ICECACHE: MEMORY-EFFICIENT KV-CACHE MANAGEMENT FOR LONG-SEQUENCE LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Key-Value (KV) cache plays a pivotal role in accelerating inference in large language models (LLMs) by storing intermediate attention outputs, thereby avoiding redundant computation during auto-regressive generation. However, the cache's memory footprint scales linearly with sequence length, often resulting in memory bottlenecks on constrained hardware. While prior work has explored offloading KV-cache to the CPU and maintaining a reduced subset on the GPU, these approaches frequently suffer from imprecise token prioritization and degraded performance in long-generation tasks such as multi-turn dialogues and chain-of-thought reasoning. In this paper, we propose a novel KV-cache management strategy called IceCache, that integrates semantic token clustering with PagedAttention, a memory-efficient paging mechanism. By clustering semantically related tokens and organizing them into a hierarchical, dynamically updateable structure, our method improves cache hit rates and memory bandwidth utilization during CPU-GPU transfers. Experimental results show that IceCache achieves over 99% accuracy with a 256-token budget and still maintains 97% accuracy with only a 64-token budget, compared to the full KV-cache model. It outperforms existing baselines even while using just 25% of the KV-cache token budget, demonstrating its superior accuracy in long-sequence scenarios.

1 INTRODUCTION

Key-Value (KV) cache is a critical component in modern large language models (LLMs) which stores the intermediate attention outputs for each token, allowing the model to reuse these computations in subsequent forward passes. This is particularly important for auto-regressive generation tasks, where tokens are generated one at a time. By caching these values, the model avoids redundant calculations, dramatically reducing the runtime required for generating long sequences. However, the main challenge for a KV cache lies in its memory consumption. As the generated sequence grows longer, the required cache size increases linearly, potentially leading to out-of-memory errors on devices with limited RAM.

Recent research (Zhang et al., 2024b; Tang et al., 2024; Xiao et al., 2023) has revealed that despite the growing size of the KV cache, a small subset of tokens plays a disproportionately important role in maintaining generation accuracy. This insight suggests that we can significantly reduce inference time by selectively loading only these crucial tokens, without compromising the quality of the output. Some research (Chen et al., 2024a; Lee et al., 2024; Chen et al., 2024b) takes this approach further by offloading the KV-cache to the CPU and dynamically maintaining a subset of the most significant KV-cache on the GPU. However, many previous methods do not identify and prioritize these critical parts of the KV-cache in a precise way so that the hit rate of the truly important tokens is low. Therefore, in scenarios involving long-generation tasks, such as long-context summarization, multi-step reasoning, and extended chain-of-thought (CoT) generation, previous methods experience significant performance degradation (Li et al., 2024a).

To improve the identification of critical parts of the KV-cache, we propose an innovative approach, which we call IceCache, that integrates token clustering with a currently prevalent method – PagedAttention (Kwon et al., 2023) which stores the KV-cache in non-contiguous paged memory. As illustrated in Figure 1, by grouping semantically related tokens into pages, our approach aims to enhance the hit rate when selecting critical pages and tokens, and increase the transmission bandwidth

during the GPU-CPU offloading for the pages. Additionally, by employing a hierarchical data structure that can be efficiently updated during the decoding phase, we can mitigate the performance degradation commonly observed in long-generation tasks in previous studies. This leads to more effective use of the KV-cache, especially in the long-generation setting. IceCache is beneficial in two scenarios: (1) it achieves better accuracy than state-of-the-art methods using the same budget size; (2) it achieves comparable accuracy using a much smaller budget size (as small as 25%).

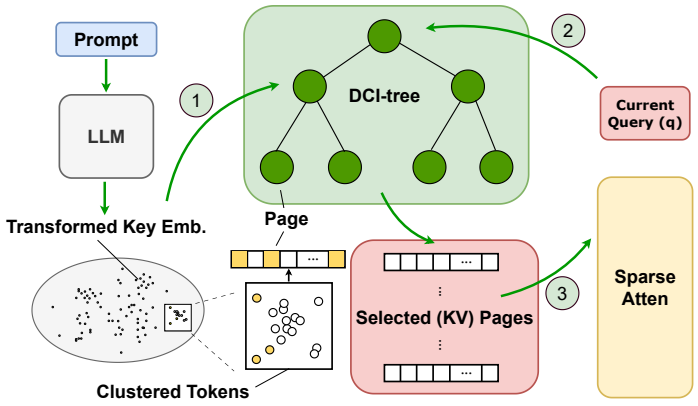


Figure 1: Illustration of IceCache. (1) During the prefill stage, tokens are indexed into a tree structure (the DCI-tree) according to their semantic similarity in the transformed key-embedding space. Each leaf node of the DCI-tree corresponds to a physical memory page. (2) During the decoding stage, given a query q, IceCache performs a tree search to identify the top-k tokens most relevant to q. The zoomed-in section at the bottom illustrates that these critical tokens (highlighted in yellow) tend to be clustered within the same leaf nodes and are stored together in corresponding memory pages. (3) After the query-aware token search, the pages (leaf nodes) containing the critical tokens are selected, and all tokens within these pages are utilized in the subsequent sparse attention with q.

IceCache has the following contributions:

1. Token Clustering for Efficient Storage: In the Prefill stage, instead of storing the KVs sequentially in their original order, we first cluster the tokens based on their similarity in a transformed key-embedding space using a maintainable tree-structured index, called DCI-tree. Tokens belonging to the same cluster are then stored together in the same memory page(s).

2. Query-aware Critical Page Selection: In the Decoding stage, given a specific query, only a subset of pages for each head are loaded to GPU to perform the attention computation for each layer and attention head. These pages are selected based on the presence of critical tokens, which is decided by an Approximated Nearest Neighbour (ANN) algorithm called Multi-level DCI (M-DCI).

3. Efficient Pipelining with CPU-GPU overlapping: IceCache performs M-DCI-based indexing and page selection on the CPU in parallel with GPU operations such as attention computation and feedforward layer execution. This pipelined design effectively overlaps computations, hiding much of the latency introduced by page selection.

We evaluated IceCache under constrained GPU memory budgets on the Passkey Retrieval (Mohtashami & Jaggi, 2023), LongBench (Bai et al., 2023), and GSM8K Chain-of-Thought (CoT) reasoning (Wei et al., 2022) using four popular open-source LLMs: Llama3.1-8B-Instruct, Mistral-7B-Instruct-v0.2, LongChat-7B-v1.5 and Qwen3-32B. Across diverse tasks, including open-domain QA, multi-hop reasoning, academic reading comprehension, long-context summarization and long-context generation, IceCache consistently outperformed six state-of-the-art KV-cache baselines. Notably, IceCache sustained near-oracle performance (over 99%) on most tasks using only a small fraction (as small as 64 tokens) of the original KV cache size. For instance, on the challenging *GovReport* task, known for its long-range dependencies and high generation demands, IceCache achieved accuracy within 1% of the full KV baseline, whereas other methods experienced sharp performance drops. Furthermore, by leveraging its hierarchical index and well-designed pipelining, IceCache achieves decoding speedups comparable to leading baselines, demonstrating both efficiency and scalability for long-context LLM inference.

2 RELATED WORK

Lots of recent methods have aimed to enhance the efficiency of attention mechanisms in large language models, especially for handling long-context inputs. H2O (Zhang et al., 2024b) only keeps a subset of tokens selected by the attention scores to save memory for the KV-cache. StreamingLLM (Xiao et al., 2023) utilizes the initial tokens, which they call sink tokens and the most recent tokens to accelerate the attention computation. Methods such as SparQ (Ribar et al., 2023) apply approximate attention by only selecting important indices across the head dimension. Similarly, MagicPiG (Chen et al., 2024b) employs a sampling technique to provide a faithful estimation of the attention output. OmniKV (Hao et al., 2025) reduces memory overhead by reusing the important tokens identified across consecutive layers. SnapKV (Li et al., 2024b) uses the last portion of the prompt to select the important key embeddings for the following decoding. PQCache (Zhang et al., 2024a) employs product quantization to manage KV-cache and approximate the attention computation.

PagedAttention (Kwon et al., 2023) is an innovative memory management technique designed to optimize the KV-cache of LLMs. It addresses the challenges by introducing a paging mechanism similar to virtual memory systems in operating systems. This approach divides the KV cache into fixed-size pages, allowing for more efficient memory allocation and management. By doing so, PagedAttention enables better utilization of GPU memory, reducing fragmentation and allowing for longer context windows without sacrificing performance.

Quest (Tang et al., 2024) and ArkVale (Chen et al., 2024a) are two query-aware criticality estimation algorithms built on the PagedAttention. They effectively identify critical KV-cache tokens and perform self-attention selectively on the chosen tokens. For each page, Quest and ArkVale calculate an upper bound using the feature values of the Key vector for each page's criticality estimation. Given all criticality scores of the pages, Top-K pages are chosen to perform approximate self-attention, where K is a preset constant (e.g. 128, 256). Additionally, ArkVale integrates the GPU-CPU offloading into the system to further save GPU memory. However, the main issue with both Quest and ArkVale is that they make all tokens in the query head attend to the same key/value blocks activated by sparse attention, which is too coarse-grained, as the information each token needs to attend to can vary significantly. Instead, IceCache allows each query head to attend to different key/value blocks, which makes the attention-approximation more accurate.

3 BACKGROUND

3.1 ATTENTION MECHANISM AND SPARSE ATTENTION

Mathematically, the attention operation takes three matrices as input, $\mathbf{K} \in \mathbb{R}^{m \times d}, \mathbf{Q} \in \mathbb{R}^{n \times d}, \mathbf{V} \in \mathbb{R}^{m \times d'}$, which denote keys, queries and values, respectively. Optionally, it may also take in a mask as input, $\mathbf{S} \in \mathbb{R}^{n \times m}$, whose entries are either 0 or 1. The ith rows of $\mathbf{K}$, $\mathbf{Q}$ and $\mathbf{V}$, denoted as $\mathbf{k}_i$, $\mathbf{q}_i$ and $\mathbf{v}_i$, represent the ith key, query, and value respectively. The entry of $\mathbf{S}$ in the ith row and jth column, denoted as $s_{i,j}$, represents whether the ith query is allowed to attend to the jth key — if it is 1, it would be allowed; if it is 0, it would not be. A common masking scheme is the causal mask, where $s_{i,j}$ is 1 if $i \geq j$ and 0 otherwise. Keys and queries have the same dimension d, and each key is associated with a value, and so the number of keys and values is the same and denoted as m. The attention operation computes the attention weight matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$. Its entry in the ith row and jth column, denoted as $a_{i,j}$, is computed with the following formula:

$$a_{i,j} = \frac{s_{i,j} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d}}\right)}{\sum_{j'=1}^{m} s_{i,j'} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_{j'}}{\sqrt{d}}\right)} \tag{1}$$

The attention matrix $\mathbf{A}$ is typically sparse (Nikita et al., 2020; Gupta et al., 2021), i.e., in each row of $\mathbf{A}$, only a few attention weights have significant (large) values, while the majority of the remaining values are close to zero. If we can somehow identify the k unmasked keys that receive the highest attention weights for each query $\mathbf{q}_i$ without computing the attention weights for all keys, the original attention matrix $\mathbf{A}$ can be approximated by only computing the inner product for the identified keys, which can save a significant amount of computational resources.

3.2 GENERATIVE INFERENCE OF LLM

The generative inference process of LLMs primarily comprises two key stages: the prefill (or prompt) stage and the decoding (or generation) stage.

In the prefill stage, the model takes an input prompt sequence of length s_{in} and processes it through all layers of the LLM. During this process, the keys and values for each token in the sequence are computed and stored as part of the KV cache. The decoding stage begins once the prompt has been processed. Here, the model generates output tokens one step at a time, using and updating the KV cache iteratively. For each decoding step, the current token's computation depends on the stored keys and values from previous tokens, allowing the model to maintain context over the sequence. The KV cache thus plays a crucial role in enabling efficient autoregressive generation by reducing redundant computations and maintaining information about past tokens.

3.3 MULTI-LEVEL DCI

Prioritized Dynamic Continuous Indexing (P-DCI) Li & Malik (2017) propose an exact, randomized algorithm designed to perform efficient k-nearest neighbour (k-NN) searches in high-dimensional spaces. Unlike traditional methods that rely on space partitioning, P-DCI avoids this by constructing multiple indices, each imposing an ordering of all data points based on their projections onto random vectors. During querying, P-DCI maintains a priority queue to process points in an order that is likely to find nearer neighbours sooner. It computes a dynamic lower bound on the distance to the nearest neighbour, allowing early termination of the search when the bound exceeds the distance to the current best candidates. This approach significantly reduces the number of distance evaluations and memory usage compared to methods like Locality-Sensitive Hashing (LSH).

Multi-level Dynamic Continuous Indexing (M-DCI) Mao et al. (2024) extend P-DCI by introducing a hierarchical structure to further enhance search efficiency. The index is organized into multiple levels, where each level contains a subset of data points. Points are randomly promoted to higher levels, forming a pyramid-like structure. Each point at a lower level is assigned a parent in the next higher level, typically the nearest neighbour among the promoted points. This creates "nodes" or clusters of points sharing the same parent. When querying, the algorithm starts at the top level, using P-DCI to find the k-closest points to the query. It then recursively searches within the nodes associated with these points at the next lower level, continuing this process down the hierarchy. This multi-level approach allows M-DCI to focus computational resources on the most promising regions of the index, effectively narrowing down the search space and improving query times, especially in indexes with high intrinsic dimensionality.

4 ICECACHE

We propose an innovative approach, named IceCache, that integrates token clustering with KV-cache storage. Our method consists of three steps: (1) Indexing; (2) Page Selection; and (3) Bulk Back-loading. The Indexing step occurs either during the prompt processing phase—when IceCache constructs a hierarchical tree structure, referred to as the DCI-tree, for the prompt key embeddings; or when new window pages are offloaded to the CPU. In this step, similar tokens are grouped into units called nodes, rather than being stored sequentially in virtual memory pages as in PagedAttention. Here, a node denotes a group of data points that share the same parent in the tree hierarchy. The next two steps take place during the token generation (decoding) phase. In the Page Selection step, IceCache employs a fast Approximate Nearest Neighbor (ANN) search algorithm, P-DCI, to independently select the top-k most relevant key pages for each attention head. Finally, in the Bulk Back-loading step, the selected pages are efficiently transferred from the CPU back to the GPU. IceCache overlaps the DCI Indexing (a CPU-intensive operation) with ongoing GPU computations, thereby minimizing additional latency.

We provide further details on each of these three steps in the following subsections and illustrate the method in Fig 2.

4.1 INDEXING: CLUSTERING KEY EMBEDDINGS INTO A HIERARCHICAL TREE

PagedAttention (Kwon et al., 2023) is a memory management strategy designed to optimize attention computation in LLMs by organizing key-value pairs into sequential memory pages. It stores these key-value pairs based on their original token indices, ensuring that tokens appearing consecutively

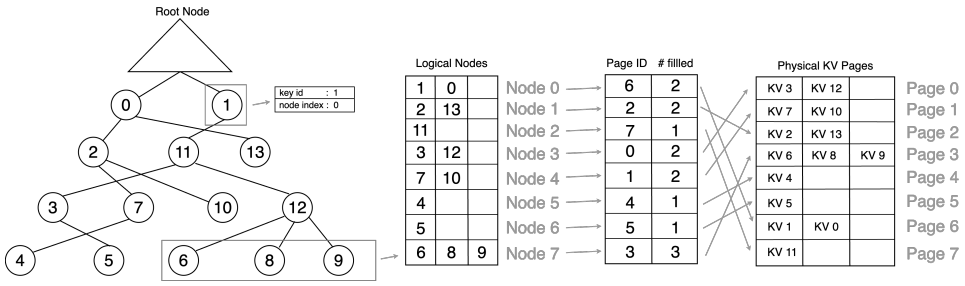


Figure 2: Illustration of DCI-tree and IceCache: The hierarchical structure on the left visualizes the result of indexing key embeddings, DCI-tree, where each tree node stores metadata for the tokens such as the key ID and node index. The tables on the right depict the mapping between nodes in the DCI-tree and the corresponding pages in physical memory. For each selected node, a mapping table is used to locate the memory region containing the associated key-value embeddings.

in the input sequence are placed contiguously in memory. This organization minimizes memory fragmentation, allowing for more efficient memory access during decoding and ultimately improving computational throughput. To inherit the benefits of PagedAttention, several subsequent KV-cache optimization techniques, such as Quest (Tang et al., 2024) and ArkVale (Chen et al., 2024a), have been developed based on its principles. These methods focus on estimating the importance of each page during KV-cache selection to approximate attention computation more efficiently.

IceCache also organizes key-value embeddings into pages, but takes a fundamentally different approach during its Indexing stage. Instead of relying on the token's original order, IceCache constructs a separate hierarchical tree structure for each attention head, called a DCI-tree, which clusters tokens based on the semantic similarity of their key embeddings. Each node in the DCI-tree represents a small group of semantically related tokens that share a common parent, effectively forming a localized cluster. From a memory system perspective, IceCache maps each node directly to a memory page, thereby preserving semantic locality in storage and enabling efficient access during decoding.

By clustering semantically similar tokens into the same nodes/pages, IceCache enables more targeted and efficient retrieval during decoding. In contrast, methods like Quest, Arkvale, or PQCache (Zhang et al., 2024a) construct pages based on the original token order, which often causes tokens relevant to a given query to be scattered across multiple pages. Retrieving them requires loading entire pages filled with many irrelevant tokens, resulting in unnecessary memory overhead. IceCache mitigates this inefficiency by grouping similar tokens, so relevant tokens tend to be concentrated within fewer pages. As a result, it achieves comparable or improved performance while reducing the number of pages retrieved.

Moreover, the DCI-tree structure used by IceCache is designed for efficient incremental updates. As new windows of tokens (e.g., from a sliding window in long-context scenarios) are offloaded to the CPU, each token is inserted into the appropriate node in the DCI-tree based on its key embedding. When a node exceeds the maximum page size, new pages are dynamically allocated to maintain balance. This adaptive tree maintenance ensures that the index remains both semantically meaningful and efficient, making IceCache particularly effective for long-sequence generation.

In summary, while prior methods treat KV-cache page layout as a static memory allocation problem, IceCache introduces a dynamic, semantically-aware structure that preserves key similarity across time. This enables more focused page retrieval, reduces memory fragmentation, and supports more efficient decoding. Furthermore, by performing indexing during the prompt or CPU offloading phase, IceCache amortizes the tree construction cost and avoids incurring additional latency during inference.

4.2 Page Selection: Head-specific ANN Search with Fine-grained Retrieval

During the decoding phase, given a query, IceCache performs a head-specific page selection to identify the most relevant key pages for each attention head. Leveraging the hierarchical DCI-tree

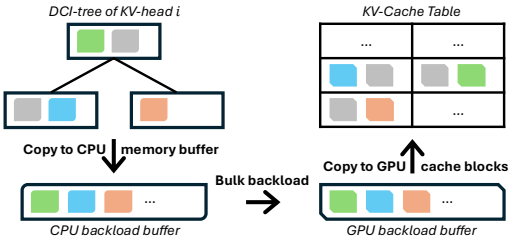


Figure 3: After IceCache selects important KV-pages, it aggregates all selected pages into a contiguous CPU preloading buffer. This buffer is then transferred via high-throughput PCIe transaction to a pre-allocated GPU buffer. Finally, the transferred blocks are scattered into their exact locations in the KV-Cache table. This bulk transfer avoids many small PCIe copies and significantly improves utilization.

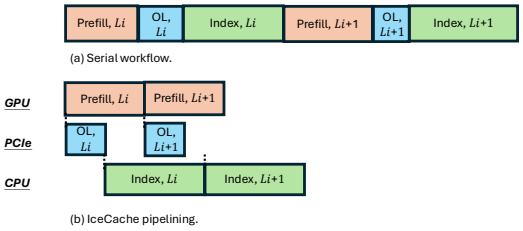


Figure 4: (a) Baseline serial workflow, where prefilling, offloading (OL), and indexing are executed strictly in sequence. (b) IceCache pipelining, where GPU prefilling overlaps with KV-offloading via PCIe and CPU-side DCI indexing. Once KVs of layer i (Li) arrive in CPU memory, Li-DCI-tree indexing progresses in parallel with GPU prefilling and offloading of the subsequent layer (Li+1). This results in significantly reduced end-to-end prefilling latency.

built during indexing, we apply a fast ANN search method mentioned in Section 3.3, M-DCI, to find the top-k pages that are closest to the current query embedding for each head independently.

This design contrasts sharply with prior methods like Quest and PQCache, which either retrieve all pages indiscriminately or use a coarse global selection strategy shared across heads. In contrast, IceCache's head-specific search recognizes that different heads often attend to different semantic aspects of the input, and thus benefit from customized retrieval strategies. This per-head granularity leads to improved attention relevance and overall model accuracy.

4.3 Bulk Loading and Pipelining

In this section, we present how we optimize the efficiency of the IceCache workflow, using bulk loading and pipelining. Our key observation is that the selected KV cache pages are not continuous in either main memory or GPU memory. As a result, individual transfer of these pages between main memory and GPU memory is highly inefficient. To overcome this issue, we designed bulk loading algorithms to efficiently offload and backload the selected pages, using two CPU and GPU backload buffers. In addition, we carefully designed an efficient pipeline of prefilling calculations, KV cache offloading, and DCI indexing, for efficient prefilling.

We illustrate our bulk back-loading workflow in Figure 3. Offloading follows the same but reversed procedure. After identifying the most relevant pages (indicated by color), we filter out those already resident in GPU memory from previous token generations. The remaining pages are then aggregated into a pre-allocated CPU-memory buffer, enabling a single high-throughput PCIe transfer into a pre-allocated GPU-memory buffer. Once the pages arrive in GPU memory, we scatter them directly into their corresponding entries in the KV-Cache table.

Figure 4(b) illustrates the IceCache prefilling pipelining (other minor stages are omitted for simplicity). After the KV states are generated on the GPU, we simultaneously trigger the prefilling calculation and the offloading transfer. The DCI index construction starts building the index along with the prefilling calculation, right after the KV states fully arrive in the main memory. This approach allows the offloading and indexing latencies to be largely hidden by the main prefilling computation. Furthermore, IceCache can be easily extended with critical page reuse techniques (Liu et al., 2023; Hao et al., 2025) to further accelerate prefilling.

5 EXPERIMENTS

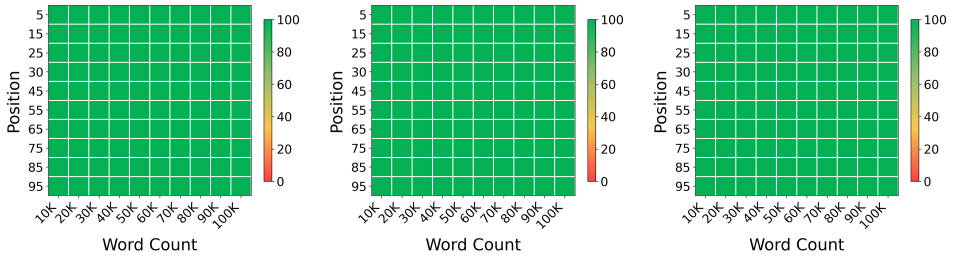
5.1 SETTINGS

We apply our method to Llama-3.1-8B-Instruct and Mistral-7B-Instruct-v0.2, two of the most popular open-source LLMs employing group-query attention (GQA) (Ainslie et al., 2023). We also test our method on a larger model, Qwen3-32B, and a multi-head attention model (Vaswani et al., 2017), LongChat-7B-v1.5. We first evaluate the recall in retrieving important tokens, followed by performance testing on 16 tasks from Longbench benchmark (Bai et al., 2023). As prior research indicates, the initial layers of the model exhibit relatively low sparsity. Therefore, neither IceCache nor baseline methods are applied to the first two layers of the models.

Our experimental platform comprises an Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz and an NVIDIA A100 40GB PCIe GPU (for small models) or an NVIDIA H100 80GB PCIe GPU (for large models). The software stack includes CUDA version 12.1, PyTorch version 2.5.1, and HuggingFace Transformers version 4.51.0. We implement IceCache on top of HuggingFace Transformers, utilizing FlashInfer for the attention kernel operation.

5.2 PASSKEY RETRIEVAL ACCURACY

We first evaluate IceCache's effectiveness in handling long-range dependencies using the passkey retrieval task (Mohtashami & Jaggi, 2023). We consider context lengths from 10k words to 100k words, and test with the size of cache budget $= \{256, 128, 64\}$. For each length, 100 test cases are generated with passkeys inserted at various positions from 0% to 95% of the total context length in increments of 5%. The results are illustrated in Fig. 5. As shown in the figure, IceCache dynamically assess the importance of evicted pages and recall crucial ones on demand, consistently maintaining 100% retrieval accuracy across all tested budget sizes.



(a) Cache Budget = 256 (b) Cache Budget = 128 (c) Cache Budget = 64

Figure 5: Passkey retrieval accuracy of IceCache on Llama3.1-8B-Instruct. The horizontal axis indicates the relative insertion position (%) of the passkey, while the vertical axis represents the context length in words. Results are presented for cache budgets of 256, 128, and 64. Notably, IceCache achieves 100% retrieval accuracy across all tested budget sizes.

5.3 LONGBENCH EVALUATION

To assess the performance of our method in long-context scenarios, we conduct a comprehensive evaluation on the LongBench benchmark. We compare IceCache (ICE) against six state-of-the-art KV cache optimization methods, including MagicPig (MPG) (Chen et al., 2024b), ArkVale (AKV) Chen et al. (2024a), SnapKV (SKV) Li et al. (2024b), StreamingLLM (SLM) Tang et al. (2024), OmniKV (OKV) Hao et al. (2025) and PQCache (PQC) Zhang et al. (2024a) as baselines. We also include the results of full KV cache (FULL) and ground-truth top-k KV cache (TOP-k) as baselines. The detailed results are presented in Table 1.

Performance on Llama-3.1-8B-Instruct. On Llama-3.1-8B, the effectiveness of IceCache is particularly pronounced. Most impressively, with a highly constrained KV Cache budget of just 64, our method achieves an average accuracy of 47.8. This result alone surpasses the strongest baseline, PQCache, which scores 47.3 while operating with a 4 times larger budget of 256. This highlights the exceptional efficiency of our approach in low-resource environments. As we increase

Table 1: Accuracy comparison of our method (ICE) with SnapKV (SKV), SteamingLLM (SLM), OmniKV (OKV), MagicPig (MPG), PQCache (PQC), ArkVale (AKV), Full KV (FULL) and ground-truth top-k (TOP-k) on LongBench for Llama-3.1-8B-Instruct and Mistral-7B-Instruct. IceCache generally outperforms other methods across various KV cache budgets and LLMs.

Budget	Method	Single-Document QA			Multi-Document QA			Summarization			Few-shot Learning			Synthetic		Code		Avg.
		NrtvQA	Qasper	MF-en	HotpotQA	2WikiMQA	Musique	GovReport	QMSum	MultiNews	TREC	TriviaQA	SAMSum	PCount	PRe	Lcc	RB-P	
		18409	3619	4559	9151	4887	11214	8734	10614	2113	5177	8209	6258	11141	9289	1235	4206	
Llama-3.1-8B-Instruct																		
N/A	FULL	30.2	45.5	54.9	55.5	46.7	31.3	35.2	25.2	27.2	72.5	91.7	43.8	8.4	99.5	65.1	58.8	49.5
256	TOP-k	30.7	44.7	55.4	55.0	46.5	31.7	34.8	25.1	26.8	71.5	92.2	44.8	7.8	100.0	67.1	63.6	49.8
256	SKV	23.7	27.3	46.3	52.3	40.8	24.2	19.2	22.6	19.2	29.0	84.1	39.0	**8.6**	97.5	57.3	56.2	40.8
	SLM	17.0	23.2	25.7	21.0	29.3	6.8	20.8	17.5	20.7	45.5	84.3	41.2	5.0	71.5	59.5	47.5	33.5
	OKV	27.2	40.7	52.8	55.1	45.6	29.7	27.6	23.0	25.4	72.5	88.9	40.4	5.6	94.5	60.8	51.5	46.3
	MPG	25.5	39.9	51.8	51.4	39.5	25.7	33.9	23.5	25.9	65.5	84.0	37.0	7.8	99.5	47.4	44.8	44.6
	PQC	28.7	43.3	52.2	55.2	45.1	28.4	27.2	24.0	22.8	69.5	91.1	41.2	6.2	99.0	59.0	54.4	47.3
	AKV	26.1	35.2	47.5	51.6	45.6	28.1	22.9	22.5	22.8	53.5	90.1	40.0	6.7	85.0	57.7	48.9	42.8
64	ICE	27.4	43.2	55.7	**55.3**	44.4	**31.2**	33.4	23.7	26.2	72.5	90.3	41.9	6.6	99.5	61.7	51.6	47.8
128	ICE	30.0	**44.7**	**56.5**	55.0	45.4	30.0	33.5	24.3	26.5	**73.0**	91.3	42.4	6.5	**100.0**	61.5	56.7	48.6
256	ICE	**30.6**	**44.7**	56.3	55.2	**45.9**	30.6	**34.6**	**24.4**	**26.7**	**73.0**	**92.0**	**43.5**	6.7	**100.0**	**62.5**	**56.4**	**49.0**
Mistral-7B-Instruct-v0.2																		
N/A	FULL	26.8	33.1	49.3	42.8	27.3	18.8	33.0	24.2	27.1	71.0	86.2	42.8	2.9	87.0	56.9	54.3	42.2
256	TOP-k	26.2	31.3	48.9	40.0	26.2	19.6	33.3	24.2	27.1	73.0	86.6	43.3	2.3	82.9	59.0	57.3	42.6
256	SKV	18.3	15.1	38.3	30.4	19.2	13.8	16.5	21.4	19.9	32.0	81.7	38.1	**4.0**	59.1	49.0	51.8	31.8
	SLM	14.2	12.3	26.4	23.2	14.8	10.1	17.5	19.8	19.8	51.0	80.5	39.9	**4.0**	15.6	52.0	45.4	27.8
	OKV	16.5	22.8	43.8	34.7	19.2	16.6	24.4	22.0	24.8	70.5	81.7	38.6	3.1	35.2	36.5	38.4	33.0
	MPG	21.0	28.0	46.5	38.4	19.5	17.5	30.8	**23.3**	25.8	70.0	83.7	39.5	2.5	85.0	49.4	48.3	39.1
	PQC	21.0	25.8	42.5	18.1	20.3	16.0	29.5	21.7	**27.1**	70.0	85.8	39.7	2.5	75.5	54.2	49.2	37.4
	AKV	18.0	15.2	41.5	30.7	17.0	13.2	21.6	21.6	23.1	58.0	85.8	40.6	2.1	41.5	51.4	47.2	33.0
64	ICE	23.9	29.0	47.4	**40.5**	23.5	18.3	30.6	21.8	26.0	70.5	85.9	41.5	3.5	56.5	53.2	50.4	39.0
128	ICE	25.1	30.5	**48.7**	40.4	**26.7**	**18.7**	30.3	22.0	26.1	70.5	85.7	42.4	3.4	70.2	53.6	51.5	40.4
256	ICE	**25.2**	**31.0**	48.4	40.4	26.0	18.3	**31.5**	23.1	26.9	**71.0**	**86.3**	**42.8**	3.9	**85.1**	**54.7**	**53.2**	**41.7**

Table 2: Accuracy comparison of our method (ICE) with Full KV (FULL) on LongBench for Qwen3-32B and LongChat-7B-v1.5.

Budget	Method	Single-Document QA			Multi-Document QA			Summarization			Few-shot Learning			Synthetic		Code		Avg.
		NrtvQA	Qasper	MF-en	HotpotQA	2WikiMQA	Musique	GovReport	QMSum	MultiNews	TREC	TriviaQA	SAMSum	PCount	PRe	Lcc	RB-P	
		18409	3619	4559	9151	4887	11214	8734	10614	2113	5177	8209	6258	11141	9289	1235	4206	
Qwen3-32B																		
N/A	FULL	32.6	45.5	50.4	59.6	56.0	40.1	33.2	23.9	25.2	72.0	70.8	37.1	18.0	100.0	12.4	18.4	43.4
64	ICE	29.5	43.9	50.9	61.4	55.8	38.8	30.8	23.8	24.3	71.0	70.0	35.7	15.0	97.0	10.5	17.3	42.2
128	ICE	32.1	44.8	53.3	61.6	54.6	38.5	31.0	23.6	24.8	72.0	70.3	36.2	15.5	100.0	10.7	17.3	42.6
256	ICE	32.2	44.1	51.9	60.2	55.1	38.9	32.4	24.3	24.7	71.5	70.7	37.4	16.5	100.0	11.8	18.0	43.1
LongChat-7B-v1.5																		
N/A	FULL	20.8	29.4	43.1	33.0	24.4	14.7	30.8	22.8	26.7	66.5	84.0	22.5	0.0	30.5	54.7	59.2	35.2
64	ICE	18.5	26.9	40.6	34.2	22.7	14.0	29.3	20.9	25.6	66.5	84.3	21.9	1.5	26.1	52.5	56.8	33.9
128	ICE	19.9	27.7	40.9	33.9	24.2	14.4	28.6	21.8	25.9	66.5	84.2	22.3	1.5	27.3	52.6	57.7	34.3
256	ICE	20.4	29.5	43.0	34.6	23.7	14.2	29.8	22.7	26.1	66.5	84.7	22.9	0.0	28.5	53.6	59.0	35.0

the budget for IceCache to 256, its performance climbs to an average score of 49.0. This not only represents a substantial 1.7 point improvement over PQCache but also closes the performance gap to the unconstrained Full KV-Cache (49.5) to a mere 0.5 points. Notably, our performance is remarkably close to the ground-truth Top-k, demonstrating a near-optimal cache management strategy across a diverse set of tasks.

Performance on Mistral-7B-Instruct. This strong performance trend is consistent on the Mistral-7B model, confirming the robustness of our method. With a budget of 256, IceCache achieves an average accuracy of 41.7, establishing a significant 2.6 point lead over the best-performing baseline, MagicPig (39.1). Again, the low-budget capability of IceCache stands out; with a budget of 64, IceCache scores 39.0, remaining highly competitive with the top baseline (MagicPig, scores 39.1) that uses four times the cache size (256).

Performance on two additional LLMs. We evaluate IceCache on LongBench using two additional models: Qwen3-32B, a large-scale model, and LongChat-7B-v1.5, which employs standard multi-head attention rather than group-query attention. As shown in Tables 2, for Qwen3-32B, IceCache with a small budget of 64 achieves an average accuracy of 42.2 on LongBench, retaining 97.2% of the full KV cache performance (43.4). This score rises to 99.3% with a budget of 256, nearly matching the vanilla model. Similarly, on LongChat-7B-v1.5, our method preserves 96.3% of the full KV cache

performance with a budget of 64, and achieves up to 99.4% at a budget of 256. These results provide strong evidence that IceCache is effective across different model scales and attention mechanisms.

5.4 GSM8K CoT Reasoning

We also evaluate IceCache on the GSM8K benchmark using Chain-of-Thought prompting, applying a 10% budget for all compared methods using Mistral-7B-Instruct-v0.2. As shown in Table 3, IceCache demonstrates superior performance, achieving an accuracy of 47.4%, significantly higher than all other methods under the same budget constraint. In particular, it improves the accuracy of 46% of the strongest baseline, PQCache, by 2.6% of the original value to 47.4%. Moreover, our approach nearly matches the full KV-cache (48.2%), highlighting the effectiveness of IceCache.

Table 3: GSM8K CoT.

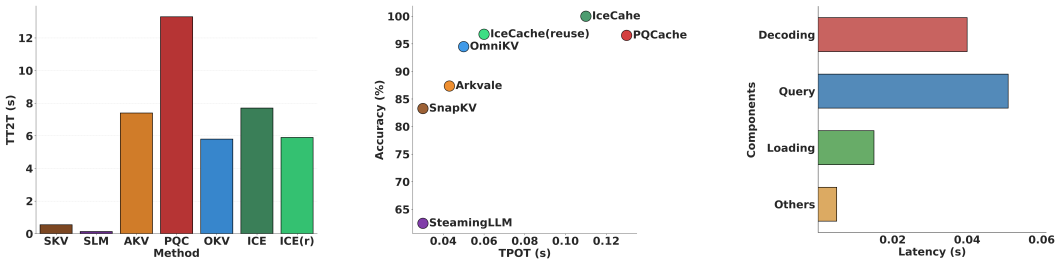
Method	Budget	Accuracy
FULL	N/A	48.2
SKV	10%	44.7
SLM	10%	44.4
OKV	10%	42.7
MPG	10%	43.1
PQC	10%	46.2
AKV	10%	30.9
ICE	10%	**47.4**

5.5 Latency Analysis

Since most baselines, including IceCache, use the entire KV cache to generate the first token, we follow PQCache (Zhang et al., 2024a) and report the Time to the second token (TT2T) in Fig. 6a, for a 36k sequence length and all the methods compared in Table 1 except MagicPig, which is restricted to costly AVX-512 CPUs. All reported numbers in this section are obtained using Llama-3.1-8B-Instruct. Our method, IceCache, achieves competitive latency among retrieval-based algorithms, recording 7.7 seconds. Its variant, IceCache (reuse), which reuses the KV-cache across layers, further reduces this to 5.9 seconds, matching OmniKV (5.8 s) and outperforming other retrieval-based baselines such as Arkvale (7.4 s) and PQCache (13.3 s). Although eviction-based methods like SnapKV (0.55 s) and StreamingLLM (0.13 s) are much faster, their speed often comes at the cost of accuracy. Overall, IceCache and IceCache(reuse) offer a strong balance between efficiency and accuracy, with the reuse variant showing how our approach can further optimize latency without significantly sacrificing performance. We include more details of IceCache(reuse) in the appendix C.

Similarly, for decoding latency, Fig. 6b shows the average time per generated token, along with the corresponding accuracy percentage relative to the full KV-cache model, for an input sequence length of 36k. Eviction-based methods, StreamingLLM and SnapKV (both are 0.03 seconds per token), continue to show the fastest speeds due to their minimal overhead. Among the more accurate retrieval-based methods, IceCache(reuse) achieves a highly competitive decoding time of 0.06 seconds per token – substantially faster than PQCache (0.13 s) and nearly matching the speed of OmniKV (0.05 s). Vanilla IceCache maintains a strong balance, combining superior accuracy with efficient decoding. It achieves the highest accuracy percentage (99.0%) while still outperforming PQCache in speed. These results further demonstrate that IceCache effectively balances accuracy and decoding latency.

Figure 6c presents a detailed breakdown of TPOT latency for IceCache at a sequence length of 36k, with a total latency of 0.11 seconds. In this figure, "Loading", "Query", and "Decoding" correspond to the overhead from CPU–GPU communication, DCI-query operations, and the overall LLM decoding process, respectively. The largest contributors to latency are the DCI-query module (0.05 s) and decoding (0.04 s), while GPU–CPU offloading and other miscellaneous operations add only 0.015 seconds and 0.005 seconds, respectively.



(a) Time to the second token. (b) Time per output token. (c) TPOT Latency breakdown.

Figure 6: Latency comparison of IceCache and baseline methods on a 36k-token sequence.

6 CONCLUSION

This paper addresses the critical challenge of managing long contexts in LLMs, where the expanding KV-cache severely impacts memory efficiency and computational performance. To address this, we introduce a novel hierarchical database, the DCI-tree, enabling lightweight updates and dynamic token management for efficient KV-cache handling. Building on this, we propose IceCache, an end-to-end page-based KV-cache manager with efficient GPU–CPU offloading and recall. Extensive experiments across diverse long-context tasks demonstrate IceCache's efficacy: it achieves over 99% accuracy with a 256-token budget and about 97% accuracy even with only a 64-token budget, surpassing existing baselines while using just 25% of the KV-cache token budget. These results establish IceCache as a scalable and practical solution for optimizing KV-cache in LLMs with long context requirements, delivering state-of-the-art accuracy and efficiency without sacrificing performance.

REFERENCES

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

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. Arkvale: Efficient generative llm inference with recallable key-value eviction. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024a.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, et al. Magicpig: Lsh sampling for efficient llm generation. *arXiv preprint arXiv:2410.16179*, 2024b.

Ankit Gupta, Guy Dar, Shaya Goodman, David Ciprut, and Jonathan Berant. Memory-efficient transformers via top-k attention. *arXiv preprint arXiv:2106.06899*, 2021.

Jitai Hao, Yuke Zhu, Tian Wang, Jun Yu, Xin Xin, Bo Zheng, Zhaochun Ren, and Sheng Guo. Omnikv: Dynamic context selection for efficient long-context llms. In *The Thirteenth International Conference on Learning Representations*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, 2024.

Ke Li and Jitendra Malik. Fast k-nearest neighbour search via prioritized dci. In *International conference on machine learning*, pp. 2081–2090. PMLR, 2017.

Yucheng Li, Huiqiang Jiang, Qianhui Wu, Xufang Luo, Surin Ahn, Chengruidong Zhang, Amir H Abdi, Dongsheng Li, Jianfeng Gao, Yuqing Yang, et al. Scbench: A kv cache-centric analysis of long-context methods. *arXiv preprint arXiv:2412.10319*, 2024a.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*, 2024b.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pp. 22137–22176. PMLR, 2023.

Yuzhen Mao, Martin Ester, and Ke Li. Iceformer: Accelerated inference with long-sequence transformers on CPUs. In *The Twelfth International Conference on Learning Representations*, 2024.

Amirkeivan Mohtashami and Martin Jaggi. Landmark attention: Random-access infinite context length for transformers. *arXiv preprint arXiv:2305.16300*, 2023.

Kitaev Nikita, Kaiser Lukasz, Levskaya Anselm, et al. Reformer: The efficient transformer. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. Sparq attention: Bandwidth-efficient llm inference. *arXiv preprint arXiv:2312.04985*, 2023.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

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

Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. Pqcache: Product quantization-based kvcache for long context llm inference. *arXiv preprint arXiv:2407.12820*, 2024a.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024b.

A METHOD OVERVIEW

We separate all tokens into three groups: *sink tokens*, which are the tokens at the very beginning of the input sequence; *window tokens*, which are the most recent tokens; and all the remaining tokens in between. The pages that store sink tokens are referred to as *sink pages*, and those that store window tokens are referred to as *window pages*. We always keep all the sink and window pages in GPU.

We provide the pseudocode below for IceCache. It operates in two main phases: (1) Prefill Phase: During the initial processing of prompt tokens, IceCache allocates paged KV memory per layer and performs self-attention computations. From the third layer onward, it copies KV embeddings to CPU and builds a dynamic index – DCI-tree. This tree enables efficient future lookup of important tokens based on query embeddings. (2) Decode Phase: During autoregressive decoding, each new token's query embedding is used to retrieve the most relevant KV pages via DCI-based query. Selected pages are back-loaded to GPU on demand, while unimportant pages are offloaded to CPU storage. When a new window page is offloaded, the DCI-tree is incrementally updated to store tokens in this page. The detailed steps are outlined in Algorithm 1. We will explain the mechanisms behind indexing and page selection in the following sections.

B METHOD DETAILS

B.1 INDEXING

For each attention head, given a set of pre-computed key embeddings, IceCache first indexes them using a hierarchical tree structure which is obtained by a novel approach called Multi-level DCI (M-DCI). It works by constructing a dynamic index called DCI-tree and applies Prioritized DCI (P-DCI) (Li & Malik, 2017) to each level of the tree recursively (more details are in Section 3.3). The data points processed in M-DCI are transformed key embeddings and query embeddings using the following transformation formulas, which we denote as $T_K : \mathbb{R}^d \to \mathbb{R}^{d+1}$ and $T_Q : \mathbb{R}^d \to \mathbb{R}^{d+1}$:

$$T_K(\mathbf{k}_j) = \left[\mathbf{k}_j/c \quad \sqrt{1 - \|\mathbf{k}_j\|_2^2/c^2}\right]^\top \tag{2}$$

$$T_Q(\mathbf{q}_i) = \left[\mathbf{q}_i/\|\mathbf{q}_i\|_2 \quad 0\right]^\top \tag{3}$$

where $c \geq \max_{j'} \|\mathbf{k}_{j'}\|_2$ is at least the maximum norm across all keys. We use the Euclidean distance as the distance function.

At the very beginning of the indexing, all data points are initially placed at the bottom level of the DCI-tree. Subsequently, some points are randomly selected to be promoted to the next higher level based on a promotion ratio $r < 1$. The ratio r is predefined during DCI-tree initialization and remains fixed throughout the process. After the indexing, we can get the total number of levels in the DCI-tree, denoted as L. The details are presented in Algorithm 2.

Specifically, let n_ℓ denote the number of data points at level ℓ, with level indices starting from the bottom (i.e., the lowest level is $\ell = 1$). Ideally, the number of points satisfies the recurrence relation $n_\ell = r \cdot n_{\ell-1}$. In other words, the distribution over level indices follows a geometric distribution. The probability that a point is assigned to the highest level ($\ell = L$) is r^{L-1}, while the probability of being assigned to level ℓ (for $1 \leq \ell \leq L-1$) is $r^{\ell-1} - r^\ell$.

After level assignment, each data point at level ℓ is linked to a parent at level $\ell + 1$, defined as the closest point in terms of key embedding distance. This parent assignment is formulated as a 1-nearest neighbor search and is efficiently solved using M-DCI query.

In the decoding stage, when a new token is generated, its key embedding is inserted into the appropriate position in the DCI-tree. A level ℓ is first assigned to the new key according to the same random promotion process. Then, its parent at level $\ell + 1$ is determined, and the key is added to the physical memory page corresponding to the node into which it is inserted.

B.2 PAGE SELECTION

As aforementioned, IceCache aims to accelerate self-attention by loading only a limited number of pages into GPU memory for computation. Therefore, the objective of page selection is to maximize the *recall* (or hit rate) of significant keys for a given query. By clustering semantically similar tokens

into the same nodes/pages, IceCache enables more targeted and efficient retrieval during decoding. In contrast, methods like Quest (Tang et al., 2024), Arkvale (Chen et al., 2024a), or PQCache (Zhang et al., 2024a) construct pages based on the original token order, which often causes tokens relevant to a given query to be scattered across multiple pages. Retrieving them requires loading entire pages filled with many irrelevant tokens, resulting in unnecessary memory overhead. IceCache mitigates this inefficiency by grouping similar tokens, so relevant tokens tend to be concentrated within fewer pages. As a result, the hit rate of significant keys during decoding increases. The detailed procedure is shown in Algorithms 3 and 4.

Specifically, when computing the attention matrix, given a query vector q_i, we follow the query process described in Section 3.3 to identify the top-k keys that are most likely to yield the highest dot-product values with q_i. Once these top-k keys are identified, we load only the pages that contain them. Suppose p pages are loaded, and each page contains d entries, since not all the pages are fully filled, the number of loaded keys N is bounded as: $N \leq pd$.

The approximate attention scores between the query q and these N selected keys are then computed using Equation 1. The masks $s_{i,j}$ are set to 1 for the selected keys and 0 for all others. Note that, IceCache constructs a separate DCI-tree for each attention head, allowing it to retrieve different sets of significant pages per head. This head-specific, fine-grained selection mechanism distinguishes IceCache from baselines such as Quest and ArkVale, which retrieve the same set of pages for all heads, potentially limiting their retrieval accuracy.

C Details of IceCache (reuse) on LongBench

We present the LongBench task scores for IceCache (reuse) in Table 4. Starting from the third layer, we build the DCI-tree and perform DCI-queries every five layers; we refer to these as "anchor layers". For the layers in between, we reuse the KV-cache indices selected at the most recent anchor layer.

Table 4: Accuracy of IceCache (reuse) on LongBench.

Budget	Method	Single-Document QA			Multi-Document QA			Summarization			Few-shot Learning			Synthetic		Code		Avg.
		NrtvQA 18409	Qasper 3619	MF-en 4559	HotpotQA 9151	2WikiMQA 4887	Musique 11214	GovReport 8734	QMSum 10614	MultiNews 2113	TREC 5177	TriviaQA 8209	SAMSum 6258	PCount 11141	PRe 9289	Lcc 1235	RB-P 4206	
Llama-3.1-8B-Instruct																		
256	ICE(r)	28.6	42.6	54.0	55.4	45.8	29.6	32.8	24.0	25.9	72.0	90.1	40.9	5.7	96.5	61.4	51.6	47.3

D Latency Scaling Across Context Lengths

Figure 7 compares the prefill latency (left) and per-token decode latency (right) of Full Attention, IceCache, and IceCache(r) as the input context grows from 32K to 128K tokens. The results show that IceCache introduces only a modest overhead during prefill, while achieving better scaling than Full Attention in the decoding stage.

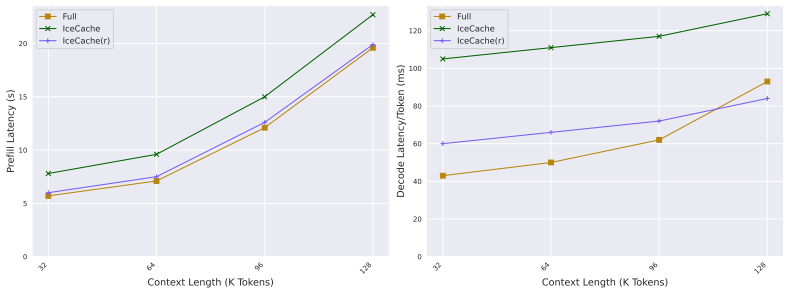


Figure 7: Latency scaling across context lengths (32k, 64k, 96k and 128k). Left: IceCache and IceCache(r) maintain close prefill latency to Full-KV across all context lengths. Although IceCache performs additional CPU-side indexing, the overhead remains small relative to the overall prefill cost. Right: both IceCache variants exhibit a slower growth rate in per-token decoding latency compared to Full-KV. Full-KV's decoding cost rises sharply with sequence length, and IceCache(r) even outperforms Full-KV at very long contexts (128k).

Algorithm 1 IceCache

Input: Sequence of tokens $x_{1:I}$, Transformer with L attention layers, Page size s
Phase 1: Prefill
for $\ell = 0$ to $L - 1$ **do**
    Allocate pages and arrange KVs to the pages for layer ℓ
    if $\ell \geq 2$ **then**
        Copy KVs of tokens between sink tokens and window tokens from GPU to CPU (denoted as S_k and S_v)
    end if
    Compute the output from the current self-attention layer ℓ
    if $\ell \geq 2$ **then**
        $T_l \leftarrow$ DCI-INDEXING(S_k, S_v)
    end if
end for
Phase 2: Decode (repeated over time steps $i > I$)
while receive new token x_i with $\mathbf{q}_i$ as its query embedding **do**
    for $\ell = 0$ to $L - 1$ **do**
        if Number of tokens in the last page $\geq s - 1$ **then**
            Offload the oldest window page P_w from GPU to CPU
            Set Flag to True
        end if
        Append KVs of x_i to the end of the newest window page
        if $\ell \geq 2$ **then**
            $S_l \leftarrow$ PAGE-SELECT$(\mathbf{q}_i, T_l, k)$
            Recall selected pages S_l from CPU to GPU
        end if
        Compute the output from the current self-attention layer ℓ
        if $\ell \geq 2$ & Flag is True **then**
            // Insert the tokens in offloaded P_w to T_l
            for i **in** P_w **do**
                // Random Promotion
                $l_i \leftarrow 1$ ▷ Start at bottom level
                while Random$(0, 1) < r$ **do**
                    $l_i \leftarrow l_i + 1$ ▷ Promote to the next higher level
                end while
                $\{p_i\} \leftarrow$ QUERY$(\mathbf{k}_i, T_l, l_i, 1)$ ▷ Get the parent index using QUERY (Alg.4)
                Insert $\mathbf{k}_i$ to T_l with p_i as its parent node index
            end for
        end if
    end for
    if x_i = EOS **then**
        Break
    end if
    $i = i + 1$
end while

Algorithm 2 Indexing

Require: A list S_k of n keys $\mathbf{k}_1, \ldots, \mathbf{k}_n \in \mathbb{R}^d$, Promotion ratio r
  function DCI-INDEXING(S_k, r)
    for $i = 1$ **to** n **do**
      // Random Promotion
      $l_i \leftarrow 1$ ▷ Start at bottom level
      while Random$(0, 1) < r$ **do**
        $l_i \leftarrow l_i + 1$ ▷ Promote to the next higher level
      end while
    end for
    Remove the empty levels
    Initialize T with an empty root node
    for $i = 1$ **to** n **do**
      $\{p_i\} \leftarrow$ QUERY$(\mathbf{k}_i, T, l_i, 1)$ ▷ Get the parent index using QUERY (Alg.4)
      Insert $\mathbf{k}_i$ to T with p_i as its parent node index
    end for
    return T
  end function

Algorithm 3 Page Selection

Require: Query vector $\mathbf{q}_i \in \mathbb{R}^d$, DCI-tree T, Number of critical keys k
  function PAGE-SELECT($\mathbf{q}_i, T, k$)
    Initialize $S_l \leftarrow \emptyset$
    $S_k \leftarrow$ QUERY$(\mathbf{q}_i, T, -1, k)$
    $S_l \leftarrow$ FIND-PAGE-INDEX(S_k)
    return S_l
  end function

Algorithm 4 k-Nearest Neighbour Querying

Require: Query vector $\mathbf{q}_i \in \mathbb{R}^d$, DCI-tree T with L levels, Target level l, Number of critical keys k
  function QUERY($\mathbf{q}_i, T, l, k$)
    if $l = -1$ **then**
      $l \leftarrow 1$
      Set Flag to True
    else
      Set Flag to False
    end if
    $S \leftarrow \emptyset$
    $P \leftarrow$ empty priority queue with size k
    for $i = L$ **to** l **do**
      $S' \leftarrow \emptyset$
      if $i = L$ **then**
        $S \leftarrow \{\text{root node}\}$
      end if
      for s **in** S **do**
        $S'' \leftarrow$ Prioritized-DCI-Query$(\mathbf{q}_i, s, k)$
        $S' \leftarrow S' \cup S''$
      end for
      if Flag is True **or** $i = l$ **then**
        for s **in** S' **do**
          $P \leftarrow$ Add-to-Priority-Queue (P, s)
        end for
      end if
      $S \leftarrow S'$
    end for
    return k nodes in P that have the keys with maximum inner-product with $\mathbf{q}_i$
  end function

E LLM USAGE

This work focuses on optimizing KV-cache management for large language models (LLMs). All the base models used in this paper can be viewed as LLMs, including Llama-3.1-8B-Instruct, Mistral-7B-Instruct-v0.2, Qwen3-32B, and LongChat-7B-v1.5. We also leveraged an LLM to help refine the manuscript's language and improve its overall readability.