RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation

Chao Jin[1] Zili Zhang[1] Xuanlin Jiang[1] Fangyue Liu[1]
Xin Liu[2] Xuanzhe Liu[1] Xin Jin[1]
[1]*Peking University* [2]*ByteDance Inc.*

Abstract

Retrieval-Augmented Generation (RAG) has shown significant improvements in various natural language processing tasks by integrating the strengths of large language models (LLMs) and external knowledge databases. However, RAG introduces long sequence generation and leads to high computation and memory costs. We propose RAGCache, a novel multilevel dynamic caching system tailored for RAG. Our analysis benchmarks current RAG systems, pinpointing the performance bottleneck (i.e., long sequence due to knowledge injection) and optimization opportunities (i.e., caching knowledge's intermediate states). Based on these insights, we design RAGCache, which organizes the intermediate states of retrieved knowledge in a knowledge tree and caches them in the GPU and host memory hierarchy. RAGCache proposes a replacement policy that is aware of LLM inference characteristics and RAG retrieval patterns. It also dynamically overlaps the retrieval and inference steps to minimize the end-to-end latency. We implement RAGCache and evaluate it on vLLM, a state-of-the-art LLM inference system and Faiss, a state-of-the-art vector database. The experimental results show that RAGCache reduces the time to first token (TTFT) by up to 4× and improves the throughput by up to 2.1× compared to vLLM integrated with Faiss.

1 Introduction

Recent advancements in large language models (LLMs) like GPT-4 [35], LLaMA2 [41], and PalM [13] have significantly enhanced performance across various natural language processing (NLP) tasks, including question answering, summarization, and translation [39, 51, 52]. Retrieval-augmented generation (RAG) [1, 27] further enhances LLMs by incorporating contextually relevant knowledge from external databases, such as Wikipedia [5], to improve the generation quality. With informative external knowledge, RAG have achieved comparable or even better performance than LLMs fine-tuned for specific downstream tasks [10].

For an RAG request, the RAG system first retrieves relevant documents from the knowledge database. The documents are typically represented as feature vectors in a vector database through embedding models, and the retrieval step is implemented by vector similarity search. Then, RAG injects the retrieved documents (i.e., external knowledge) into the original request and feeds the augmented request to the LLM for generation. With the help of the retrieved documents, RAG expands LLMs' knowledge base and contextual understanding, thereby improving the generation quality [10].

With knowledge injection, RAG introduces long sequence generation for the augmented request, which leads to high computation and memory costs. For instance, the initial request contains 100 tokens, and the retrieved documents may contain 1000 tokens in total. Consequently, the extra computation and memory costs for the augmented request are >10× higher than the original request. This escalation in resource requirements poses a substantial challenge in scaling systems for efficient processing of RAG requests.

Recent work [26, 57], focusing on system optimizations of LLM inference, has made significant progress in sharing the intermediate states of LLM inference to reduce recomputation costs. vLLM [26] manages the intermediate states in non-contiguous memory blocks to allow fine-grained memory allocation and state sharing for a single request's multiple generation iterations. SGLang [57] identifies the reusable intermediate states across different requests for LLM applications like multi-turn conversations and tree-of-thought [48]. However, these efforts only optimize for LLM inference without considering the characteristics of RAG. They cache the intermediate states in GPU memory, which has limited capacity considering the long sequences in augmented requests, leading to suboptimal performance.

We conduct a system characterization of RAG, which measures the performance of current RAG systems under various datasets and retrieval settings with representative LLMs. Our analysis highlights a significant performance limitation rooted in the processing of augmented sequences due to document injection. In addition, we uncover two potential *opportunities* for system optimizations to mitigate this constraint. First, the recurrence of identical documents across multiple requests enables the sharing of LLM inference's intermediate states for such documents. Second, a small fraction of documents accounts for the majority of retrieval requests. This allows us to cache the intermediate states of these frequently accessed documents to reduce the computational burden.

To this end, we propose RAGCache, a novel multilevel dynamic caching system tailored for RAG. RAGCache is the first system to cache the intermediate states of retrieved documents (i.e., external knowledge) and share them across multiple requests. The core of RAGCache is a knowledge tree that adapts the intermediate states of the retrieved documents to the GPU and host memory hierarchy. Documents with more frequent accesses are cached in the fast GPU memory and those with fewer accesses are cached in the slow host memory. There are mainly two challenges in designing the caching system for RAG.

First, RAG systems are sensitive to the referred order of the retrieved documents. For instance, there are two documents D_1 and D_2 and two requests Q_1 and Q_2. Let Q_1's and Q_2's relevant documents be $[D_1, D_2]$ and $[D_2, D_1]$, respectively, where $[D_1, D_2]$ means D_1 is more relevant than D_2. The intermediate states (i.e., the key value tensors) of $[D_1, D_2]$ are different from that of $[D_2, D_1]$ because the key-value tensor of the new token is calculated based on the preceding tokens in the attention mechanism of LLMs [44]. Unfortunately, we also cannot swap the order of D_1 and D_2. As recent efforts have shown, the generation quality of the LLMs will be affected by the referred order [9, 30]. We use knowledge tree to organize the intermediate states of the retrieved documents in the GPU and host memory hierarchy and design a prefix-aware Greedy-Dual-Size-Frequency (PGDSF) replacement policy that comprehensively considers the document order, size, frequency, and recency to minimize the miss rate. We also propose a cache-aware request scheduling approach to further improve the hit rate when the request rate is high.

Second, vector retrieval (processed on CPU) and LLM inference (processed on GPU) are two independent steps in RAG. The two steps are executed sequentially in the current RAG systems, leading to idle GPU resources during retrieval and long end-to-end latency. We propose a dynamic speculative pipelining strategy to dynamically overlap the computation of the two steps and minimize the end-to-end latency while keeping the system load under control.

We implement a RAGCache prototype and evaluate it on various datasets and representative LLMs. The experimental results show that RAGCache outperforms the state-of-the-art solution, vLLM [26] integrated with Faiss [4], by up to 4× on time to first token (TTFT) and improves the throughput by up to 2.1×. Compared to SGLang [57], which reuse the intermediate states in GPU memory, RAGCache reduces the TTFT by up to 3.5× and improves the throughput by up to 1.8×.

In summary, we make the following contributions.

- We conduct a detailed system characterization of RAG, which reveals the performance bottleneck and optimization opportunities.
- We propose RAGCache, to the best of our knowledge, the first RAG system that caches the intermediate states of external knowledge and shares them across multiple queries to reduce the redundant computation.
- We design a prefix-aware GDSF replacement policy that leverages the characteristics of RAG to minimize the miss rate and a dynamic speculative pipelining approach to minimize the end-to-end latency.

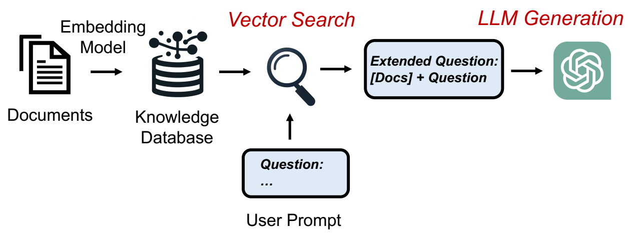


Figure 1. RAG workflow.

- We implement a RAGCache prototype. The evaluation shows that RAGCache outperforms vLLM integrated with Faiss by up to 4× on TTFT and 2.1× on throughput. Compared to the state-of-the-art caching system for LLM applications, SGLang, RAGCache achieves up to 3.5× lower TTFT and up to 1.8× higher throughput.

2 Background

Retrieval-Augmented Generation (RAG) represents a significant advancement in the field of natural language processing (NLP) and machine learning, combining LLMs with the vast information accessible in external knowledge databases. Specifically, RAG is employed to enhance the generative models' ability to produce more accurate, relevant, and contextually rich responses by dynamically retrieving information from a corpus during the generation process. This hybrid approach combines the strengths of two major strands: the deep contextual understanding of LLMs and the precision of knowledge database retrieval. Recent work [1, 8, 22, 27, 37, 42] has demonstrated that RAG can significantly improve the generation quality across various benchmarks compared to solely generative models. The RAG framework has since been applied across various tasks, including question answering [39], content creation [24], and even code generation [33, 43], showcasing its versatility and promise.

As shown in Figure 1, RAG operates on a two-step workflow: *retrieval* and *generation*, integrating offline preparation with real-time processing for enhanced performance. Initially, in its offline phase, RAG transforms the external knowledge sources, such as documents, into high-dimensional vectors using advanced embedding models. RAG then indexes these vectors into a specialized vector database designed for efficient retrieval. Upon receiving a user request, RAG first accesses this vector database to conduct a vector similarity search, retrieving the documents that best match the request based on their semantic content. Following this, RAG combines the content of these retrieved documents with the original user request, creating an augmented request. This augmented request is then provided to an LLM, which leverages the combined information to generate a response that is more informed and contextually rich.

In an RAG workflow, the retrieval step is mainly performed on CPUs, while the generation step is executed on GPUs. From a system perspective, the end-to-end performance of

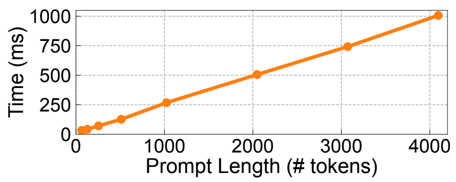


Figure 2. Inference time with different input lengths.

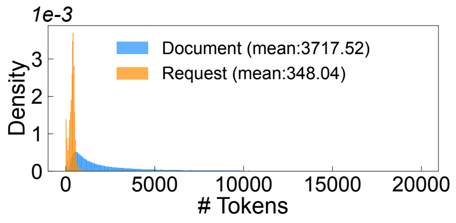


Figure 3. The distribution of token number.

RAG is affected by both the retrieval or generation steps. The retrieval time is mainly determined by the vector database's scale, and the generation time is decided by the model size and the sequence length. Our subsequent characterization will identify RAG's performance bottleneck and highlight potential areas for optimization.

3 RAG System Characterization

In this section, we conduct a comprehensive system characterization. First, we identify the performance bottlenecks for RAG, i.e., the LLM generation step. Next, we evaluate the performance improvement of caching the intermediate states of the retrieved knowledge. Finally, we analyze the question patterns and the potential for caching optimization.

3.1 Performance Bottleneck

LLM inference can be divided into two distinct phases: prefill and decoding. The prefill phase involves computing the key-value tensors of the input tokens, while the decoding phase generates the output token in an auto-regressive manner based on the previously generated key-value tensors. The prefill phase is particularly time-consuming, as it requires to compute the entire input sequence's key-value tensors.

As discussed in § 2, RAG comprises two steps: retrieval and generation. Recent work [53, 54] shows that the retrieval step executes in milliseconds per request with a high accuracy for billion-scale vector databases. Meanwhile, the generation step, conducted on GPUs, is heavily affected by sequence length and model size. To identify the performance bottleneck, we evaluate the inference time with fixed output length and different input lengths on LLaMA2-7B, the smallest model in the LLaMA2 series [41]. Larger models will have longer inference time. The backend system is vLLM [26] equipped with one NVIDIA A10G GPU. Figure 2 shows that the inference time, mainly dominated by the prefill phase, increases rapidly with sequence length and reaches one second when the sequence length is larger than 4000 tokens.

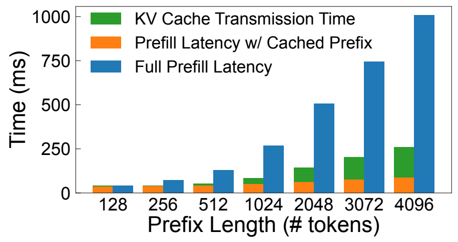


Figure 4. Prefill latency characterization.

The sequence length in the LLM generation step is the token number of the original request plus the retrieved document. We generate a document dataset based on the Wikipedia corpus [5] with ~0.3 million documents from most popular Wikipedia pages. Figure 3 demonstrates the distribution of the document length and the request length. The document length is significantly longer than the request length of the MMLU dataset [18]. With an average document length of 3718 tokens, the corresponding inference time is markedly higher than the retrieval step in most cases. Notably, the retrieval step may take comparable time to the generation step when users require relevant knowledge with extremely high accuracy [54], which necessitates extensive searching in the vector database and further complicates the performance bottleneck.

3.2 Optimizations Opportunities

Caching knowledge. The generation step's performance bottleneck primarily arises from processing the long sequence's key-value tensors in attention blocks. A simple yet effective optimization for RAG involves caching these key-value tensors of previously retrieved documents. For example, let requests, Q_1 and Q_2, both refer to the same document, D_1. If Q_1 arrives first, the key-value tensors of D_1 are computed, and we can cache the key-value tensors. When Q_2 arrives, we can reuse the cached key-value tensors to reduce the prefill latency of Q_2. The average prefill latency with caching is calculated as follows:

$$\begin{aligned}&\mathit{Prefill\ Latency} = \mathit{Miss\ Rate} \times \mathit{Full\ Prefill\ Latency}\\&+ (1 - \mathit{Miss\ Rate}) \times \mathit{Cache\ Hit\ Latency}\end{aligned}$$

To explore caching's optimization opportunity, we consider three crucial factors: full prefill computation, cache hit, and miss rate.

Full prefill computation. To quantify the full computation, we compare the LLM prefill phase's latency with and without caching these partial intermediate states (i.e., the key-value cache of prefixes). We set the original request length to 32 tokens and vary the prefix length from 128 to 4096 tokens. Figure 4 illustrates that the prefill latency is significantly reduced when caching is employed. In the scenario of cached prefix, only the request tokens' key-value tensors are computed. Conversely, in the scenario of full prefill computation, the key-value tensors of the entire sequence need to be calculated. The full prefill latency is up to 11.5× lower than that in the cached prefix scenario. These results underscore the substantial performance improvement achieved by caching intermediate states of accessed documents.

Cache hit. The cache hit comprises two components: prefill computation of the request tokens and loading the key-value cache of the retrieved documents. The former is negligible compared to miss penalty. As for the latter one, the limited GPU memory contrasts sharply with the substantial size of the key-value cache from retrieved documents. This discrepancy necessitates leveraging the host memory to extend the caching system, accommodating a greater volume of documents. However, this introduces a potential overhead: the transmission of key-value cache between the GPU and host memory. To assess this, we conduct an evaluation of the transmission overhead. Figure 4 adds the KV cache transmission time with the given prefix length to the prefill time with cached prefix, representing the cache hit latency. Even with the transmission overhead, the cache hit latency is significantly better (up to 3.9× lower) than the full prefill latency, highlighting the advantages of caching intermediate states of retrieved documents.

Miss rate. The final consideration lies in the retrieval pattern of RAG systems. The cache performance is dominated by the miss rate, which is directly influenced by the retrieval pattern. For example, a 100% cache miss rate occurs when each request retrieves a unique document. In such a scenario, caching intermediate states of retrieved documents is meaningless. We analyze the document retrieval pattern in four representative question-answering datasets for RAG: MMLU [18], Google Natural Questions [25], HotpotQA [47], and TriviaQA [23]. We convert the documents on Wikipedia to vectors through `text-embedding-3-small` model [3] from OpenAI [2] for retrieval. The number of documents referred by one request is top-1. The ANN index is FlatL2, i.e., exact search on the entire dataset with Euclidean distance. Figure 5 shows the CDF of the accessed documents. We observe that the retrieval pattern is skewed, with a small fraction of documents accounting for the majority of retrieval requests. For example, the top 3% documents are referred to by 60% requests in the MMLU dataset, which is 20× less than the uniform distribution. This observation reveals a low miss rate to cache the frequently accessed documents.

Further analysis on additional embedding models and ANN indexes for vector search is shown in Figure 6. All of the results exhibit a similar retrieval pattern no matter which embedding model or ANN index is used. The results are consistent with FlatL2 index, which indicates the potential for caching optimization under different settings.

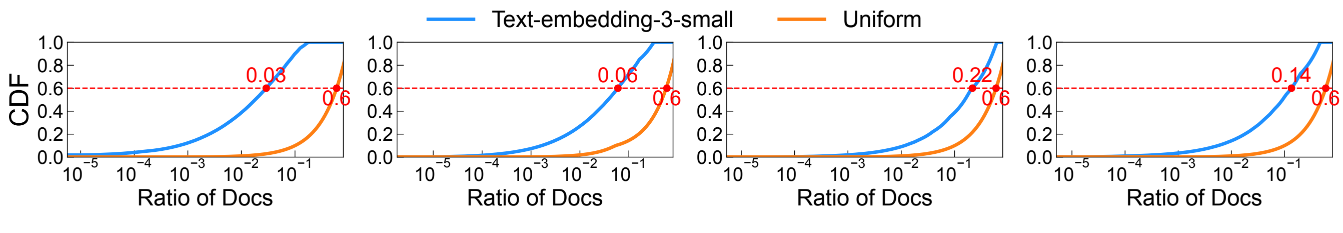
4 RAGCache Overview

We present RAGCache, a novel multilevel dynamic caching system tailored for RAG. RAGCache caches the key-value tensors of retrieved documents across multiple requests to minimize redundant computation. The core of RAGCache is a knowledge tree with a prefix-aware Greedy Dual-Size Frequency (PGDSF) replacement policy that ensures caching the most critical key-value tensors. RAGCache also implements a global RAG controller that orchestrates interactions between the external knowledge database and LLM inference engine. The controller is enhanced by system optimizations including cache-aware reordering and dynamic speculative pipelining.

Architecture overview. We provide a brief overview of RAGCache in Figure 7. When a request arrives, the RAG controller first retrieves the relevant documents from the external knowledge database. These documents are then forwarded to the cache retriever to locate matching key-value tensors. If the key-value tensors are absent from the cache, RAGCache directs the LLM inference engine to produce new tokens. Conversely, if the tensors are available, the request with the key-value tensors are forwarded to the LLM inference engine, which then employs a prefix caching kernel for token generation. After generating the first token, the key-value tensors are relayed back to the RAG controller, which caches the tensors from the accessed documents and refreshes the cache's status. Finally, the generated answer is delivered to the user as the response.

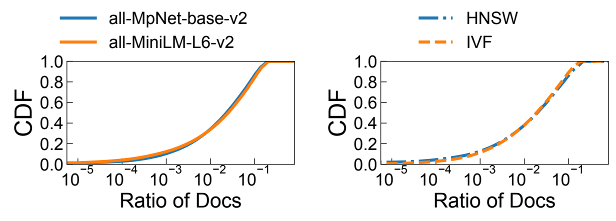
Cache retriever. The cache retriever efficiently locates the key-value tensors for documents stored in the in-memory cache, utilizing a knowledge tree to organize these tensors. This tree, structured as a prefix tree based on document IDs, aligns with the LLM's position sensitivity to the document order. Each path within this tree represents one specific sequence of documents referenced by a request, with each node holding the key-value tensor of a referred document. Different paths may share the same nodes, which indicates the shared documents across different requests. This structure enables the retriever to swiftly access the key-value tensors of documents in their specified order.

RAG controller. The RAG controller orchestrates the interactions with some system optimizations tailored for RAG. Prefix-aware Greedy Dual-Size Frequency (PGDSF) policy is employed to minimize the cache miss rate. PGDSF calculates a priority based on the frequency, size of key-value tensors, last access time, and prefix-aware recomputation cost. The cache eviction is determined by the priority, which ensures the most valuable tensors are retained. Cache-aware reordering schedules the requests to improve cache hit rate and prevent thrashing, while also ensuring request fairness to mitigate starvation issues. Dynamic speculative pipelining is designed to overlap the knowledge retrieval and LLM inference to minimize the latency. This optimization leverages the mid-process generation of retrieval results to initiate LLM inference early.



(a) MMLU. **(b)** Google Natural Questions. **(c)** HotpotQA. **(d)** TriviaQA.

Figure 5. Retrieval pattern on different datasets.



(a) Different embedding models. **(b)** Different ANN indexes.

Figure 6. Retrieval pattern under different settings.

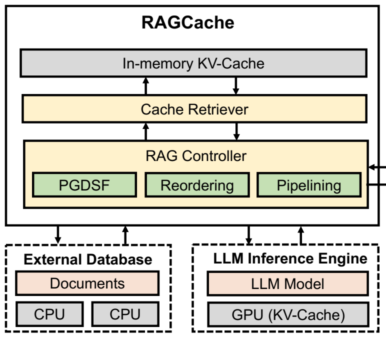


Figure 7. RAGCache overview.

5 RAGCache Design

In this section, we present the design of RAGCache. We first introduce the cache structure and the prefix-aware replacement policy (§5.1). Then, we describe the cache-aware reordering strategy to improve the cache hit rate (§5.2). Finally, we present the dynamic speculative pipelining approach to overlap knowledge retrieval and LLM inference (§5.3).

5.1 Cache Structure and Replacement Policy

Different from traditional cache systems that cache individual objects, RAGCache caches the key-value tensors of the retrieved documents that are sensitive to the referred order. For example, consider two document sequences: $[D_1, D_3]$ with key-value tensors KV and $[D_2, D_3]$ with KV'. Although $KV[1]$ and $KV'[1]$ both pertain to D_3, they are different in values. This discrepancy arises because the key-value tensor for a given token is generated based on the preceding tokens, underscoring the order-dependence of key-value tensors.

To facilitate fast retrieval while maintaining the document order, RAGCache structures the documents' key-value tensors with a knowledge tree, as depicted in Figure 8. This tree assigns each document to a node, which refers to the

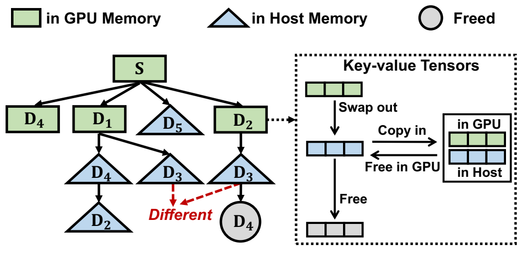


Figure 8. Knowledge tree.

memory addresses of the document's key-value tensors. Following vLLM [26], RAGCache stores the key-value tensors in non-continuous memory blocks for KV cache reuse. The root node S denotes the shared system prompt. A path from the root to a particular node represents a sequence of documents.

This design inherently allows RAGCache to serve multiple requests simultaneously through overlapping paths in the tree. RAGCache retrieves tensors by prefix matching along these paths. During the prefix matching process, if a subsequent document is not located among the child nodes, the traversal is promptly terminated, and the identified document sequence is returned. This method ensures efficiency with a time complexity of $O(h)$, where h represents the tree's height.

Prefix-aware Greedy-Dual-Size-Frequency (PGDSF) replacement policy. With the knowledge tree, RAGCache has to decide each node's placement within a hierarchical cache. Nodes that are accessed more frequently are ideally stored in GPU memory for faster access speeds, while those accessed less often are allocated to the slower host memory or simply freed. To optimize node placement, RAGCache employs a prefix-aware Greedy-Dual-Size-Frequency (PGDSF) replacement policy, which is based on the classic GDSF policy [12]. Unlike traditional caching strategies such as LRU, which neglect the variable sizes of documents, PGDSF evaluates each node based on its access frequency, size, and access cost. This method utilizes limited storage capacity by maintaining the most beneficial nodes, whose *priority* is defined as follows:

$$Priority = Clock + \frac{Frequency \times Cost}{Size} \tag{1}$$

Nodes with lower priority are evicted first. *Clock* tracks node access recency. We maintain two separate logical clocks in the RAG controller for GPU and host memory, respectively, to adapt to the cache hierarchy. Each clock starts at zero and

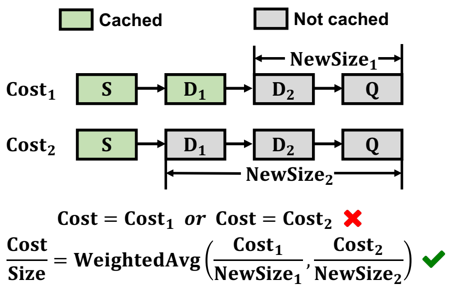

Cached
Not cached
NewSize₁
Cost₁
S
D₁
D₂
Q
Cost₂
NewSize₂
Cost = Cost₁ or Cost = Cost₂ ✗
Cost/Size = WeightedAvg(Cost₁/NewSize₁, Cost₂/NewSize₂) ✓

Figure 9. Cost estimation in PGDSF.

updates with every eviction. When a document is retrieved, its node's clock is set and its priority is adjusted. Nodes with older clock, indicating less recent use, receive lower priorities. Let E be the set of evicted nodes in one eviction operation. The clock is updated accordingly:

$$Clock = \max_{n \in E} Priority(n) \tag{2}$$

Frequency represents the total retrieval count for a document within a time window. This count is reset to zero upon system start or cache clearance. The priority is proportional to the frequency, and thus more frequently accessed documents have higher priorities. *Size* reflects the number of tokens in a document post-tokenization, directly influencing the memory required for its key-value tensors. *Cost*, defined as the time taken to compute a document's key-value tensors, varies with GPU computational capacity, document size, and the sequence of preceding documents.

PGDSF achieves prefix awareness for RAG systems in two aspects: *Cost* estimation and node placement. Unlike GDSF, where costs are straightforward (e.g., object size in web caching), RAG costs involve complex LLM generation dynamics. For example, Figure 9 shows varying costs incurred by the same request denoted as $[S, D_1, D_2, Q]$. To estimate the cost for D_2, directly using the the cost where $[S, D_1]$ is cached or only S is cached is imprecise. Besides, the latter case's cost also includes the time to compute the key-value tensors for D_1 and Q. PGDSF addresses this problem by replacing $Cost/Size$ in Formula 1 as follows:

$$\frac{Cost}{Size} = \frac{1}{m} \sum_{i=1}^{m} \frac{Cost_i}{NewSize_i} \tag{3}$$

where m is the number of requests that access the document but do not have the document cached. $Cost_i/NewSize_i$ represents the compute time per non-cached token for the i-th request. Such estimation inherently considers the document size by amortizing the cost to all non-cached tokens. As for $Cost_i$, RAGCache profiles the LLM prefill time with varying cached and non-cached token lengths offline and uses bilinear interpolation to estimate the cost for a given request. Document retrieval triggers an update in node frequency, cost estimation and clock within the knowledge tree, or initiates a new node for documents not previously cached.

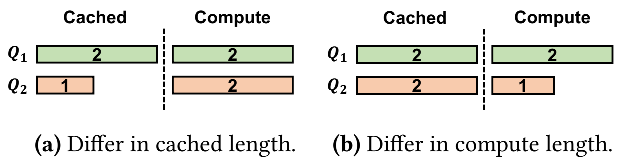
Algorithm 1 Knowledge Tree Operations

```
1:  function UPDATE_NODE_IN_GPU($node$, $is\_cached$, $\alpha$, $\beta$)
2:      // $\alpha$ and $\beta$ are cached and non-cached sizes of the request
3:      $node.Frequency \leftarrow node.Frequency + 1$
4:      if $is\_cached$ is false then
5:          // Bilinear interpolation to estimate the cost
6:          Find $\alpha_l < \alpha < \alpha_h$ and $\beta_l < \beta < \beta_h$ from the profiler
7:          $T_l \leftarrow T(\alpha_l, \beta_l) + \frac{\alpha - \alpha_l}{\alpha_h - \alpha_l} \cdot [T(\alpha_h, \beta_l) - T(\alpha_l, \beta_l)]$
8:          $T_h \leftarrow T(\alpha_l, \beta_h) + \frac{\alpha - \alpha_l}{\alpha_h - \alpha_l} \cdot [T(\alpha_h, \beta_h) - T(\alpha_l, \beta_h)]$
9:          $T(\alpha, \beta) \leftarrow T_l + \frac{\beta - \beta_l}{\beta_h - \beta_l} \cdot (T_h - T_l)$
10:         $node.TotalCost \leftarrow node.TotalCost + \frac{T(\alpha,\beta)}{\beta}$
11:         $node.numComputed \leftarrow node.numComputed + 1$
12:         $node.AvgCost \leftarrow \frac{node.TotalCost}{node.numComputed}$
13:     $node.Priority \leftarrow Clock + node.AvgCost \times node.Frequency$
14:
15: function EVICT_IN_GPU($required\_size$)
16:     $E \leftarrow \emptyset$ // Evicted nodes in GPU
17:     $S \leftarrow \{n \in GPU \wedge n.Children \notin GPU\}$ // Leaf nodes in GPU
18:     while $\sum_{n \in E} n.Size < required\_size$ do
19:         $n \leftarrow \arg\min_{n \in S} n.Priority$
20:         $E \leftarrow E \cup \{n\}$
21:         $Clock \leftarrow \max\{Clock, n.Priority\}$
22:         if $n.Parent.Children \notin GPU$ then
23:             $S \leftarrow S \cup \{n.Parent\}$
```

PGDSF orchestrates node placement in the knowledge tree, which is divided into GPU, host, and free segments, as illustrated in Figure 8. Nodes in GPU memory serve as parent nodes to those in host memory, establishing a hierarchical structure. RAGCache dynamically manages node eviction across these segments for efficiency. Specifically, when the GPU memory is full, RAGCache swaps the least priority node in leaf nodes to the host memory. RAGCache applies a similar process for host memory oversubscription. This eviction strategy upholds the tree's hierarchical partitioning, which is pivotal for aligning with the memory hierarchy and prefix sensitivity in LLM generation. A node relies on its parent node for key-value tensor calculation, emphasizing the need for prioritizing parent node placement for rapid retrieval.

Algorithm 1 outlines the operations for updating and evicting nodes in the GPU memory of the knowledge tree. $T(\alpha, \beta)$ represents the estimated compute time for a request with α cached tokens and β non-cached tokens. Upon a document retrieval by a request, RAGCache updates the cost using bilinear interpolation (line 6–9) if the document is not cached, EVICT_IN_GPU evicts nodes from the GPU memory to accommodate new requests and updates the clock according to Formula 2. If a parent node becomes a leaf following eviction, it is added to the candidate set S.

Swap out only once. The GPU connects to the host memory via the PCIe bus, which typically offers much lower bandwidth than the GPU HBM. To minimize data transfer between the GPU and host memory, RAGCache adopts a



(a) Differ in cached length. **(b)** Differ in compute length.

Figure 10. Cache-aware reordering.

swap-out-only-once strategy as shown in Figure 8. The key-value tensors of a node are swapped out to the host memory only for the first eviction. The host memory keeps the key-value tensors until the node is evicted from the whole cache. For subsequent evictions in GPU memory, RAGCache directly frees the node with zero data copy. Given that the capacity of the host memory is one or two orders of magnitude larger than the GPU memory, keeping one copy of the key-value tensors in the host memory is acceptable.

5.2 Cache-aware Reordering

Cache hit rate is vital for RAGCache's cache efficiency, yet the unpredictable arrival pattern of user requests results in substantial cache trashing. The requests referring to the same documents may not be issued together, affecting cache efficiency. For illustration, let requests $\{Q_i, i\%2 == 0\}$ and $\{Q_i, i\%2 == 1\}$ target documents D_1 and D_2, respectively. The cache capacity is one document. The sequence $\{Q_1, Q_2, Q_3 ...\}$ causes frequent swapping of the key-value cache of D_1 and D_2, rendering a zero cache hit rate. Conversely, rearranging requests to $\{Q_1, Q_3, Q_5, Q_2, Q_4, Q_6, Q_7, \ldots\}$ optimizes cache utilization, which improves the hit rate to 66%. This exemplifies how strategic request ordering can mitigate cache volatility and enhance cache efficiency.

Before introducing the cache-aware reordering algorithm, we first consider two scenarios to illustrate the key insights. We assume that the recomputation cost is proportional to the recomputation length in this example. The first scenario (Figure 10a) considers requests with identical recomputation demands but varying cached context lengths, under a cache limit of four. With an initial order of $\{Q_1, Q_2\}$, the system must clear Q_2's cache space to accommodate Q_1's computation, then reallocate memory for Q_1's processing. It effectively utilizes Q_1's cache while discarding Q_2's. This results in a total computation cost of $2 + 1 + 2 = 5$. Conversely, ordering as Q_2, Q_1 utilizes Q_2's cache but discards Q_1's, which increases computation to $2 + 2 + 2 = 6$. Thus, cache-aware reordering advocates prioritizing requests with larger cached contexts to enhance cache efficiency, as they bring larger benefits.

In the second scenario, we examine requests with identical cached context lengths but varying recomputation demands, given a cache capacity of five. For a sequence $\{Q_1, Q_2\}$, the system must clear Q_2's cache to allocate space for Q_1's computation, given only one available memory slot. This necessitates recomputing Q_2 entirely, resulting in a computation cost of $2 + 2 + 1 = 5$. In contrast, the sequence $\{Q_2, Q_1\}$ allows for direct computation of Q_2 due to adequate cache availability. It reduces the total computation to $2 + 1 = 3$. Hence, cache-aware reordering is beneficial when it prioritizes requests with shorter recomputation segments, as this approach minimizes the adverse effects on cache efficiency.

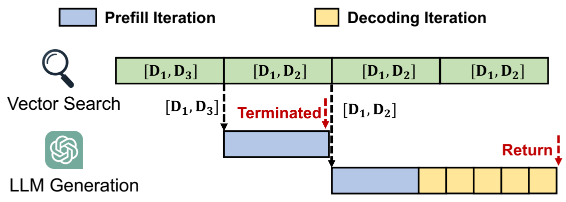


Figure 11. Speculative pipelining.

Drawing from these insights, we introduce a cache-aware reordering algorithm aimed at improving cache efficiency. RAGCache employs a priority queue for managing incoming requests, prioritizing them based on their impact on cache performance. Specifically, requests are selected for processing based on a priority metric, defined as:

$$OrderPriority = \frac{Cached\ Length}{Computation\ Length}$$

This formula prioritizes requests that are likely to enhance cache efficiency—–those with a larger cached portion relative to their computation needs. By adopting this cache-aware reordering, RAGCache increases the cache hit rate and decreases the total computation time, optimizing resource use and system performance. To avoid starvation, RAGCache sets a window for each request to ensure that all requests are processed no later than the window size.

5.3 Dynamic Speculative Pipelining

As we discuss in §3.1, the LLM generation is the key performance bottleneck in RAG systems. However, if the vector database grows to a larger scale or the retrieving requires a higher accuracy, the retrieval step may incur a substantial latency. To mitigate the impact of retrieval latency, RAGCache employs dynamic speculative pipelining to overlap knowledge retrieval and LLM inference. The key insight behind this technique is that the vector search may produce the final results early in the retrieval step, which can be leveraged by LLM for speculative generation ahead of time.

Specifically, vector search maintains a queue of top-k candidate documents, which are ranked by their similarity to the request. During the retrieval process, the top-k documents in the queue are continuously updated i.e., some documents with greater similarity are inserted into the queue. However, the final top-k documents may emerge early in the retrieval step [28, 53]. Based this observation, RAGCache introduces a speculative pipelining strategy that splits a request's retrieval process into several stages. In each stage, RAGCache ticks the vector database to send the candidate documents to the LLM engine for speculative generation. Then, the LLM

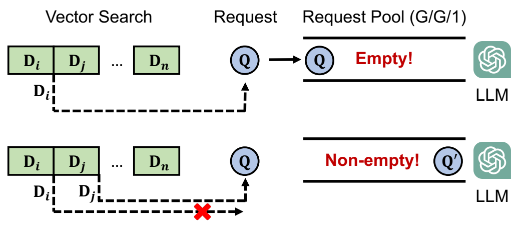


Figure 12. The optimal speculative pipelining strategy in the simplified analysis.

engine starts a new speculative generation and terminates the previous generation if the received documents are different from the previous ones. If there is no difference, the LLM engine remains processing the previous generation. When the final top-k documents are produced, RAGCache sends the final results to the LLM engine. At this moment, the LLM engine returns the results of the latest speculative generation to users if it matches the final top-k documents. Otherwise, the LLM engine performs re-generation.

As shown in Figure 11, we split the retrieval process into four stages. The top-2 documents in candidate queue are $[D_1, D_3]$, $[D_1, D_2]$, $[D_1, D_2]$, and $[D_1, D_2]$ in the four stages. After stage one is finished, RAGCache sends $[D_1, D_3]$ to the LLM engine for speculative generation. When stage two is finished, RAGCache sends $[D_1, D_2]$ to the LLM engine. The LLM engine finds that $[D_1, D_3]$ are different from $[D_1, D_2]$, and thus terminates the previous speculative generation and starts a new one. As for stage three, the LLM engine receives the same documents as stage two, and thus remains processing the previous generation. After the final stage, RAGCache sends the final top-2 documents to the LLM engine which is the same as the latest speculative generation. The LLM engine directly returns the speculative generation results to users.

Dynamic speculative generation. The speculative pipelining allows RAGCache to overlap the retrieval and generation steps, which reduces the end-to-end latency of RAG systems. However, it may introduce extra LLM computation as some speculative generations are incorrect, potentially leading to performance degradation under high system loads. To address this problem, RAGCache dynamically enables speculative pipelining based on the system load. We begin with a simplified analysis to demonstrate how to minimize the end-to-end latency while keeping the system load under control.

For simplicity, we assume the vector search and the LLM both serve one request at a time. The vector search produces candidate retrieval results at the end of each stage with a fixed time interval d. Since the batch size is one, we can immediately terminate any incorrect speculative generation request. We model the LLM engine with a G/G/1 queue [38]. Since priority scheduling is widely used in LLM serving (e.g., MLFQ in FastServe [45] and cache-aware reordering in RAGCache), we assume that the LLM engine can schedule

Algorithm 2 Dynamic Speculative Pipelining Strategy

```
1:  function DYNAMIC_SPECULATIVE_PIPELINING($request$)
2:      $D \leftarrow []$
3:      while the vector search of $request$ is not finished do
4:          // Produce the candidate documents at the next stage
5:          $D_{temp} \leftarrow$ VECTOR_SEARCH($request, D$)
6:          if $D_{temp} \neq D$ then
7:              if $\{request, D\}$ in $pool$ then
8:                  Terminate $\{request, D\}$ after the current iteration
9:              if $pool.size < max\_prefill\_bs$ then
10:                 $pool.insert(\{request, D_{temp}\})$
11:         $D \leftarrow D_{temp}$
```

the requests in the queue with arbitrary order but executes the speculation generation requests from a single request in order. Figure 12 shows the optimal speculative pipelining strategy under this setting, which we summarize as a theorem below.

Theorem 5.1. *Let d be the speculative time interval and T be the serving time distribution for the LLM in the G/G/1 request pool, satisfying $T \geq d$ for all serving events. The optimal strategy is: start a speculative generation if the request pool is empty after a stage ends with a different document sequence; if not, continue vector search until the pool empties.*

Proof. After the i-th stage, the vector search produces a document sequence D_i. There are four cases.

(1) The pool is empty and D_i is the final result. In this case, deferring the speculative generation incurs additional waiting time, while starting the speculative generation immediately utilizes the LLM generation engine. As a result, starting a speculative generation leads to a lower latency.

(2) The pool is non-empty and D_i is not the final result. In this case, inserting the speculative generation may delay the request in the pool due to the reordering policy. Furthermore, computing the speculative generation is unnecessary. In such case, deferring the speculative generation is more efficient.

(3) The pool is non-empty and D_i is the final result. In this case, inserting the speculative generation into the LLM request pool allows the LLM engine scheduler to conduct more efficient scheduling optimizations. It is more efficient to start the speculative generation immediately.

(4) The pool is empty and D_i is not the final result. The incorrect speculative generation is terminated immediately upon vector search produces the first differing result. Though the speculative generation is incorrect, it utilizes only the idle GPU resources. Both deferring and starting the speculative generation are equally efficient.

This concludes the proof. □

The general RAG system is more complex with larger LLM batch sizes and parallel vector search. In addition, we cannot terminate a speculative generation immediately when the request is batched with other requests. Based on Theorem 5.1,

Model	Layers	Q/KV Heads	MoE	Model Size	KV Size
Mistral-7B	32	32/8	no	14 GiB	0.125 MiB/token
LLaMA2-7B	32	32/32	no	14 GiB	0.5 MiB/token
Mixtral-8×7B	32	32/8	yes	96.8 GiB	0.125 MiB/token
LLaMA2-70B	80	64/8	no	140 GiB	0.3125 MiB/token

Table 1. Models used in the evaluation.

we design a dynamic speculative pipelining strategy in Algorithm 2. The main idea is to start a speculative generation only if the retrieved documents change and the number of pending LLM requests falls below a predetermined maximum batch size for the prefill iteration ($max\_prefill\_bs$). The maximum prefill batch size is determined by the smaller number of tokens that can either fit within the GPU memory or fully utilize the SMs. The strategy terminates the incorrect speculative generation after the current LLM iteration, which does not affect other requests in the batch. This strategy overlaps the retrieval and generation steps as much as possible while gauging the system load.

6 Implementation

We implement a system prototype of RAGCache with ~5000 lines of code in C++ and Python. Our implementation is based on vLLM [26] v0.3.0, a state-of-the-art LLM serving system. We extend its prefill kernel in Pytorch [36] and Triton [40] to support prefix caching for different attention mechanisms, e.g., multi-head attention [44] and grouped-query attention [6].

Pipelined vector search. We implement dynamic speculative pipelining on top of Faiss [4], an open-source widely-used vector database, and adapt it for two types of indexes: IVF [7] and HNSW [34]. IVF divides the vector space into multiple clusters and stores the vectors in the corresponding clusters. During vector search, IVF first locates the top-n closest clusters to the request vector and subsequently searches within these clusters. HNSW constructs multi-level graphs to map the vector space, connecting vectors with edges to represent similarity. HNSW searches the vectors by traversing the graph and maintains a candidate list to store the current top-k nearest vectors. To support pipelined vector search, we modify the search process of these two indexes. Specifically, we split the IVF search into multiple stages, each searching the vectors in some clusters and returning the current top-k vectors. For HNSW, we measure the average search time for a specified search configuration and split the entire time into smaller time slices. After each time slice of searching, it returns the current top-k vectors. These modifications aim to facilitate the dynamic speculative pipeline while maintaining the final search semantics.

Fault tolerance. We implement two fault-tolerant mechanisms in RAGCache to handle GPU failures and request processing failures. The GPU memory serves as RAGCache's first-level cache, storing the KV cache of the upper-level nodes in the knowledge tree hierarchy. Given the prefix sensitivity of LLM inference, a GPU failure would invalidate the lower-level nodes and therefore the entire tree. We replicate a portion of the most frequently accessed upper-level nodes (e.g., the system prompt) in the host memory for fast recovery. We also employ a timeout mechanism to retry the failed requests. If a request fails before completing its first iteration, it will be recomputed. Otherwise, the request can continue computation by reusing the stored KV cache.

7 Evaluation

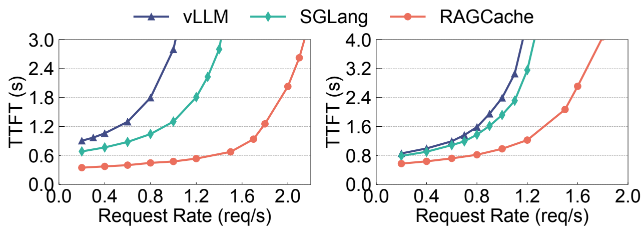
In this section, we evaluate RAGCache from the following aspects: (i) overall performance against state-of-the-art approaches; (ii) performance under general settings; (iii) ablation studies on the techniques used in RAGCache; and (iv) scheduling time of RAGCache.

Testbed. Most of our experiments are conducted on AWS EC2 g5.16xlarge instances, each with 64 vCPUs (AMD EPYC 7R32), 256 GiB host memory, and 25 Gbps NIC. Each instance is configured with one NVIDIA A10G GPU with 24 GiB memory and the GPU is connected to the host via PCIe 4.0×16. We run experiments with 7B models on a single g5.16xlarge instance and use 192 GiB host memory for caching unless otherwise stated. For large models, we use two NVIDIA H800 GPUs, each with 80 GiB memory and interconnected by NVLink. The two GPUs are connected to the host via PCIe 5.0×16. We use 384 GiB host memory for caching in this case.

Models. We evaluate RAGCache with the LLaMA 2 chat models [41] and the Mistral AI models [20, 21]. The model details are listed in Table 1. Most of the experiments are conducted with Mistral-7B and LLaMA2-7B. The two models have the same size but employ different attention mechanisms, i.e., grouped-query attention and multi-head attention. We also evaluate RAGCache with large models, Mixtral-8×7B and LLaMA2-70B, to demonstrate the scalability of RAGCache. Mixtral-8×7B is a mixture-of-experts (MoE) model with eight experts, and two experts are activated for each token. We deploy the large models on two H800 80GB GPUs for tensor parallelism and expert parallelism.

Retrieval. We use the Wikipedia dataset collected in § 3.2 as the knowledge base. For vector search, we use the IVF index with 1024 clusters and set the default top-k to 2. We deploy the vector database with four separate vCPUs and 30 GiB host memory on the same instance as the GPU and expose a RESTful API for document retrieval.

Workloads. Our evaluation uses two representative QA datasets, MMLU [18] and Natural Questions [25]. MMLU is a multi-choice knowledge benchmark where the LLM outputs a single token (A/B/C/D) per question. Natural Questions comprises anonymized questions from Google Search and provides the reference answers for each question. For Natural Questions, we sample the output length for each question



(a) Mistral-7B. **(b)** LLaMA2-7B.

Figure 13. Overall performance on MMLU.

from the token length distribution of the reference answers. The average output length is 6 tokens, and 99% of the answers contain no more than 32 tokens. We sample a subset of the questions from the dataset, respecting the document retrieval distribution in § 3.2, and randomly shuffle the questions to generate 1-hour workloads. In line with prior work [26, 45], we assign the arrival time for each request using a Poisson process parameterized by the arrival rate.

Metrics. We report the average time-to-first-token (TTFT) as the main metric. We also evaluate the system throughput, which is defined as the request rate that the system can process while maintaining a TTFT below a certain threshold, e.g., 5× of the TTFT at the lowest request rate. In addition, we measure the cache hit rate in the ablation study.

Baselines. We compare RAGCache with two baselines.

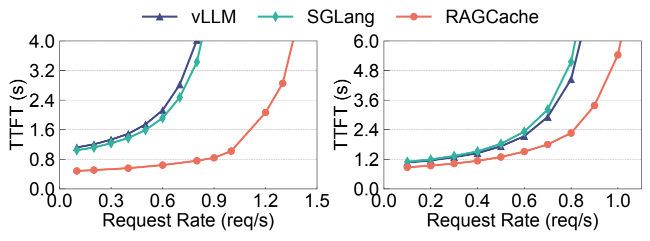
- **vLLM** [26], a state-of-the-art LLM serving system that supports iteration-level scheduling [50] and uses PagedAttention [26] to reduce memory fragmentation.
- **SGLang** [57], a high-performance LLM serving system that allows KV cache reuse across different requests in GPU memory and employs LRU as the replacement policy.

For fair comparison, the baselines are configured with the same model parallelism, maximum batch size, and vector database settings as RAGCache.

7.1 Overall Performance

We first compare the overall performance of RAGCache against the baselines. We use MMLU and Natural Questions as the workloads and Mistral-7B and LLaMA2-7B as the models. The maximum batch size is set to 4. We vary the request rate and measure the average TTFT. Figure 13 and Figure 14 show the results on MMLU and Natural Questions, respectively, which we summarize as follows.

- RAGCache reduces the average TTFT by 1.2–4× compared to vLLM and 1.1–3.5× compared to SGLang under the same request rate. This is because RAGCache utilizes the GPU memory and host memory to cache the KV cache of hot documents and avoids frequent recomputation.
- Due to faster request processing, RAGCache achieves 1.3–2.1× higher throughput than vLLM and 1.2–1.8× higher throughput than SGLang.
- RAGCache outperforms the baselines across models with varying attention mechanisms on different datasets.



(a) Mistral-7B. **(b)** LLaMA-2-7B.

Figure 14. Overall performance on Natural Questions.

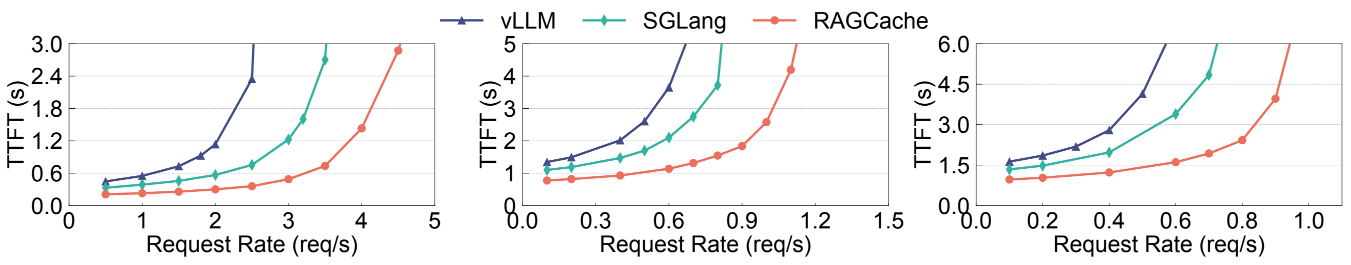
The results also reflect the differences between the two models and datasets. Notably, the performance gap between RAGCache and vLLM is greater for Mistral-7B than for LLaMA2-7B This is because LLaMA2-7B has a KV cache size 4× that of Mistral-7B for the same token count. resulting in a lower cache hit rate for LLaMA2-7B with the same cache size. According to our characterization in § 3.2, MMLU benefits more from document caching than Natural Questions, with a wider performance improvement for MMLU than for Natural Questions. SGLang performs closely to vLLM for Natural Questions because the limited GPU memory restricts document locality. RAGCache, however, with its multilevel caching and the adapted knowledge tree, outperforms in both datasets.

7.2 Case Study

We then conduct two case studies to demonstrate the benefits of RAGCache over the baselines under general settings. We use MMLU and Mistral-7B in the case studies.

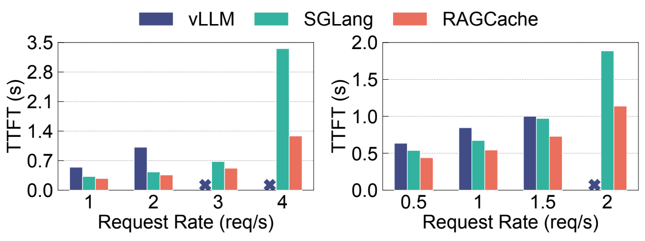
Different top-*k* values. Users may have varying requirements for the number of retrieved documents. We evaluate the performance of RAGCache and the baselines with commonly used top-*k* values: 1, 3, and 5. We set the maximum batch size to 4 and truncate the documents in the top-5 experiment to fit within GPU capacity limits. Figure 15 shows that RAGCache outperforms vLLM by 1.7–3.1× and SGLang by 1.2–2.5× in average TTFT across these top-*k* values. Despite the factorial growth in document permutations with increasing top-*k* values, RAGCache maintains its advantage by caching frequently-used documents. This is because the knowledge tree always evicts the node furthest from the root, ensuring that the most frequently used prefixes remain in the cache.

Large models. The second case study evaluates RAGCache with larger models using MMLU as the workload. We deploy Mixtral-8×7B and LLaMA2-70B on two H800 80GB GPUs. For each model, we set the maximum batch size to the lesser of what fits in GPU memory or fully utilizes the SMs (e.g., 8 for Mixtral-8×7B and 4 for LLaMA2-70B). With a limited budget, we run the experiments with four different request rates per model and set a TTFT SLO at 5× the TTFT of the lowest request rate. Figure 16 shows that under low request rates, RAGCache reduces the average TTFT by 1.4–2.1×



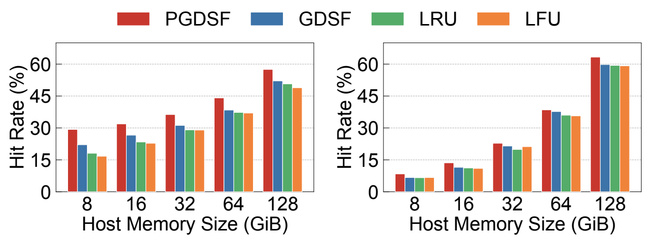
(a) Top-1. **(b)** Top-3. **(c)** Top-5.

Figure 15. Performance with different top-k values.



(a) Mixtral-8×7B. **(b)** LLaMA2-70B.

Figure 16. Performance under large models.



(a) MMLU. **(b)** Natural Questions.

Figure 17. Ablation study on cache replacement policy.

Host Memory	MMLU				Natural Questions			
Size	**PGDSF**	**GDSF**	**LRU**	**LFU**	**PGDSF**	**GDSF**	**LRU**	**LFU**
8 GiB	1.38	1.68	1.78	1.81	2.85	3.35	3.41	3.36
16 GiB	1.32	1.55	1.61	1.63	2.50	2.89	2.92	2.98
32 GiB	1.23	1.45	1.50	1.49	2.00	2.09	2.25	2.20
64 GiB	1.06	1.27	1.28	1.29	1.32	1.47	1.56	1.55
128 GiB	0.83	0.98	1.01	1.03	0.78	0.92	0.95	0.95

Table 2. Average TTFT (seconds) of different replacement policies with varying host memory size.

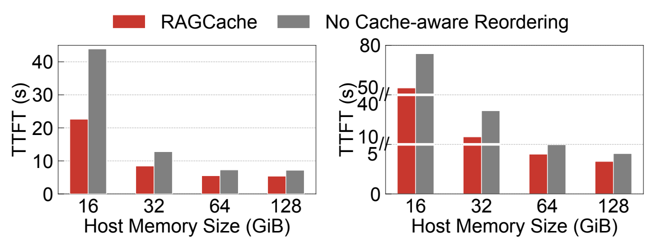
compared to vLLM. vLLM fails to meet the SLO above 2 and 1.5 requests per second for Mixtral-8×7B and LLaMA2-70B, respectively, whereas RAGCache maintains the TTFT below 1.4 seconds across varying request rates. SGLang performs better on H800 than on A10G GPUs due to increased GPU memory for caching, but RAGCache still surpasses SGlang by 1.2–2.6× in average TTFT.

7.3 Ablation Study

Prefix-aware GDSF policy. We compare RAGCache with versions of RAGCache that use native GDSF, LRU, and LFU as the replacement policy. For the GDSF policy, we set the recomputation cost of a document proportional to the document size, which aligns with our profiling results in Figure 2. We vary the host memory size for caching from 8 GiB to 128 GiB, set the request rate to 0.8 req/s, and report the hit rate and average TTFT. The hit rate for top-2 retrieval is defined as the ratio of the number of hit documents to the number of retrieved documents. For example, if the stored document sequence is $[D_1, D_2]$ and the requested one is $[D_1, D_3]$, then the hit rate would be 50%. Figure 17 shows the hit rate for MMLU and Natural Questions, and Table 2 lists the corresponding average TTFT. PGDSF achieves the highest hit rate across different host memory sizes, with a 1.02–1.32× improvement over GDSF, 1.06–1.62× improvement over LRU, and 1.06–1.75× improvement over LFU. This is because PGDSF captures the varying sizes, access patterns, and recomputation costs of different document prefixes. With the higher hit rate, RAGCache achieves 1.05–1.29× lower average TTFT than the baseline policies.

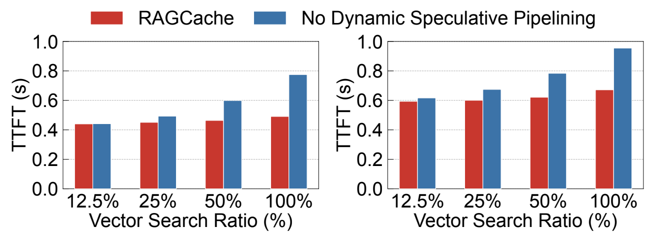
Cache-aware reordering. Then we evaluate the impact of cache-aware reordering. Reordering works when the request queue is saturated. We set the request rate to 2.5 req/s for MMLU and 1.4 req/s for Natural Questions, which are slightly higher than the throughput of RAGCache. We set the reordering window size to 32 and vary the host memory size from 16 GiB to 128 GiB. Figure 18 shows that RAGCache reduces the average TTFT by 1.2–2.1× with cache-aware reordering, which demonstrates its effectiveness under high request rates.

Dynamic speculative pipelining. Finally, we evaluate the effectiveness of dynamic speculative pipelining against a baseline, No Dynamic Speculative Pipelining (No DSP), which waits for vector search completion before starting LLM generation. We vary the ratio of the number of searched vectors to the total number of vectors from 12.5% to 100%. Note that while the search accuracy increases with the vector search ratio, the search time also extends. We use MMLU and Natural Questions as the workloads and set the request rate to 0.1 req/s. Figure 19 demonstrates that RAGCache achieves up to 1.6× TTFT reduction with dynamic speculative pipelining. Table 3 presents the average non-overlapping vector



(a) MMLU. **(b)** Natural Questions.

Figure 18. Ablation study on cache-aware reordering.



(a) MMLU. **(b)** Natural Questions.

Figure 19. Ablation study on speculative pipelining.

search time, which refers to the duration that the vector search does not overlap with the LLM generation using the final retrieval result. Dynamic speculative pipelining allows RAGCache to decrease non-overlapping vector search time by 1.5–4.3× and leads to a lower TTFT.

7.4 Scheduling Time

We measure RAGCache's scheduling time, including the time for knowledge tree lookup and update, request reordering, and speculative pipelining decisions. Using MMLU as the workload and Mistral-7B as the model, we range the request rate from 0.5 to 2 req/s. Table 4 indicates that the scheduling time remains below one millisecond across all request rates, which is negligible compared to the second-level TTFT.

8 Discussion

Time per output token (TPOT). In addition to time to first token (TTFT), TPOT is crucial for LLM serving [58]. RAG augments the request with documents retrieved from the external knowledge base, which significantly increases the input length and thus the latency of the prefill phase, i.e., TTFT. Consequently, the primary concern for RAG systems is the prolonged TTFT due to the extended input length. RAGCache reduces TTFT by caching the KV cache of the most frequently retrieved documents and can also lower TPOT by accelerating the prefill iteration, as decoding iterations typically take far less time than the prefill iteration [45].

Large top-*k*. As the top-k value increases, the number of document permutations explodes in factorial growth, making them less likely to be reused. RAGCache mitigates this by caching documents with a lower top-*k* value (e.g., caching the top-3 documents for requests with top-5 documents), thereby achieving a balance between hit rate and cache efficiency.

Vector Search	MMLU		Natural Questions	
Ratio	**RAGCache**	**No DSP**	**RAGCache**	**No DSP**
12.5%	52.1 ms	78.5 ms	67.7 ms	105.8 ms
25%	59.2 ms	135.9 ms	72.9 ms	163.4 ms
50%	69.7 ms	243.7 ms	94.2 ms	282.5 ms
100%	97.4 ms	422.3 ms	145.0 ms	446.1 ms

Table 3. Average non-overlapping vector search time under different settings.

9 Related Work

RAG. RAG [1, 8, 22, 27, 37, 42] enhances the generation quality of LLMs by incorporating relevant knowledge from external databases. Several works [22, 37, 42, 56] suggest iterative retrieval throughout generation to further improve the response quality. RAGCache supports iterative retrieval by treating the intermediate iterations as separate requests and caching the corresponding KV cache of the documents.

Vector search. RAG systems convert user prompts into vectors and uses approximate nearest neighbor (ANN) indexes like IVF [7, 11, 53, 54] and graph indexes [15, 19, 34] for efficient and accurate similarity search. RAGCache extracts the temporary search results for speculative LLM generation and thus pipelines the search process with LLM inference.

KV cache management. KV cache is widely used to accelerate the decoding phase of LLM inference [26, 45, 46, 50, 58]. Recent efforts aim to reduce the KV cache's memory footprint by quantization [14], compression [16, 29, 32], and self-attention with a subset of tokens [46, 55]. These methods introduce approximation to the generation process, while RAGCache preserves the exact KV cache of documents without affecting generation quality. Inspired by virtual memory in operating systems, vLLM [26] manages the KV cache at page granularity and proposes PagedAttention to prevent external fragmentation. RAGCache integrates the page-level management for KV cache sharing and improves over vLLM by leveraging the characteristics of RAG to cache the KV cache of the knowledge documents.

KV cache reusing. Recent efforts [17, 31, 49, 57] propose to reuse the KV cache across requests to reduce redundant computation. Prompt Cache [17] allows flexible reuse of the same tokens at different positions, while CacheGen [31] compresses the KV cache for efficient reuse. Both approaches may generate inaccurate responses. SGLang [57] and ChunkAttention [49] identify the reusable KV cache in GPU memory. RAGCache leverages RAG's retrieval pattern and builds a multilevel caching system, leading to higher performance with unchanged generation results.

Request Rate	Scheduling Time
0.5 req/s	0.880 ms
1.0 req/s	0.872 ms
1.5 req/s	0.902 ms
2.0 req/s	0.906 ms

Table 4. Scheduling time of RAGCache.

10 Conclusion

We present RAGCache, a multilevel caching system tailored for RAG. Based on a detail RAG system characterization, RAGCache employs a knowledge tree with a prefix-aware replacement policy to minimize redundant computation and a dynamic speculative pipelining mechanism to overlap the knowledge retrieval and LLM inference in the RAG workflow. We evaluate RAGCache with a variety of models and workloads. The experimental results show that RAGCache outperforms the state-of-the-art solution, vLLM integrated with Faiss, by up to 4× on TTFT and 2.1× on throughput. j

References

[1] 2024. LangChain. https://python.langchain.com/docs/get_started/introduction. (2024).

[2] 2024. OpenAI. https://openai.com/. (2024).

[3] 2024. OpenAI text-embedding-3 model. https://openai.com/blog/new-embedding-models-and-api-updates/. (2024).

[4] 2024. Pinecone: Introduction to Facebook AI Similarity Search (Faiss). (2024). https://www.pinecone.io/learn/series/faiss/faiss-tutorial/.

[5] 2024. Wikipedia (en) embedded with cohere.ai multilingual-22-12 encoder. https://huggingface.co/datasets/Cohere/wikipedia-22-12-en-embeddings/. (2024).

[6] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* (2023).

[7] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* (2014).

[8] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International Conference on Machine Learning (ICML)*.

[9] Hung-Ting Chen, Fangyuan Xu, Shane A Arora, and Eunsol Choi. 2023. Understanding retrieval augmentation for long-form question answering. *arXiv preprint arXiv:2310.12150* (2023).

[10] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking large language models in retrieval-augmented generation. In *AAAI Conference on Artificial Intelligence*.

[11] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* (2021).

[12] Ludmila Cherkasova. 1998. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy.* Hewlett-Packard Laboratories Palo Alto, CA, USA.

[13] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[14] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems* (2022).

[15] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. In *Proceedings of the VLDB Endowment*.

[16] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801* (2023).

[17] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2023. Prompt cache: Modular attention reuse for low-latency inference. *arXiv preprint arXiv:2311.04934* (2023).

[18] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* (2020).

[19] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* (2019).

[20] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[21] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[22] Wenqi Jiang, Shuai Zhang, Boran Han, Jie Wang, Bernie Wang, and Tim Kraska. 2024. Piperag: Fast retrieval-augmented generation via algorithm-system co-design. *arXiv preprint arXiv:2403.05676* (2024).

[23] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. 2017. 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)*.

[24] Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp. *arXiv preprint arXiv:2212.14024* (2022).

[25] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. 2019. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics* (2019).

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *ACM SOSP*.

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* (2020).

[28] Conglong Li, Minjia Zhang, David G Andersen, and Yuxiong He. 2020. Improving approximate nearest neighbor search through learned adaptive early termination. In *ACM SIGMOD*.

[29] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. 2020. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on Machine Learning (ICML)*.

[30] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* (2024).

[31] Yuhan Liu, Hanchen Li, Kuntai Du, Jiayi Yao, Yihua Cheng, Yuyang Huang, Shan Lu, Michael Maire, Henry Hoffmann, Ari Holtzman, et al. 2023. CacheGen: Fast Context Loading for Language Model Applications. *arXiv preprint arXiv:2310.07240* (2023).

[32] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2024. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems* (2024).

[33] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[34] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* (2018).

[35] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* (2019).

[37] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics* (2023).

[38] John F Shortle, James M Thompson, Donald Gross, and Carl M Harris. 2018. *Fundamentals of queueing theory*. Vol. 399. John Wiley & Sons.

[39] Shamane Siriwardhana, Rivindu Weerasekera, Elliott Wen, Tharindu Kaluarachchi, Rajib Rana, and Suranga Nanayakkara. 2023. Improving the domain adaptation of retrieval augmented generation (RAG) models for open domain question answering. *Transactions of the Association for Computational Linguistics* (2023).

[40] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*.

[41] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. CoRR, abs/2302.13971, 2023. doi: 10.48550. *arXiv preprint arXiv.2302.13971* (2023).

[42] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *arXiv preprint arXiv:2212.10509* (2022).

[43] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* (2017).

[45] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920* (2023).

[46] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453* (2023).

[47] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[48] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems* (2024).

[49] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. *arXiv preprint arXiv:2402.15220* (2024).

[50] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for {Transformer-Based} Generative Models. In *USENIX OSDI*.

[51] Biao Zhang, Barry Haddow, and Alexandra Birch. 2023. Prompting large language model for machine translation: A case study. In *International Conference on Machine Learning (ICML)*.

[52] Tianyi Zhang, Faisal Ladhak, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B Hashimoto. 2024. Benchmarking large language models for news summarization. *Transactions of the Association for Computational Linguistics* (2024).

[53] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. 2023. Fast, Approximate Vector Queries on Very Large Unstructured Datasets. In *USENIX NSDI*.

[54] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining. In *USENIX NSDI*.

[55] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2024. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* (2024).

[56] Zhihao Zhang, Alan Zhu, Lijie Yang, Yihua Xu, Lanting Li, Phitchaya Mangpo Phothilimthana, and Zhihao Jia. 2024. Accelerating retrieval-augmented language model serving with speculation. *arXiv preprint arXiv:2401.14021* (2024).

[57] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104* (2023).

[58] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. *arXiv preprint arXiv:2401.09670* (2024).