Where Matters More Than What: Decoding-aligned KV Cache Compression via Position-aware Pseudo Queries

Anonymous ACL submission

Abstract

The Key-Value (KV) cache is crucial for efficient Large Language Models (LLMs) inference, but excessively long contexts drastically increase KV cache memory footprint. Existing KV cache compression methods typically rely on input-side attention patterns within a prompt observation window to estimate token importance during the prefill stage. They fail to preserve critical tokens for future generation since these assessments are not derived from the decoding process. Intuitively, an effective observation window should mirror the decoding-stage queries to accurately reflect which tokens the generation process will attend to. However, ground-truth decoding queries are inherently unavailable during inference. For constructing pseudo queries to approximate them, we find that positional information plays a more critical role than semantic content. Motivated by this insight, we propose decoding-aligned KV cache compression via position-aware pseudo queries (**DapQ**), a novel and lightweight eviction framework that leverages position-aware pseudo queries to simulate the output tokens, thereby establishing an effective observation window for importance assessment. It aligns closely with the actual generation context and enables precise token eviction. Extensive evaluations across multiple benchmarks and LLMs demonstrate that DapQ achieves superior performance, particularly under strict memory constraints (e.g., up to nearly lossless performance 99.5% on NIAH with 3% KV cache budgets).

1 Introduction

Large Language Models (Achiam et al., 2023; Jiang et al., 2023; Team et al., 2024; Liu et al., 2024a; Grattafiori et al., 2024; Yang et al., 2025a) have achieved significant success across various domains and demonstrated exceptional abilities for processing long-context tasks, such as contextual question answering and document summarization (Liu et al., 2024c; Guo et al., 2024; Liu et al., 2025). A key enabler of efficient inference is the KV cache mechanism, which significantly accelerates autoregressive decoding by reducing the computational complexity of self-attention from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. However, with the growth of context length, the memory footprint of KV cache and the high computational overhead increase dramatically, posing a severe obstacle to the efficient deployment and application of LLMs (Bai et al., 2023).

To tackle these challenges, various methods have been proposed to compress the KV cache, such as token eviction or merging (Zhang et al., 2023; Tian et al., 2025), quantization (Liu et al., 2024d; Hooper et al., 2024), head or layer-wise sharing (Ainslie et al., 2023; Yang et al., 2024), low-rank decomposition (Dong et al., 2024; Singhania et al., 2024). Among these, token eviction remains a widely-adopted strategy. Nevertheless, the rapid growth of input length has further intensified the demand for more effective eviction strategies. In response, as implemented in SnapKV(Li et al., 2024), the observation window has proven superior for retaining critical tokens by combining with pooled accumulated attention scores. This approach is further extended by PyramidKV(Cai et al., 2024), which dynamically allocates layer-wise cache budgets and selects important KV pairs for compression using the window-based attention mechanism. These studies demonstrate the potential of observation windows for effective KV cache compression.

However, the input-centric observation window is inherently misaligned with the dynamic query of actual decoding and relies solely on static prompt-based features, typically the last 16-32 tokens. Consequently, they fail to reflect the importance distribution determined by the output-side generation process, leading to misidentification of the critical tokens for decoding, particularly in complex or noisy contexts. Crucially, ground-truth decoding queries are unavailable during inference, rendering them impractical for directly guiding eviction. To

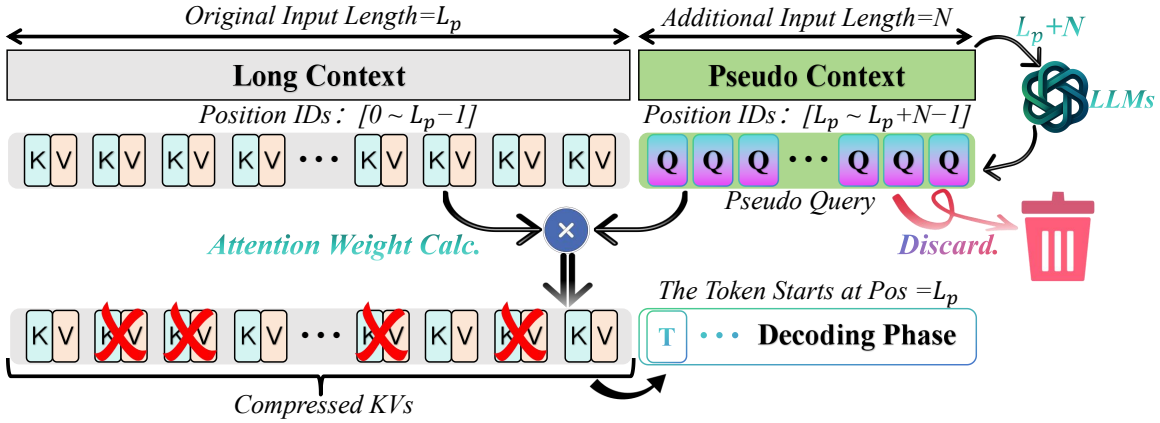


Figure 1: An overview of DapQ. A synthetic pseudo context (length N) is appended to the original context (length L_p), forming an extended sequence of length L_p+N. The model processes this sequence during the prefill phase and then obtains pseudo queries for the synthetic tokens, which are endowed with the correct positional encodings of the first N decoding steps. These pseudo queries compute attention scores with all keys from the original prompt, establishing the token importance distribution. The $topK$ tokens are retained in the compressed KV cache, while the others, along with all the synthetic tokens are evicted. Autoregressive decoding then begins from position L_p.

mitigate this, the recent approach LAQ++ (Wang et al., 2025) attempts to better align the observation window with decoding queries by pre-generating pseudo responses. But its two-stage eviction process introduces a significant memory peak issue that undermines its practical efficiency. Therefore, constructing effective pseudo queries (Q_{pseudo}) to approximate unavailable future queries without incurring any memory overheads is highly desirable.

Inspired by CaliDrop (Su et al., 2025), where queries at adjacent positions exhibit high similarity, our experiments uncover a pivotal insight: **positional information plays a more critical role than semantic content in constructing query approximations and determining attention patterns**. This discovery implies that high-quality pseudo queries, capable of reliably assessing the importance distribution of KV cache, can be synthesized based on future positional encodings.

Motivated by this insight, we propose decoding-aligned KV cache compression via position-aware pseudo queries (**DapQ**), a novel and lightweight KV cache eviction framework that constructs pseudo queries using future positional encodings to accurately simulate the output tokens. These queries collectively serve as an effective observation window for importance scoring that aligns closely with the actual generation context, enabling precise cache eviction. Extensive experiments across multiple benchmarks and different LLMs demonstrate that DapQ achieves superior performance and outperforms existing eviction baselines, particularly under strict memory constraints.

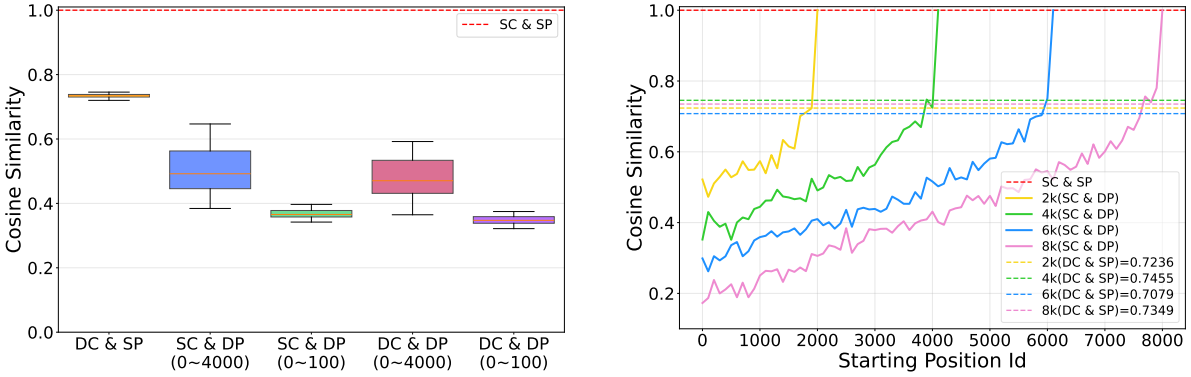
2 Related Work

Long-Context LLMs. The growing demand for LLMs to process long contexts intensifies computational and memory challenges. Prior works address these issues through specialized fine-tuning (Chen et al., 2023b) and extending effective context windows via refined positional encodings, such as interpolation and extrapolation (Chen et al., 2023a; Peng and Quesnelle, 2023). To mitigate computational overhead, sparse attention and linear attention have been widely explored (Kitaev et al., 2020; Beltagy et al., 2020; Wang et al., 2020). Beyond traditional Transformer, novel architectures like State-Space Models (SSMs) (Ye et al., 2025; Gu et al., 2021) provide linear complexity solutions for processing long sequences. Additionally, memory optimization techniques, such as KV cache compression (Xiao et al., 2023; Li et al., 2024) and memory offloading (Yang et al., 2025c; Aminabadi et al., 2022), have been developed. These multifaceted techniques collectively advance LLMs' capabilities in handling ultra-long sequence tasks.

KV Cache Compression. KV cache compression is crucial for enhancing the inference efficiency and deployability of LLMs, particularly in resource-constrained scenarios. Various methods have been developed to reduce KV cache memory footprint. Token eviction strategies aim to retain only the most important tokens based on metrics like attention scores (Li et al., 2024; Zhang et al., 2023), positional heuristics (Xiao et al., 2023), special tokens (Ge et al., 2023; Chen et al., 2024), or

Experiment	Content Similarity	Positional Similarity	Post ROPE	Pre ROPE
SC & SP	Same("The report discusses the Federal..... Airport Improvement Program (AIP). The program")	Same(4424,4425,4426..... 4453,4454,4455)	1.0000	1.0000
DC & SP	Different("Sorry, I don't know. Sorry, I don't know. Sorry, I don't know. Sorry, I don't know. Sorry, I")	Same(4424,4425,4426..... 4453,4454,4455)	0.7238	0.7238
SC & DP	Same("The report discusses the Federal..... Airport Improvement Program (AIP). The program")	Different(0,1,2..... 29,30,31)	0.3522	0.7913
DC & DP	Different("Sorry, I don't know. Sorry, I don't know. Sorry, I don't know. Sorry, I don't know. Sorry, I")	Different(0,1,2..... 29,30,31)	0.3267	0.7434

Table 1: Query similarity comparison under different content and position conditions. Post ROPE denotes similarity after ROPE has been applied to query vectors. Pre ROPE indicates similarity measured before ROPE application.



(a) (b)

Figure 2: Analysis of Positional Dominance and Offset Sensitivity in Query Similarity. We set the pseudo queries of fixed length 32. (a) Boxplot of query similarity distributions for a 4k context under different content and position conditions, each aggregated from 100 independent trials. DC: pseudo-queries content is constructed by randomly sampling 32 tokens from the model's vocabulary; DP: pseudo-queries positions are assigned by randomly sampling a consecutive span of 32 index positions from the context length range [0, m] (e.g., [0, 4000]). (b) Query similarity curves over offset positions for contexts of lengths 2k, 4k, 6k, and 8k. The x-axis denotes the starting position assigned to pseudo queries (e.g., an x-axis value of 3500 corresponds to position IDs $3500 \sim 3531$).

norm-based criteria (Devoto et al., 2024). Quantization (Liu et al., 2024d; Hooper et al., 2024) reduces memory by storing less important KV pairs with lower precision, some approaches even achieving sub-2-bit quantization via token-aware and channel-aware techniques. Sharing-based approaches deliver memory savings and accelerate inference through head-wise sharing (Shazeer, 2019; Ainslie et al., 2023), inter-layer sharing (Sun et al., 2024; Wu and Tu, 2024; Brandon et al., 2024), or prefix sharing across sequences (Juravsky et al., 2024; Zhu et al., 2024). Low-rank decomposition (Kang et al., 2024; Chang et al., 2024) projects KV cache into lower-dimensional spaces to exploit inherent redundancy, as demonstrated by the Multi-Head Latent Attention (Liu et al., 2024b) of DeepSeek, which reduces cache size through low-rank compression and decoupled RoPE while preserving model performance. KV merging (Tian et al., 2025; Cui and Xu, 2025) employs attention-pattern similarity or reparameterization to merge similar semantic information, achieving effective compression with minimal performance loss.

3 Observation

Given the discussion in Section 1, constructing pseudo queries to accurately approximate the unavailable ground-truth decoding queries becomes crucial. Building upon CaliDrop's (Su et al., 2025) insight that queries at adjacent positions exhibit high similarity, we hypothesize that this similarity is strongly correlated with positional information rather than semantic content. This prompts us to investigate whether positional information alone can effectively approximate future decoding queries without relying on true decoding content. See Appendix A for details of preliminary experiments.

3.1 Position Drives Query Representation

As detailed in Table 1,We compare cosine similarities between ground-truth decoding queries and pseudo queries across four conditions: SC (Same Content), DC (Different Content), SP (Same Position), and DP (Different Position). Specifically, pseudo queries assigned correct future positional IDs but composed of completely irrelevant or nonsensical content (DC&SP), exhibit strong

cosine similarity (0.7238) to the actual target decoding queries. Conversely, queries with the identical semantic content but incorrect positional IDs (SC&DP vs SC&SP) fail to accurately approximate the target queries (0.3522 vs 1.0000). Notably, the comparison between DC&SP and SC&DP highlights that maintaining correct positional alignment achieves 2.1× higher similarity than maintaining correct content alone. The stark contrast underscores that the semantic content of queries plays a secondary role compared to positional encodings in constructing query approximations. The consistently high similarity under Pre-ROPE conditions (0.7238 to 0.7913) confirms that the model's underlying processing does not rely heavily on semantic content to distinguish between these query vectors, thereby underscoring that the dramatic disparity observed in Post-ROPE scores is almost attributable to the positional information. A large-scale statistical analysis (Figure 2a) confirms the pervasiveness and consistency of the above phenomenon.

3.2 Precise Positional Alignment is Necessary

Building upon the dominance of positional information, we further quantify how the positional alignment of pseudo queries affects their similarities to true decoding queries. As shown in Figure 2b, we fix the pseudo-query semantic content to match the true output and systematically vary their assigned positional IDs. **Results reveal a monotonic decay in the query similarity as the absolute offset increases between the assigned position and the correct position**. This decay phenomenon is consistently observed across diverse context lengths (2k to 8k), which is more pronounced in longer context scenarios. The strong sensitivity to positional misalignment underscores the critical dependence of query approximations on precise positional information. Consequently, accurately simulating decoding queries requires precise alignment with the future generation positional IDs.

3.3 Position-Based Queries Enable More Accurate Token Eviction

The critical question is whether higher query similarity translates into more accurate token eviction. To quantify this, we evaluate the recall of eviction strategies, which measures its ability to retain the tokens that are most important for the actual generation. Following the methodology of prior work (Wang et al., 2025), the recall rate of the selected KV cache is defined as the proportion of indices

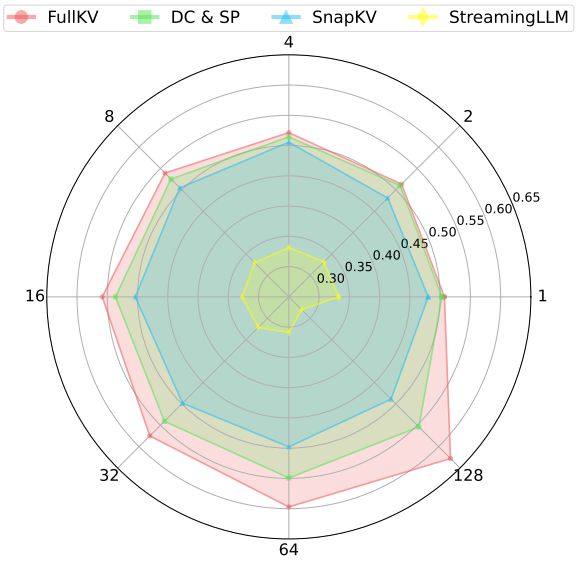


Figure 3: Recall Performance of different methods across various Window Sizes.

selected by the observation window that overlap with those selected by all response tokens from the model. We define the recall metric as follows:

Let R be the set of all tokens in the ground-truth response, and let K be the full key cache from the prefill stage. The gold standard set of indices M_{gold} for a given budget B is determined by accumulated attention scores from all true response queries:

$$M_{gold} = \underset{i\in[0,N)}{TopK}\left(\sum_{j\in R}\text{Attention}(q_j, K)\right), \quad (1)$$

where N is the number of all prefill tokens, and $TopK$ returns the indices of the tokens with the highest accumulated scores. The predicted set of indices M_{pred} is defined analogously to M_{gold}, but computed using only the queries in a candidate observation window W:

$$M_{pred} = \underset{i\in[0,N)}{TopK}\left(\sum_{j\in W}\text{Attention}(q_j, K)\right). \quad (2)$$

The recall is the proportion of the gold-standard tokens that are correctly retained:

$$Recall_W = \frac{|M_{\text{gold}} \cap M_{pred}|}{|M_{gold}|}. \quad (3)$$

We evaluate this recall metric on the GovReport dataset for different observation windows. As shown in Figure 3, the window composed of pseudo queries with randomized content but correct future positions achieves significantly better recall than baselines like SnapKV and StreamingLLM.

Notably, it maintains high recall even as the window size is reduced to 32 or 16, showing the high effectiveness and accuracy of position-based estimation. This demonstrates that a small set of pseudo queries, informed solely by precise positional forecasting, provides a highly effective basis for importance assessment, enabling accurate eviction even under extreme window constraints.

In summary, our experiments converge on a pivotal insight: the representation of a query vector is dominated by its positional encoding, with semantic content playing a secondary role. From the perspective of query-key interactions, this further implies that the attention pattern relies heavily on positional information to establish importance distribution, while exhibiting considerable robustness to variations in semantic content. This leads to a profound practical implication: high-fidelity decoding pseudo queries can be synthesized from positional encodings, entirely bypassing the computationally expensive and memory-intensive process of token generation. This position-aware query approximation forms the foundation of our method.

4 Method

Motivated by the pivotal insight that positional information dominates query representations, we propose DapQ [1] (as illustrated in Figure 1), a novel KV cache compression framework that accurately simulates decoding-stage contextual positioning during the prefill phase. DapQ synthesizes a decoding-aligned observation window, composed of pseudo queries endowed with future positional encodings, which mirrors the dynamic context of the actual decoding process. This precisely assesses token importance, enabling accurate token eviction without altering the intended timeline.

4.1 Construct Decoding-Aligned Q_{pseudo}

The core of DapQ is to simulate the dynamic positional query of the decoding phase. For a prompt sequence of length L_p, we append a set of N artificially constructed tokens, denoted $\mathbf{T}_{\text{pseudo}}$, to form an extended input sequence. $\mathbf{T}_{\text{pseudo}}$ can be constructed or arbitrarily chosen from the existing context (e.g., uniformly sampled or prefix-suffix concatenation), as their semantic content is secondary to the positional assignment. The crucial operation is to assign $\mathbf{T}_{\text{pseudo}}$ the correct positional indices that they would occupy as the first N tokens generated by the model, rather than arbitrarily:

$$\text{Positions}(\mathbf{T}_{\text{pseudo}}) = [L_p, L_p+1, ..., L_p+N-1].$$

This yields an input sequence with length $L_{\text{total}} = L_p + N$. The model processes this extended sequence during the prefill phase, computing KV cache for L_{p} prompt tokens. The primary purpose of this step is to obtain pseudo queries (Q_{pseudo}) of these $\mathbf{T}_{\text{pseudo}}$, which are endowed with correct positional encodings for the start of decoding phase.

4.2 Importance Assessment and Eviction

We leverage the Q_{pseudo} to assess the importance of all Keys derived from the original prompt. The importance score for the j-th ($j \in [0, L_p - 1]$) prompt token is computed by aggregating its attention scores from each pseudo query $q_i \in Q_{\text{pseudo}}$:

$$S(j) = \sum_{i=L_p}^{L_p+N-1} \text{Attention}(q_i, k_j). \quad (4)$$

The $TopK$ tokens with the highest scores $S(j)$ are retained:

$$M_{\text{retain}} = \underset{j\in[0,L_p-1]}{TopK} \left(S(j)\right). \quad (5)$$

The KV cache is pruned, discarding all key-value pairs not in M_{retain}. **Crucially, the entire synthetic segment $\mathbf{T}_{\text{pseudo}}$ is discarded immediately after performing the importance scoring.** Autoregressive decoding phase then begins from position L_p, utilizing only the compressed cache of size K. This ensures the model's generation remains consistent with the intended timeline.

5 Experiments

5.1 Settings

Models and Benchmarks. To evaluate the applicability and generalization of DapQ in various models, we conduct experiments on LLaMA-3-8B-Instruct, LLaMA-3.1-8B-Instruct (Grattafiori et al., 2024), Qwen2.5-7B-Instruct (Yang et al., 2025b), and Qwen3-8B (Yang et al., 2025a). To ensure a more comprehensive and robust assessment, we use five benchmarks: LongBench (Bai et al., 2023), LongBenchV2 (Bai et al., 2024), Ruler (Hsieh et al., 2024), HELMET (Yen et al., 2024), and Needle-in-a-Haystack (Kamradt, 2024), each designed to assess distinct aspects of long-context inference, thereby forming a solid foundation for validating

[1] https://anonymous.4open.science/r/DapQ_code

	Methods	Single-Document QA		Multi-Document QA		Summarization		Few-shot Learning			Synthetic		Code		Avg.
		Qasper	MF-en	HotpotQA	2WikiMQA	GovReport	MultiNews	TREC	TriviaQA	SAMSum	PCount	PRe	Lcc	RB-P	
Llama3-8B-Instruct	FullKV	37.72	40.64	50.17	34.88	31.03	25.64	70.00	89.85	40.55	13.30	83.67	58.96	52.71	48.39
	KV Cache Size = 256														
	H2O	28.11	36.63	48.62	31.50	21.87	21.44	45.67	89.49	38.28	12.11	83.67	61.49	53.36	44.02
	PyramidKV	30.88	38.11	50.20	33.88	**22.54**	21.84	60.00	89.26	37.07	**12.78**	83.67	61.34	52.51	45.70
	SnapKV	30.84	**38.39**	49.75	33.80	22.18	21.53	57.00	89.65	36.97	12.11	**84.00**	61.78	54.92	45.61
	DapQ	**32.55**	38.18	**50.67**	**34.35**	22.25	**21.89**	**60.67**	**90.48**	**38.34**	11.78	83.67	**62.78**	**55.64**	**46.40**
	KV Cache Size = 128														
	H2O	25.95	36.25	48.65	31.90	**20.79**	20.30	40.00	87.29	36.25	12.33	**83.67**	59.81	53.14	42.79
	PyramidKV	28.80	**38.29**	49.52	31.60	20.67	20.55	49.00	87.68	36.73	**12.44**	82.00	60.36	52.03	43.82
	SnapKV	**29.52**	37.80	49.36	32.40	19.87	20.08	47.67	87.82	35.63	11.44	82.33	61.49	52.40	43.68
	DapQ	28.76	37.24	**50.04**	**33.59**	20.47	**20.63**	**50.00**	**90.06**	**36.87**	12.11	81.67	**61.81**	**53.92**	**44.40**
	KV Cache Size = 64														
	H2O	24.02	30.83	48.27	31.70	**19.37**	**19.14**	37.33	86.27	35.18	7.72	**82.33**	59.20	**51.10**	40.96
	PyramidKV	22.04	31.80	47.01	31.54	15.70	16.34	39.00	76.80	32.31	10.33	79.67	55.19	47.90	38.90
	SnapKV	25.06	32.92	47.16	31.71	16.85	17.09	**40.67**	86.02	33.99	11.78	78.00	57.95	50.91	40.78
	DapQ	**25.99**	**37.36**	**49.11**	**32.88**	18.46	18.70	38.67	**87.38**	**35.30**	**11.89**	77.67	**60.19**	49.90	**41.81**
Qwen3-8B	FullKV	36.87	53.67	57.67	44.73	33.39	23.69	71.67	91.79	42.07	11.98	86.67	70.64	59.26	52.60
	KV Cache Size = 256														
	H2O	27.80	44.92	51.37	40.50	22.38	18.26	46.33	90.04	38.98	12.33	86.67	66.11	53.93	46.12
	PyramidKV	30.41	47.81	51.00	41.37	22.36	17.41	61.67	90.96	37.65	13.00	86.33	67.01	50.11	47.47
	SnapKV	**32.40**	49.29	54.38	41.40	23.79	18.99	**63.00**	91.07	38.57	**14.67**	86.67	**67.99**	53.77	48.92
	DapQ	32.14	**50.78**	**54.79**	**44.47**	**24.16**	**19.01**	62.67	**91.15**	**39.61**	14.17	**86.67**	67.20	**53.83**	**49.28**
	KV Cache Size = 128														
	H2O	26.60	41.37	48.10	39.85	20.83	17.27	41.33	90.21	38.16	**11.72**	**86.67**	65.22	52.77	44.22
	PyramidKV	26.22	40.48	48.41	39.46	18.99	15.10	48.33	89.31	36.92	9.67	86.67	60.82	48.78	43.78
	SnapKV	**29.41**	46.43	51.20	41.66	20.37	16.64	51.33	**91.07**	37.37	11.00	86.67	65.36	51.65	46.17
	DapQ	29.10	**47.15**	**53.84**	**43.00**	**21.11**	**17.27**	**54.00**	90.45	**38.50**	11.67	86.00	**66.29**	**52.03**	**46.95**
	KV Cache Size = 64														
	H2O	25.55	38.94	46.66	39.27	**18.55**	**15.23**	39.00	88.13	35.98	9.67	**86.67**	59.48	**48.95**	42.47
	PyramidKV	25.32	40.44	46.61	39.20	16.25	12.93	**44.67**	88.27	34.63	11.33	83.67	59.73	46.48	42.27
	SnapKV	25.09	39.89	46.58	39.38	15.28	12.38	42.67	87.93	35.12	11.33	84.33	57.96	46.49	41.88
	DapQ	**25.78**	**43.17**	**49.84**	**41.28**	17.16	13.95	43.00	**88.97**	**36.00**	**12.67**	83.00	**60.31**	46.90	**43.23**

Table 2: Main Results on LongBench: performance comparison of different KV cache compression methods

Method	Batch Size					
	1	10	20	30	40	50
FullKV	11.59	26.43	25.60	26.59	OOM	OOM
LaCache	11.44	35.77	40.64	42.49	OOM	OOM
SLM	10.81	34.49	38.21	39.25	39.98	40.46
H2O	10.56	34.34	37.71	39.02	39.62	40.01
PyramidKV	10.54	34.12	38.00	39.03	39.86	40.11
SnapKV	10.77	34.22	38.09	39.10	39.77	40.23
DapQ	10.68	34.16	37.99	38.97	39.73	40.12

Table 3: Comparison of throughput (tokens/s) with different batch sizes.

Method	Context Length				
	8K	16K	32K	64K	128K
FullKV	1.1106	2.5607	6.5718	18.9441	60.8399
LaCache	1.1283	2.5921	6.6356	19.0519	61.5180
SLM	1.1236	2.5801	6.6008	18.9853	61.0339
H2O	1.1348	2.6046	6.6352	19.0379	61.5029
PyramidKV	1.1253	2.5958	6.6337	19.0462	61.5345
SnapKV	1.1278	2.5909	6.6242	19.0318	61.5017
DapQ	1.1298	2.5974	6.6289	19.0423	61.5097

Table 4: Comparison of Time-to-First-Token (TTFT (s)) with different context lengths.

DapQ's performance across diverse scenarios. Due to space limitations, complete experiment results and details are presented in Appendix B.

Baselines. To comprehensively validate the performance of DapQ, we select six representative KV cache compression methods as baselines: **FullKV** caches all keys and values for every token, which is the standard approach for KV Cache in transformer-based models; **SnapKV** (Li et al., 2024) captures attention signals from an observation window and uses pooling-based clustering to select important KV pairs for compression; **PyramidKV** (Cai et al., 2024) leverages cross-layer attention distribution characteristics to dynamically allocate different KV cache budgets and selects important KV pairs for compression; **H2O** (Zhang et al., 2023) identifies Heavy Hitter (H2) tokens based on cumulative attention scores and dynamically balances the retention of recent and H2 tokens to compress KV cache; **StreamingLLM (SLM)** (Xiao et al., 2023) identifies the attention sink and dynamically balances the retention of recent and initial tokens to compress KV cache; **LaCache** (Shi et al., 2025) adopts a ladder-shaped pattern in the prefilling stage to retain KV of early tokens in shallow layers and gradually shift to later tokens in deeper layers. **Note:** To ensure rigor and consistency, compression is performed solely during the prefill stage.

Implementation Details. For all methods, we set the observation window size to 32 unless otherwise specified (e.g., LaCache use its default settings). In DapQ, pseudo queries are constructed by concatenating a small number of tokens from the beginning and the end of the input sequence(e.g., the first 4 and last 28 tokens, the first 2 and last 30 tokens). This design is motivated by two key considerations: the beginning tokens, often high-frequency special tokens (e.g.,<|begin_of_text|>), possess stable and generalizable embeddings due to their extensive exposure during training; the ending tokens carry the most recent context, making their semantic state highly relevant to the imminent decoding step. This finding is further supported by Liu et al.

(2023). We also validate it through experiments in Figure 4a, where concatenating prefix and suffix tokens consistently yields superior performance compared to using random or intermediate consecutive tokens from the input as query contents.

5.2 Improvement on Accuracy

Consistent accuracy gains across benchmarks: **LongBench Results.** As shown in Table 2, DapQ consistently outperforms all baselines across different models and cache budgets. The advantage is particularly pronounced under aggressive compression (e.g., budget=64), where DapQ shows a robust ability to retain critical information (e.g., preserving the long-range contextual dependencies especially on complex information integration tasks like HotpotQA and 2WikiMQA) and mitigate high-compression performance degradation. **LongBenchV2 Results.** Under a 64 cache budget, DapQ achieves 29.26% accuracy in the category of "Hard", marking a +6.75% absolute improvement over SnapKV (22.51%) on LLaMA3-8B-Instruct. **Ruler Results.** Table 11 shows that DapQ attains a notable 59.6% accuracy on the challenging S-NIAH-3 task with a cache budget of 512, substantially outperforming SnapKV (1.4%) and H2O (2.4%). **HELMET Results.** As reported in Table 14, DapQ achieves an average score of 48.10 on Qwen2.5-7B-Instruct with a low cache budget of 512, surpassing strong baselines SnapKV (43.74), H2O (40.36), and PyramidKV (42.49). **Needle-in-a-Haystack Results.** As shown in Figure 5 and Table 15, a striking example is on LLaMA3-8B with a cache size of 256: DapQ achieves 99.5% accuracy, closely approaching full-cache performance. This exceptional performance shows DapQ effectively simulates the decoding-stage positional context via prospectively encoded pseudo queries, enabling precise identification and retention of the key "needles" amidst a vast "haystack" of tokens.

Across diverse benchmarks, DapQ demonstrates the highest average score across nearly all budget settings on different models and especially delivers strong performance gains under strict cache constraints. These results underscores its effectiveness, robustness and generalizability in identifying and preserving critical contextual information.

5.3 Analysis on Efficiency

We conduct a comprehensive efficiency evaluation of DapQ using Llama-3.1-8B-Instruct. We first focus on memory usage and throughput across different batch sizes (Input 8k, Output 150 tokens, Budget=256). As shown in Table 3 and Table 7, DapQ maintains robust performance, exhibiting memory and throughput highly on par with other compression methods. Furthermore, to assess the algorithmic overhead introduced during the prefill stage, we measure the Time-to-First-Token (TTFT) across varying sequence lengths from 8K to 128K. Table 4 indicates that the latency of DapQ is nearly identical to that of SnapKV. The results confirms that the additional prefill-stage overhead is almost negligible. Overall, DapQ achieves a balanced performance between long-context understanding capability and inference efficiency, ensuring its practicality for real-world long-context applications.

6 Analysis

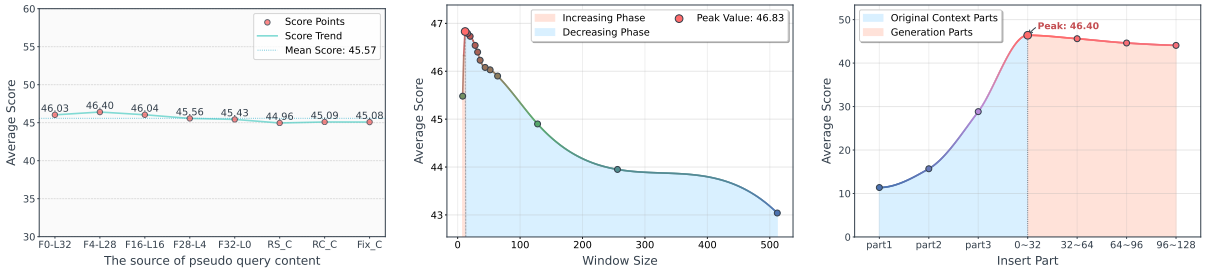
6.1 The Impact of Q_{pseudo} Semantic Content

To further investigate the practical impact of semantic variation on KV cache compression performance, we conduct an ablation study by evaluating DapQ under a fixed cache budget while altering the semantic content of pseudo queries. As shown in Figure 4a, the average performance remains highly stable (e.g., coefficient of variation $\approx 1\%$), regardless of whether the pseudo-query window is constructed from different semantic contents (e.g., the input's prefix and suffix, random in-context tokens, or nonsensical sequences). This consistency provides compelling empirical evidence that the attention pattern relies significantly on positional information to establish importance distribution, rendering the semantic content a secondary factor.

6.2 The Impact of Q_{pseudo} Length

The length of pseudo queries (the size of an observation window, N) is a crucial hyper-parameter, controlling the breadth of the simulated decoding context used for importance estimation. Figure 4b reveals a non-monotonic relationship between N and performance, characterized by distinct increasing and decreasing phases.

Increasing Phase (Small N): For small window sizes, performance increases sharply as N grows. This is because a small window lacks the contextual breadth to robustly estimate the importance of all relevant tokens, causing high uncertainty in the importance assessment. Adding more pseudo queries can provide a more comprehensive simulation of the decoding process, leading to a more accurate and holistic importance distribution. This



(a) The Impact of Q_{pseudo} Quality (b) The Impact of Q_{pseudo} Length (c) The Impact of Q_{pseudo} Insert Part

Figure 4: Ablation analysis of pseudo queries with respect to quality, length and insert positions. (a) Performance under a fixed Q_{pseudo} length 32, with varying semantic content of Q_{pseudo}: `Fm_Ln` is the concatenation of the first `m` and the last `n` tokens from the input context; `RS_C` is constructed by concatenating 32 randomly sampled individual tokens from the context; `RC_C` is a randomly sampled consecutive span of 32 tokens from the context; and `Fix_C` is a fixed, repetitive nonsensical sequence (e.g., "Sorry, I don't know. Sorry, I don't know..."). (b) Performance under varying observation window sizes N. (c) Performance under varying insert positions of Q_{pseudo}. Left Parts: we divide the original context interval $[0, L_p)$ equally into three segments: `part1` (beginning), `part2` (middle), and `part3` (end). For each segment, we randomly select a continuous sequence of 32 positions and map the position IDs of Q_{pseudo} to it. Right Parts: we only move backwards the position IDs of Q_{pseudo} to continuous 32 positions at different offsets after the context L_p (e.g., $[L_p + 0, L_p + 32)$), $[L_p + 32, L_p + 64)$).

expanded window thereby enables the model to identify and retain a greater number of critical tokens for future effective generation.

Decreasing Phase (Large N): Beyond a certain point, further increasing N leads to a performance decline. We attribute this to a dilution effect: while initial queries in the window are precisely aligned with the start of decoding, later queries represent increasingly speculative future positions. The attention pattern for these distant positions becomes progressively diffuse and attends to tokens less relevant for initial generation steps, introducing noisier and less reliable signals into the aggregated importance scores. And mutual attention among these queries introduces additional interference, further diverting the focus from critical tokens.

This analysis identifies that the window should be sufficiently sized to capture a representative decoding context but not so large as to dilute the attentional signal. The existence of this optimum further confirms that our method is not relying on a brute-force approach. Instead, it performs a precise and efficient simulation by concentrating on the most relevant segment of the decoding trajectory.

6.3 The Impact of Q_{pseudo} Insert Positions

We observe that deviating from the proposed Q_{pseudo} placement (i.e., immediately following the prompt) leads to performance degradation. This phenomenon can be explained by two key factors.

Insufficient context visibility when inserted within the context: Under the standard causal attention mask, Q_{pseudo} placed at position t can only attend to the prefix $[0, t)$ and is unable to access the subsequent context. This conflicts with our goal that Q_{pseudo} should approximate the attention patterns over the full context as seen by future decoding steps. Consequently, as shown in the left part of Figure 4c, Q_{pseudo} placed earlier in the context fail to reconstruct global attention patterns and exhibit degraded performance.

The assigned position of Q_{pseudo} shifts backwards away from the correct interval, introduces increasing misalignment in the RoPE embedding space. Large backward shifts in positions lead to accumulated rotational offsets, causing the Q_{pseudo} representation space to progressively diverge from those of real queries. This misalignment makes it harder for Q_{pseudo} to approximate the target attention distribution, explaining the observed performance drop with larger positional shifts, as clearly illustrated in the right part of Figure 4c.

7 Conclusion

This work underscores the primacy of positional information over semantic content in constructing query approximations and determining attention patterns. We introduce DapQ, a novel KV cache compression framework that leverages position-aware pseudo queries to simulate the output tokens, thereby establishing an effective observation window for importance assessment. During the prefill stage, it aligns closely with the actual generation context and enables precise token eviction. Extensive experiments demonstrate that DapQ consistently outperforms existing baselines and achieves superior performance in long-context scenarios, particularly under strict memory constraints.

Limitations

Although DapQ achieves excellent results, there are still some limitations.

While preliminary experiments indicate that positional information dominates query approximation, semantic content still plays a non-negligible role (as shown in Table 1: similarity drops to 0.3267 when content is the same but positions differ, whereas it remains at 0.7238 when positions are correct but content is irrelevant). This suggests that further optimizing the semantic content of Q_{pseudo} to better mirror actual decoding could lead to additional performance gains and enhanced robustness. Future work may explore how to intelligently construct or select semantically more meaningful Q_{pseudo} content without introducing significant computational overhead, thereby approaching near-lossless performance even under extreme compression scenarios.

The sensitivity of Q_{pseudo} to positional and semantic information may vary across layers, which could lead to suboptimal compression in certain layers. Future work could explore layer-wise or adaptively-aware Q_{pseudo} approximation mechanisms to better align with the attention distribution of each layer, thereby further improving overall compression efficiency and generation quality.

References

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

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

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and 1 others. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, and 1 others. 2023. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, and 1 others. 2024. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv preprint arXiv:2412.15204*.

Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan-Kelley. 2024. Reducing transformer key-value cache size with cross-layer attention. *Advances in Neural Information Processing Systems*, 37:86927–86957.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and 1 others. 2024. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*.

Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed S Abdelfattah, and Kai-Chiang Wu. 2024. Palu: Compressing kv-cache with low-rank projection. *arXiv preprint arXiv:2407.21118*.

Guoxuan Chen, Han Shi, Jiawei Li, Yihang Gao, Xiaozhe Ren, Yimeng Chen, Xin Jiang, Zhenguo Li, Weiyang Liu, and Chao Huang. 2024. Sepllm: Accelerate large language models by compressing one segment into one separator. *arXiv preprint arXiv:2412.12094*.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023a. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. 2023b. Longlora: Efficient fine-tuning of long-context large language models. *arXiv preprint arXiv:2309.12307*.

Wanyun Cui and Mingwei Xu. 2025. Homogeneous keys, heterogeneous values: Exploiting local kv cache asymmetry for long-context llms. *arXiv preprint arXiv:2506.05410*.

Alessio Devoto, Yu Zhao, Simone Scardapane, and Pasquale Minervini. 2024. A simple and effective l_2 norm-based strategy for kv cache compression. *arXiv preprint arXiv:2406.11430*.

Harry Dong, Xinyu Yang, Zhenyu Zhang, Zhangyang Wang, Yuejie Chi, and Beidi Chen. 2024. Get more with less: Synthesizing recurrence with kv cache compression for efficient llm inference. *arXiv preprint arXiv:2402.09398*.

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

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Albert Gu, Karan Goel, and Christopher Ré. 2021. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*.

Dongsheng Jiang, Yuchen Liu, Songlin Liu, Jin'e Zhao, Hao Zhang, Zhen Gao, Xiaopeng Zhang, Jin Li, and Hongkai Xiong. 2023. From clip to dino: Visual encoders shout in multi-modal large language models. *arXiv preprint arXiv:2310.08825*.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y Fu, Christopher Ré, and Azalia Mirhoseini. 2024. Hydragen: High-throughput llm inference with shared prefixes. *arXiv preprint arXiv:2402.05099*.

Gregory Kamradt. 2024. Needle in a haystack-pressure testing llms, 2023. *URL https://github.com/gkamradt/LLMTest_NeedleInAHaystack*.

Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. 2024. Gear: An efficient kv cache compression recipe for near-lossless generative inference of llm. *arXiv preprint arXiv:2403.05527*.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024b. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Jiaheng Liu, Dawei Zhu, Zhiqi Bai, Yancheng He, Huanxuan Liao, Haoran Que, Zekun Wang, Chenchen Zhang, Ge Zhang, Jiebin Zhang, and 1 others. 2025. A comprehensive survey on long context language modeling. *arXiv preprint arXiv:2503.17407*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*.

Ran Liu, Ming Liu, Min Yu, He Zhang, Jianguo Jiang, Gang Li, and Weiqing Huang. 2024c. Sumsurvey: An abstractive dataset of scientific survey papers for long document summarization. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 9632–9651.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024d. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*.

Bowen Peng and Jeffrey Quesnelle. 2023. Ntk-aware scaled rope allows llama models to have extended (8k+) context size without any fine-tuning and minimal perplexity degradation.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*.

Dachuan Shi, Yonggan Fu, Xiangchi Yuan, Zhongzhi Yu, Haoran You, Sixu Li, Xin Dong, Jan Kautz, Pavlo Molchanov, and 1 others. 2025. Lacache: Ladder-shaped kv caching for efficient long-context modeling of large language models. *arXiv preprint arXiv:2507.14204*.

Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. 2024. Loki: Low-rank keys for efficient sparse attention. *Advances in Neural Information Processing Systems*, 37:16692–16723.

Yi Su, Quantong Qiu, Yuechi Zhou, Juntao Li, Qingrong Xia, Ping Li, Xinyu Duan, Zhefeng Wang, and Min Zhang. 2025. Calidrop: Kv cache compression with calibration. *arXiv preprint arXiv:2507.19906*.

Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. 2024. You only cache once: Decoder-decoder architectures for language models. *Advances in Neural Information Processing Systems*, 37:7339–7361.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, and 1 others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Yuxuan Tian, Zihan Wang, Yebo Peng, Aomufei Yuan, Zhiming Wang, Bairen Yi, Xin Liu, Yong Cui, and Tong Yang. 2025. Keepkv: Eliminating output perturbation in kv cache compression for efficient llms inference. *arXiv preprint arXiv:2504.09936*.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*.

Yixuan Wang, Shiyu Ji, Yijun Liu, Yuzhuang Xu, Yang Xu, Qingfu Zhu, and Wanxiang Che. 2025. Lookahead q-cache: Achieving more consistent kv cache eviction via pseudo query. *arXiv preprint arXiv:2505.20334*.

Haoyi Wu and Kewei Tu. 2024. Layer-condensed kv cache for efficient inference of large language models. *arXiv preprint arXiv:2405.10637*.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, and 1 others. 2025b. Qwen2. 5-1m technical report. *arXiv preprint arXiv:2501.15383*.

Dongquan Yang, Yifan Yang, Xiaotian Yu, Xianbiao Qi, and Rong Xiao. 2025c. Hcattention: Extreme kv cache compression via heterogeneous attention computing for llms. *arXiv preprint arXiv:2507.19823*.

Yifei Yang, Zouying Cao, Qiguang Chen, Libo Qin, Dongjie Yang, Hai Zhao, and Zhi Chen. 2024. Kvsharer: Efficient inference via layer-wise dissimilar kv cache sharing. *arXiv preprint arXiv:2410.18517*.

Zhifan Ye, Kejing Xia, Yonggan Fu, Xin Dong, Jihoon Hong, Xiangchi Yuan, Shizhe Diao, Jan Kautz, Pavlo Molchanov, and Yingyan Celine Lin. 2025. Longmamba: Enhancing mamba's long context capabilities via training-free receptive field enlargement. *arXiv preprint arXiv:2504.16053*.

Howard Yen, Tianyu Gao, Minmin Hou, Ke Ding, Daniel Fleischer, Peter Izsak, Moshe Wasserblat, and Danqi Chen. 2024. Helmet: How to evaluate long-context language models effectively and thoroughly. *arXiv preprint arXiv:2410.02694*.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, and 1 others. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710.

Lei Zhu, Xinjiang Wang, Wayne Zhang, and Rynson WH Lau. 2024. Relayattention for efficient large language model serving with long system prompts. *arXiv preprint arXiv:2402.14808*.

A Preliminary Experiment Details

A.1 Query similarity comparison under different content and position conditions.

We conduct the query similarity analysis using a representative example from the GovReport dataset, which is designed for document summarization tasks. The input sequence consists of 4424 tokens. The ground-truth decoding queries are obtained by extracting the first 32 output tokens generated by LLaMA-3-8B-Instruct for this input. The semantic content of these output tokens is: "The report discusses the Federal Aviation Administration's (FAA) state block grant pilot program, which is part of its Airport Improvement Program (AIP). The program", and they are assigned the positional indices 4424-4455. We then construct a Q_{pseudo} of length 32 with the repetitive content: "Sorry, I don't know. Sorry, I don't know. Sorry, I don't know. Sorry, I don't know. Sorry, I". Notably, the positional IDs for the Q_{pseudo} can be flexibly configured to emulate various decoding scenarios.

A.1.1 Experimental Details of Query Similarity Comparison

In Table 1, we evaluate Q_{pseudo} by varying two key attributes relative to the ground-truth decoding queries: semantic content and positional assignment. The content is either **consistent with** the true output (i.e., the actual beginning of the model's summary, "The report discusses...") or **different from** it (e.g., a fixed and nonsensical sequence, "Sorry, I don't know..."). Similarly, the positional indices are either **aligned with** the true future decoding positions (i.e., 4424-4455) or **deviated from** them (e.g., assigned to a random consecutive span 0-31). The cosine similarity between each set of Q_{pseudo} and the ground-truth queries is reported under two measurement conditions: *Post ROPE*, which captures the final query representation after the application of Rotary Position Embedding (ROPE), and *Pre ROPE*, which reflects the similarity before the positional encoding is applied.

A.1.2 Experimental Details of Query Similarity Distribution Analysis

To quantitatively assess the impact of positional and content variations on query representation, we conduct a large-scale statistical analysis as depicted in Figure 2a. Each box is aggregated from 100 independent trials, providing a stable estimate of the similarity distribution. The **Different Content (DC)** condition is implemented by randomly sampling 32 tokens from the model's full vocabulary, effectively removing any meaningful semantic correlation with the true output. The **Different Position (DP)** condition is implemented by assigning a consecutive span of 32 positions randomly sampled from two distinct ranges to introduce positional deviation: a general deviation range (0-4000), which represents a random mismatch within the context window, and an extreme deviation range (0-100), which is specifically chosen to maximize the absolute offset from the correct positions (i.e., 4424-4455), thereby rigorously testing the hypothesis that positional accuracy is dominant.

A.2 Experimental Details of Positional Offset Sensitivity

To systematically quantify the sensitivity of query representations to positional miscalibration, we conduct the analysis presented in Figure 2b. The experiment investigates how the similarity between Q_{pseudo} and the true decoding decays based on the absolute offset between their assigned positional indices and the correct future positions. For this purpose, we select input examples of varying context lengths (2k, 4k, 6k, and 8k tokens) from the GovReport dataset. For each context length, we construct Q_{pseudo} with fixed semantic content (aligned with the true output) but systematically vary their assigned starting position. The x-axis represents this starting position assigned to Q_{pseudo} (e.g., an x-axis value of 3500 indicates that the 32 Q_{pseudo} are assigned the consecutive position IDs from 3500 to 3531). The y-axis measures the resulting cosine similarity between the Q_{pseudo} and the ground-truth decoding queries. This approach allows us to observe the monotonic decay in similarity with increasing positional offset.

A.3 Generalization Analysis of Positional Dominance

To further validate the universality of the "Positional Dominance" phenomenon and address potential concerns regarding model-specific or dataset-specific biases, we extended our experimental analysis beyond the Llama-3-8B model and the GovReport dataset. We conducted supplementary experiments using the Qwen2.5-7B-Instruct model across two distinct task domains: Multi-Document QA (using the 2WikiMQA dataset) and Code Completion (using the LCC dataset). Strictly adhering to the experimental configuration outlined in Table

Experiment	Content Sim.	Positional Sim.	Post ROPE	Pre ROPE
SC & SP	Same	Same	1.0000	1.0000
DC & SP	Different	Same	0.7142	0.7142
SC & DP	Same	Different	0.4270	0.7341
DC & DP	Different	Different	0.4113	0.7224

Table 5: Query cosine similarity comparison on the LCC dataset using Qwen2.5-7B-Instruct.

Experiment	Content Sim.	Positional Sim.	Post ROPE	Pre ROPE
SC & SP	Same	Same	1.0000	1.0000
DC & SP	Different	Same	0.7236	0.7236
SC & DP	Same	Different	0.4671	0.7359
DC & DP	Different	Different	0.4428	0.7192

Table 6: Query cosine similarity comparison on the 2WikiMQA dataset using Qwen2.5-7B-Instruct.

1, we randomly sampled 100 examples from each dataset to compute the average cosine similarities. As shown in Table 5 and Table 6, the quantitative results and experimental phenomena for the LCC and 2WikiMQA dataset are consistent with our observations on Llama. This evidence strongly supports the generalization of our central insight.

B Complete Main Experiment Results and Details

B.1 Results and Details on LongBench

We comprehensively evaluate the performance of DapQ and baselines on LongBench benchmark shown in Table 9, with the following setup:

- **Models:** LLaMA-3-8B-Instruct, Qwen2.5-7B-Instruct, Qwen3-8B(Reasoning OFF);
- **KV Cache Budgets:** 256, 128, 64 tokens.

B.2 Results and Details on LongBenchV2

We comprehensively evaluate the performance of DapQ and baselines on LongBenchV2 benchmark shown in Table 10, with the following setup:

- **Models:** LLaMA-3-8B-Instruct, LLaMA-3.1-8B-Instruct, Qwen2.5-7B-Instruct, Qwen3-8B(Reasoning OFF);
- **KV Cache Budgets:** 128, 64 tokens.

B.3 Results and Details on Ruler

We comprehensively evaluate the performance of DapQ and baselines on Ruler benchmark shown in Table 11, Table 12, Table 13, with the following setup:

- **Models:** LLaMA-3-8B-Instruct, Qwen2.5-7B-Instruct, Qwen3-8B(Reasoning OFF);
- **KV Cache Budgets:** 4096, 2048, 1024, 512, 256, 128, 64 tokens.

Method	Batch Size					
	1	10	20	30	40	50
FullKV	16.87	33.74	52.49	71.24	OOM	OOM
LaCache	16.87	33.74	52.49	71.24	OOM	OOM
SLM	16.01	25.17	35.36	45.55	55.73	65.92
H2O	16.01	25.17	35.36	45.55	55.73	65.92
PyramidKV	16.01	25.23	35.47	45.72	55.96	66.20
SnapKV	16.01	25.17	35.36	45.55	55.73	65.92
DapQ	16.01	25.21	35.43	45.65	55.87	66.02

Table 7: Comparison of memory usage (GB) with different batch sizes.

B.4 Results and Details on HELMET

We comprehensively evaluate the performance of DapQ and baselines on HELMET benchmark shown in Table 14, with the following setup:

- **Models:** LLaMA-3-8B-Instruct, Qwen2.5-7B-Instruct;
- **KV Cache Budgets:** 2048, 1024, 512, 256, 128 tokens.

B.5 Results and Details on NIAH

We comprehensively evaluate the performance of DapQ and baselines on Needle-in-a-Haystack(NIAH) benchmark shown in Table 15, with the following setup:

- **Models:** LLaMA-3-8B-Instruct, LLaMA-3.1-8B-Instruct, Qwen2.5-7B-Instruct, Qwen3-8B(Reasoning OFF);
- **KV Cache Budgets:** 256, 128, 64 tokens.

B.6 Efficiency Evaluation Setup

We conduct a comprehensive efficiency evaluation of DapQ, focusing on memory usage (Table 7), throughput (Table 3), and Time-to-First-Token (Table 4). The specific experimental environment and configurations are detailed as follows:

- **Hardware & Environment:** All efficiency experiments are conducted on a single H20 96G GPU. The environment consists of Python 3.10, PyTorch 2.6.0, and Transformers 4.53.0.
- **Attention Kernel:** Utilize Flash-Attention (version 2.7.4) to accelerate attention computation.
- **Inference Framework:** Adopt the native Hugging Face transformers implementation rather than vLLM.

C Theoretical and Empirical Evidence for Query-Attention Alignment

C.1 Theoretical Analysis

To rigorously verify that position-Based queries enable more accurate token eviction, we derive a formal theorem to establish that the similarity of

query vectors directly constrains both the upper bound of the attention score error and the lower bound of the attention distribution similarity. This provides a principled justification that high query similarity necessarily leads to high attention distribution.

(1) Theorem: Given a fixed KV set $\{K, V\}$ with $K \in \mathbb{R}^{n\times d_k}$ and $V \in \mathbb{R}^{n\times d_v}$, let two unit query vectors $q, q' \in \mathbb{R}^{1\times d_k}$ have cosine similarity defined as $\text{sim}(q, q') = q \cdot q'$. The difference between their corresponding attention scores satisfies:

$$\|\text{Attention}(q, K) - \text{Attention}(q', K)\|_1 \leq \frac{K_{\max}\cdot\sqrt{2(1-\text{sim}(q,q'))}}{\sqrt{d_k}} \tag{1}$$

$$\text{sim}(\text{Attention}(q, K), \text{Attention}(q', K)) \geq 1 - \frac{K_{\max}\cdot\sqrt{2(1-\text{sim}(q,q'))}}{2\sqrt{d_k}} \tag{2}$$

where:

- d_k is the key-vector dimension;
- $K_{\max} = \max_j \|k_j\|$, with k_j the j-th key vector;
- $\text{Attention}(q, K) = \text{softmax}\left(\frac{qK^T}{\sqrt{d_k}}\right)$

and these quantities (i.e., d_k and $K_{\max}$) are constants under the fixed KV set.

(2) Core conclusion: **Inequality (1)** directly establishes a positive correlation between the *cosine similarity of queries* and the *similarity of their attention scores*. Specifically, as $\text{sim}(q, q') \to 1$, the upper bound on the scores difference approaches 0, meaning that the attention scores produced by q and q' become almost identical. **Inequality (2)** further quantifies the lower bound constraint that query similarity imposes on attention score similarity — the higher the query similarity, the higher the lower bound on attention score similarity.

(3) Detailed Derivation:

Step 1: Define key variables and similarity measures

- **Cosine similarity of queries.** In modern LLMs, query vectors typically originate from normalization layers (e.g., LayerNorm), ensuring their norms remain a stable constant (i.e., $\|q\| \approx \|q'\| \approx C$). Without loss of generality, we assume unitary magnitude ($C = 1$) to simplify the notation. Consequently, their cosine similarity reduces to:

$$\text{sim}(q, q') = \frac{q \cdot q'}{\|q\| \cdot \|q'\|} = q \cdot q'.$$

- **Pre-softmax scores.** The raw matching score between the query and each key is

$$s_j = \frac{q \cdot k_j}{\sqrt{d_k}}, \quad s'_j = \frac{q' \cdot k_j}{\sqrt{d_k}}, \quad (j = 1, 2, \ldots).$$

- **Attention weights.** After softmax normalization, the attention weights are

$$\alpha = \text{softmax}(s), \qquad \alpha' = \text{softmax}(s'),$$

where $\alpha_j \geq 0$ and $\sum_j \alpha_j = 1$.

Step 2: Upper bound the difference of pre-softmax scores

Using the Cauchy–Schwarz inequality $|x \cdot y| \leq \|x\| \cdot \|y\|$, the score difference induced by two queries satisfies

$$\begin{aligned} |s_j - s'_j| &= \left|\frac{q \cdot k_j}{\sqrt{d_k}} - \frac{q' \cdot k_j}{\sqrt{d_k}}\right| \\ &= \frac{1}{\sqrt{d_k}}|(q - q') \cdot k_j| \\ &\leq \frac{1}{\sqrt{d_k}}\|q - q'\| \cdot \|k_j\|. \end{aligned} \tag{3}$$

Next, for unit vectors q and q' we have the standard identity

$$\|q-q'\|^2 = \|q\|^2+\|q'\|^2-2q\cdot q' = 2(1-\text{sim}(q, q')),$$

hence

$$\|q - q'\| = \sqrt{2\big(1 - \text{sim}(q, q')\big)}.$$

Substituting this into the score bound Inequality (3) and taking the maximum key norm $K_{\max} = \max_j \|k_j\|$, we obtain

$$\max_j \left|s_j - s'_j\right| \leq \frac{K_{\max} \cdot \sqrt{2\big(1 - \text{sim}(q, q')\big)}}{\sqrt{d_k}} \tag{4}$$

Step 3: Upper bound the difference of attention weights

The softmax function is Lipschitz continuous: for any score vectors s, s', we have

$$\|\text{softmax}(s) - \text{softmax}(s')\|_1 \leq \|s - s'\|_\infty.$$

We note that this bound corresponds to a global worst-case Lipschitz constant of the softmax function; while tighter, input-dependent bounds can be derived via the Jacobian, the above inequality suffices for establishing a conservative and general theoretical guarantee.

Here, the infinity norm is defined as $\|s-s'\|_\infty = \max_j |s_j - s'_j|$, and the L_1 norm of the attention-weight difference is $\|\alpha - \alpha'\|_1 = \sum_j |\alpha_j - \alpha'_j|$.

Combining this with Inequality (4), we obtain

$$\|\alpha - \alpha'\|_1 \leq \frac{K_{\max} \cdot \sqrt{2\big(1 - \mathrm{sim}(q, q')\big)}}{\sqrt{d_k}} \quad (5)$$

Step 4: Lower bound the difference of attention weights

We adopt a standard similarity measure for attention scores based on the Total Variation Distance:

$$\mathrm{sim}(\alpha, \alpha') = 1 - \frac{1}{2}\|\alpha - \alpha'\|_1$$

where $\|\alpha - \alpha'\|_1$ takes values in $[0, 2]$.

First, we scale the L1 difference $[0, 2]$ to $[0, 1]$: $\frac{1}{2}\|\alpha - \alpha'\|_1$.

Then, by subtracting this scaled value from 1, we convert the "difference" into "similarity":

- Smaller difference $\rightarrow$ smaller scaled value $\rightarrow$ similarity closer to 1;
- Larger difference $\rightarrow$ larger scaled value $\rightarrow$ similarity closer to 0.

The above definition is a standard measure of similarity in the field of probability distributions. Substituting Inequality (5) into this definition yields the final lower bound:

$$\mathrm{sim}(\alpha, \alpha') \geq 1 - \frac{K_{\max} \cdot \sqrt{2(1 - \mathrm{sim}(q, q'))}}{2\sqrt{d_k}} \quad (6)$$

C.2 Empirical Evaluation

We evaluate the cosine similarity, between Snapkv, DapQ's observation window attention weights for the input context and ground-truth decoding query's attention weights for the input context under different window sizes. As shown in the Table 8, across all window sizes, DapQ consistently achieves substantially higher attention-weight similarity compared to SnapKV. This directly demonstrates that better query approximation indeed translates into more accurate attention distributions.

Table 8: Cosine similarity of softmax-normalized attention weights for DapQ vs. SnapKV.

Window Size	128	64	32	16	8	4	2	1
DapQ	0.9785	0.9747	0.9726	0.9713	0.9705	0.9644	0.9615	0.8768
SnapKV	0.9463	0.9429	0.9508	0.9466	0.9416	0.9385	0.9332	0.8426

Table 9: Performance comparison of different methods across various LLMs on LongBench.

		Single-Document QA		**Multi-Document QA**		**Summarization**		**Few-shot Learning**			**Synthetic**		**Code**		
	Methods	**Qasper**	**MF-en**	**HotpotQA**	**2WikiMQA**	**GovReport**	**MultiNews**	**TREC**	**TriviaQA**	**SAMSum**	**PCount**	**PRe**	**Lcc**	**RB-P**	**Avg.**
Llama3-8B-Instruct	FullKV	37.72	40.64	50.17	34.88	31.03	25.64	70.00	89.85	40.55	13.30	83.67	58.96	52.71	48.39
	KV Cache Size = 256														
	H2O	28.11	36.63	48.62	31.50	21.87	21.44	45.67	89.49	38.28	12.11	83.67	61.49	53.36	44.02
	PyramidKV	30.88	38.11	50.20	33.88	**22.54**	21.84	60.00	89.26	37.07	**12.78**	83.67	61.34	52.51	45.70
	SnapKV	30.84	**38.39**	49.75	33.80	22.18	21.53	57.00	89.65	36.97	12.11	**84.00**	61.78	54.92	45.61
	DapQ	**32.55**	38.18	**50.67**	**34.35**	22.25	**21.89**	**60.67**	**90.48**	**38.34**	11.78	83.67	**62.78**	**55.64**	**46.40**
	KV Cache Size = 128														
	H2O	25.95	36.25	48.65	31.90	**20.79**	20.30	40.00	87.29	36.25	12.33	**83.67**	59.81	53.14	42.79
	PyramidKV	28.80	**38.29**	49.52	31.60	20.67	20.55	49.00	87.68	36.73	**12.44**	82.00	60.36	52.03	43.82
	SnapKV	**29.52**	37.80	49.36	32.40	19.87	20.08	47.67	87.82	35.63	11.44	82.33	61.49	52.40	43.68
	DapQ	28.76	37.24	**50.04**	**33.59**	20.47	**20.63**	**50.00**	**90.06**	**36.87**	12.11	81.67	**61.81**	**53.92**	**44.40**
	KV Cache Size = 64														
	H2O	24.02	30.83	48.27	31.70	**19.37**	**19.14**	37.33	86.27	35.18	7.72	**82.33**	59.20	**51.10**	40.96
	PyramidKV	22.04	31.80	47.01	31.54	15.70	16.34	39.00	76.80	32.31	10.33	79.67	55.19	47.90	38.90
	SnapKV	25.06	32.92	47.16	31.71	16.85	17.09	**40.67**	86.02	33.99	11.78	78.00	57.95	50.91	40.78
	DapQ	**25.99**	**37.36**	**49.11**	**32.88**	18.46	18.70	38.67	**87.38**	**35.30**	**11.89**	77.67	**60.19**	49.90	**41.81**
Qwen2.5-7B-Instruct	FullKV	36.50	49.70	55.91	44.70	31.64	22.84	66.33	89.34	42.49	11.00	86.33	61.97	59.97	50.67
	KV Cache Size = 256														
	H2O	27.76	42.94	47.89	40.97	22.13	18.31	44.33	84.51	39.77	10.67	86.00	57.07	54.09	44.31
	PyramidKV	29.65	46.37	49.89	40.77	20.42	16.80	53.00	87.89	39.61	10.67	86.00	52.61	49.49	44.82
	SnapKV	**31.12**	**47.87**	51.76	40.78	22.25	**18.41**	53.33	87.66	39.34	10.67	86.00	56.87	**54.36**	46.19
	DapQ	30.21	45.00	**51.92**	**41.46**	**22.35**	18.40	**56.67**	**88.64**	**39.92**	**11.00**	**86.00**	**58.40**	53.82	**46.45**
	KV Cache Size = 128														
	H2O	26.83	37.80	45.14	39.77	**20.13**	16.64	40.67	81.43	38.56	10.67	**85.67**	53.97	51.52	42.22
	PyramidKV	26.37	43.09	46.57	39.07	18.01	15.15	43.33	84.69	38.40	10.67	84.67	49.45	46.90	42.03
	SnapKV	**27.46**	41.81	48.62	41.40	19.42	16.19	42.33	83.91	**38.89**	10.67	85.00	52.14	51.39	43.02
	DapQ	26.81	**43.12**	**49.62**	**41.36**	19.89	**16.68**	**47.00**	**84.92**	38.07	**11.00**	85.00	**54.46**	**51.70**	**43.82**
	KV Cache Size = 64														
	H2O	24.33	32.36	44.94	39.06	**17.96**	**15.05**	37.33	82.76	35.27	10.67	**85.00**	**49.44**	45.52	39.98
	PyramidKV	22.36	35.19	43.51	38.08	14.04	11.10	37.33	84.81	35.58	10.67	80.33	44.67	42.22	38.45
	SnapKV	22.90	40.66	45.56	40.28	15.42	12.03	37.67	**85.11**	36.92	10.67	82.00	46.16	45.75	40.09
	DapQ	**25.58**	**42.65**	**49.66**	**41.25**	16.90	14.05	**39.67**	84.16	**35.49**	**11.00**	80.67	49.38	**46.77**	**41.33**
Qwen3-8B	FullKV	36.87	53.67	57.67	44.73	33.39	23.69	71.67	91.79	42.07	11.98	86.67	70.64	59.26	52.60
	KV Cache Size = 256														
	H2O	27.80	44.92	51.37	40.50	22.38	18.26	46.33	90.04	38.98	12.33	86.67	66.11	53.93	46.12
	PyramidKV	30.41	47.81	51.00	41.37	22.36	17.41	61.67	90.96	37.65	13.00	86.33	67.01	50.11	47.47
	SnapKV	**32.40**	49.29	54.38	41.40	23.79	18.99	**63.00**	91.07	38.57	**14.67**	86.67	**67.99**	53.77	48.92
	DapQ	32.14	**50.78**	**54.79**	**44.47**	**24.16**	**19.01**	62.67	**91.15**	**39.61**	14.17	**86.67**	67.20	**53.83**	**49.28**
	KV Cache Size = 128														
	H2O	26.60	41.37	48.10	39.85	20.83	17.27	41.33	90.21	38.16	**11.72**	**86.67**	65.22	52.77	44.22
	PyramidKV	26.22	40.48	48.41	39.46	18.99	15.10	48.33	89.31	36.92	9.67	86.67	60.82	48.78	43.78
	SnapKV	**29.41**	46.43	51.20	41.66	20.37	16.64	51.33	**91.07**	37.37	11.00	86.67	65.36	51.65	46.17
	DapQ	29.10	**47.15**	**53.84**	**43.00**	**21.11**	**17.27**	**54.00**	90.45	**38.50**	11.67	86.00	**66.29**	**52.03**	**46.95**
	KV Cache Size = 64														
	H2O	25.55	38.94	46.66	39.27	**18.55**	**15.23**	39.00	88.13	35.98	9.67	**86.67**	59.48	**48.95**	42.47
	PyramidKV	25.32	40.44	46.61	39.20	16.25	12.93	**44.67**	88.27	34.63	11.33	83.67	59.73	46.48	42.27
	SnapKV	25.09	39.89	46.58	39.38	15.28	12.38	42.67	87.93	35.12	11.33	84.33	57.96	46.49	41.88
	DapQ	**25.78**	**43.17**	**49.84**	**41.28**	17.16	13.95	43.00	**88.97**	**36.00**	**12.67**	83.00	**60.31**	46.90	**43.23**

Table 10: Performance comparison of different methods across various LLMs on LongBenchv2. For DapQ, the pseudo queries are constructed via prefix-suffix concatenation: using the first 8 and last 24 tokens for LLaMA series models, and the first 2 and last 30 tokens for Qwen models. Notably, several compressed methods surpass the FullKV baseline. We attribute this phenomenon to the noise reduction mechanism of cache eviction. By selectively retaining critical tokens, these methods effectively reduce noise and sparsify the context, potentially leading to more focused and effective model reasoning. This effect is particularly pronounced in long-context benchmarks.

		Difficulty		Length			
LLMs	**Methods**	**Easy**	**Hard**	**Short**	**Medium**	**Long**	**Overall**
Llama3-8B Instruct	FullKV	28.65	26.37	32.22	24.19	25.00	27.24
	KV Cache Size = 128						
	H2O	**32.29**	25.40	32.22	26.98	23.15	28.03
	PyramidKV	30.21	24.44	31.67	26.05	19.44	26.64
	SnapKV	30.73	25.72	32.22	26.05	23.15	27.63
	DapQ	30.73	**27.65**	**33.33**	**27.91**	**23.15**	**28.83**
	KV Cache Size = 64						
	H2O	30.73	24.76	28.89	26.51	25.00	27.04
	PyramidKV	27.08	23.47	24.44	27.44	20.37	24.85
	SnapKV	**31.25**	22.51	23.89	26.98	26.85	25.84
	DapQ	30.73	**29.26**	**31.11**	**28.84**	**29.63**	**29.82**
Llama3.1-8B Instruct	FullKV	25.00	28.62	31.67	25.58	23.15	27.24
	KV Cache Size = 128						
	H2O	26.56	29.58	34.44	25.58	24.07	28.43
	PyramidKV	**29.17**	28.94	32.22	27.44	26.85	29.03
	SnapKV	27.08	29.90	33.89	26.05	25.93	28.83
	DapQ	27.08	**30.55**	**34.44**	**27.44**	**24.07**	**29.22**
	KV Cache Size = 64						
	H2O	23.44	27.01	30.56	23.72	21.30	25.65
	PyramidKV	28.12	28.62	33.89	24.19	**27.78**	28.43
	SnapKV	25.00	26.69	29.44	24.65	23.15	26.04
	DapQ	**29.17**	**28.94**	**33.89**	**26.98**	25.00	**29.03**
Qwen2.5-7B Instruct	FullKV	28.65	27.33	30.56	27.44	24.07	27.83
	KV Cache Size = 128						
	H2O	28.65	**27.65**	30.56	27.91	**24.07**	28.03
	PyramidKV	29.17	25.40	29.44	26.51	23.15	26.84
	SnapKV	**29.69**	26.37	30.56	27.91	22.22	27.63
	DapQ	29.17	27.33	**30.56**	**28.37**	23.15	**28.03**
	KV Cache Size = 64						
	H2O	28.65	26.37	31.11	26.51	22.22	27.24
	PyramidKV	**31.25**	27.97	32.22	28.37	25.93	29.22
	SnapKV	30.73	27.33	**32.78**	27.44	24.07	28.63
	DapQ	30.73	**28.30**	31.67	**28.37**	**26.85**	**29.22**
Qwen3-8B	FullKV	31.25	28.30	33.33	25.58	30.56	29.42
	KV Cache Size = 128						
	H2O	34.38	27.97	35.56	25.58	**31.48**	30.42
	PyramidKV	34.38	27.33	**36.11**	26.05	27.78	30.02
	SnapKV	**34.90**	27.33	34.44	26.05	31.48	30.22
	DapQ	33.85	**28.30**	33.89	**27.44**	30.56	**30.42**
	KV Cache Size = 64						
	H2O	35.42	27.65	34.44	28.84	27.78	30.62
	PyramidKV	**38.02**	26.69	34.44	28.37	30.56	31.01
	SnapKV	36.46	27.33	33.33	**29.30**	29.63	30.82
	DapQ	35.94	**28.62**	**36.67**	26.51	**32.41**	**31.41**

Table 11: Performance comparison of different methods across various kv cache size on Ruler for llama3-8B-Instruct.

LLM	Methods	Single NIAH			Multi-key NIAH								
		S-NIAH-1	S-NIAH-2	S-NIAH-3	MK-NIAH-1	MK-NIAH-2	MK-NIAH-3	MQ-NIAH	MV-NIAH	CWE	FWE	VT	AVG
Llama3-8B-Instruct	FullKV	100	100	100	99.2	91.8	95.8	99.75	97.5	97.82	82.93	98.32	96.65
	KV Cache Size = 2048												
	LaCache	21	27.4	1.8	29.6	29	3.2	12.2	6.45	86.08	86.07	11.04	28.53
	SLM	26.6	25.4	25.4	22.6	24.2	17.8	24.1	24.3	4.9	78.6	21.16	26.82
	H2O	100	90	14	78	47	12.6	74.7	45	90.04	79.33	97.68	66.21
	PyramidKV	100	100	40.8	99	72.2	17.4	99	96.3	83.25	67.87	98.2	79.46
	SnapKV	100	98.4	61.4	99	69	19.8	98.8	94.45	90.34	73.13	97.56	81.99
	DapQ	100	99	96	99.2	67.2	24.4	99.05	95.45	90.68	75	97.64	**85.78**
	KV Cache Size = 1024												
	LaCache	0.2	4.8	2.4	4.2	4	0	2.3	2.45	55.02	86.87	1.56	14.89
	SLM	13.4	13.8	11.4	11.4	13.2	10.8	10.95	11.3	0.38	75.07	8.16	16.35
	H2O	98.2	75.8	8	62	38.8	4.8	49.05	10.6	63.4	75.07	92.2	52.54
	PyramidKV	100	98.2	6.2	98.6	47.8	2	97	90.8	56.14	61.53	97.08	68.67
	SnapKV	100	96.8	15.4	98.4	44.8	3.4	96.45	87.9	65.42	64.27	96.48	69.94
	DapQ	100	98.4	85.8	99.2	41.8	6.2	97.4	92.5	65.74	69.33	96.76	**77.56**
	KV Cache Size = 512												
	LaCache	0	0.2	0	2.6	0.4	0	0.05	1.4	16.52	78.8	0.08	9.10
	SLM	3.6	6.2	5.6	6.6	6.2	5.4	6.25	6.75	0.18	75.33	1.08	11.20
	H2O	88.4	63.8	2.4	45	25.4	1.4	24.4	2.85	46.56	64.6	69.64	39.50
	PyramidKV	100	95.6	0	97.4	35	0.2	91.8	73.5	23.56	52.8	92.96	60.26
	SnapKV	100	95.6	1.4	96.8	30.4	0.4	91.1	71.65	30.82	53.73	94.48	60.58
	DapQ	100	97.8	59.6	98.6	29.8	1	91.95	82.9	27.16	61.27	95.2	**67.75**
	KV Cache Size = 256												
	LaCache	0	0	0	0	0	0	0	0	3.64	58.2	0	5.62
	SLM	1.2	1.2	1.2	2	3.4	2.4	2.3	2.45	0.14	78.6	0	8.63
	H2O	67.2	56.8	2.4	25.4	15.6	0	9.05	1.25	31.1	48.2	20.64	25.24
	PyramidKV	100	94.8	0	89.6	29.4	0	73.5	35.15	9.42	39.8	75.8	49.77
	SnapKV	100	95	0	90.2	26	0	76.5	36.3	13.66	45.2	91.56	52.22
	DapQ	100	97.6	23	97.8	19.8	0	79.7	55	12.8	51.87	88.04	**56.87**
	KV Cache Size = 128												
	LaCache	0	0	0	0	0	0	0	0	0.58	8.4	0	0.82
	SLM	0.6	1.2	1.2	2	2	0	2.25	2.45	0.2	44.93	0	5.17
	H2O	41.4	38.8	2.4	14.8	2.8	0	2.2	0.3	18.06	13	7.88	12.88
	PyramidKV	99.2	91	0	68.2	33.2	0	26.5	9.7	2.38	25.6	18.76	34.05
	SnapKV	98.8	89	0	60.2	35.4	0	17.25	7.45	5.18	28.53	13.24	32.28
	DapQ	99.6	97.6	1.4	94.4	21.4	0	28.85	20.05	4.32	30.33	6.84	**36.80**
	KV Cache Size = 64												
	LaCache	0	0	0	0	0	0	0	0	0.18	0.67	0	0.08
	SLM	0	0	0	0	0	0	0	0	0.06	27.53	0	2.51
	H2O	22	21.2	0	3.8	0.2	0	0.4	0.25	5.7	0.07	3.52	5.19
	PyramidKV	48.8	47.2	0	13.4	7.2	0	0.4	0.25	0.08	0	0.6	10.72
	SnapKV	58.8	65.4	0	20.4	14	0	0.75	0.4	0.14	0.07	2.28	14.75
	DapQ	85.2	87.4	0	26.2	19.6	0	3.65	1.35	0.78	0.33	3.6	**20.74**

Table 12: Performance comparison of different methods across various kv cache size on Ruler for Qwen2.5-7B-Instruct.

LLM	Methods	Single NIAH			Multi-key NIAH								
		S-NIAH-1	S-NIAH-2	S-NIAH-3	MK-NIAH-1	MK-NIAH-2	MK-NIAH-3	MQ-NIAH	MV-NIAH	CWE	FWE	VT	AVG
Qwen2.5-7B-Instruct	FullKV	100	99.8	99.8	99.8	98	93.2	99.8	93.9	77.38	87.67	95.36	94.97
	KV Cache Size = 4096												
	LaCache	3	1.8	2.4	4	4.8	3	3.3	2.25	62.78	87.73	6.32	16.49
	SLM	26.4	28	27	24.4	19.6	11.6	26.1	27.1	36.96	91.53	22.84	31.05
	H2O	100	98.4	24	96.8	19.4	9.4	85.75	68.15	67.56	92.33	93.92	68.70
	PyramidKV	100	99.4	41.8	99.4	19.8	3.6	91.6	83.4	47.66	91.07	93.96	70.15
	SnapKV	100	99.8	86	99.8	39.6	13	97.15	88.6	67.72	91.2	93.64	79.68
	DapQ	100	99.2	82	99	56.8	23.6	97.25	82.15	68.04	92.53	94.52	**81.37**
	KV Cache Size = 2048												
	LaCache	0.2	1.6	0	2.4	0.4	0	0	1.2	36.74	87.33	0.68	11.87
	SLM	13.8	13.4	11	11.4	9	6.2	10.8	11.3	5.56	94.67	9.6	17.88
	H2O	98	90.8	7.8	90.2	7.2	3.8	65.5	35	55.78	93.07	89.72	57.90
	PyramidKV	99.4	97.2	9.2	97.8	11	0.8	75.9	55	28.7	94.73	96.04	60.52
	SnapKV	100	99.2	47.6	98.8	25	2.4	92.25	79.85	55.72	95.33	95.6	71.98
	DapQ	100	96.4	53	97.2	43.2	7	93	69.4	56.44	95.6	94.44	**73.24**
	KV Cache Size = 1024												
	LaCache	0	1.6	2.4	3.2	0	0	1.8	2.35	13.56	84.47	0	9.94
	SLM	4.4	6.2	5.6	6.6	4	4.2	6.2	6.7	0.26	96.93	3.16	13.11
	H2O	96.4	73.8	2.4	78.2	2.8	1.6	38.2	11.15	42.12	87.53	71.16	45.94
	PyramidKV	99	91	0.6	91.6	3.6	0	48.25	25.1	12.08	95.2	93.32	50.89
	SnapKV	99.4	98.4	14.4	97.8	13.8	0.8	78.1	58.7	37.94	96.4	92.48	62.57
	DapQ	99.8	91.8	18.4	94.2	28.4	2.4	82.15	50.8	37.34	97	93.64	**63.27**
	KV Cache Size = 512												
	LaCache	0	0	0	0	0	0	0	0	5	71.47	0	6.95
	SLM	1.6	1.2	1.2	2	2.2	2.6	2.3	2.4	0.26	98.87	0.36	10.45
	H2O	91	53.4	2.4	52.4	0.6	0.8	14	3.75	25.64	64.87	39.92	31.71
	PyramidKV	96.8	74.8	0	62.6	1	0	15.6	8.6	2.52	82.47	59.84	36.75
	SnapKV	99	89.6	1	93.8	5	0.2	56.8	29.55	22.1	93.33	92.12	52.95
	DapQ	99.6	82.8	3.4	87.4	16.6	0.4	62.4	29.75	21.9	95.2	87.52	**53.36**
	KV Cache Size = 256												
	LaCache	0	0	0	0	0	0	0	0	1.58	21.27	0	2.08
	SLM	0.6	1.2	1.2	3.6	0.6	0	2.3	2.4	0.22	98.07	0	10.02
	H2O	63.8	22.4	2.4	13.6	0.6	0	4.25	1.5	12.46	31.13	8.76	14.63
	PyramidKV	67.4	37.6	0	16.6	0.8	0	0.75	0.6	0.36	60.67	23.96	18.98
	SnapKV	97.6	73.4	0	69.6	2	0	18.95	5.8	10.52	81.6	54.84	37.66
	DapQ	98.8	63.2	0.2	71.4	10.8	0	21.4	6.35	10.7	83.73	56.96	**38.5**
	KV Cache Size = 128												
	LaCache	0	0	0	0	0	0	0	0	0.6	2.67	0	0.30
	SLM	0.4	1.4	0	2	0.2	0	2.3	2.4	0.34	96.13	0	9.56
	H2O	6	3.4	0	3.6	0.2	0	0.05	2.05	7.26	3.4	0.48	2.40
	PyramidKV	4.6	6	0	3.2	0	0	0	0	0.26	21.27	2.48	3.44
	SnapKV	57.2	30.8	0	16.8	0.6	0	0.4	0.3	1.02	39.27	5.64	13.82
	DapQ	58	31.4	0	39.2	3.2	0	0.6	0.9	1.46	34.73	9.96	**16.31**
	KV Cache Size = 64												
	LaCache	0	0	0	0	0	0	0	0	0.7	0.67	0	0.12
	SLM	0	0	0	0	0	0	0	0	0.38	0	0.04	0.04
	H2O	0.2	0	0	0	0	0	0	0	0.98	0	0.04	0.11
	PyramidKV	0	0	0	0	0	0	0	0	0.2	0	0	0.02
	SnapKV	0	0	0	0.4	0	0	0	0	0.28	0.07	0.44	0.11
	DapQ	5.4	2.6	0	3.2	0.4	0	0	0	0.28	2.2	1.16	**1.39**

Table 13: Performance comparison of different methods across various kv cache size on Ruler for Qwen3-8B.

LLM	Methods	Single NIAH			Multi-key NIAH								
		S-NIAH-1	S-NIAH-2	S-NIAH-3	MK-NIAH-1	MK-NIAH-2	MK-NIAH-3	MQ-NIAH	MV-NIAH	CWE	FWE	VT	AVG
Qwen3-8B	FullKV	100	100	100	99.6	99.6	99.6	99.9	99.75	83.98	90.67	100	97.55
	KV Cache Size = 4096												
	LaCache	17.6	14.6	13.2	16.8	20.2	7.4	14.75	6.2	62.46	68.47	12.48	23.11
	SLM	26.6	28	27	24.4	19.6	18.4	26.15	27.1	36.62	93	22.96	31.80
	H2O	100	99.8	22	99.8	84.8	32.4	99.5	89.9	46.74	93	99.92	78.90
	PyramidKV	100	100	24.8	99.6	91	51.2	99.9	99.55	57.4	92.87	100	83.30
	SnapKV	100	100	75.2	99.8	96	58	99.9	99.75	72.7	93.27	100	90.42
	DapQ	100	100	96.4	99.8	93	58.4	99.9	99.85	71.82	93.6	100	**92.07**
	KV Cache Size = 2048												
	LaCache	2	1.6	2.4	3.4	1.2	0	2.45	0.9	43.86	69.4	2.04	11.75
	SLM	13.8	13.4	11	11.4	9.4	9	10.85	11.3	10.38	95.47	9.72	18.70
	H2O	100	99.4	7.8	97	64.8	10.6	94.7	48.3	28.48	94.53	99.68	67.75
	PyramidKV	100	100	1.6	100	79.4	19	99.9	93.6	35.46	95.6	100	74.96
	SnapKV	100	100	20	100	91.6	30.4	99.85	97.75	49.59	96	100	80.47
	DapQ	100	100	55	100	88.05	31	99.95	95.15	45.76	96.07	100	**82.82**
	KV Cache Size = 1024												
	LaCache	0.2	1.6	2.4	3.2	0.4	0	2.45	1.05	15.86	62.07	0.32	8.14
	SLM	4.6	6.2	5.6	6.6	3.8	5	6.25	6.7	0.8	96.47	3.24	13.21
	H2O	97.8	92	2.4	81.6	38.2	2.4	72	15.85	23.98	96.2	93.92	56.03
	PyramidKV	100	99.8	0	98.6	61.8	2.4	97.75	70.2	17.84	92.47	99.12	67.27
	SnapKV	100	99.6	1	99.6	79.2	13	99.25	83.05	26.52	96.27	98.84	72.39
	DapQ	100	99.8	14.2	99.6	75.4	14.8	99.75	78.25	23.78	97.73	99.12	**72.95**
	KV Cache Size = 512												
	LaCache	0	1.6	0	3	0	0	0	0.8	7.06	26.07	0.08	3.51
	SLM	1.8	4	1.2	3.8	2.4	3	3.75	4.4	0.76	97.6	0.56	11.21
	H2O	90.4	66.2	2.4	57.2	15	0.8	30.6	5.1	14.8	89.2	65.56	39.75
	PyramidKV	99	97.2	0	84.4	40.8	0.2	70.7	30	6.2	70.27	70.92	51.79
	SnapKV	99.6	99.2	0	97.2	60.6	2	94.35	47.25	14.78	93.6	98.64	64.29
	DapQ	100	99.6	5.8	95.4	68.6	5.2	94.85	42.45	12.38	96.27	96.08	**65.15**
	KV Cache Size = 256												
	LaCache	0	0	0	0	0	0	0	0	1.7	7.93	0	0.88
	SLM	0.8	1.2	1.2	2	2.4	0	2.3	2.4	0.72	95	0	9.82
	H2O	68	21.4	2.4	19	2.6	0	6.15	1.4	11.2	61.27	19.2	19.33
	PyramidKV	94.8	60.4	0	41.6	14.4	0	5.8	5.6	1.22	34.07	24	25.63
	SnapKV	97.4	87.8	0	81.6	35.2	0	47.2	13	9.7	84.47	72.32	48.06
	DapQ	100	88.8	0	75.4	59.4	0.2	47.55	10.2	6.1	88	58.04	**48.52**
	KV Cache Size = 128												
	LaCache	0	0	0	0	0	0	0	0	0.46	0.2	0	0.06
	SLM	0.6	1.2	1.2	2	0.2	0	2.3	2.4	0.64	96	0	9.69
	H2O	19.8	2.4	1.6	3.6	0.2	0	0.05	0.25	8.4	22.53	3.44	5.66
	PyramidKV	11.4	2.8	0	0	1.2	0	0	0	0.84	4.6	3.68	2.23
	SnapKV	68.8	31.2	0	2.6	3.2	0	0.15	0.45	2.06	40.33	5.24	14.00
	DapQ	97.8	34	0	7	20.8	0	0.2	0.1	1.1	40.53	5.47	**18.82**
	KV Cache Size = 64												
	LaCache	0	0	0	0	0	0	0	0	0.72	0	0	0.07
	SLM	0	0	0	0	0	0	0	0	0.4	32.33	0	**2.98**
	H2O	0	0	0	0	0	0	0	0	3.42	0	1.24	0.42
	PyramidKV	0	0	0	0.2	0	0	0	0	0.78	0.4	0.56	0.18
	SnapKV	0	0	0	0	0	0	0	0	0.8	0.07	0.88	0.16
	DapQ	0.8	0.2	0	0	1.2	0	0	0	1	2.4	1.68	0.66

Table 14: Performance comparison of different methods across various LLMs on sub-task categories of the HELMET benchmark.

		ICL					LONGQA		RAG				
		exact_match					*f1*	*rougeL_f1*	*substring_exact_match*				
	Methods	**icl_banking77**	**icl_clinic150**	**icl_nlu**	**icl_trec_coarse**	**icl_trec_fine**	**narrativeqa**	**infbench_qa_eng**	**kilt_hotpotqa**	**kilt_nq**	**kilt_popqa_3**	**kilt_triviaqa**	**Avg.**
Llama3-8B-Instruct	FullKV	38.60	73.60	76.20	39.40	25.00	12.00	16.56	52.00	42.17	47.67	80.00	45.75
	KV Cache Size = 1024												
	H2O	**32.00**	64.00	70.60	36.00	**21.20**	11.30	15.28	47.67	44.17	47.67	81.50	42.85
	PyramidKV	24.80	55.40	68.60	29.60	16.00	11.61	16.00	52.33	44.17	48.17	81.33	40.73
	SnapKV	23.00	54.40	70.60	29.00	18.20	11.04	17.07	**52.67**	43.83	48.33	81.67	40.89
	DapQ	26.80	**64.40**	**72.60**	**39.80**	15.40	**11.81**	**16.67**	50.00	**45.00**	**48.33**	**81.83**	**42.97**
	KV Cache Size = 512												
	H2O	18.00	51.80	63.00	28.80	18.00	10.65	14.45	48.33	44.17	48.00	82.00	38.84
	PyramidKV	17.40	39.80	56.00	22.20	12.20	**11.69**	15.87	50.67	44.17	48.00	82.17	36.38
	SnapKV	17.60	41.40	63.60	22.20	14.20	11.34	16.67	**50.67**	44.50	47.17	82.83	37.47
	DapQ	**21.60**	**52.80**	**64.00**	**39.20**	**15.40**	11.18	**16.89**	49.00	**46.00**	**48.17**	**82.17**	**40.58**
	KV Cache Size = 256												
	H2O	11.60	35.40	**54.20**	23.40	11.20	11.21	12.82	46.67	42.33	47.00	80.67	34.23
	PyramidKV	14.00	27.40	36.20	17.00	10.00	11.36	14.94	**52.00**	42.83	48.00	81.17	32.26
	SnapKV	16.80	27.80	47.60	17.80	9.00	11.52	14.76	50.67	42.67	46.83	81.67	33.37
	DapQ	**17.00**	**37.40**	47.00	**38.40**	**13.80**	**11.66**	**15.70**	48.67	**44.00**	**48.17**	**81.83**	**36.69**
	KV Cache Size = 128												
	H2O	6.80	20.00	**32.80**	18.00	6.20	10.76	12.52	45.33	41.00	46.00	81.33	29.16
	PyramidKV	11.20	21.80	19.60	14.80	8.20	10.70	14.03	48.00	39.33	46.50	83.50	28.88
	SnapKV	13.80	19.60	21.40	14.60	8.80	10.92	13.66	46.00	38.83	47.07	84.33	29.00
	DapQ	**16.40**	**23.40**	25.60	**31.00**	**13.20**	**10.99**	**15.14**	**49.33**	**41.50**	**47.33**	**84.67**	**32.60**
Qwen2.5-7B-Instruct	FullKV	74.00	71.00	53.80	75.60	31.80	20.53	30.33	56.00	49.83	57.67	86.67	55.20
	KV Cache Size = 2048												
	H2O	63.20	54.00	51.00	**79.40**	31.40	**20.37**	28.74	52.00	**49.33**	58.17	87.00	52.24
	PyramidKV	68.40	61.40	53.20	77.40	33.20	19.46	28.52	53.67	48.00	**60.83**	85.33	53.58
	SnapKV	68.40	62.00	51.40	78.00	33.20	20.35	29.41	55.67	47.83	58.33	86.67	53.75
	DapQ	**71.00**	**67.80**	**55.00**	76.00	**33.60**	19.05	**29.46**	**56.33**	48.83	58.50	**87.00**	**54.78**
	KV Cache Size = 1024												
	H2O	44.60	33.20	28.00	**79.40**	25.40	19.52	27.24	50.67	**48.50**	57.00	85.50	45.37
	PyramidKV	58.40	40.60	39.00	78.20	29.20	20.30	28.11	50.67	44.50	**61.00**	84.50	48.59
	SnapKV	56.80	42.10	38.00	75.00	30.80	**20.53**	28.05	56.00	48.00	58.33	85.37	49.00
	DapQ	**66.00**	**56.40**	**49.80**	76.20	**31.20**	20.20	**28.76**	**54.33**	46.67	57.50	**85.67**	**52.07**
	KV Cache Size = 512												
	H2O	28.60	19.40	17.00	**76.20**	17.80	18.51	26.88	51.27	45.33	57.67	85.33	40.36
	PyramidKV	45.00	20.00	22.80	70.60	25.80	**21.91**	26.60	48.67	43.50	**59.67**	82.83	42.49
	SnapKV	44.60	23.60	22.20	71.80	26.80	20.17	27.85	53.00	**47.17**	58.17	85.83	43.74
	DapQ	**52.00**	**43.20**	**40.80**	72.20	**28.40**	20.54	**29.50**	**54.00**	45.33	57.33	**85.83**	**48.10**
	KV Cache Size = 256												
	H2O	21.20	12.00	9.20	**74.40**	14.40	18.26	26.12	48.67	43.00	58.67	83.33	37.20
	PyramidKV	34.80	14.00	13.20	50.80	15.00	17.36	25.93	45.00	40.00	59.17	75.17	35.49
	SnapKV	38.40	15.20	18.80	60.00	19.80	**19.90**	26.55	51.00	**45.33**	**59.83**	82.17	39.73
	DapQ	**41.40**	**25.40**	**27.60**	67.40	**21.00**	17.64	**26.55**	**51.33**	47.00	59.00	**85.17**	**42.68**

Table 15: Performance comparison of different methods across various LLMs on Needle-in-a-Haystack.

LLM	KV Cache Size	Method	Acc
Llama3-8B-Instruct		**FullKV**	**100.00**
	256	H2O	66.81
		PyramidKV	93.94
		SnapKV	90.97
		DapQ	**99.46**
	128	H2O	50.92
		PyramidKV	79.67
		SnapKV	74.67
		DapQ	**95.75**
	64	H2O	42.37
		PyramidKV	55.45
		SnapKV	61.70
		DapQ	**68.34**
Llama3.1-8B-Instruct		**FullKV**	**98.02**
	256	H2O	61.18
		PyramidKV	78.30
		SnapKV	74.84
		DapQ	**84.70**
	128	H2O	47.61
		PyramidKV	65.23
		SnapKV	61.45
		DapQ	**70.34**
	64	H2O	40.36
		PyramidKV	52.25
		SnapKV	56.50
		DapQ	**62.20**
Qwen2.5-7B-Instruct		**FullKV**	**94.23**
	256	H2O	75.64
		PyramidKV	83.80
		SnapKV	84.30
		DapQ	**85.11**
	128	H2O	70.45
		PyramidKV	74.80
		SnapKV	73.64
		DapQ	**76.25**
	64	H2O	63.70
		PyramidKV	56.11
		SnapKV	72.84
		DapQ	**75.75**
Qwen3-8B		**FullKV**	**96.52**
	256	H2O	74.55
		PyramidKV	88.50
		SnapKV	90.41
		DapQ	**91.73**
	128	H2O	67.50
		PyramidKV	72.36
		SnapKV	75.39
		DapQ	**77.89**
	64	H2O	**62.70**
		PyramidKV	61.32
		SnapKV	59.73
		DapQ	61.98

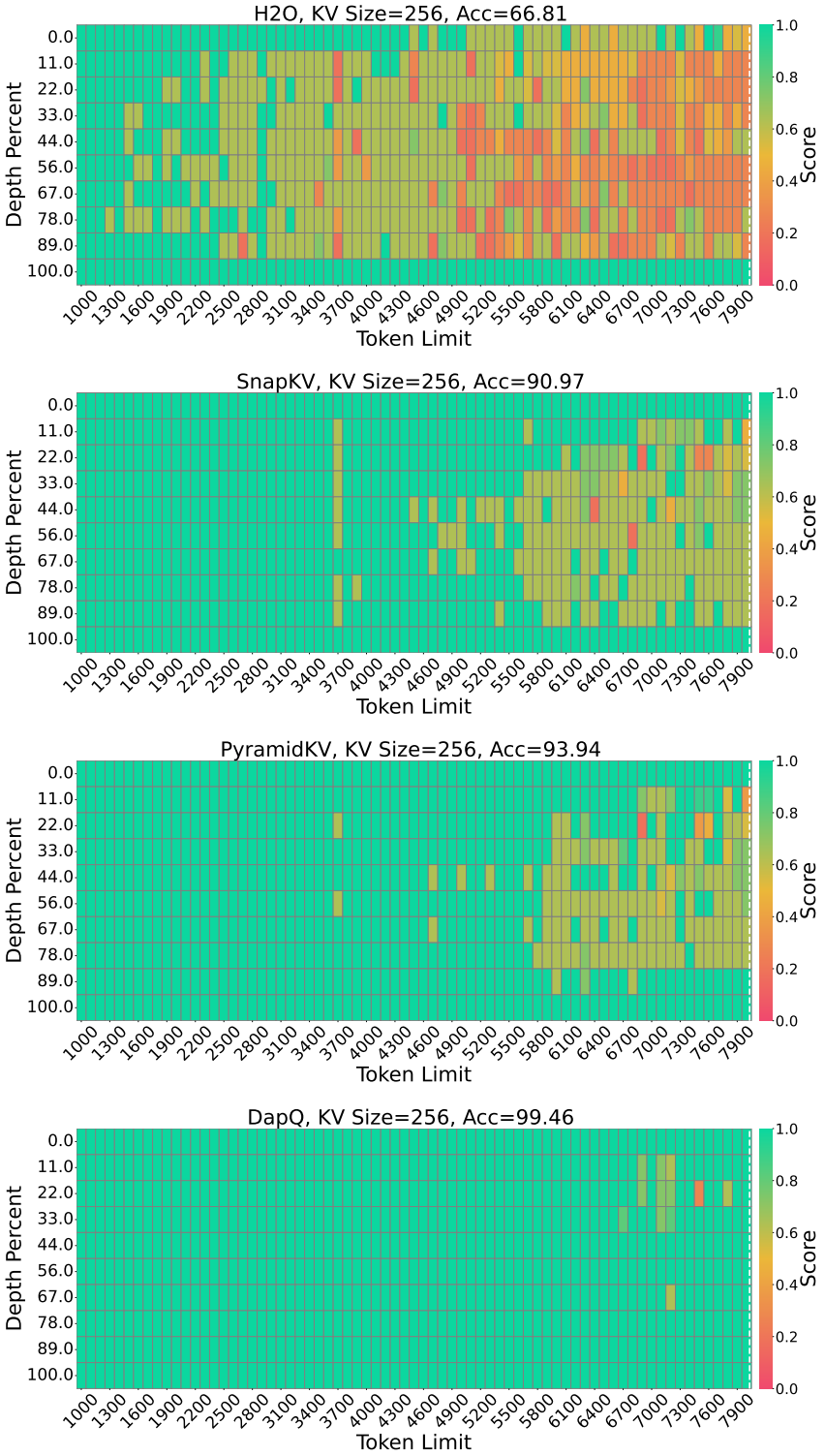


Figure 5: Visualization of Needle-in-a-Haystack results. We take LLaMA-3-8B-Instruct (8k context, 256 KV size) as a representative example to demonstrate the performance differences. The vertical axis represents the depth percentage, and the horizontal axis represents the token length.