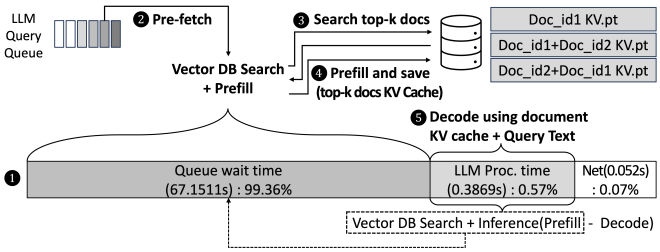


Fig. 6: Shared RAG-DCache operation sequence.

IV. Evaluation

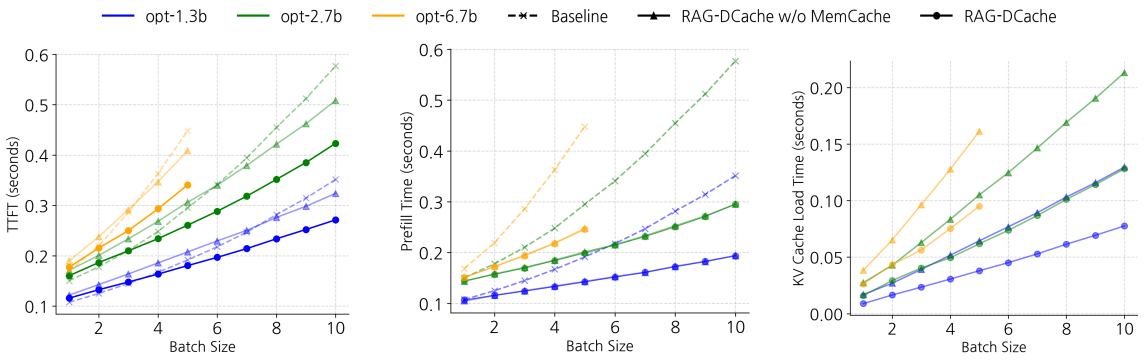
We implemented the above system and conducted experiments to evaluate the performance gains of RAG-DCache and Shared RAG-DCache. All experiments were performed on a single-host server with the following specifications(Table I):

TABLE I: Experimental hardware specifications.

Component	Specification
CPU	AMD Ryzen 9 3900XT (12 cores)
GPU	NVIDIA RTX 2080 SUPER(2 units, 8GB each)
Memory	64GB RAM
Storage	SAMSUNG 970 EVO NVMe SSD 500GB (Read 3.4GB/s Write:2,4GB/s)

We used the SQuAD v1.1 dataset as a source of queries and documents for retrieval. This dataset was created by crowdsourcing questions and answers from Wikipedia articles, where each answer is a direct span of text from the corresponding passage. It is primarily used as a benchmark to train and evaluate machine reading comprehension models, specifically for their ability to perform extractive question answering. Furthermore, SQuAD v1.1 drives research in natural language understanding and is often utilized for transfer learning to other related NLP tasks. For similarity search, we employed a Faiss [21] vector database. Different large language models were used for single-instance vs. multi-instance tests, and we measured key metrics including TTFT and throughput, as well as breakdowns of latency where appropriate. TTFT captures the latency from when a query is submitted to when the LLM outputs the first token of the answer – this primarily reflects the time spent in the prefill stage since decoding the very first token is usually quick once the model has the prompt. Throughput is measured as the number of queries that can be completed per second, reflecting the system's capacity under load.

The increase in the vector database storage capacity due to disk-based KV cache usage is shown in Table II. The FAISS Vector DB size is the sum of the index, the document embedding vectors, and the original text size of the documents.



(a) TTFT (b) Prefill Time (c) KV Cache Loading Time

Fig. 7: Results of TTFT, Prefill Time, and KV Cache Loading Time Measurements by LLM and Batch Size.

RAG-DCache and Shared RAG-DCache require additional disk space for storing KV caches in addition to the FAISS vector database storage. As expected, the KV cache size increases with the number of model parameters. Even with the same number of parameters, the KV Cache size can differ depending on the embedding method used by each LLM model. Additionally, if the number of documents extracted in RAG (top-k) increases, the size of the KV Cache generated may also grow. However, by adjusting the disk-based KV Cache that leverages query-document locality, the required storage space can be reduced—this may represent a tradeoff between query throughput and the necessary disk size.

TABLE II: Size of Normal FAISS Vector DB and KV Cache when using SQuAD dataset and different LLMs.

FAISS Vector DB	RAG-DCache and Shared RAG-DCache			
	opt-1.3b	opt-2.7b	opt-6.7b	LLAMA-1B
0.5MB	5.9GB	9.9GB	16GB	1GB

A. RAG-Dcache Results and Analysis

For the single-instance scenario, we evaluated RAG-DCache using the SQuAD dataset and Facebook's OPT [25] decoder-only LLM of varying sizes. We tested three model sizes to see how cache benefits scale with model complexity. The vector database was implemented with Faiss, and we chunked the SQuAD documents into passages for retrieval. Before inference, we pre-generated the KV caches for all document chunks that might be retrieved for the SQuAD queries, by running each chunk through the respective OPT model's prefill stage and storing the resulting KV pairs on disk. The KV Cache Manager was given a 16GB memory cache to hold recently used caches, as described earlier. Table III summarizes the experimental setup, including the LLM models, dataset size (2,000 queries from SQuAD v1.1 train), and other components. We compared two settings: a baseline RAG (meaning the LLM processes raw text of retrieved documents for every query) and RAG-DCache enabled, across different batch sizes. Here, "batch size" refers to the number of queries processed simultaneously by the model. For each combination of model size and batch size, we measured the average TTFT and the throughput in both the baseline and RAG-DCache configuration.

TABLE III: Experimental Environment.

Component	Specification
LLM	facebook/opt-1.3b, 2.7b, 6.7b
Embedding Model	all-MiniLM-L6-v2
Dataset	SQuAD v1.1 Train (2,000 samples)
Vector DB	Faiss DB, IndexFlatIP

Performance Breakdown: Fig. 7 presents the results for TTFT, Prefill time, and KV cache loading time under various conditions. Fig. 7(a) shows the overall TTFT for each model and batch size, comparing the baseline to RAG-DCache. Fig. 7(b) and Fig. 7(c) break this TTFT into two components for the RAG-DCache case: the time spent in the LLM's prefill stage, and the time spent loading KV caches from disk. In the baseline, TTFT is essentially all prefill.

As shown in Fig. 7(a), RAG-DCache consistently reduces TTFT in almost all cases. This is because using the disk-based KV cache drastically reduces the prefill computation time on the GPU, and the memory cache further cuts down repeated disk reads. In Fig. 7(b), we see that the prefill time with RAG-DCache is much lower than baseline since the LLM doesn't need to encode the full documents from scratch. Fig. 7(c) shows the KV cache loading time incurred for RAG-DCache – this is an overhead not present in the baseline. However, because of our caching optimizations, this overhead is kept relatively small: many cache loads are served from the 16GB memory cache, and even disk loads are fast on NVMe SSD. The result is that Prefill time savings outweigh the KV load time, yielding a net gain. For example, with the OPT-6.7B model at batch size 4, RAG-DCache might add a few milliseconds to load caches but saves far more time in GPU computation, leading to a substantially lower TTFT overall.

However, as shown in the Fig. 8 RAG-DCache works well when only a single document (top-k=1) is retrieved. We measured the accuracy of the answer using the formula F1 × 0.5 + Exact Match × 0.5, and the results showed that when top-k=1, the accuracy was the same as when RAG-DCache was not used. When multiple documents (top-k>1) were retrieved, the accuracy was higher than when RAG was not used but lower than when RAG-DCache was not applied. This is because we do not calculate cross-attention between the documents. To address these issues, researchs like Cache-

Craft and CacheBlend employ a selective KV recalculation process, updating KV values by including cross-attention calculations between tokens when necessary. In contrast, our approach leverages LLM queue waiting time within the LLM service runtime to pre-calculate and store the KV cache for retrieved top-k document groups, incorporating cross-attention from the outset. Therefore, during inference, using this stored, combined cache improves performance while maintaining accuracy, without the need for separate selective recalculation.

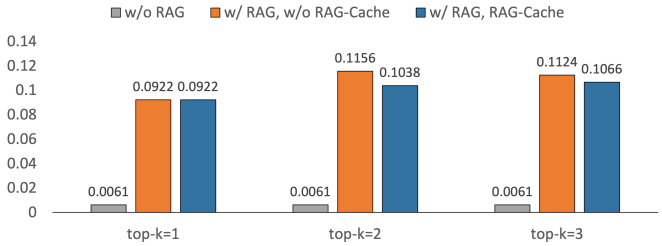


Fig. 8: Accuracy based on the number of retrieved documents (top-k) when using RAG-DCache.

Impact of Model Size: The benefits of RAG-DCache become more pronounced for larger models and batch sizes. Larger models have heavier per-token computation, so removing a chunk of tokens from their workload yields a bigger time savings. Similarly, at higher batch sizes, the GPU is encoding multiple queries' documents at once in the baseline, which is very compute-intensive; with caching, those computations are skipped, freeing the GPU to handle more queries.

In our experiments, we observed TTFT reductions of roughly 10%–20% with RAG-DCache compared to the baseline, depending on the configuration. These percentages tended towards the higher end (closer to 20%) for larger OPT models and larger batches. Concretely, Table IV shows the throughput achieved in each scenario, averaged per model. Without RAG-DCache, the throughput for OPT-1.3B was approximately 23.74 QPS, which increased to 26.63 QPS with RAG-DCache – approximately a 12% improvement. For the OPT-2.7B model, throughput went from 15.32 QPS to 18.01 QPS, approximately a 17.6% gain. The OPT-6.7B model was the slowest, but improved to 11.05 QPS with caching (15.7% increase). The average relative improvement in these models was around 14%–15%, aligning well with the TTFT savings noted above. These results validate that RAG-DCache not only lowers the latency per query but also increases the overall throughput of the system, as the GPUs spend less time on redundant tasks and can handle more queries.

TABLE IV: Average Throughput of Baseline and RAG-DCache.

Component	opt-1.3b	opt-2.7b	opt-6.7b	Average
Baseline	23.74	15.32	9.55	17.53
RAG-DCache	26.63	18.01	11.05	20.07

B. Shared RAG-DCache Results and Analysis

We next evaluate Shared RAG-DCache in a multi-instance LLM service scenario. The test environment remained the same dual-GPU server described above. We switched the LLM to Meta's LLaMA-3.2-1B model for these experiments. We used

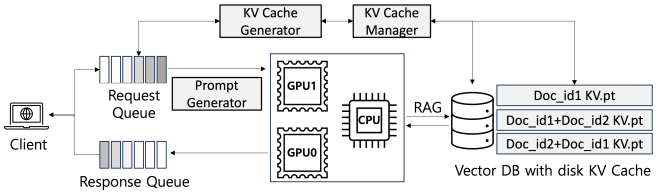


Fig. 9: Experimental System Configuration.

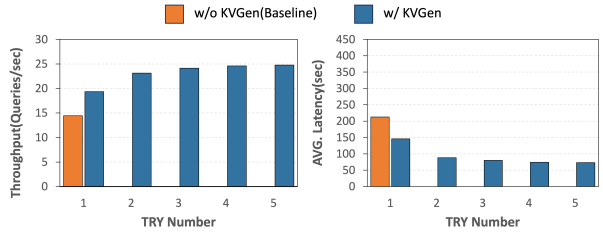
1,000 queries from the SQuAD v1.1 dataset and for each query. We retrieved either k=1 or k=2 documents to examine the effect of different context sizes. These queries were sent in a continuous stream at a rate of 40 requests per second to simulate a heavy multi-user load. This high query rate ensured that at any given moment there were multiple queries waiting in the queue, which is necessary to fully leverage the prefetching mechanism of Shared RAG-DCache. If the system is not under load, queries won't wait in queue and the KV Cache Generator might not trigger.

And whether Shared RAG-DCache was used or not, the KV cache values generated during the prefill stage remained the same regardless of changes in the top-k value. Therefore, accuracy was not measured separately. Furthermore, to control for variability in the decode phase, this experiment focused solely on measuring the prefill stage. This is because response latency can vary depending on the length of the generated answer during the decode stage. Our goal was to reduce noise and highlight the optimization benefits of the prefill stage. And to isolate the effect of shared disk caching, we disabled the memory caching in the KV Cache Manager for these experiments. This means all cache fetches go to disk, ensuring that any performance improvements observed are due to the multi-instance sharing and prefetching, not just RAM hits. Finally, to measure the performance improvement effect as the Disk-based KV Cache expands, we processed the entire dataset multiple times in a random order.

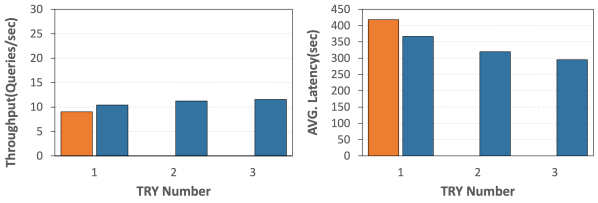
The test configuration for this experiment is shown in Fig. 9. To determine the optimal configuration in the given environment, we evaluated two resource allocation strategies for CPU and GPU.

- In Configuration (A) "GPU-Only KV Cache Generation", we dedicate one of the two GPUs entirely to KV cache generation tasks, and use only the other GPU to run the LLM inference. In other words, GPU0 handles all LLM inference requests, and GPU1 is reserved for computing KV caches of retrieved documents in the background.
- In Configuration (B) "CPU-Based KV Cache Generation", we use both GPUs for LLM inference, and assign all KV cache generation to the CPU. In this setup, the KV Cache Generator runs on the CPU, while both GPU0 and GPU1 are busy serving LLM inference requests.

Configuration (A) tests the scenario where we sacrifice a GPU to speed up cache prep, whereas (B) tests using no GPU for cache prep at the expense of slower cache generation on CPU. In both cases, Shared RAG-DCache is active, which means that caches are shared across the two LLM instances, and prefetching is enabled. The baseline for comparison is

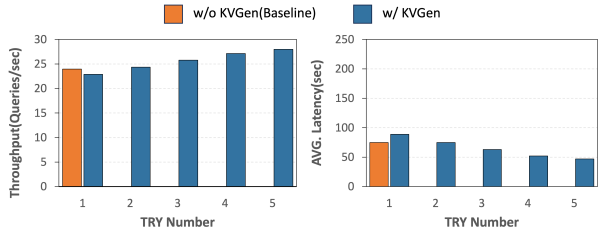


(a) top-k=1

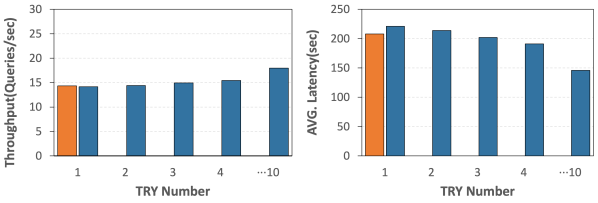


(b) top-k=2

Fig. 10: Throughput and Avg. Latency with Configuration (A) when top-k=1 and 2.



(a) top-k=1



(b) top-k=2

Fig. 11: Throughput and Avg. Latency with Configuration (B) when top-k=1 and 2.

a multi-instance system without Shared RAG-DCache(w/o KVGen). We measure the system's throughput in queries/sec and the average end-to-end latency which in our prefill-only measurement corresponds to how long a query waits plus its prefill time for each configuration and each top-k documents.

Overall Improvements: Shared RAG-DCache showed significant performance improvements over the baseline, though the magnitude depended on the configuration. In Configuration (A), as shown in Fig. 10, enabling the shared cache system to improve throughput by approximately 35-71% and reduce average latency by 31–65% compared to the baseline, according to our measurements. As the number of times the dataset was processed (TRY Number in the figure) increased, the performance improved further. (For the baseline, since there is no disk-based KV Cache expansion according to the number of tries, it was measured only once.)

For a example, in the k=1 scenario under Configuration (A), throughput increased from 23.96 QPS to 24.78 QPS with caching, and the average latency dropped from 75.75s to 73.25s. These particular numbers represent a modest 3.4% throughput gain and 3.3% latency reduction – relatively small, because with only one document the baseline was already not very slow and the single inference GPU was not heavily bottlenecked.

In the k=2 scenario with Configuration (A), we observed mixed results: the caching system sometimes incurred overhead that offset its benefit. Specifically, handling two documents per query on only one inference GPU proved challenging – the throughput and latency with caching in some trials were on par with or slightly worse than the baseline. This is likely because in Configuration (A) the single GPU had to handle the combined work of two documents' KV insertion plus the query itself, sequentially, which increased contention. The KV generation GPU could produce caches quickly, but the inference GPU became a bottleneck when k was larger. However, as the TRY Number increased, throughput improved, and latency decreased progressively. This is attributed to the accumulation of the Disk KV Cache, which increased the likelihood that queries processed by the LLM would reference the same KV Cache, thereby improving the Disk KV Cache hit ratio.

In summary, Configuration (A) demonstrated the viability of shared caching, but it showed limited parallelism – dedicating a GPU for caches helped less than expected when the remaining GPU was overloaded with inference work for larger contexts. However, as the Disk KV Cache accumulated, throughput improved, and latency progressively decreased.

In Configuration (B), as shown in Fig. 11, with caching enabled, overall throughput increased by approximately 15%-28% and latency decreased by 12%–29% compared to baseline in this configuration. For example, as shown in Fig.10, in the k=1 case, as the number of times the dataset was processed increased, the throughput improved from 23.96 to 27.98 QPS, and the average latency fell from 75.75s to 47.92s. This is a significant improvement: 17% higher throughput and 37% lower latency. For k=2, the system with caching went from 14.34 QPS to 17.96 QPS, and latency dropped from 208.22s to 146.17s. That's roughly a 25% increase in throughput and a 30% reduction in latency, respectively.

We note that k=2 queries are inherently slower – even with caching, the latency was higher than any k=1 scenario simply because processing two documents' worth of context takes extra time and resources. However, the relative improvement with caching is still substantial for k=2. These results confirm that Shared RAG-DCache is effective even when additional context is included, although the best absolute performance naturally occurs with fewer documents. Indeed, comparing k=1 vs k=2 across the board, we see that k=1 had lower latency and higher throughput in all configurations (baseline and caching). This is expected because more documents means more work.

Importantly, however, Shared RAG-DCache mitigates the performance penalty of larger k: for instance, going from 1 to 2 documents in the baseline caused throughput to drop by 40% and latency to almost triple, whereas with Shared RAG-DCache

the drop in throughput was less severe and the latency increase was much smaller – and some of that remaining latency was due to the heavier workload rather than idle waiting. Even with k=2, the caching system delivered significantly better performance than baseline.

Comparison of Configurations (A) vs (B): The comparison result is shown in Table V. The CPU-based KV generation (B) clearly emerges as the preferable configuration in our experiments. It achieved the highest throughput in all cases and more consistent latency reductions. In Configuration (A), when one GPU was taken away from inference, the remaining single inference GPU became a choke point under heavier loads (like k=2). It could not parallelize the work enough, and the benefit of offloading some computation to the second GPU was negated by the loss of overall inference capacity. In configuration (B), both GPUs were fully utilized for inference tasks, doubling the inference parallelism, while the CPU handled cache generation without impeding the GPUs. The CPU was effectively leveraging otherwise idle time since GPUs were busy, CPU cycles could be used to precompute caches. This leads to better pipeline balance: GPU power focused on what GPUs do best running the model for answers, and CPU cycles used for background prep work.

As a result, configuration (B) achieved the highest observed throughput in our tests – for example, 27.98 QPS at k=1 with caching, which was higher than even the baseline with two GPUs. It even outperformed Configuration (A)'s throughput despite Configuration (A) having a whole GPU doing cache work, indicating that GPU might have been under-utilized or its benefit was offset by the other GPU's overload. Additionally, the latency in configuration (B) was dramatically better: in Table III, configuration (B) brought average latency down to 47.9s for k=1 and 146.2s for k=2, whereas Configuration (A) had 73.3s (k=1) and a very high 295.4s.

The k=2 result for Configuration (A) suggests that the single GPU was so overwhelmed that queries ended up waiting a long time despite caching – possibly because the KV generation GPU was producing caches faster than the inference GPU could use them, leading to a queue buildup. In contrast, configuration (B) kept latencies much lower by always utilizing both GPUs for serving queries. These findings underscore that offloading KV generation to a non-GPU resource yields better overall system performance, which aligns with our resource allocation optimization strategy.

TABLE V: Average Performance Results by Configuration

Top-k	Avg. Perf.	Conf. (A)	GPU: 2 LLM(Baseline)	Conf. (B)
1	Throughput	24.78	23.96	**27.98**
	Latency	73.25	75.75	**47.92**
2	Throughput	11.57	14.34	**17.96**
	Latency	295.37	208.22	**146.17**

In summary, Shared RAG-DCache provided some performance improvements in both configurations compared to the baseline, Utilizing not only GPU but also CPU resources for cache generation can maximize effectiveness. Our optimal setup in this experiment was configuration (B), which improved throughput by 16.8% and 25.2% for k=1 and 2 respectively, and cut latencies by 36.8% and 29.8%, relative to the no-caching baseline as summarized in Table V. Configuration (A) was suboptimal, showing the importance of a balanced resource allocation when using Shared RAG-Dcache.

V. Discussion

While Shared RAG-DCache demonstrates performance improvements, several aspects need further discussion. The first of these lies in its application within heterogeneous multi-model environments. The KV Caches are inherently model-specific. Different LLM models and distinct tokenization schemes will generate unique KV Cache values even for identical input prompts. This specificity means that a KV Cache generated by one model cannot be directly utilized by another. Consequently, in real-world scenarios where multiple diverse LLMs might be deployed concurrently, the performance benefits of Shared RAG-DCache would be siloed per model type. Extending the system to enable a form of cross-model cache utility is a direction for future research to broaden its applicability.

Further considerations involve group KV caching for multi-document contexts (top-k > 1). Although Shared RAG-DCache was implemented to preserve accuracy, this method still introduces storage overhead due to the growth of document combinations. Its efficiency also relies on the locality of these document groups, which, unlike established individual document locality, is less understood and requires dedicated analysis. Thus, investigating selective caching policies for group caches is needed. Similarly, to confirm broader applicability beyond current evaluations, further research across a wider range of datasets and LLMs is needed. Since LLM and RAG performance varies with factors like data domains, document characteristics, query complexity, and model specifics, these expanded evaluations are necessary. Such testing will assess Shared RAG-DCache's robustness and generalizability, and refine its operational heuristics.

VI. Conclusion

In this paper, We proposed Shared RAG-DCache, a disk-based shared KV cache management system, to optimize LLM inference in multi-instance environments. By leveraging query locality and service queue waiting times, our system prefetches and shares KV caches of frequently accessed documents. This significantly reduces redundant prefill computations, boosts overall throughput, and decreases response latency. Experiments showed throughput improvements up to 70% and notable latency reductions. Optimal performance was achieved when GPUs were dedicated to LLM inference and the CPU handled KV cache generation.

Acknowledgment

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2025-00564249), and by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2022-0-00498, Development of high-efficiency AI computing software core technology for high-speed processing of large-scale learning models).

REFERENCES

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems 33 (NIPS 2020)* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), pp. 9459–9474, Curran Associates Inc., 2020.

[2] S. Siriwardhana, R. Weerasekera, E. Wen, T. Kaluarachchi, R. Rana, and S. Nanayakkara, "Improving the Domain Adaptation of Retrieval Augmented Generation (RAG) Models for Open Domain Question Answering," *Transactions of the Association for Computational Linguistics*, vol. 11, pp. 1–17, 2023.

[3] J. Chen, H. Lin, X. Han, and L. Sun, "Benchmarking Large Language Models in Retrieval-Augmented Generation." https://arxiv.org/abs/2309.01431, 2023.

[4] Q. Fu, M. Cho, T. Merth, S. Mehta, M. Rastegari, and M. Najibi, "LazyLLM: Dynamic Token Pruning for Efficient Long Context LLM Inference," in *Workshop on Efficient Systems for Foundation Models II at ICML 2024*, ICML 2024 Workshop, 2024.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All You Need," in *Advances in Neural Information Processing Systems 30 (NIPS 2017)* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 5998–6008, Curran Associates, Inc., 2017.

[6] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," in *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), pp. 16344–16359, Curran Associates, Inc., 2022.

[7] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ Questions for Machine Comprehension of Text," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2383–2392, Association for Computational Linguistics, 2016.

[8] Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. W. Cohen, R. Salakhutdinov, and C. D. Manning, "HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2369–2380, Association for Computational Linguistics, 2018.

[9] M. Joshi, E. Choi, D. S. Weld, and L. Zettlemoyer, "TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, Association for Computational Linguistics, 2017.

[10] C. Jin, Z. Zhang, X. Jiang, F. Liu, X. Liu, X. Liu, and X. Jin, "RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation." https://arxiv.org/abs/2404.12457, 2024.

[11] S. Lu, H. Wang, Y. Rong, Z. Chen, and Y. Tang, "Turborag: Accelerating retrieval-augmented generation with precomputed kv caches for chunked text." https://arxiv.org/abs/2410.07590, 2024.

[12] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang, "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion," in *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*, (New York, NY, USA), pp. 94–109, Association for Computing Machinery, 2025.

[13] S. Agarwal, S. Sundaresan, S. Mitra, D. Mahapatra, A. Gupta, R. Sharma, N. J. Kapu, T. Yu, and S. Saini, "Cache-Craft: Managing Chunk-Caches for Efficient Retrieval-Augmented Generation." https://arxiv.org/abs/2502.15734, 2025.

[14] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, (New York, NY, USA), pp. 611–626, Association for Computing Machinery, 2023.

[15] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast Distributed Inference Serving for Large Language Models." https://arxiv.org/abs/2305.05920, 2023.

[16] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis, "Efficient Streaming Language Models with Attention Sinks," in *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[17] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, (Carlsbad, CA), pp. 521–538, USENIX Association, July 2022.

[18] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving," in *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*, (Santa Clara, CA), pp. 193–210, USENIX Association, July 2024.

[19] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2022.

[20] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense Passage Retrieval for Open-Domain Question Answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (B. Webber, T. Cohn, Y. He, and Y. Liu, eds.), (Online), pp. 6769–6781, Association for Computational Linguistics, Nov. 2020.

[21] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021.

[22] W. Wang, H. Bao, S. Huang, L. Dong, and F. Wei, "MiniLMv2: Multi-Head Self-Attention Relation Distillation for Compressing Pretrained Transformers." https://arxiv.org/abs/2012.15828, 2020.

[23] D. Zhang, Y. Luo, Y. Wang, X. Kui, and J. Ren, "BatOpt: Optimizing GPU-Based Deep Learning Inference Using Dynamic Batch Processing," *IEEE Transactions on Cloud Computing*, vol. 12, no. 1, pp. 174–185, 2024.

[24] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models." https://arxiv.org/abs/2302.13971, 2023.

[25] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models." https://arxiv.org/abs/2205.01068, 2022.