DepCache: A KV Cache Management Framework for GraphRAG with Dependency Attention

HAO YUAN, XIN AI, QIANGE WANG*, PEIZHENG LI, JIAYANG YU, CHAOYI CHEN, XINBO YANG, YANFENG ZHANG*, and ZHENBO FU, Northeastern University, China
YINGYOU WEN, Neusoft AI Magic Technology Research, China
GE YU, Northeastern University, China

Graph-based Retrieval-Augmented Generation (GraphRAG) has emerged as a promising paradigm for enhancing LLM reliability by enabling multi-hop reasoning over graph-structured knowledge. However, existing LLMs struggle to efficiently process graph-structured inputs, as traditional attention mechanisms are sequence-based and introduce significant redundancy when serializing graphs into prompt sequences, leading to excessive computation and memory overhead. To address this, we introduce dependency attention, a novel graph-aware attention mechanism that restricts attention computation to token pairs with structural dependencies in the retrieved subgraph. Unlike standard self-attention that computes fully connected interactions, dependency attention prunes irrelevant token pairs and reuses computations along shared relational paths, substantially reducing inference overhead. Building on this idea, we develop DepCache, a KV cache management framework tailored for dependency attention. DepCache enables efficient KV cache reuse through (i) a graph-based KV cache reuse strategy that aligns KV caches across varying prompt contexts, enabling efficient cross-request reuse in GraphRAG, and (ii) a locality-aware replacement policy that leverages spatial and temporal access patterns to improve KV cache hit rate. Evaluations across diverse models and datasets show that DepCache improves LLM inference throughput by 1.5×–5.0× and reduces time-to-first-token latency by up to 3.2×, without compromising generation accuracy.

CCS Concepts: • **Information systems** → **Data management systems**; • **Computing methodologies** → **Artificial intelligence**.

Additional Key Words and Phrases: Large Language Models; Retrieval-Augmented Generation; LLM Inference; Efficient Attention; KV Cache Reuse

ACM Reference Format:
Hao Yuan, Xin Ai, Qiange Wang, Peizheng Li, Jiayang Yu, Chaoyi Chen, Xinbo Yang, Yanfeng Zhang, Zhenbo Fu, Yingyou Wen, and Ge Yu. 2025. DepCache: A KV Cache Management Framework for GraphRAG with Dependency Attention. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 313 (December 2025), 29 pages. https://doi.org/10.1145/3769778

1 Introduction

Retrieval-Augmented Generation (RAG) [21, 37, 99] enhances the capabilities of large language models (LLMs) by incorporating external knowledge, enabling the generation of high-quality and

*Yanfeng Zhang and Qiange Wang are the corresponding authors.

Authors' Contact Information: Hao Yuan, yuanhao@stumail.neu.edu.cn; Xin Ai, aixin0@stumail.neu.edu.cn; Qiange Wang, wangqiange94@gmail.com; Peizheng Li, lipeizheng@stumail.neu.edu.cn; Jiayang Yu, yujiayang@stumail.neu.edu.cn; Chaoyi Chen, 2210700@stu.neu.edu.cn; Xinbo Yang, 20215857@stu.neu.edu.cn; Yanfeng Zhang, zhangyf@mail.neu.edu.cn; Zhenbo Fu, fuzhenbo@stumail.neu.edu.cn, Northeastern University, China; Yingyou Wen, Neusoft AI Magic Technology Research, China, wenyingyou@mail.neu.edu.cn; Ge Yu, yuge@mail.neu.edu.cn, Northeastern University, China.

ACM 2836-6573/2025/12-ART313
https://doi.org/10.1145/3769778

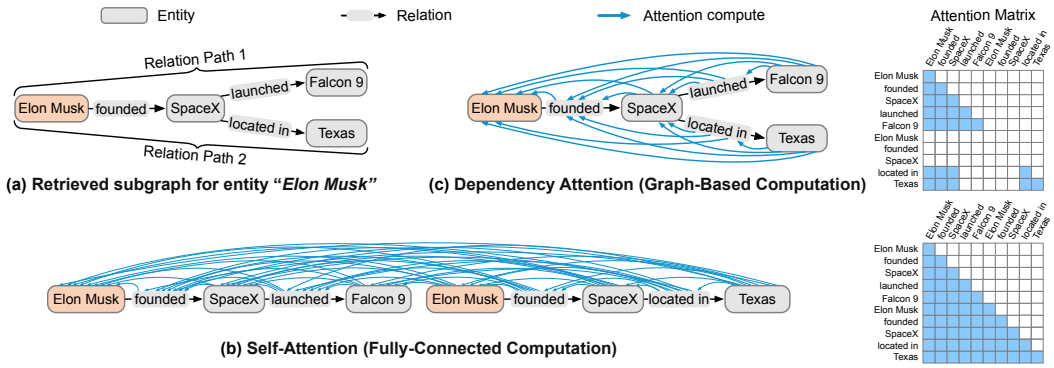


Fig. 1. Comparison of attention computation in self-attention and dependency attention. Arrows indicate attention computation between tokens, and attention matrices highlight computed scores in blue. (a) A 2-hop retrieved subgraph for the entity "*Elon Musk*", containing two relation paths. (b) Self-Attention: Transforming the retrieved subgraph into a token sequence and performing fully-connected attention computation. (c) Dependency Attention: Computing attention based on graph dependencies eliminates redundant computations by pruning non-dependent attention and reusing shared prefix attention.

reliable responses in domain-specific tasks such as legal consultation [79], healthcare [82], and financial services [93]. Recently, GraphRAG [17, 26, 28, 29, 39, 41, 56, 60, 61, 83, 96], an emerging variant of RAG, has attracted increasing attention for its ability in multi-hop reasoning and fine-grained contextual information capturing by modeling the external knowledge sources into knowledge graphs.

The key components of a GraphRAG system are a graph database and an LLM. The graph database stores a knowledge graph, which is extracted and constructed from domain-specific documents, and consists of many entities (nodes) and their relationships (edges). During the retrieval phase, GraphRAG extracts entities from the query and retrieves their L-hop subgraphs from the knowledge graph. In the generation phase, the retrieved subgraphs are serialized into text sequences and fed into the LLM along with the original query to generate the final response. Figure 1 illustrates the overall process, where (a) shows the 2-hop retrieved subgraph for the entity "Elon Musk" with two relation paths, and (b) shows how these paths are concatenated into a single text sequence to match the LLM's sequence-based input format and support attention computation. Modern LLMs [54, 74, 100, 103] rely on self-attention and autoregressive mechanisms to process input prompt sequences. They first employ a `prefill` phase to transform the input prompt into key and value tensors (i.e., KV cache) and then feed them into `decode` phase, where each token computes attention scores with respect to all previous tokens in the sequence (e.g., the arrow between tokens in Figure 1b), allowing the model to aggregate global contextual information and generate output sequence token by token.

The extensive entity dependency relationships in the graph enable GraphRAG to acquire rich structured information, thereby improving the accuracy of generation. However, there exists a fundamental gap between structured graph data and the sequence-to-sequence nature of LLMs, where serializing the retrieved graph into input prompts leads to redundant attention from non-dependent tokens and repeated prefixes, resulting in significant computational and memory overhead. The experimental results show that, compared to LLM inference without retrieval augmentation, GraphRAG incurs a 6×–16× increase in end-to-end latency. We identify two key limitations

in how GraphRAG-based LLM inference fails to effectively utilize graph dependencies and graph access patterns.

First, ignoring graph dependencies leads to inefficient attention computation. The lack of graph-aware serialization in GraphRAG results in excessive redundant attention, as LLMs compute attention over token sequences without considering underlying graph dependencies. Since LLMs are inherently sequence-to-sequence models, the retrieved subgraph must be transformed into a text sequence format suitable for LLM input. The graph-to-text transformation splits the retrieved subgraph into multiple relation paths from the query entity to leaf entities and concatenates these paths into a long text sequence. The entities in the sequence are fed into the LLM in order, regardless of whether dependencies exist between adjacent entities in the original subgraph. This results in excessive non-dependent attention among unrelated tokens, as the self-attention mechanism cannot effectively constrain attention to structurally relevant nodes. Furthermore, shared prefixes across multiple relation paths are duplicated in the input, leading to repeated attention computations. Our analysis reveals that approximately 75% of attention computations in GraphRAG occur between tokens without structural dependencies, and these computations typically yield low attention scores (see details in Section 2.3).

Second, ignoring graph access patterns hinders efficient KV cache reuse. While each individual query may retrieve a distinct subgraph, we observe that the overall graph access patterns exhibit strong locality across multiple queries. When entities are shared across queries, the KV caches of their retrieved subgraphs can be reused to avoid redundant computation. Our analysis on three representative RAG QA datasets, RGB [8], DragonBall [104], and Multihop [68], reveals that the top 20% most frequent entities account for 41%, 60%, and 70% of all retrieval requests, respectively (see details in Section 2.3). This indicates that a small subset of popular entities is repeatedly accessed across queries, presenting a valuable opportunity for KV cache reuse. Unfortunately, GraphRAG-based LLM inference fails to exploit this graph access pattern: even when the same entity is retrieved multiple times, its KV cache is repeatedly recomputed, resulting in excessive memory and computation overhead during generation.

Bridging this gap requires efficient mechanisms for adapting the graph-structured knowledge to sequence-based LLMs. To this end, we propose a novel attention mechanism, called dependency attention, which leverages dependency relationships within the retrieved subgraph to guide attention computation. Unlike traditional self-attention that computes attention in a fully connected manner, dependency attention restricts attention computation to tokens with dependency relations, while preserving output quality. This graph-aware mechanism not only improves computational efficiency but also introduces new opportunities for managing KV caches more effectively. The independence of retrieved subgraphs for different entities enables parallel KV cache computation, and shared prefixes across relation paths support efficient KV cache reuse. Figure 1 illustrates the computational differences between self-attention and dependency attention. In self-attention (Figure 1b), each token computes attention with all preceding tokens. In contrast, dependency attention (Figure 1c) computes attention based on dependency edges in the graph. Figures 1b and 1c also show the corresponding attention matrices, highlighting the reduced computational overhead achieved by dependency attention.

Based on the above design, we develop DepCache, an efficient GraphRAG-based LLM inference system that enables effective KV cache reuse for dependency attention through two key techniques. First, DepCache introduces a graph-based KV cache reuse mechanism that aligns the positional encoding of shared graph structures to enable KV cache reuse for the retrieved subgraph across queries. Second, it provides a locality-aware cache replacement policy that considers both historical access frequency and the temporal locality of future requests to improve KV cache reuse efficiency under limited memory resources. Our evaluations on various models and workloads show that

DepCache improves the end-to-end LLM inference throughput by 1.5×–5.0× compared to state-of-the-art systems [36, 101], without compromising the answer accuracy. In summary, we make the following contributions:

- We provide an in-depth analysis of the data management challenges arising from the interaction between graph-structured knowledge and LLMs in GraphRAG.
- We propose dependency attention, a graph-aware attention mechanism that eliminates redundant attention computations while preserving generation quality.
- We design and implement DepCache, an LLM inference framework that integrates dependency-aware attention with efficient KV cache reuse mechanisms to improve inference efficiency.
- We conduct a comprehensive evaluation of DepCache. Experimental results show that DepCache achieves up to 3.2× speedup in time-to-first-token (TTFT) and 5.0× improvement in throughput compared to the state-of-the-art LLM inference system.

2 Background and Motivation

2.1 LLM Architecture and Inference

2.1.1 Transformer-Based Large Language Models. Large language models (LLMs) demonstrate exceptional performance in the field of natural language processing. The widely used LLMs such as GPTs [55], Gemini [59, 69], and LLaMAs [15, 71, 72] are based on the transformer architecture. The transformer [74] model is composed of multiple transformer layers, each consisting of two steps: self-attention and a feed-forward network (FFN).

For the input token sequences $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n]$, the transformer applies linear transformations to each token in $\boldsymbol{X}$ using the weights $\boldsymbol{W}_q$, $\boldsymbol{W}_k$ and $\boldsymbol{W}_v$ to generate a set of queries $\boldsymbol{Q}$, keys $\boldsymbol{K}$ and values $\boldsymbol{V}$:

$$\boldsymbol{Q} = \boldsymbol{W}_q \boldsymbol{X}; \quad \boldsymbol{K} = \boldsymbol{W}_k \boldsymbol{X}; \quad \boldsymbol{V} = \boldsymbol{W}_v \boldsymbol{X} \tag{1}$$

The self-attention scores are then computed as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{2}$$

where d_k is the dimension of $\boldsymbol{K}$. The self-attention mechanism is the core design of the transformer, enabling the model to capture dependencies across any distance within the input data.

The feed-forward network (FFN) is another critical structure in a transformer layer. The FFN typically consists of two linear transformations with an intermediate non-linear activation function, represented as:

$$\text{FFN}(\boldsymbol{X}) = \boldsymbol{W}_2 \sigma(\boldsymbol{W}_1 \boldsymbol{X}) \tag{3}$$

where $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ are two learnable parameters, σ is a non-linear activation function (e.g., ReLU). The output of the FFN is then fed to the next transformer layer as input. Finally, after all transformer layers have completed their computations, the model outputs a probability vector that predicts the next token.

2.1.2 Autoregressive Generation. During the inference phase, a transformer-based LLM processes the user's prompt and generates a response in an autoregressive manner. The inference process of an LLM can be divided into two phases: `prefill` and `decode`. In the `prefill` phase, the input token sequence $\boldsymbol{X}[1:n]$ is fed into the transformer. Then the transformer generates intermediate $\boldsymbol{K}$ and $\boldsymbol{V}$ tensors for each token and outputs the predicted token $\boldsymbol{X}_{n+1}$. These $\boldsymbol{K}$ and $\boldsymbol{V}$ tensors, referred to as the KV cache, are stored in the GPU for use during the decode phase. In the `decode`

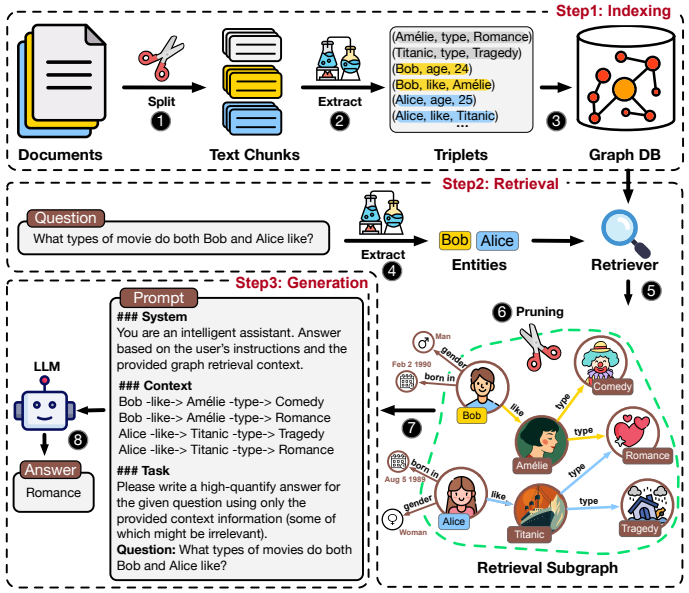


Fig. 2. GraphRAG workflow.

phase, the token X_{n+1} and the KV cache generated in the prefill stage are used to compute the next token X_{n+2}. The decode phase iteratively generates tokens in an autoregressive manner until either an end-of-sequence token (<eos>) is produced or the sequence reaches a maximum length. In the RAG scenario, the retrieved documents significantly increase the input length, which typically contains thousands of tokens. For the long inputs, running the `prefill` phase can cause substantial delays, making users wait until the first token is generated [84].

2.2 Graph Retrieval-Augmented Generation

The knowledge encoded in LLMs is inherently time-sensitive, as it reflects only data available during training (e.g., Llama3-70B up to Dec 2023 [48]). Consequently, periodic retraining is required to remain up to date, but the sheer scale of modern LLMs renders this process costly and inefficient. Retrieval-Augmented Generation (RAG) mitigates this limitation by integrating external knowledge bases to provide up-to-date and contextually relevant information, improving answer quality. Recently, GraphRAG [17, 26, 29, 41, 56, 60, 61, 83, 96], an emerging variant of RAG, has attracted increasing attention for its ability in multi-hop reasoning and fine-grained contextual information capturing by modeling external knowledge into graph structures. Existing GraphRAG methods can be broadly categorized into two types based on how external knowledge is retrieved and represented: chunk-based and KG-based. The former retrieves chunk-level summaries to support thematic or summarization QA [17, 26, 41], while the latter adopts entity-relationship graphs to enable fine-grained, multi-hop reasoning in knowledge-intensive tasks [29, 51, 52, 67, 78]. Our work specifically targets the latter by leveraging explicit graph dependencies, which are critical for ensuring efficient inference. Beyond GraphRAG, prior work on large-scale graph computation systems [3, 76, 77, 90] has also emphasized the importance of efficient dependency management.

Figure 2 illustrates the execution workflow of GraphRAG, which typically consists of three components: indexing, retrieval, and generation. In the indexing phase, external documents are split into multiple text chunks (❶) and then fed into a knowledge extractor to generate triplets (❷),

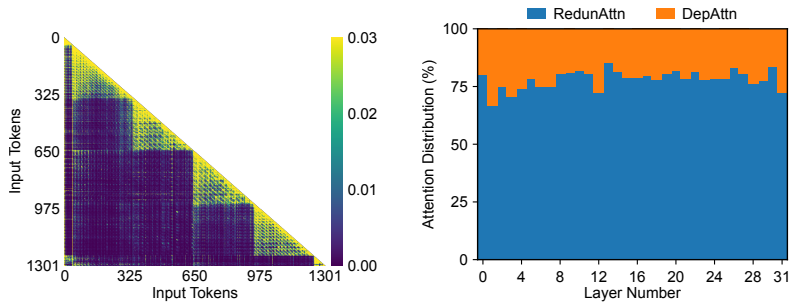
such as (*Bob*, *age*, *24*). These triplets are then inserted into a graph database to construct a complete knowledge graph (❸), supporting subsequent retrieval. In the retrieval phase, GraphRAG extracts entities from the user's query (❹) and retrieves the L-hop subgraph associated with these entities from the graph database (❺). The subgraph is then pruned based on semantic similarity to the query, retaining the top-k relation paths most relevant to the user's query (❻). Since LLMs cannot directly process graph-structured data, the system first need to convert the retrieved subgraph into a text sequence in the generation phase. Specifically, the retrieved subgraph is transformed into multiple relation paths starting from the query entity and extending to leaf nodes (e.g., the entity "*Alice*" corresponds to two relation paths: "*Alice -like-> Titanic -type-> Tragedy*" and "*Alice -like-> Titanic -type-> Romance*"). These relation paths are concatenated to form a complete text sequence (❼). Subsequently, the text sequence is used as context and combined with the user's query through the predefined prompt template to construct a prompt. Finally, this prompt is input into the LLM to generate the answer (❽).

2.3 Motivation

GraphRAG significantly improves generation quality by incorporating structured information from extensive entity dependencies in the knowledge graph. However, existing LLM inference engines lack effective strategies for managing complex graph data in GraphRAG, leading to significant inference latency. We summarize the two major limitations below.

Limitation #1: Ignoring graph dependencies leads to inefficient attention computation. The self-attention mechanism performs fully connected attention computations among all tokens, regardless of whether there are dependency relationships between the tokens in the retrieved subgraph. This introduces a substantial amount of redundant attention computation. Specifically, the redundant attention computation consists of two parts: **(1) Attention is redundantly computed for the non-dependent tokens.** For example, the node "*located in*" in Path 2 computes attention to "*-launched-> Falcon 9*" in Path 1, even though no semantic dependency exists between them. We visualize the attention matrix in GraphRAG (Figure 3a). The results show that high attention scores are primarily concentrated within dependent nodes within individual retrieved subgraphs, while attention scores between nodes with no dependencies, especially across subgraphs, are relatively low. This confirms that attention among non-dependent nodes is redundant and can be saved, as removing low-scoring attention typically does not harm inference accuracy [45, 80, 84, 92, 98]. **(2) Attention is redundantly computed for the shared prefixes.** After serialization, shared prefixes across relation paths appear multiple times in the input sequence, causing remaining entities to repeatedly compute attention scores with the same prefixes during self-attention. For example, as shown in Figure 1b, the node "*located in*" computes attention scores twice with the same prefix "*Elon Musk -founded-> SpaceX*" from both Path 1 and Path 2. Finally, we quantify the proportion of two types of redundant attention computations, as shown in Figure 3b. The results show that approximately 75% of the attention computations are redundant.

Limitation #2: Ignoring graph access patterns hinders efficient KV cache reuse. Although GraphRAG retrieval exhibits strong locality, GraphRAG-based LLM inference fails to leverage this graph access pattern for efficient KV cache reuse. We empirically analyze the reuse potential and identify key challenges that limit its effectiveness. Our evaluation on three representative RAG QA datasets, RGB [8], DragonBall [104], and Multihop [68]. As shown in Figure 4, top 20% frequent entities are referred to by 41%, 60%, and 70% requests in the RGB, DragonBall, and Multihop dataset, respectively. For these frequently accessed entities, the KV caches of their retrieved subgraphs can be stored in GPU memory and reused across multiple requests to reduce the generation overhead in GraphRAG. However, achieving efficient KV cache reuse in GraphRAG is non-trivial. First, due to the position sensitivity of the attention mechanism, KV caches of the same retrieved subgraph



(a) Attention Matrices. (b) Redundant Attention.

Fig. 3. (a) Visualization of the attention matrix in GraphRAG. Each pixel represents the attention weight between a pair of tokens, where brighter colors (yellow) indicate higher attention, and darker colors (purple) indicate lower attention. (b) Proportion of redundant attention in GraphRAG.

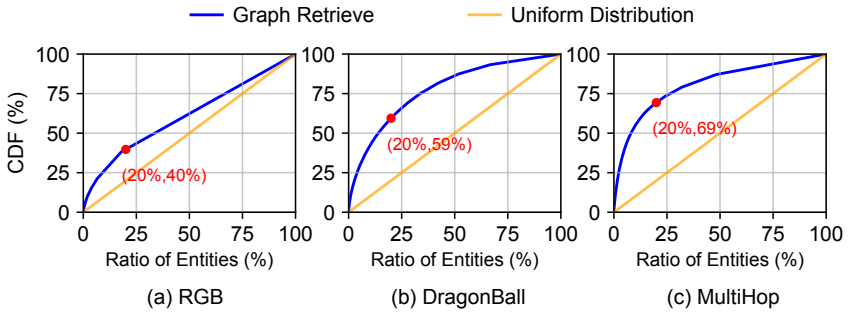
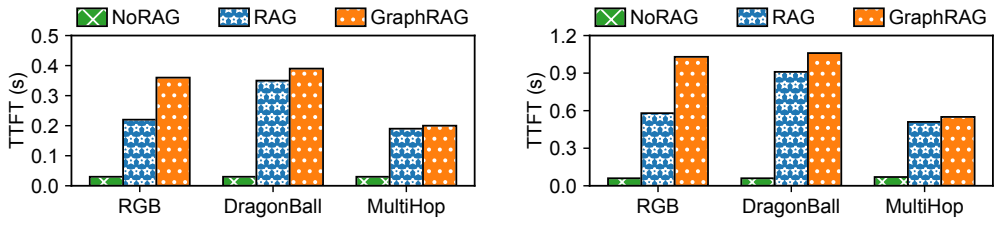


Fig. 4. Retrieval pattern of GraphRAG on three datasets.

cannot be directly reused when it appears at different positions in the prompt [2, 32, 50, 66, 74, 84]. We observe that the KV cache reuse rate in GraphRAG-based LLM inference is only 1.1%, 2.1%, and 2.3% on the RGB, DragonBall, and Multihop datasets, respectively. Second, limited GPU memory makes it impractical to store all KV caches. Under dynamic and unpredictable query workloads, designing a replacement policy that maximizes memory utilization remains a significant challenge.

The aforementioned limitations in GraphRAG lead to significant inference overhead during LLM execution. We evaluate the time to first token (TTFT) across three settings: NoRAG, RAG, and GraphRAG. To highlight the inference overhead introduced by the substantial retrieved context in RAG and GraphRAG, we include NoRAG as a baseline, where the LLM directly generates answers without any retrieval. This comparison clearly quantifies the additional latency incurred by incorporating retrieved context into the prompt. The experiments are conducted using the Llama3-8B [47] (small-scale) and Qwen2.5-32B [70] (medium-scale) models. The experiments run on two NVIDIA A6000 GPUs using benchmark datasets from three RAG tasks: RGB [8], Multihop [68], and DragonBall [104]. For RAG, we retrieve five text chunks per question from the vector database, each with a chunk size of 512 tokens. For GraphRAG, we retrieve five subgraphs per question from the graph database, with each subgraph containing up to 30 relation paths. The retrieved content is concatenated with the question and fed into the LLM. As shown in Figure 5, GraphRAG increases the average TTFT by 10.26× on the Llama3-8B model compared to NoRAG. As the model size grows, the computational overhead of GraphRAG becomes more pronounced, with the average TTFT reaching 13.71× on the Qwen2.5-32B model.



(a) Llama3-8B. (b) Qwen2.5-32B.

Fig. 5. Comparison of time to first token (TTFT) between NoRAG, RAG, and GraphRAG.

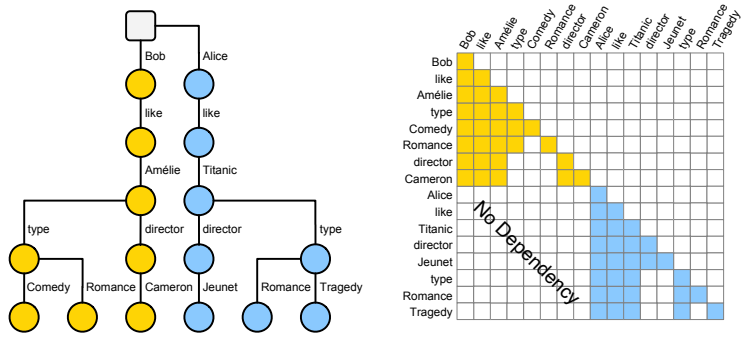
3 Dependency Attention

Overview. To address the aforementioned limitations, we propose dependency attention, an attention mechanism that computes attention between tokens based on graph dependencies. Dependency attention introduces a graph-aware approach to organizing input sequences, offering a more efficient alternative to the conventional strategy of serializing graphs into flat prompt sequences. By abstracting multiple relation paths into a unified dependency tree, our method captures semantic dependencies explicitly while avoiding redundant representations of shared prefixes across paths. Building upon this dependency tree, we construct a graph-aware attention mask to restrict attention computation to structurally relevant token pairs. Furthermore, we introduce a lightweight dependency-aware fine-tuning process that enables the model to adapt to the new attention paradigm while maintaining inference accuracy.

3.1 Dependency Attention Workflow

We present the workflow of dependency attention to illustrate how dependency trees are constructed from retrieved subgraphs and how they guide graph-aware attention computation through structured attention masks. In GraphRAG, each retrieved subgraph consists of relation paths rooted at query entities, where the entities and relations exhibit explicit semantic dependencies. As shown in Figure 6a, we abstract these relation paths into a dependency tree, where nodes represent entities and relationships, and edges capture their semantic dependencies. Unlike traditional self-attention mechanisms that compute attention among all token pairs in a fully connected manner, dependency attention computes the attention between node v and node u if and only if there exists a path from u to v in the tree (i.e. node u is an ancestor of v). As shown in Figure 6b, we generate an attention mask based on the dependency tree, where the mask value is set to 1 for token pairs with a dependency relationship (i.e., the coloured cells) and 0 otherwise (i.e., the empty cells). Under this abstraction, the traditional self-attention mechanism can be viewed as a special case of a dependency tree, specifically a chain-like structure.

We present the implementation details of dependency trees and attention mask construction through Algorithm 1, which outlines the workflow of dependency attention. During the dependency tree construction, each relation path from the retrieved subgraph is tokenized (Line 3) and inserted into a prefix tree to form the dependency tree (Line 4). This prefix-based structure merges common prefixes to reduce redundancy and supports efficient retrieval with a complexity of $O(h)$, where h is the tree height. In the attention mask generation phase, we construct two key inputs for dependency attention: the input token sequence $\mathcal{S}$ and the attention mask matrix $\mathcal{M}$. We first perform a pre-order traversal of the dependency tree to generate $\mathcal{S}$ for the LLM (Line 5). Unlike naïve path-by-path serialization, this method preserves structural dependencies while eliminating duplicated prefixes, thus reducing input length and lowering computational overhead. To construct



(a) Dependency Tree (b) Dependency Attention Mask

Fig. 6. Two key steps of dependency attention: dependency tree construction and attention mask generation.

$\mathcal{M}$, we perform DFS from each node to collect its ancestors (Lines 8–9) and mark dependent token pairs with 1 and others with 0 (Lines 10–12), thereby restricting attention to graph-relevant token interactions. The resulting token sequence $\mathcal{S}$ and attention mask matrix $\mathcal{M}$ are returned and will be used as inputs for the subsequent attention computation to generate the final token output.

3.2 Computation Reduction Analysis

We analyze and compare the computational complexity of dependency attention with that of standard self-attention. Assume that each query retrieves N entities, each entity retrieves an L-hop subgraph, and K relation paths are retained after pruning. For simplicity, we assume that each relation path has a token length of L. Let $M \in [1, L)$ denote the length of the common prefix shared by the K paths. In the standard self-attention mechanism, all NK relation paths are concatenated into a flat sequence of length NKL, resulting in a computational complexity of $O(N^2K^2L^2)$ due to the quadratic scaling of self-attention with sequence length. In contrast, the computational cost of dependency attention can be divided into two parts: (1) the attention computation for the common prefix, which has a complexity of $O(NM^2)$, and (2) the attention computation for the remaining tokens in the NK paths, which has a complexity of $O\left(NK\frac{(M+1+L)(L-M)}{2}\right) \approx O\left(NK(L^2-M^2)\right)$. Thus, the overall computational complexity of dependency attention is $O\left(NM^2 + NK(L^2-M^2)\right)$. This formulation reveals that the computational cost of dependency attention is directly influenced by the prefix sharing degree M. In the best case where $M = L-1$ (i.e., all paths share nearly the same prefix), the complexity reduces to $O(NL^2 + NKL)$, significantly lower than self-attention. Even in the worst case ($M = 1$), where only the root is shared, the complexity is $O(NKL^2)$, which is still $\frac{1}{NK}$ of standard self-attention.

3.3 Dependency-aware Fine-Tuning

LLMs are pretrained with fully connected self-attention, so directly switching to dependency attention may introduce a mismatch between training and inference, potentially degrading accuracy. To address this, we propose dependency-aware fine-tuning, which replaces the standard lower-triangular attention mask with a structure-aware dependency attention mask (Figure 6b), aligning the model's computation with the structure of retrieved subgraphs.

We randomly sample 6,000 question-answer pairs from TriviaQA [34] to construct the training dataset for dependency fine-tuning. Each sample includes: (1) a question, (2) relevant subgraphs retrieved from a graph database, and (3) an answer generated by GPT-4o [1] based on the subgraph. The knowledge base is built by extracting relational triplets from context documents using GPT-4o and storing them in a graph database. For each question, we retrieve 10 related entities using the

Algorithm 1: Workflow of Dependency Attention

Input: $\mathcal{G} = [p_1, p_2, \ldots, p_n]$; Tokenizer T
Output: Token sequence S; Dependency attention mask $\mathcal{M}$

Phase 1: Dependency Tree Construction

Initialize $\mathcal{T} \leftarrow$ PrefixTree() //initialize dependency tree
for *each* $p_i \in \mathcal{G}$ **do**
    $s_i \leftarrow$ Tokenize(T, p_i) //convert path to tokens
    $\mathcal{T} \leftarrow$ Insert($\mathcal{T}$, s_i) //insert token sequence into tree

Phase 2: Attention Mask Generation

$\mathcal{S} \leftarrow$ PreOrderTraversal($\mathcal{T}$) //obtain token sequence of dependency tree
$\mathcal{M} \leftarrow \mathbf{0}^{|S|\times|S|}$ //initialize dependency attention mask matrix
$\mathcal{D} \leftarrow \{\}$ //initialize dependency list
for *each* $v \in \mathcal{T}$ **do**
    $\mathcal{D}[v] \leftarrow$ FindAncestors($\mathcal{T}$, v) //find ancestor nodes of v
for *each* $v \in \mathcal{D}$ **do**
    for *each* $u \in \mathcal{D}[v]$ **do**
        $\mathcal{M}[u, v] \leftarrow 1$ //fill the dependency attention mask
return S, $\mathcal{M}$

bge-large-en-v1.5 [81] model. For each entity, a 2-hop subgraph is extracted, and up to 30 relation paths are retained based on semantic similarity. GPT-4o then generates answers grounded in the retrieved subgraphs, which serve as ground-truth labels for training.

Although dependency-aware fine-tuning introduces an additional training stage, its cost is relatively modest. Our experiments demonstrate that with only a small amount of data, dependency-aware fine-tuning enables LLMs to learn the computation paradigm of dependency attention, achieving inference accuracy comparable to or surpassing self-attention (see Section 6.5).

3.4 Discussions of Dependency Attention and Other Sparse Attention Methods.

The quadratic cost of self-attention limits its scalability to long sequences [7, 53, 103]. To address this, sparse attention methods [5, 10, 25, 49, 80, 91, 92] restrict computation to local windows, achieving linear complexity. Recent works further reduce inference latency by confining attention to within individual documents [2, 22, 50, 66, 84]. However, these methods are designed for sequential inputs and overlook the structural characteristics of graphs. Dependency attention, in contrast, is tailored for graph-structured data. While Graph Transformers [62, 89] and related studies [16, 58, 63, 95] exploit vertex dependencies to guide attention, they still rely on fully connected attention to capture global context in large graphs. GraphRAG differs by operating on multiple small, query-relevant subgraphs, which exhibit strong intra-graph but weak inter-graph dependencies (see Section 2.2 for details). This design allows GraphRAG to maintain accuracy under dependency attention while also enabling more efficient KV cache reuse by processing batches of subgraphs instead of a single large graph (see Section 4 for details).

4 The DepCache

Building on the structure-aware design of dependency attention, we develop DepCache, an efficient GraphRAG-based LLM inference system that enables effective KV cache reuse. Dependency attention exposes two key opportunities for efficient KV cache reuse. First, the retrieved k-hop subgraphs for different entities are structurally independent, enabling parallel computation of their

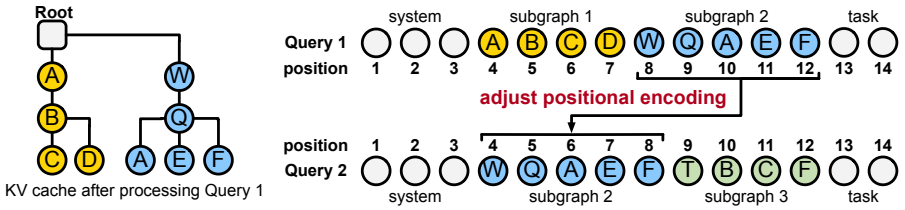


Fig. 7. Graph-based positional re-encoding for KV cache reuse across queries.

KV caches. For example, in Figure 6a, the subgraphs of *Bob* and *Alice* do not overlap, allowing their attention computations to proceed independently. Second, the shared prefixes within relation paths across queries create reuse opportunities for KV cache segments, particularly for frequently accessed entities. By caching the KV representations of these shared substructures, DepCache can significantly reduce redundant computation and inference latency. To realize this potential, DepCache integrates two key techniques tailored to dependency attention: a graph-based KV cache reuse mechanism and a locality-aware cache replacement strategy.

4.1 Graph-based KV Cache Reuse

Graph-based KV cache reuse aims to enable cross-query reuse of cached key-value representations for frequently retrieved subgraphs, thereby reducing redundant computation in dependency attention. However, realizing this reuse is non-trivial, as the same retrieval subgraph may appear at different positions within the prompt across requests, resulting in different KV representations and hindering effective cache reuse. This challenge arises from the autoregressive nature of LLMs, where the KV cache is inherently position-sensitive due to the underlying attention mechanism [2, 32, 50, 66, 74, 84]. Specifically, during inference, position encoding is incorporated into the attention mechanism to reflect the position of each token within the sequence. This design constrains the KV cache to be valid only at its original token positions [74]. As a result, even if the same subgraph appears across multiple queries, its KV cache cannot be directly reused if its position shifts in the prompt [2, 32, 50, 66, 84].

Benefiting from the dependency attention mechanism design, which restricts attention computation to token pairs with dependency relations, the retrieved subgraphs for different query entities are typically independent (Figure 6a). This structural independence implies that the KV cache of each subgraph is solely determined by its internal token dependencies, rather than its absolute position within the prompt. Based on this observation, we employ a graph-based positional re-encoding mechanism that leverages the translation invariance [35, 46, 75] of relative positional encoding to enable cross-request KV cache reuse by shifting the positional encodings of the KV cache associated with the retrieved subgraphs.

The graph-based positional re-encoding method works in conjunction with Rotary Position Embedding (RoPE) [65], a widely adopted relative positional encoding technique in modern LLMs [4, 23, 24, 30, 71, 88], which has shown superior performance over absolute positional encoding (APE) for long-context modeling [13, 74, 97]. RoPE encodes relative positional information by leveraging the rotation-invariant property of complex-valued multiplication. It computes the relative position between tokens by applying rotation matrices to their absolute positions and incorporates this information into the key and value vectors for self-attention. This rotation-invariant property also allows us to directly compute the relative positional information of a prompt in a new query using a rotation matrix that encodes the difference in absolute positions between the two queries. As shown in Figure 7, both Query 1 and Query 2 include Subgraph 2, but due to differing relevance, its position in the two prompts varies. The graph-based position re-encoding method adjusts the

positional encoding of the subgraph's KV cache from its position in Query 1 (positions 8–12) to its new location in Query 2 (positions 4–8), addressing the KV cache reuse issue caused by position mismatch. Specifically, suppose we aim to update the positional encoding of the m-th token to that of the n-th token. Our method consists of the following two steps:
Firstly, the positional encoding vector $f_{\{q,k\}}(\boldsymbol{x}_m, m)$ for the m-th token is calculated using the following formula:

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m}\boldsymbol{W}_{\{q,k\}}\boldsymbol{x}_m \tag{4}$$

where, $\boldsymbol{R}^d_{\Theta,m}$ is a block-diagonal matrix, whose diagonal elements correspond to 2D rotation sub-matrices $\boldsymbol{R}^{(i)}_m$:

$$\boldsymbol{R}^d_{\Theta,m} = \mathrm{diag}\big(\boldsymbol{R}^{(0)}_m, \boldsymbol{R}^{(1)}_m, \cdots, \boldsymbol{R}^{(d/2-1)}_m\big)$$
$$\boldsymbol{R}^{(i)}_m = \begin{bmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{bmatrix}$$

Secondly, we left-multiply the positional embedding of the m-th token by the matrix $\boldsymbol{R}^d_{\Theta,n-m}$, rotating its positional embedding to the target position n:

$$\begin{aligned} f_{\{q,k\}}(\boldsymbol{x}_n, n) &= f_{\{q,k\}}(\boldsymbol{x}_m, n) \\ &= \boldsymbol{R}^d_{\Theta,n-m} f_{\{q,k\}}(\boldsymbol{x}_m, m) \end{aligned} \tag{5}$$

4.2 Locality-aware KV Cache Replacement

Locality-aware KV cache replacement improves cache utility under limited memory by leveraging both spatial and temporal locality to retain KV entries with high reuse potential. This is especially critical in dependency attention, where complex graph dependencies lead to large subgraph retrievals that quickly exhaust high-bandwidth GPU memory (HBM) under intensive request streams [20]. To reduce memory pressure, existing systems adopt eviction policies based on historical access behavior. LRU retains recently used entries and is widely adopted in LLM inference systems [36, 101], while LFU prioritizes frequently accessed items. GDSF [9] improves upon these by integrating access frequency, object size, and recomputation cost. PGDSF [32] further refines this approach by incorporating prefix structures to estimate node-level cost in retrieval tasks. However, all these methods rely solely on historical access patterns and lack awareness of future query behaviors.

In practice, LLM inference engines typically receive a large number of concurrent requests from multiple users. The queue of pending requests implicitly defines the future access pattern of the KV cache. By leveraging this future access pattern, we can tailor a cache replacement strategy that optimizes temporal locality within a bounded time window. Meanwhile, historical access patterns can be exploited to identify hot requests (i.e., frequently occurring queries), thereby enhancing the spatial locality of KV cache access. Therefore, we propose a locality-aware cache replacement (LACR) approach that combines historical and future access information to better adapt to the highly dynamic request workloads of LLM inference tasks. LACR consists of two key components: (1) a prefix-based cache structure that captures token dependencies and improves search efficiency. (2) a locality-aware replacement policy that guides eviction decisions by balancing historical access knowledge with future access hints. We detail each component in the following sections.

4.2.1 Prefix-based Cache Structure. DepCache adopts a compressed prefix tree to organize and manage KV cache entries generated during inference. Unlike conventional KV cache systems

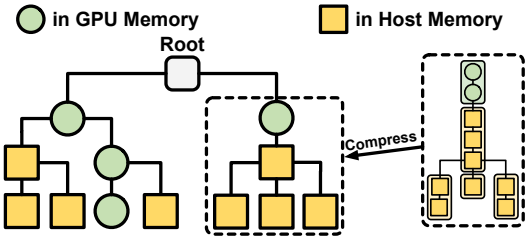


Fig. 8. Cache structure in DepCache.

that perform reuse checks based on flat token sequence matching, DepCache must respect the hierarchical structure imposed by dependency attention. Specifically, it performs prefix dependency matching from the root entity of each retrieved subgraph, ensuring that KV reuse is structurally valid within the graph-aware context. To support this hierarchical matching efficiently, DepCache compresses the prefix tree structure by merging consecutive nodes along non-branching paths into a single compressed node. This compression reduces the overall height of the tree, thereby accelerating prefix traversal and cache lookup. Moreover, since all tokens within a compressed node lie on the same dependency path, they inherently share a common ancestry in the graph. As a result, their KV cache reuse behavior is consistent, allowing DepCache to manage them as a unified unit. This enables batch reuse and eviction operations, significantly simplifying cache management. As illustrated in Figure 8, each node in the compressed prefix tree maps to the memory location of the KV cache for one or more tokens. Upon receiving a new query, DepCache constructs the corresponding dependency tree (e.g., Figure 6a) and performs prefix matching against the compressed prefix tree. Matched prefixes allow direct reuse of cached KV entries, while unmatched tokens are recomputed and inserted back into the tree for future reuse.

4.2.2 Cache replacement policy. When the cache space is insufficient to accommodate the KV cache generated by new requests, the system must decide which cache nodes in the compressed prefix tree should be evicted from the GPU to the CPU, or released from the CPU entirely. We propose a locality-aware cache replacement method. This approach overcomes the constraints of traditional strategies that rely solely on historical access information by incorporating limited future access knowledge, making it better suited to highly dynamic request patterns in LLM inference workloads. Specifically, for each node in the compressed prefix tree, we define two priority metrics: a historical priority and a future priority. The final replacement priority is computed as a weighted sum of these two metrics:

$$\text{Priority} = \alpha\,\text{HistoryPriority} + (1-\alpha)\,\text{FuturePriority} \tag{6}$$

LFU is used as the `HistoryPriority` metric, as it naturally captures high-frequency accessed entries. This choice is motivated by our empirical observation that GraphRAG exhibits strong retrieval locality, where a small subset of entities are frequently accessed across multiple requests (as shown in Figure 4). On the other hand, the `FuturePriority` is computed by analyzing a fixed-size lookahead window from the scheduler to count the number of upcoming accesses for each node. The parameter $\alpha \in [0, 1]$ controls the relative weight between historical and future access information. A smaller α places more emphasis on future access patterns, while a larger α favors historical behavior. In our experiments, we set the α parameter to 0.2 based on empirical observations across various workloads. This fixed value was found to perform well under a range of representative scenarios. We leave the exploration of adaptive α tuning strategies as future work.

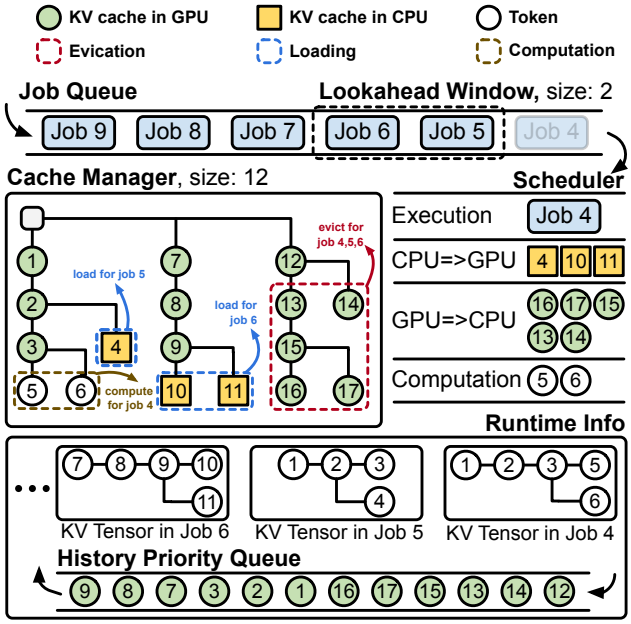


Fig. 9. KV Cache management workflow in DepCache.

Figure 9 illustrates the KV cache management workflow in DepCache. Assume that jobs 1–3 have been completed, and the system currently maintains 15 KV cache nodes in `CacheManager`. Among them, 12 nodes (in green) reside in GPU memory, while the remaining 3 nodes (in yellow) have been offloaded to CPU memory. The node indices correspond to token IDs. The `RuntimeInfo` records the specific tokens required for the computation of each job. In addition, it maintains a `HistoryPriorityQueue` to track the `HistoryPriority` of each node in the cache. Nodes with lower priority are placed at the front of the queue and are evicted first during cache replacement. When job 4 is scheduled, the `RuntimeInfo` indicates that it needs to process tokens [1, 2, 3, 5, 6]. Among them, nodes [1, 2, 3] are hit in the GPU cache, while nodes [5, 6] result in cache misses and require recomputation. Since the GPU cache is already full, the system must evict some nodes to make room for the newly generated KV entries. When relying solely on historical information, the system evicts nodes 9 and 8 since they are ranked lowest in the `HistoryPriorityQueue`, even though they will soon be reused in job 6. This results in frequent evictions and reloads. By applying a lookahead window to capture future access patterns and re-evaluating node priorities using Equation 6, DepCache raises the priorities of nodes [1, 2, 3] and [7, 8, 9], while lowering the priorities of nodes [16, 17], which are not accessed in the window. This avoids unnecessary eviction and reload of nodes [8, 9].

In our cache replacement strategy, non-leaf nodes are not evicted directly. A non-leaf node becomes eligible for eviction only after all of its children have been evicted, at which point it is treated as a new leaf. This design is grounded in our dependency-aware prefix-matching mechanism, which operates from the root of the KV cache tree. A non-leaf node represents a shared prefix and is reused by multiple nodes in its subtree. Prematurely evicting such a node would invalidate the reuse of all dependent cache entries, significantly undermining cache efficiency.

Algorithm 2: Workflow of DepCache

Input: Prompt P; maximum decode length L; Tokenizer T
Output: Generated token sequence O

1 $O \leftarrow [\,]$ //initialize the output token sequence
2 $C \leftarrow$ CacheManager() //initialize the KV cache manager

Phase 1: Prefill (KV Cache Preparation)

3 $S, \mathcal{M} \leftarrow$ DepAttn(P, T) //construct token sequence and attention mask via DepAttn (Algorithm 1)
4 $HitKV, MissSeq \leftarrow C$.match$(S)$ //retrieve reusable KV cache and unmatched tokens
5 C.alloc$(|MissSeq|)$ //allocate cache slots for unmatched tokens
6 $PastKV \leftarrow$ GKVR$(HitKV, \mathcal{S})$ //adjust positional encoding of reused KV via GKVR
7 $NewKV, t \leftarrow$ LLM.Forward$(MissSeq, \mathcal{M}, PastKV)$ //compute KV for unmatched tokens and generate first token
8 C.insert$(NewKV)$ //insert new KV into cache
9 $PastKV \leftarrow PastKV \| NewKV$ //concatenate reused and new KV
10 $O \leftarrow O \| t$ //append first token

Phase 2: Decode (Token Generation)

11 **while** $|O| < L$ ***and*** *last token* $t \neq$ <eos> **do**
12 $NewKV, t \leftarrow$ LLM.Forward$(t, PastKV)$
13 $PastKV \leftarrow PastKV \| NewKV$ //update KV cache
14 $O \leftarrow O \| t$ //append generated token
15 **return** O

4.3 Overall Execution Flow in DepCache

Algorithm 2 outlines the execution workflow of DepCache using a user request as an example. For simplicity, the request prompt P is assumed to contain a single retrieved subgraph. The workflow consists of two phases: prefill (lines 3-10) and decode (lines 11-14). In the prefill stage, the input prompt P is first processed by the DepAttn procedure (Algorithm 1) to generate the token sequence $\mathcal{S}$ and its dependency attention mask $\mathcal{M}$ (line 3). The cache manager then performs a dependency-aware prefix match over $\mathcal{S}$ to retrieve the reusable KV entries (*HitKV*) and identify unmatched tokens (*MissSeq*) (line 4). To prepare for KV computation of unmatched tokens, memory slots are allocated in advance (line 5). If the requested memory exceeds the available GPU capacity, our locality-aware KV cache replacement (LKVR) strategy is triggered during `alloc`, which evicts low-priority KV entries based on a weighted combination of historical and future access patterns (Equation 6) until sufficient space is freed. The reusable KV entries are then positionally adjusted via our graph-based KV reuse mechanism to align with the new token sequence (line 6). These adjusted entries, together with the attention mask $\mathcal{M}$ and the unmatched tokens *MissSeq*, are passed to the LLM to compute new KV entries (*NewKV*) and produce the first output token t (line 7). The newly generated entries are inserted into the cache for future reuse (line 8). Finally, the complete KV cache *PastKV* is assembled by concatenating the adjusted and newly generated KV entries (line 9), which will be reused in the subsequent decode phase. During the decode phase, DepCache generates tokens autoregressively until generating <eos> or reaching the maximum length L. At each step, it feeds the latest token t along with *PastKV* into the LLM to compute the KV entry for t and generate the next token (Line 12).

5 Implemetation

We implement DepCache on top of SGLang [101] with about 11K lines of code in Python based on PyTorch v2.4 and CUDA 12.4.

Prompt dependency management: In GraphRAG, the input prompt to the LLM comprises three semantically distinct components: **system**, **context**, and **task**, corresponding to system instructions, the retrieved subgraph, and the user query, respectively (as illustrated in Figure 2). These components exhibit fundamentally different structural dependencies—`system` and `task` follow a linear sequence, while `context`, derived from the graph retrieval process, exhibits graph-structured dependencies. To uniformly manage these heterogeneous dependency structures, we introduce the `DepPrompt` data structure. `DepPrompt` models linear sequences using `LinearNode` and graph-structured inputs using `TreeNode`, enabling a consistent representation across all prompt segments. By default, the prompt follows a hierarchical information flow: context depends on system, and task depends on context, forming a top-down dependency chain. Moreover, `DepPrompt` supports customizable dependency structures to accommodate complex prompt designs tailored to different task requirements [6, 85].

Integrating into LLM inference framework: DepCache operates the LLM through two interfaces:

- `compute_depkv(dep_prompt)` → `KVCache`: given a `DepPrompt` instance (dep_prompt), DepCache invokes LLM through this function call to obtain the corresponding KV cache.
- `generate_with_depkv(KVCache)` → `text`: DepCache passes the KV cache to the LLM and enables it to generate tokens. The `prefill` step is skipped since the KV cache for the prompt has already been computed in the previous step.

For `compute_depkv`, we first invoke `tree_cache.match_prefix` over a compressed prefix tree to identify the token IDs that miss the cache. For unmatched suffix tokens, DepCache allocates fresh KV space. We then call `DepPrompt.compute_attention_mask` to generate dependency-aware attention masks for these tokens. These masks are stored in SGLang's `forward_batch` and subsequently consumed during the forward pass. We adopt the batched decoding strategy from SGLang [101], which maintains an index to determine whether KV cache entries should be reused or recomputed. For each request, DepCache performs prefix matching over a compressed prefix tree to determine which KV cache entries can be reused and which require recomputation. The result is stored as per-request metadata in SGLang's `forward_batch` and passed to the attention backend, which uses it to selectively read from and write to the KV cache during execution. We leverage FlashInfer [87] as the attention backend and enable dependency-aware attention by configuring the `custom_mask` parameter.

KV cache management in DepCache**:** DepCache pre-allocates contiguous memory regions on both the GPU and CPU to store KV cache entries. We extend the `RadixCache` data structure from SGLang, which uses a radix tree for efficient prefix matching, by incorporating vertex-level locality priority scores to support a locality-aware replacement strategy. DepCache leverages a lookahead window to capture temporal locality in query workloads. In practice, we set the window size to match the number of pending queries in the queue. Each request is assigned a decreasing temporal priority based on its position in the queue, ensuring that queries closer to execution have higher influence on eviction decisions.

Table 1. Dataset description.

Dataset	Question	Type	Entity	Relationship
RGB [8]	500	Inference	54,544	74,394
Multihop [68]	816	Inference	30,953	26,876
DragonBall [104]	533	Summary	5,978	6,716

6 Evaluation

6.1 Experimental Setup

Environments. The experimental platform consists of two Intel Xeon Silver 4316 CPUs, each with 20 cores at 2.30 GHz, providing a total of 40 cores, and is equipped with 512 GB of DRAM. The system includes two NVIDIA RTX A6000 GPUs, each with 84 streaming multiprocessors (SMs), 10,752 CUDA cores, and 48 GB of GDDR6 memory. The GPUs run on the CUDA 12.4 runtime with driver version 560.35. The host environment is configured with Ubuntu 20.04 LTS and Linux kernel 5.15.0.

Datasets. Our evaluation covers the following datasets:

- *RGB* [8]: It is a question-answering dataset constructed from recent news articles, designed to evaluate different capabilities of LLMs in RAG tasks. It supports multilingual evaluation in both English and Chinese.
- *MultiHop* [68]: This dataset is composed of news articles published between 2013 and 2023, specifically designed for multi-hop query tasks. Answering these questions requires integrating information from multiple articles to form a complete response.
- *DragonBall* [104]: This RAG benchmark dataset covers three critical domains: finance, law, and medicine. It is designed to assess diverse reasoning and retrieval capabilities.

We evaluate our system on representative types of queries from each benchmark dataset. Specifically, we use all 500 English-language queries from the RGB dataset, all 816 inference-type queries from the MultiHop dataset, and all 533 summary-type English-language questions from the DragonBall dataset. Since the datasets we use are primarily designed for vector-based RAG methods and only provide raw documents without corresponding structured knowledge graphs, we construct a knowledge graph from the documents in each dataset to support the GraphRAG task. Specifically, we first split each document into text chunks of 512 tokens. Then, we leverage the ChatGPT-4o model to extract entities and their relationships from each text chunk. These entities and relationships are inserted into a graph database to construct the knowledge graph. Table 1 summarizes the dataset statistics and the scale of the constructed knowledge graphs.

We also construct a synthetic dataset to evaluate the effectiveness of caching strategies under different query workloads. We generate 500 question-answer pairs based on entities and relation paths sampled from the RGB knowledge graph. Specifically, for each synthetic query, we sample 10 entities from the RGB knowledge graph and extract up to 30 two-hop relation paths per entity. These paths form contextual subgraphs, which are encoded into prompt templates and passed to an LLM to generate question–answer pairs. Since no public user query traces are available for RAG scenarios, we construct synthetic query workloads using three widely adopted access patterns to simulate realistic query distributions: (1) Temporal: models user interactions in sessions or multi-turn dialogues, where recent entities are more likely to be queried. (2) Zipfian: captures heavy-tailed access distributions, where a small number of entities dominate query volume, as is common in LLM-based generation services with frequent hotspot queries. (3) Uniform: provides a baseline where all entities are equally likely to be queried, enabling a controlled comparison

across caching strategies. These workload patterns are commonly used to construct synthetic query workloads in evaluation [14, 33, 38, 40, 64].

Models. We evaluate DepCache on four open-source large language models, including two small-scale models (Llama3-8B and Mistral-7B-v0.3) and two medium-scale models (Qwen2.5-14B and Qwen2.5-32B). Llama3-8B, Mistral-7B-v0.3, and Qwen2.5-14B are deployed on a single GPU, while Qwen2.5-32B is served using tensor parallelism across two GPUs.

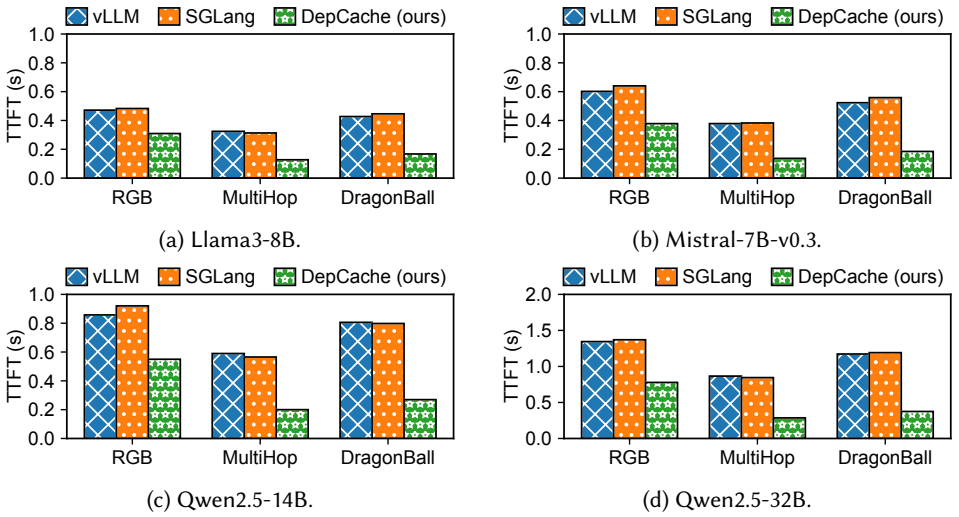
Baseline. We compare DepCache with two state-of-the-art LLM serving systems: vLLM [36] and SGLang [101]. vLLM introduces PagedAttention, an attention algorithm inspired by virtual memory and paging techniques in operating systems, to manage the KV cache memory efficiently. This innovation addresses memory fragmentation and redundancy issues, enabling near-zero waste in KV cache memory. SGLang introduces RadixAttention, a technique for automatic and efficient KV cache reuse across multiple LLM generation calls, thereby reducing redundant memory usage and computation time.

Metrics. We compare DepCache with the baselines using three system-wide metrics (*TTFT*, *Throughput*, and *Cache hit rate*) and two standard metrics (*Accuracy* and *Rouge-L score* [42]) for evaluating generation quality.

- *Time-to-first-token (TTFT)* measures the latency between the arrival of a user query and the generation of the first token. Minimizing TTFT is crucial for real-time interactions to ensure a responsive user experience.
- *Throughput* measures the maximum request load (requests per second) that a system can sustain while meeting a specified latency target. Higher throughput allows the system to handle more requests with the same resources, thereby reducing service costs.
- *Cache hit rate* evaluates the effectiveness of different caching strategies. We measure the cache hit rate at the token level. For each request, the hit rate is the ratio of cache-hit tokens to total tokens, and the system-wide hit rate is the average across all requests.
- *Accuracy* is used to evaluate the model's performance on the RGB and MultiHop datasets. A model-generated answer is deemed correct if it appears in the candidate answer set for a given question. The accuracy is defined as the percentage of correctly answered questions over the total number of questions.
- *Rouge-L score* is used to evaluate the DragonBall dataset. It measures the similarity between the model's output and the ground-truth summaries based on the longest common sequence.

6.2 End-to-end Performance

We conduct an end-to-end evaluation comparing two state-of-the-art inference systems, vLLM and SGLang, across three representative RAG datasets: RGB, MultiHop, and DragonBall, using four LLMs of varying scales: Llama3-8B, Mistral-7B-v0.3, Qwen2.5-14B, and Qwen2.5-32B. The maximum batch size is set to 4. We report the average Time-To-First-Token (TTFT) and evaluate system throughput under varying request rates. Figure 10 presents the TTFT results. DepCache reduces the average TTFT by 1.5×−3.1× compared to vLLM, and by 1.6×−3.2× compared to SGLang. These improvements stem from two key factors: (1) the use of dependency attention, which eliminates a significant amount of redundant attention computation; and (2) the design of DepCache, which exploits the locality of graph access patterns to enable efficient KV cache reuse through a graph-based KV cache reuse method and a locality-aware cache replacement policy. Figure 11 shows the throughput comparison. Benefiting from reduced request latency, DepCache achieves 1.5×-5.0× higher throughput than vLLM and 1.6×-4.3× higher than SGLang. The results also reveal performance differences across datasets. Notably, DepCache achieves significantly better TTFT and throughput on the MultiHop dataset compared to RGB and DragonBall. This improvement is



(a) Llama3-8B. (b) Mistral-7B-v0.3.

(c) Qwen2.5-14B. (d) Qwen2.5-32B.

Fig. 10. DepCache reduces TTFT by 1.5×–3.2× compared to SOTA LLM inference frameworks (vLLM and SGLang) across three datasets and four models.

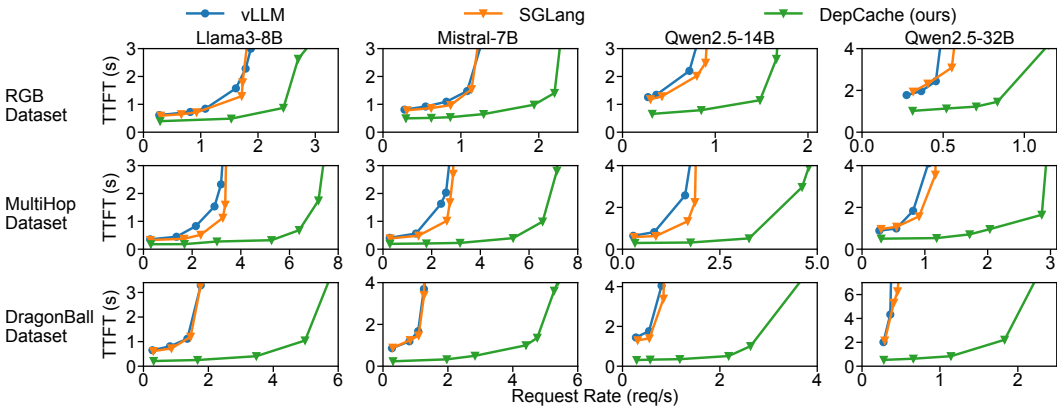
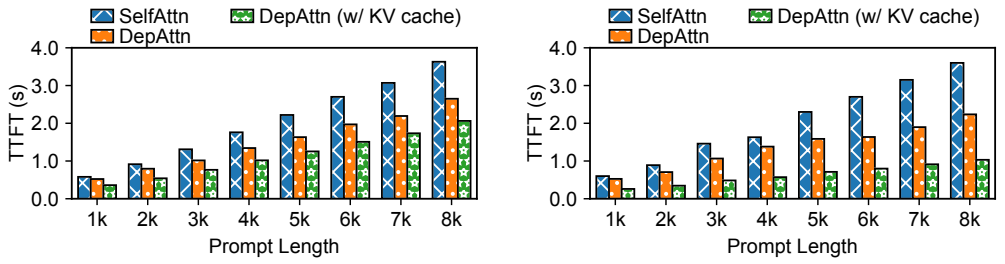


Fig. 11. DepCache improves throughput by 1.5–5.0× compared to SOTA LLM inference frameworks (vLLM and SGLang) across three datasets and four models.

related to the locality of the dataset. As shown in Figure 4, MultiHop exhibits stronger retrieval locality, enabling more effective KV cache reuse and further reducing computational overhead.

6.3 Performance Gain Analysis

We analyze the performance gain of dependency attention (DepAttn) and KV cache management of DepCache (KV cache) under varying prompt lengths ranging from 1K to 8K tokens. To ensure a fair comparison, we start with a self-attention baseline based on DepCache codebase and gradually integrate the two optimization methods. We conduct experiments using the Qwen2.5-32B model, deployed with tensor parallelism across two A6000 GPUs. The batch size is set to 4, and evaluations are performed on the RGB and MultiHop datasets. For each query, we retrieve 20 subgraphs from



(a) Qwen2.5-32B on RGB. (b) Qwen2.5-32B on MultiHop.

Fig. 12. Comparison of self-attention and dependency attention across varying prompt lengths.

the graph database, each containing up to 50 relation paths. To construct prompts of a target length, the subgraphs are appended sequentially until the predefined length constraint is met.

Figure 12 shows the results. Compared to the baseline (SelfAttn), DepAttn reduces TTFT by 1.06×–1.73×. This improvement comes from avoiding redundant attention by restricting computation to structurally dependent tokens. The performance gain of DepAttn is less significant for shorter prompts, as retrieved subgraphs account for a smaller portion of the input and thus offer limited opportunity for reducing attention computation. Compared to DepAttn, DepAttn (w/ KV cache) achieves a 1.27×–2.69× reduction in TTFT. This is enabled by the ability of DepCache to reuse subgraph-level KV caches across queries, leveraging strong locality in graph access patterns (Figure 4). The performance gain of DepAttn (w/ KV cache) remains stable across different prompt lengths, as GraphRAG's retrieval exhibits strong locality and its access patterns do not vary significantly with sequence length.

6.4 Analysis of Locality-aware KV Cache Replacement

6.4.1 Comparison with Cache Replacement policies. We evaluate our locality-aware cache replacement strategy (LACR) with three commonly used baselines—LRU, LFU, and PGDSF—focusing on improvements in cache hit rates under varying GPU memory capacities (4GB, 8GB, 16GB) and access distributions (Uniform, Temporal, Zipf).

Figure 13 shows the results. Overall, LACR consistently outperforms all three baselines across settings, achieving average improvements of 10.1%, 6.7%, and 7.2% over LRU, LFU, and PGDSF, respectively. This is attributed to LACR effectively combining LFU's ability to retain high-frequency nodes with a lookahead window that captures upcoming reuse, allowing it to adapt to both spatial and temporal locality.

Under the uniform distribution, where accesses are nearly random and historical frequency offers little predictive value, traditional strategies perform poorly, with hit rates not exceeding 40%. In contrast, LACR leverages short-term future access information to achieve hit rate gains of 16.7%, 8.1%, and 11.9% over LRU, LFU, and PGDSF, respectively.

Under the temporal distribution, where access exhibits strong short-term locality, LRU and PGDSF benefit from their recency-based design and achieve improvements over the uniform setting by 41.9% and 34.7%, respectively. However, LACR still achieves higher hit rates, with improvements of 4.5% over LRU and 7% over PGDSF, demonstrating its ability to complement temporal reuse with future-aware decisions. LFU performs worse in this setting due to its lack of recency awareness, but LACR closes this gap and outperforms it by up to 8.3%.

Under the Zipf distribution, where accesses are heavily skewed toward a small number of popular entities, LFU performs best among the baselines, aligning well with the distribution's long-tail

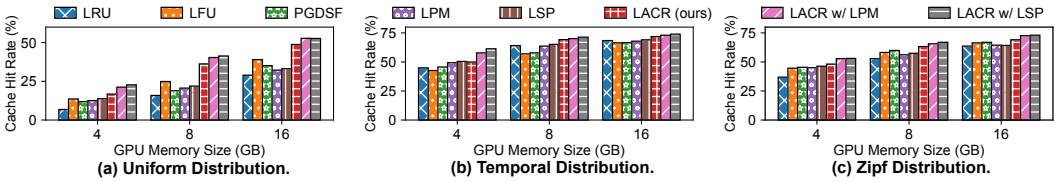
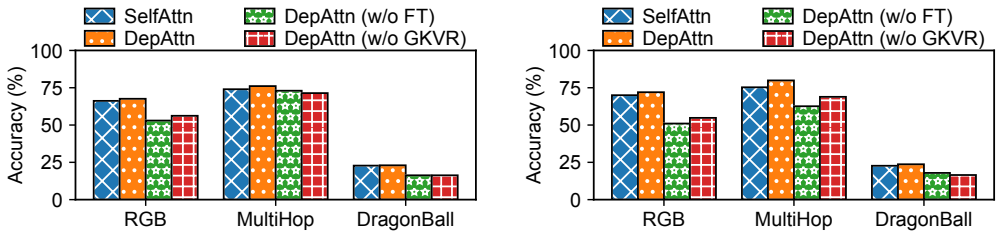


Fig. 13. Cache hit rate comparison under different query workloads.



(a) Llama3-8B. (b) Mistral-7B-v0.3.

Fig. 14. Accuracy of different attention methods.

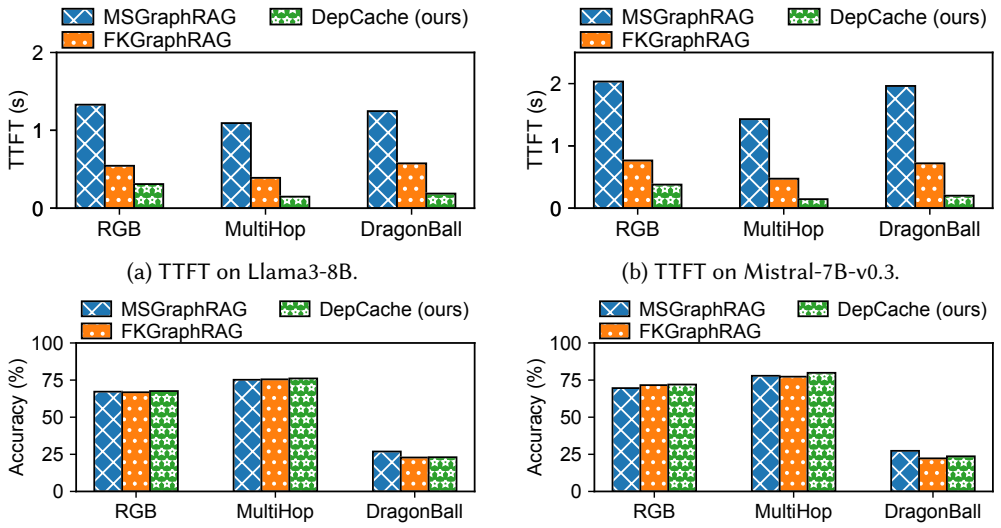
nature. On the other hand, LACR still achieves additional gains of 9%, 3.7%, and 2.7% over LRU, LFU, and PGDSF, respectively, by retaining both frequently and imminently used entries. These results highlight the robustness and adaptability of LACR across different workload characteristics, enabling consistently higher cache efficiency regardless of the underlying data access pattern.

6.4.2 Comparison with Cache-aware Request Scheduling Strategies. In addition to classic cache replacement baselines (LRU, LFU, PGDSF), we also compare against two recent cache-aware request scheduling methods—Longest Prefix Match (LPM, as in RadixAttention [101]) and Longest Shared Prefix (LSP, as in BatchLLM [102])—which dynamically adjust the execution order of requests based on current cache contents to improve cache utilization. Unlike traditional replacement policies that decide which entries to evict, these methods enhance cache hit rates by optimizing request scheduling. Notably, cache-aware scheduling and cache replacement policies are complementary: the former improves cache utilization through intelligent scheduling, while the latter enhances cache retention through optimized eviction.

As shown in Figure 13, LACR consistently achieves higher hit rates than the cache-aware scheduling methods LPM and LSP across all access patterns, with average improvements of 6.8% and 5.7%, respectively. While LPM and LSP improve hit rates over LRU by 5.2% and 1.8% on average under uniform and temporal distributions by scheduling requests with shared prefixes to enhance prefix reuse, they perform worse than LFU and PGDSF under the skewed Zipf distribution due to their inability to capture global popularity beyond the local request queue. Furthermore, combining LACR with cache-aware scheduling yields additional gains, with hit rates improved by 3.7% and 4.8% over LACR alone, demonstrating the complementary nature of scheduling and replacement strategies.

6.5 Accuracy Comparisons

We evaluate the inference accuracy of our proposed Dependency Attention (DepAttn) against the standard Self-Attention (SelfAttn) baseline across three downstream tasks. To enhance the accuracy of DepAttn, we incorporate two key techniques: dependency-aware fine-tuning and a graph-based KV cache reuse method (GKVR). Fine-tuning enables the model to adapt to the structure-aware



(a) TTFT on Llama3-8B.

(b) TTFT on Mistral-7B-v0.3.

(c) Accuracy on Llama3-8B.

(d) Accuracy on Mistral-7B-v0.3.

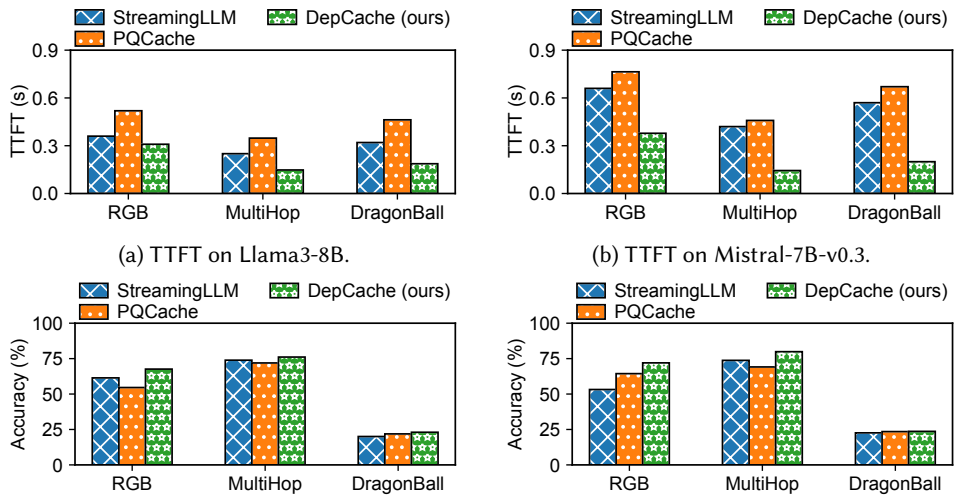
Fig. 15. Comparison of DepCache and GraphRAG baselines in terms of TTFT and accuracy.

attention patterns imposed by dependency masks, while GKVR facilitates cross-query KV reuse for entities appearing in different positions via relative positional encoding. The fine-tuning procedure is as follows: we fine-tune Llama3-8B and Mistral-7B-v0.3 using the DeepSpeed [12, 57] training framework on two NVIDIA A6000 GPUs. ZeRO Stage-3 optimization with CPU offloading is enabled for model parameters and optimizer states, allowing efficient training under constrained GPU memory. The fine-tuning is lightweight, using a learning rate of $\alpha = 2 \times 10^{-5}$, a batch size of $b = 64$, and a total of $n = 1$ training epoch.

We conduct experiments on three datasets: RGB and MultiHop (question answering, evaluated by *Accuracy*), and DragonBall (summarization, evaluated by *ROUGE-L F1*). As shown in Figure 14, DepAttn achieves comparable or slightly better inference accuracy than SelfAttn, with up to 4.6% higher accuracy on MultiHop. To isolate the contributions of each component, we evaluate two ablated variants. The first, DepAttn w/o FT, directly applies dependency attention without any fine-tuning. This leads to substantial degradation, with accuracy dropping by 17.8% on RGB, 10.2% on MultiHop, and 6.3% on DragonBall, as the model has not been exposed to the structural attention patterns during training. The second variant, DepAttn w/o GKVR, disables the graph-based KV reuse mechanism. In this setting, direct reuse of KV cache without position alignment causes a 9.7% average accuracy drop, highlighting the importance of position-aware reuse in maintaining generation quality.

6.6 GraphRAG Baseline Comparisons

Existing GraphRAG approaches can be broadly categorized into two types: chunk-based and KG-based. We compare two representative GraphRAG methods. The first is Microsoft's GraphRAG [17] (MSGraphRAG), a chunk-based method tailored for thematic and summarization-style QA, which generates hierarchical community summaries through graph clustering. The second is FKGraphRAG, a KG-based method built on the FalkorDB [18] graph database, which supports multi-hop reasoning by directly retrieving entity–relation graphs from the knowledge graph. Experiments are conducted using Llama3-8B and Mistral-7B-v0.3 on three datasets (RGB, MultiHop, and DragonBall), focusing



(a) TTFT on Llama3-8B.

(b) TTFT on Mistral-7B-v0.3.

(c) Accuracy on Llama3-8B.

(d) Accuracy on Mistral-7B-v0.3.

Fig. 16. Comparison of DepCache and sparse attention baselines in terms of TTFT and accuracy.

on TTFT and accuracy. As shown in Figure 15, DepCache consistently achieves the lowest TTFT, with average speedups of 7.8× over MSGraphRAG and 2.9× over FKGraphRAG. These gains stem from its graph-aware attention and KV cache reuse, which reduce redundant computation and enable efficient cross-query cache reuse. In terms of accuracy, DepCache achieves comparable performance to FKGraphRAG and MSGraphRAG. MSGraphRAG slightly outperforms on DragonBall due to its chunk-based global summarisation, which is specifically designed for summarisation-oriented QA.

6.7 Sparse Attention Comparisons

Recently, researchers have proposed various sparse attention methods [27, 80, 94] to address the high KV cache memory consumption in long-context LLM inference. We conduct a comprehensive evaluation of DepCache and the two sparse attention baselines (StreamingLLM [80] and PQCache [94]) using Llama3-8B and Mistral-7B-v0.3 on three datasets (RGB, MultiHop, and DragonBall), focusing on TTFT and accuracy. As shown in Figure 16, DepCache consistently outperforms both baselines in terms of TTFT and accuracy across all datasets and models. Specifically, it achieves average TTFT speedups of 2.2× over StreamingLLM and 2.7× over PQCache. Although sparse attention reduces computation by focusing on a subset of important tokens, it overlooks retrieval locality and incurs redundant KV computation for repeated subgraphs. In contrast, DepCache eliminates this overhead via graph-aware attention and cross-query KV reuse. In terms of accuracy, DepCache achieves average accuracy improvements of 6.3% over StreamingLLM and 6.1% over PQCache. This is because heuristic sparse attention methods typically restrict attention to a small subset of important tokens, which may miss critical dependencies due to the complex structures in graph-based contexts. In contrast, DepCache explicitly models token dependencies based on the underlying graph structure, leading to higher accuracy.

7 Discussion

Compatibility with Common Graph Serialization Strategies. For GraphRAG methods based on entity–relationship graphs, graph serialization is essential to enable LLMs to process graph-structured inputs. Two widely adopted serialization strategies are: triplet-based serialization [11,

29, 43], which avoids repeated prefixes but captures only one-hop dependencies, and path-based serialization [51, 52, 67, 78], which explicitly encodes multi-hop paths at the cost of repeating shared prefixes. Our approach does not rely on any specific serialization format. Both serialization strategies can benefit from our dependency-aware attention mechanism. For path-based serialization, it reduces redundant attention computations over shared prefixes and unrelated tokens. Similarly, for triplet-based serialization, it effectively reduces unnecessary attention between tokens without explicit dependency.

Inference Optimization Orthogonal to Graph Construction. While graph construction—often involving LLM-based entity extraction, precomputation, and index maintenance—can be computationally expensive, it is typically performed offline and reused across all inference tasks. Periodic updates can be executed in batch mode without affecting online performance. In contrast, online LLM inference directly impacts query latency, significantly increasing TTFT by 6–16× compared to the NoRAG baseline, due to longer input prompts introduced by graph-structured retrieval. Therefore, optimizing inference overhead is crucial, especially in real-world scenarios that require high throughput and low latency. Our work targets the inference inefficiency in GraphRAG caused by these long inputs. It is orthogonal to graph construction and can be seamlessly integrated into any GraphRAG method using entity–relationship graphs, regardless of the construction strategy.

8 Related Work

LLM Inference Optimization for RAG: RAG [8, 17, 19, 21, 31, 37, 73, 99] enhances the reliability of LLMs by incorporating external knowledge, but incurs substantial generation overhead. RAGCache [32] accelerates inference through a multi-level cache that stores document-level KV pairs. CacheBlend [84] and CacheCraft [2] extend reuse to non-prefix positions by selectively recomputing a small subset tokens to preserve output quality. In contrast, DepCache focuses on graph-based RAG, where knowledge graphs serve as structured external sources [17, 26, 29, 41, 56, 60, 61, 83, 96].

KV Cache Management in LLM Systems: Recent studies [20, 22, 36, 44, 86, 101] aim to reduce redundant computation in LLM inference by reusing the KV cache across requests. PromptCache [22] use domain-specific languages (DSLs) to define shared prompt prefixes. BatchLLM [102] schedules prefix-sharing requests and reorders them by decoding ratios to improve reuse and GPU efficiency. CachedAttention [20] introduces a hierarchical KV cache with decoupled position encoding for efficient reuse. In contrast, our work targets KV cache management in GraphRAG by leveraging its unique graph structure and access patterns, enabling effective cross-query reuse.

9 Conclusion

This paper proposes dependency attention, a novel mechanism that restricts attention computation to structurally dependent token pairs, significantly reducing redundant computation. Building on this mechanism, we present DepCache, a low-latency and high-throughput inference framework tailored for GraphRAG. DepCache integrates a graph-based KV cache reuse strategy and a locality-aware cache replacement policy to support efficient KV cache reuse and maximize cache efficiency under memory constraints. Experimental results demonstrate that DepCache achieves up to 5× throughput improvement and up to 3.2× reduction in TTFT, while preserving generation quality comparable to state-of-the-art LLM inference systems.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (62461146205, U2241212), the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501), and the CAAI–Huawei AI Computing Acceleration Program research grant.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Shubham Agarwal, Sai Sundaresan, Subrata Mitra, Debabrata Mahapatra, Archit Gupta, Rounak Sharma, Nirmal Joshua Kapu, Tong Yu, and Shiv Saini. 2025. Cache-Craft: Managing Chunk-Caches for Efficient Retrieval-Augmented Generation. *arXiv preprint arXiv:2502.15734* (2025).

[3] Xin Ai, Hao Yuan, Zeyu Ling, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. NeutronTP: Load-Balanced Distributed Full-Graph GNN Training with Tensor Parallelism. *Proceedings of the VLDB Endowment* 18, 2 (2024), 173–186.

[4] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[5] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *arXiv preprint arXiv:2004.05150* (2020).

[6] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michał Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of thoughts: solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '24)*.

[7] Tianle Cai, Kaixuan Huang, Jason D Lee, and Mengdi Wang. 2023. Scaling In-Context Demonstrations with Structured Attention. *arXiv preprint arXiv:2307.02690* (2023).

[8] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking large language models in retrieval-augmented generation. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '24, Vol. 38)*. 17754–17762.

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

[10] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* (2019).

[11] Nurendra Choudhary and Chandan K Reddy. 2023. Complex logical reasoning over knowledge graphs using large language models. *arXiv preprint arXiv:2305.01157* (2023).

[12] DeepSpeed. 2020. https://github.com/deepspeedai/DeepSpeed.

[13] Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Mingzhe Du and Michael L Scott. 2024. Buffered Persistence in B+ Trees. *Proceedings of the ACM on Management of Data* 2, 6, Article 226 (2024).

[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[16] Vijay Prakash Dwivedi and Xavier Bresson. 2020. A Generalization of Transformer Networks to Graphs. *arXiv preprint arXiv:2012.09699* (2020).

[17] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* (2024).

[18] FalkorDB. 2023. https://github.com/FalkorDB/FalkorDB.

[19] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD '24)*. 6491–6501.

[20] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-efficient large language model serving for multi-turn conversations with CachedAttention. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference (ATC '24)*.

[21] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* (2023).

[22] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems* 6 (2024), 325–338.

[23] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793* (2024).

[24] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[25] Qipeng Guo, Xipeng Qiu, Pengfei Liu, Yunfan Shao, Xiangyang Xue, and Zheng Zhang. 2019. Star-Transformer. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL '19)*. 1315–1325.

[26] Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. 2024. LightRAG: Simple and Fast Retrieval-Augmented Generation. *arXiv preprint arXiv:2410.05779* (2024).

[27] Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. 2024. LM-Infinite: Zero-Shot Extreme Length Generalization for Large Language Models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers) (NAACL '24)*. 3991–4008.

[28] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A Rossi, Subhabrata Mukherjee, Xianfeng Tang, et al. 2024. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309* (2024).

[29] Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. 2024. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. *Advances in Neural Information Processing Systems* 37 (2024), 132876–132907.

[30] 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).

[31] 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).

[32] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *arXiv preprint arXiv:2404.12457* (2024).

[33] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th symposium on operating systems principles (SOSP '17)*. 121–136.

[34] 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) (ACL '17)*. 1601–1611.

[35] Shun Kiyono, Sosuke Kobayashi, Jun Suzuki, and Kentaro Inui. 2021. Shape: Shifted absolute position embedding for transformers. *arXiv preprint arXiv:2109.05644* (2021).

[36] 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 *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. 611–626.

[37] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS '20)*.

[38] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with In-Network coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. 387–406.

[39] Peizheng Li, Chaoyi Chen, Hao Yuan, Zhenbo Fu, Hang Shen, Xinbo Yang, Qiange Wang, Xin Ai, Yanfeng Zhang, Yingyou Wen, and Ge Yu. 2025. NeutronRAG: Towards Understanding the Effectiveness of RAG from a Data Retrieval Perspective *(SIGMOD '25)*. 163–166.

[40] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.

[41] Lei Liang, Mengshu Sun, Zhengke Gui, Zhongshu Zhu, Zhouyu Jiang, Ling Zhong, Yuan Qu, Peilong Zhao, Zhongpu Bo, Jin Yang, et al. 2024. KAG: Boosting LLMs in Professional Domains via Knowledge Augmented Generation. *arXiv preprint arXiv:2409.13731* (2024).

[42] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Association for Computational Linguistics (ACL '04)*. 74–81.

[43] Guangyi Liu, Yongqi Zhang, Yong Li, and Quanming Yao. 2025. Dual Reasoning: A GNN-LLM Collaborative Framework for Knowledge Graph Question Answering. In *The Second Conference on Parsimony and Learning (Proceedings Track) (CPAL '25)*.

[44] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM '24)*. 38–56.

[45] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems* 36 (2023), 52342–52364.

[46] Antoine Liutkus, Ondřej Cıfka, Shih-Lun Wu, Umut Simsekli, Yi-Hsuan Yang, and Gael Richard. 2021. Relative positional encoding for transformers with linear complexity. In *International Conference on Machine Learning (PMLR '21)*. 7067–7079.

[47] Meta Llama. 2024. https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct.

[48] Llama3-70B. 2024. https://huggingface.co/meta-llama/Meta-Llama-3-70B.

[49] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, et al. 2025. MoBA: Mixture of Block Attention for Long-Context LLMs. *arXiv preprint arXiv:2502.13189* (2025).

[50] Songshuo Lu, Hua Wang, Yutian Rong, Zhi Chen, and Yaohua Tang. 2024. TurboRAG: Accelerating Retrieval-Augmented Generation with Precomputed KV Caches for Chunked Text. *arXiv preprint arXiv:2410.07590* (2024).

[51] LINHAO LUO, Yuan-Fang Li, Gholamreza Haffari, and Shirui Pan. 2024. Reasoning on Graphs: Faithful and Interpretable Large Language Model Reasoning. In *The Twelfth International Conference on Learning Representations (ICLR '24)*.

[52] Shengjie Ma, Chengjin Xu, Xuhui Jiang, Muzhi Li, Huaren Qu, Cehao Yang, Jiaxin Mao, and Jian Guo. 2025. Think-on-Graph 2.0: Deep and Faithful Large Language Model Reasoning with Knowledge-guided Retrieval Augmented Generation. In *The Thirteenth International Conference on Learning Representations (ICLR '25)*.

[53] Thomas Merth, Qichen Fu, Mohammad Rastegari, and Mahyar Najibi. 2024. Superposition Prompting: Improving and Accelerating Retrieval-Augmented Generation. In *International Conference on Machine Learning (ICML '24)*.

[54] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. arXiv 2024. *arXiv preprint arXiv:2402.06196* (2024).

[55] OpenAI. 2024. https://openai.com/blog/chatgpt.

[56] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. 2024. Graph retrieval-augmented generation: A survey. *arXiv preprint arXiv:2408.08921* (2024).

[57] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. 1–16.

[58] Ladislav Rampásek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. 2022. Recipe for a General, Powerful, Scalable Graph Transformer. In *Advances in Neural Information Processing Systems 35 (NeurIPS '22)*.

[59] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).

[60] Diego Sanmartin. 2024. Kg-rag: Bridging the gap between knowledge and creativity. *arXiv preprint arXiv:2405.12035* (2024).

[61] Bhaskarjit Sarmah, Benika Hall, Rohan Rao, Sunil Patel, Stefano Pasquali, and Dhagash Mehta. 2024. HybridRAG: Integrating Knowledge Graphs and Vector Retrieval Augmented Generation for Efficient Information Extraction. *arXiv preprint arXiv:2408.04948* (2024).

[62] Ahsan Shehzad, Feng Xia, Shagufta Abid, Ciyuan Peng, Shuo Yu, Dongyu Zhang, and Karin Verspoor. 2024. Graph transformers: A survey. *arXiv preprint arXiv:2407.09777* (2024).

[63] Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J. Sutherland, and Ali Kemal Sinop. 2023. Exphormer: Sparse Transformers for Graphs. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (PMLR '23, Vol. 202)*. 31613–31632.

[64] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2025. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. In *The Thirteenth International Conference on Learning Representations (ICLR '25)*.

[65] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.

[66] East Sun, Yan Wang, and Lan Tian. 2024. Block-Attention for Efficient RAG. *arXiv preprint arXiv:2409.15355* (2024).

[67] Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel Ni, Heung-Yeung Shum, and Jian Guo. 2024. Think-on-Graph: Deep and Responsible Reasoning of Large Language Model on Knowledge Graph. In *The Twelfth International Conference on Learning Representations (ICLR '24)*.

[68] Yixuan Tang and Yi Yang. 2024. MultiHop-RAG: Benchmarking Retrieval-Augmented Generation for Multi-Hop Queries. arXiv:2401.15391 [cs.CL]

[69] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[70] Qwen Team. 2024. https://huggingface.co/Qwen/Qwen2.5-32B-Instruct.

[71] 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. *arXiv preprint arXiv:2302.13971* (2023).

[72] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[73] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2023. Interleaving Retrieval with Chain-of-Thought Reasoning for Knowledge-Intensive Multi-Step Questions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL '23)*.

[74] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS '17)*. 5998–6008.

[75] Benyou Wang, Donghao Zhao, Christina Lioma, Qiuchi Li, Peng Zhang, and Jakob Grue Simonsen. 2019. Encoding word order in complex embeddings. *arXiv preprint arXiv:1912.12333* (2019).

[76] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs. *Proceedings of the ACM on Management of Data* 1, 4, Article 246 (2023).

[77] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. 1301–1315.

[78] Song Wang, Junhong Lin, Xiaojie Guo, Julian Shun, Jundong Li, and Yada Zhu. 2025. Reasoning of Large Language Models over Knowledge Graphs with Super-Relations. In *The Thirteenth International Conference on Learning Representations (ICLR '25)*.

[79] Nirmalie Wiratunga, Ramitha Abeyratne, Lasal Jayawardena, Kyle Martin, Stewart Massie, Ikechukwu Nkisi-Orji, Ruvan Weerasinghe, Anne Liret, and Bruno Fleisch. 2024. CBR-RAG: Case-Based Reasoning for Retrieval Augmented Generation in LLMs for Legal Question Answering. In *Case-Based Reasoning Research and Development - 32nd International Conference, ICCBR 2024, Merida, Mexico, July 1-4, 2024, Proceedings (ICCBR '24, Vol. 14775)*. 445–460.

[80] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient Streaming Language Models with Attention Sinks. In *The Twelfth International Conference on Learning Representations (ICLR '24)*.

[81] Shitao Xiao, Zheng Liu, Peitian Zhang, and N Muennighof. 2023. C-pack: packaged resources to advance general Chinese embedding. 2023. *arXiv preprint arXiv:2309.07597* (2023).

[82] Ran Xu, Wenqi Shi, Yue Yu, Yuchen Zhuang, Bowen Jin, May Dongmei Wang, Joyce C. Ho, and Carl Yang. 2024. RAM-EHR: Retrieval Augmentation Meets Clinical Predictions on Electronic Health Records. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, ACL 2024 - Short Papers, Bangkok, Thailand, August 11-16, 2024 (ACL '24)*. 754–765.

[83] Zhentao Xu, Mark Jerome Cruz, Matthew Guevara, Tie Wang, Manasi Deshpande, Xiaofeng Wang, and Zheng Li. 2024. Retrieval-augmented generation with knowledge graphs for customer service question answering. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '24)*. 2905–2909.

[84] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2024. CacheBlend: Fast Large Language Model Serving with Cached Knowledge Fusion. *arXiv preprint arXiv:2405.16444* (2024).

[85] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: deliberate problem solving with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NeurIPS '23)*.

[86] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL '24)*.

[87] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. 2025. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005* (2025).

[88] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, et al. 2024. Yi: Open foundation models by 01. ai. *arXiv preprint arXiv:2403.04652* (2024).

[89] Chaohao Yuan, Kangfei Zhao, Ercan Engin Kuruoglu, Liang Wang, Tingyang Xu, Wenbing Huang, Deli Zhao, Hong Cheng, and Yu Rong. 2025. A survey of graph transformers: Architectures, theories and applications. *arXiv preprint arXiv:2502.16533* (2025).

[90] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. Comprehensive Evaluation of GNN Training Systems: A Data Management Perspective. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1241–1254.

[91] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, et al. 2025. Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention. *arXiv preprint arXiv:2502.11089* (2025).

[92] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems* 33 (2020), 17283–17297.

[93] Boyu Zhang, Hongyang Yang, Tianyu Zhou, Ali Babar, and Xiao-Yang Liu. 2023. Enhancing Financial Sentiment Analysis via Retrieval Augmented Large Language Models. In *4th ACM International Conference on AI in Finance, ICAIF 2023, Brooklyn, NY, USA, November 27-29, 2023 (ICAIF '23)*. 349–356.

[94] Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. 2025. PQCache: Product Quantization-based KVCache for Long Context LLM Inference. *Proceedings of the ACM on Management of Data* 3, 3, Article 201 (2025).

[95] Meng Zhang, Jie Sun, Qinghao Hu, Peng Sun, Zeke Wang, Yonggang Wen, and Tianwei Zhang. 2024. TorchGT: A Holistic System for Large-Scale Graph Transformer Training. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '24)*. 1–17.

[96] Qinggang Zhang, Shengyuan Chen, Yuanchen Bei, Zheng Yuan, Huachi Zhou, Zijin Hong, Junnan Dong, Hao Chen, Yi Chang, and Xiao Huang. 2025. A Survey of Graph Retrieval-Augmented Generation for Customized Large Language Models. *arXiv preprint arXiv:2501.13958* (2025).

[97] Zhengyan Zhang, Xu Han, Zhiyuan Liu, Xin Jiang, Maosong Sun, and Qun Liu. 2019. ERNIE: Enhanced language representation with informative entities. *arXiv preprint arXiv:1905.07129* (2019).

[98] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2023), 34661–34710.

[99] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, Jie Jiang, and Bin Cui. 2024. Retrieval-augmented generation for ai-generated content: A survey. *arXiv preprint arXiv:2402.19473* (2024).

[100] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[101] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems* 37 (2024), 62557–62583.

[102] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. 2024. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching. *arXiv preprint arXiv:2412.03594* (2024).

[103] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. 2024. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294* (2024).

[104] Kunlun Zhu, Yifan Luo, Dingling Xu, Ruobing Wang, Shi Yu, Shuo Wang, Yukun Yan, Zhenghao Liu, Xu Han, Zhiyuan Liu, et al. 2024. Rageval: Scenario specific rag evaluation dataset generation framework. *arXiv preprint arXiv:2408.01262* (2024).

Received April 2025; revised July 2025; accepted August 2025