AlayaDB: The Data Foundation for Efficient and Effective Long-context LLM Inference

Yangshen Deng*
AlayaDB AI
Shenzhen, China
yangshen.deng@alayadb.ai

Zhengxin You*
SUSTech
AlayaDB AI
Shenzhen, China
zhengxin.you@alayadb.ai

Long Xiang*
SUSTech
AlayaDB AI
Shenzhen, China
long.xiang@alayadb.ai

Qilong Li
AlayaDB AI
SUSTech
Shenzhen, China
qilong.li@alayadb.ai

Peiqi Yuan
AlayaDB AI
SUSTech
Shenzhen, China
peiqi.yuan@alayadb.ai

Zhaoyang Hong
AlayaDB AI
SUSTech
Shenzhen, China
zhaoyang.hong@alayadb.ai

Yitao Zheng
AlayaDB AI
SUSTech
Shenzhen, China
yitao.zheng@alayadb.ai

Wanting Li
AlayaDB AI
SUSTech
Shenzhen, China
wanting.li@alayadb.ai

Runzhong Li
AlayaDB AI
SUSTech
Shenzhen, China
runzhong.li@alayadb.ai

Haotian Liu
AlayaDB AI
SUSTech
Shenzhen, China
haotian.liu@alayadb.ai

Kyriakos Mouratidis
Singapore Management University
Singapore, Singapore
kyriakos@smu.edu.sg

Man Lung Yiu
The Hong Kong Polytechnic University
Hong Kong, China
csmlyiu@comp.polyu.edu.hk

Huan Li
Zhejiang University
Hangzhou, China
lihuan.cs@zju.edu.cn

Qiaomu Shen
Beijing Institute of Technology, Zhuhai
Zhuhai, China
shenqm@sustech.edu.cn

Rui Mao†
Shenzhen University
Shenzhen, China
mao@szu.edu.cn

Bo Tang†
SUSTech
AlayaDB AI
Shenzhen, China
tangb3@sustech.edu.cn

Abstract

AlayaDB is a cutting-edge vector database system natively architected for efficient and effective long-context inference for Large Language Models (LLMs) at AlayaDB AI. Specifically, it decouples the KV cache and attention computation from the LLM inference systems, and encapsulates them into a novel vector database system. For the Model as a Service providers (MaaS), AlayaDB consumes fewer hardware resources and offers higher generation quality for various workloads with different kinds of Service Level Objectives (SLOs), when compared with the existing alternative solutions (e.g., KV cache disaggregation, retrieval-based sparse attention). The crux of AlayaDB is that it abstracts the attention computation and cache management for LLM inference into a query processing procedure, and optimizes the performance via a native query optimizer. In this work, we demonstrate the effectiveness of AlayaDB via (i) two use cases from our industry partners, and (ii) extensive experimental results on LLM inference benchmarks.

SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724428

CCS Concepts

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

Keywords

Vector database, Large language model, Machine learning systems

ACM Reference Format:
Yangshen Deng, Zhengxin You, Long Xiang, Qilong Li, Peiqi Yuan, Zhaoyang Hong, Yitao Zheng, Wanting Li, Runzhong Li, Haotian Liu, Kyriakos Mouratidis, Man Lung Yiu, Huan Li, Qiaomu Shen, Rui Mao, and Bo Tang. 2025. AlayaDB: The Data Foundation for Efficient and Effective Long-context LLM Inference. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724428

1 Introduction

Large Language Models (LLMs) have been widely used in various real-world applications such as personal assistants [4, 6, 9, 14, 41], search engines [2, 3, 10, 17], code generators [5, 7, 11, 41] and document analyzers [33, 44]. Efficient and effective LLM inference is an open problem in the industry [12, 20, 25, 50], especially for long-context (e.g., millions of tokens) inference. In particular, the performance of LLM inference systems is evaluated by three metrics: (1) inference latency, the end-to-end time cost for user tasks, (2)

* These authors contributed equally to this work.
† Corresponding authors: Prof. Bo Tang and Prof. Rui Mao.

generation quality, the capabilities of LLM in various workloads, and (3) GPU memory consumption, the used hardware resources for the user tasks.

Many solutions have been proposed to optimize these metrics in long-context LLM inference. They can be classified into three categories: (i) coupled architecture; (ii) KV cache disaggregation; and (iii) retrieval-based sparse attention. vLLM [42], SGLang [69] and HuggingFace transformers [61] are the most widely-used LLM inference systems in (i) coupled architecture. LLM model computation and KV cache management are tightly coupled in these systems. These systems achieve high generation quality as they use a full attention mechanism. Mooncake [51] and LMCache [15, 46] are representative LLM inference systems in (ii) KV cache disaggregation. They store the KV cache of contexts in external storage and reuse them among different LLM inference instances. Thus, the inference latency of these systems is improved as it reuses the KV cache and reduces the expensive computations (e.g., inner product and softmax). Recently, retrieval-based sparse attention solutions have been proposed (e.g., InfLLM [63] and RetrievalAttention [45]) to alleviate the large GPU memory consumption of these systems in both (i) and (ii). The core idea behind them is the sparse attention mechanism, i.e., only a subset of critical key and value tokens are selected to perform the attention computation. Unfortunately, existing systems cannot simultaneously optimize the three aforementioned performance metrics, as we will elaborate in Section 3.

At AlayaDB.AI, we designed an LLM-native vector database AlayaDB to overcome the limitations of existing LLM inference systems/solutions and enable efficient and effective long-context inference in LLM era. Specifically, for Model as a Service (MaaS) [38] providers, the SLOs of different kinds of workloads indicate their requirements for the inference latency. Thus, the core challenge of AlayaDB is solving a bi-objective optimization problem, i.e., *meet the SLOs of different workloads by consuming less GPU memory and offering higher generation quality simultaneously.* The core idea of AlayaDB is to decouple both KV cache and attention computation and to encapsulate them into a monolithic vector database. The major benefits of the novel disaggregation level are three-fold.

- **Lightweight LLM Inference System.** The cache management and attention computation can be separated from the LLM inference engine, which lightens its burden.
- **Interface Simplification.** It simplifies the interface between LLM inference engine and KV cache service by only returning the attention result, instead of the KV cache content.
- **Co-optimization Opportunity.** It sheds light on co-optimizing attention computation and KV cache management in a monolithic vector database together.

At a high level, AlayaDB's role in LLM inference is comparable to the role of traditional databases [21, 30, 40, 47, 53, 57] in web applications. Specifically, the LLM application developers only need to pay attention to the logic of their applications while AlayaDB offers efficient long-context management from their developed LLM applications. This is analogous to web application developers focusing on the logic of their applications and leaving efficient data management to a traditional relational database.

To achieve the above vision, there are three design goals of AlayaDB: (i) ease-to-use, (ii) high generation quality, and (iii) good efficiency. AlayaDB employs a novel system architecture and introduces end-to-end optimizations. Firstly, it provides simple-yet-powerful abstractions and APIs, which are compatible with the software ecosystem of LLMs. Secondly, it handles sparse attention computation as a vector search query. To improve the generation quality and reduce the memory consumption simultaneously, AlayaDB defines a novel query type, i.e., dynamic inner product range query (DIPR), which overcomes the limitations of the traditional top-k query. To accelerate query processing, AlayaDB includes a native query optimizer, which selects the best execution plan for efficient vector search. Last but not least, a suite of optimization techniques (from algorithm-side to index-side, from computation to storage) has been employed in AlayaDB.

Compared to existing LLM inference systems, AlayaDB enjoys low latency, high generation quality, and low resource consumption at the same time from long-context inference. Our experience shows that AlayaDB greatly lowers the cost of hardware resources for handling long contexts and lightens the labor for optimizing the LLM infrastructure. AlayaDB has already been used to support several online LLM services including chatting apps and knowledge-base QA services in our industry partners.

To sum up, the technical contributions of AlayaDB are as follows.

- **Novel Decoupling Level for LLM Inference Systems:** We classify existing LLM inference solutions into three categories and analyze their limitations to handle the challenges of long-context LLM inference. Then, we decouple the KV cache and attention computation from the LLM inference system and encapsulate them into a novel vector database system.
- **Dynamic Vector Search Query for Sparse Attention:** We analyze the internal characteristics of sparse attention in various LLM benchmarks and real-world applications, then propose a novel dynamic vector search query, i.e., Dynamic Inner Product Range (DIPR), to capture the dynamic nature of sparse attention, which overcomes the limitation of traditional top-k query.
- **AlayaDB System Architecture and Implementation:** We architect and implement AlayaDB for efficient and effective long-context inference. It consists of user interface, query processing engine, and vector storage engine. AlayaDB has been used in several LLM applications by our industry partners. To the best of our knowledge, it is the first vector database natively built for LLM inference.
- **Extensive Evaluation:** We conduct in-depth evaluation of AlayaDB. The results show that it is able to reduce resource consumption and offer better generation quality while guaranteeing the SLOs for various LLM workloads.

The remainder of the paper is organized as follows. Section 2 introduces the LLM inference procedure; Sections 3 and 4 present the motivation, design goals and architecture of AlayaDB; Sections 5, 6, and 7 describe AlayaDB's components; Section 8 elaborates two use cases of AlayaDB; Section 9 presents the experimental study results and Section 10 concludes this work.

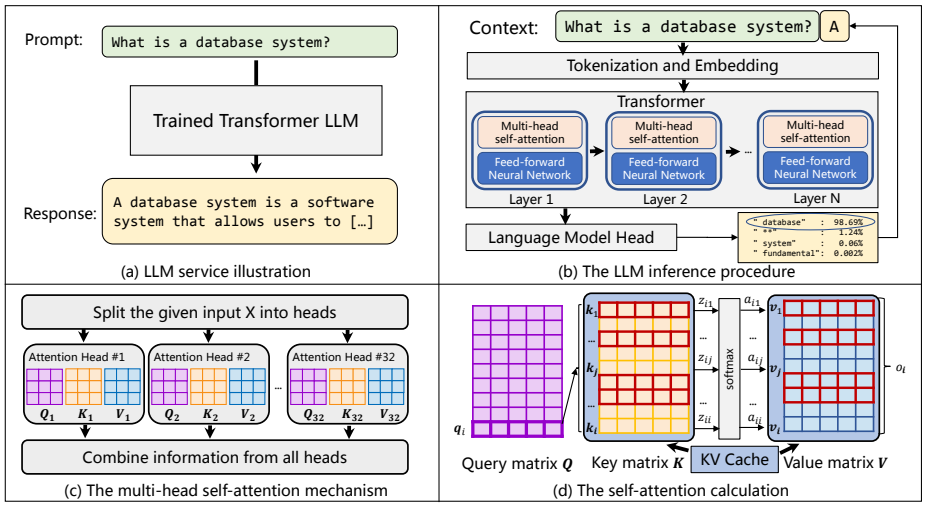


Figure 1: The concepts and illustrations of LLM inference

2 LLM Inference

A large language model (LLM) is a deep neural language model with billions of parameters. The decoder-only transformer is the most prevalent architecture in LLMs, such as GPT [4], Llama [56], and Qwen [41]. Given a well-trained LLM model, the LLM inference generates the text in response to the user input prompt, as shown in Figure 1(a). Actually, it generates the tokens in response text one by one. Each token generation is a forward pass of the language model. The generated token will be appended to the end of the input prompt to form the new context. The context is used to generate the next token by following the same forward pass of the language model. The generation procedure terminates when a special token <eot> (end of text) is generated or the generated text reaches the predefined maximum length.

Figure 1(b) depicts the major components in the LLM inference procedure. For the input context of LLM, it first breaks down the text into small chunks (i.e., tokens), then turns tokens into numeric representations to capture the meaning via the tokenization and embedding modules. The embeddings of the context are the input of the transformer model, which consists of a stack of transformer layers that do all the processing. The output of the transformer is probability scores for what the most likely next token is via the language modeling head. Transformer LLMs include a stack of transformer layers, e.g., Llama 3.1 has 32 layers. Each transformer layer processes its inputs and passes the results to the next layer. For each transformer layer, it has two successive modules: (i) self-attention module, and (ii) feed-forward neural network. The feed-forward neural network emphasizes the important features to make the output more informative.

We next elaborate the core of transformer LLM, i.e., self-attention mechanism, via the illustrated Figures 1(c) and (d). In general, the self-attention mechanism involves two major steps: (i) measuring how relevant each of the previous context tokens is to the current token being processed; and (ii) combining the information from them into a single output vector. A well-trained LLM has three projection matrices, i.e., a query projection matrix $\boldsymbol{W}_Q$, a key projection matrix $\boldsymbol{W}_K$ and a value projection matrix $\boldsymbol{W}_V$, which are used to calculate the attention. In particular, the self-attention mechanism starts by multiplying the input matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, where n is the number of input vectors and d is the dimensionality of the embedding vector of each token, by the projection matrices to create three new matrices, i.e., query matrix $\boldsymbol{Q}$, key matrix $\boldsymbol{K}$ and value matrix $\boldsymbol{V}$, as shown in Figure 1(c). These three matrices are the information of the input tokens in three different spaces, which are used to calculate the attention. In recent transformer LLMs (e.g., Llama 3.1), multi-query and multi-head self-attention mechanisms are employed to improve the scalability of larger models. For simplicity, we utilize one self-attention head for illustration in Figure 1(d) as every head of multi-head attention has a distinct version of matrices of queries, keys and values, see Figure 1(c).

$$z_{ij} = \frac{\boldsymbol{q}_i \cdot \boldsymbol{k}_j^T}{\sqrt{d}}; \quad a_{ij} = \text{softmax}(z_{ij}); \quad \boldsymbol{o}_i = \sum_{s=1}^{i} a_{is} \cdot \boldsymbol{v}_s \tag{1}$$

As shown in Figure 1(d), to generate the $i+1$-th token t_{i+1}, the self-attention mechanism in each head computes the inner product between the query vector $\boldsymbol{q}_i \in \mathbb{R}^{1 \times d}$ and the key vector of the past tokens $\boldsymbol{k}_j$ where $j \in [1, i]$. The computed product is scaled by $\sqrt{d}$ and normalized via a Softmax function to derive the attention score a_{ij}. These attention scores multiply with the value vectors $\boldsymbol{v}_s$ in value matrix $\boldsymbol{V}$ to compute the output $\boldsymbol{o}_i$, see Equation (1).

LLM Inference Phases. In LLM services, the LLM inference procedure of a prompt can be decomposed into two phases: prefill phase and decode phase. Specifically, in the prefill phase the LLM processes all the input tokens in user prompts and generates the first output token. The service level objective (SLO) of the prefill phase in LLM service is its duration, i.e., *Time-To-First-Token (TTFT)*. In the decode phase the LLM sequentially generates the answers. This phase completes when an end-of-sequence token <eot> is generated or when the context reaches a specified maximum length. The SLO in the decode phase is *Time-Per-Output-Token (TPOT)*.

KV Cache. Recall that the last generated token is appended to the previous context and then input into the LLM for the next token generation. In particular, the new context does another forward pass of the model. Obviously, the performance of the decode phase can be significantly improved by caching the key and value matrices of the previous context (see KV cache in Figure 1(d)) as they do not need

to be recomputed. The KV cache is one of the core components in the self-attention mechanism which is widely used in recent LLMs and offers significant speedup of the decode phase.

Sparse Attention. The attention calculation in Equation (1) is the computationally expensive part of LLM inference. To make matters worse, the key and value matrices consume large GPU memory space. The sparse attention mechanism has been proposed to improve the efficiency of the attention calculation and reduce the GPU memory consumption during the LLM inference. The intuition of sparse attention is that only a small proportion of tokens, not all tokens in previous context, dominates the generation quality/accuracy [45]. For example, only the key vectors and value vectors in KV cache with red rectangles in Figure 1(d) are critical vectors for the high-quality token generation. The computation cost of the attention calculation is significantly reduced as the sparse attention only calculates a fixed size of keys (resp. values) in key matrix $\boldsymbol{K}$ (resp. value matrix $\boldsymbol{V}$), instead of all keys and values in both key and value matrices, see Equation (1). Key vectors with high inner product scores relative to the query vector are considered important, as they have high attention scores, significantly contributing to the final output.

3 Motivation of AlayaDB

The context length of an LLM request becomes very large with the rapid development of LLM applications. For example, users may ask LLM questions about long documents, including understanding academic papers [8], getting legal assistance from law documents [22, 35], or analyzing financial documents [33]. Chat applications [4, 6, 14] utilize the long chat log to produce better responses for users. The AI programming assistants leverage all code in the project to accurately generate code or identify bugs/errors [5, 11].

The long-context LLM inference is extremely expensive as its self-attention mechanism incurs high memory consumption and numerous computation operations. In particular, it requires $O(n)$ memory to store the KV cache, where n is the length of the long context. The compute complexity for the prefill phase is $O(n^2)$ due to the self-attention computation in Equation (1) that applies to each input token. Thus, the TTFT of prefill phase is several minutes to tens of minutes when the context length is quite large. For the decode phase, it needs $O(n)$ for each token generation. In practice, it takes about 141.38 GB memory and 6 minutes to answer a question about the book "*Database System Concepts, 7th Edition*" [52] (495.5K tokens), with a bfloat16 version Llama-3-8B model [13] on 2 NVIDIA A800 GPUs (each has 80 GB memory). To reduce the memory consumption and computation cost of long context LLM inference, several research studies have been proposed to reuse the KV cache of the long context and use them to serve different requests from the users. For example, the users may ask various questions about the same book "*Database System Concepts*". Thus, the KV cache of this book can be reused to answer these different questions. The reused KV cache reduces the latency of TTFT in prefill phase significantly, and it becomes a de facto standard in LLM inference systems. However, the performance of long-context LLM inference still has a lot of room for improvement. We next analyze the existing LLM inference systems/techniques in four dimensions: (i) GPU memory consumption, (ii) inference latency, (iii) generation quality, and (iv) solution usability.

Table 1: LLM inference solutions analysis

Existing solution	GPU memory consumption	Inference latency	Generation quality	Solution usability
①	Large	High	Good	Good
②	Large	Medium	Good	Medium
③	Small	–	Medium	Bad
AlayaDB	**Small**	**Low**	**Good**	**Good**

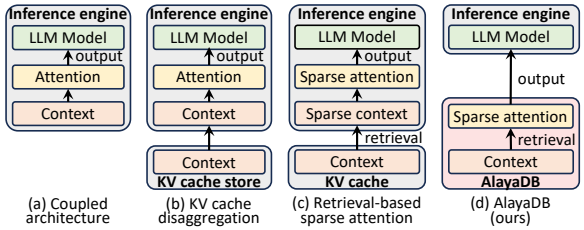


Figure 2: Summary of LLM inference solutions

3.1 Analysis of Existing Solutions

In this section, we classify existing work into three categories: ① coupled architecture, ② KV cache disaggregation, and ③ Retrieval-based sparse attention mechanism. We introduce the core idea of each category and analyze the characteristics of them in detail. Table 1 summarizes the analyzed results of existing solutions.

① **Coupled Architecture.** It is the widely-used LLM inference system architecture in industry, e.g., vLLM [42], SGLang [69], and transformers [61]. The core idea of the coupled architecture is the LLM model computation and KV cache management are tightly coupled and it processes the user request in a holistic manner, as shown in Figure 2(a). It offers good usability with a simple user interface and high generation quality. However, it fails to handle long context. The major reasons are: (i) the large GPU memory consumption for KV cache and (ii) the high TTFT in prefill phase as it reuses the KV cache in a coarse manner, e.g., vLLM employs LRU policy to maintain the KV cache in limited GPU memory.

② **KV Cache Disaggregation.** As depicted in Figure 2(b), several systems decouple the KV cache into a separate storage service and manage it in a stateful way. For example, LMCache [15] and Mooncake [51] store the KV cache of a long context in external cheap storage (e.g., CPU memory, disk or remote memory) after its prefill phase such that the KV cache can be reused by everyone in the future as it only needs to be loaded into the inference engine. The inference latency of the KV cache disaggregation solutions is slightly lower than the coupled architecture as it reduces the TTFT of prefill phase by reusing KV cache better. The generation quality of it is the same as the coupled solution as both employ full attention mechanism. However, it is not easy to use, as it involves a lot of intrusive modifications (i.e., lots of engineering work) to the inference engine. Moreover, the KV cache disaggregation still consumes a large GPU memory during the decoding stage.

③ **Retrieval-based Sparse Attention.** Recently, InfLLM [63] and RetrievalAttention [45] use the sparse attention mechanism to alleviate the high GPU memory consumption of these systems in both ① and ②. In particular, they only retrieve a small subset of keys and values from offloaded KV cache for attention computation, see Figure 2(c). Although these retrieval-based solutions can significantly reduce GPU memory consumption, almost all (if not all) of them are not easy to use as (i) the retrieval algorithm is hard-coded in the underlying specific LLM model and cannot be directly used on

other LLM models and (ii) they lack the ability to manage and reuse the long contexts among different requests and inference engines. Moreover, they trade off between memory consumption/inference latency and generation quality. In particular, the generation quality of these methods is determined by the retrieved critical keys and values. However, it is challenging to retrieve all the critical keys and values efficiently. Existing work assumes that the number of critical vectors is fixed (i.e., k) and then retrieves top-k critical keys and values from the offloaded KV cache. This static method cannot achieve the good generation quality of ① and ②, as we will elaborate in Section 6. Regarding inference latency, the retrieval-based sparse attention methods introduce extra overhead to identify the critical key and value vectors. However, they gains benefits during the attention computation as only the selected critical keys and values will be used. According to our internal experimental evaluation, there is no clear winner between the extra overhead and the reduced attention computation. Thus, we use '—' in the inference latency column of ③, see Table 1.

3.2 Design Goals of AlayaDB

Motivated by the above limitations, we propose a novel architecture for efficient and effective long-context LLM inference by decoupling both the KV cache and sparse attention computation from the LLM inference engine. In particular, we architect a vector database AlayaDB to manage the offloaded KV cache and compute the sparse attention for LLM inference, as illustrated in Figure 2(d). The design goals of AlayaDB are as follows.

G1: Ease-of-use. The first design goal of AlayaDB for long-context LLM inference engine is ease-of-use. Thus, the user interface abstraction of AlayaDB should be simple and compatible with LLM inference engines. With these abstractions, the LLM developers could use AlayaDB easily for efficient and effective inference in their LLM applications, e.g., analogue to how web developers use traditional database systems in their web applications.

G2: High Quality. The second design goal of AlayaDB for long-context LLM inference engines is to provide high generation quality. As mentioned above, the generation quality is determined by the quality of retrieved critical keys and values. Thus, AlayaDB should offer the capability to identify the critical tokens.

G3: Good Efficiency. The third design goal of AlayaDB for long-context LLM inference engine is good efficiency. Specifically, AlayaDB should achieve higher generation quality and lower memory consumption as much as possible for user-specified SLOs.

We are aware that many vector database systems/techniques have been proposed [1, 18, 19, 24, 27, 34, 39, 54, 59, 60, 62], both in academia and industry. However, to the best of our knowledge, none of them are natively designed to support efficient and effective long-context LLM inference. In subsequent, we introduce the architecture and key components of AlayaDB. As the last row in Table 1 shows, AlayaDB incurs small memory consumption, low inference latency, and high generation quality simultaneously.

4 System Overview of AlayaDB

Figure 3 depicts the overview of AlayaDB we built at AlayaDB.AI. It consists of three components: (i) user interface, (ii) query processing engine, and (iii) vector storage engine. We briefly introduce each component in AlayaDB to elaborate on the designs for the aforementioned three design goals.

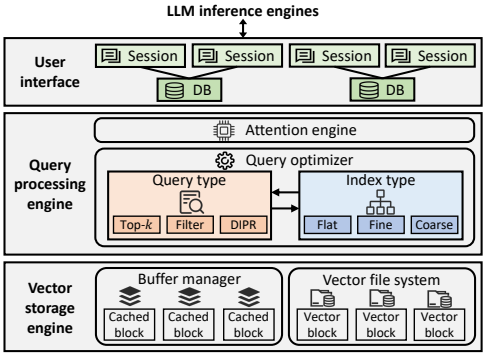


Figure 3: System overview of AlayaDB

User Interface. The top layer of AlayaDB is the user interface component. It abstracts the complex attention computation and KV cache management to offer easy-to-use APIs. Thus, LLM developers can simply leverage efficient and effective long-context LLM inferences by invoking the abstracted APIs in AlayaDB. This is similar to how web developers can build various applications without worrying about the underlying database management system. Specifically, we use two widely-used concepts `DB` and `Session` in database community to abstract the `context` and `request` in the LLM inference procedure. We will introduce the details in Section 5.

Query Processing Engine. The middle layer of AlayaDB is the query processing engine, which is essential to achieve high quality and good efficiency goals. It consists of a native attention engine and a query optimizer. The native attention engine is designed for efficient sparse attention computation in Equation (1). The query optimizer is devised to identify the optimal query processing plan, which efficiently computes the critical tokens. Unlike traditional database query optimizers, the query optimizer in AlayaDB has two major modules: (i) query type module, and (ii) index type module. The query type module includes a set of predefined queries (e.g., top-k) that are used to retrieve the critical tokens from the KV cache. The index type module has a set of indices that can be used to accelerate the predefined queries. It is worth pointing out that both query type and index type in the query optimizer are extensible in AlayaDB. The details of this engine are presented in Section 6.

Vector Storage Engine. To further improve the efficiency (both memory consumption and inference latency), we equipped AlayaDB with the vector storage engine in the bottom layer. It includes a buffer manager and a novel vector file system. A novel vector data layout scheme is designed in the vector file system, which could be used to improve the data access locality during query processing. The buffer manager manages the buffered blocks of KV cache and supports high-performance keys and values retrieval. We will show the optimizations of the vector storage engine in Section 7.3.

5 User Interface

AlayaDB provides simple and flexible abstractions and easy-to-use APIs for users to import context, reuse context and compute sparse attention result for efficient and effective long-context LLM inference. Two core abstractions in AlayaDB are `DB` and `Session`. A `DB` in AlayaDB manages all the contexts, including prompts, KV cache

Table 2: AlayaDB APIs

DB abstraction and provided APIs
`DB.create_session(prompts) -> Session, prompts`
`DB.import(prompts, kv_cache)`
`DB.store(session)`
Session abstraction and provided APIs
`Session.attention(q, layer) -> o`
`Session.update(q, k, v, layer) -> k, v`

and vector indexes, e.g., an analogue of `DB` instance in traditional relational database systems, which include the schema, tables, and data tuples. In a traditional database system, a database session is the connection established between an application server and a database server to enable communication and data retrieval. Inspired by it, in AlayaDB, a `Session` connects the contexts and the running inference requests from a user. AlayaDB provides compatible APIs with HuggingFace transformers [61] and flash-attention [31, 32] library, which are the de facto standards of LLM inference and attention computation. The core APIs provided by AlayaDB are summarized in Table 2. We briefly introduce them as follows.

- `DB.create_session(prompts)` takes a list of prompts as input and returns a `Session` object and the truncated prompts. Given the input prompts, it reuses the longest common prefix with the stored contexts. The reused context is in the `Session` object. The non-reused part of input prompts are the truncated prompts.
- `DB.import(prompts, kv_cache)` imports a list of computed contexts to AlayaDB for further reuse. Thus, its inputs are the prompts and KV cache of these contexts.
- `DB.store(session)` persists all states in a `session` into a reusable context in the database. It takes the `session` with the corresponding prompts and KV cache as the input.
- `Session.attention(q, layer) -> o` generates the attention results of one LLM model layer for the session. It accepts the query vectors and the layer id as the input, and returns computed attention output. This API can be used to replace the flash-attention APIs.
- `Session.update(q, k, v, layer) -> k, v` updates a session with the new inputs or generated tokens for one model layer. This API is compatible with `DynamicCache.update` in huggingface transformers. It provides an option to return the full key and value cache for manual management.

Example. With the above APIs, it is easy for users to import context, reuse context and compute sparse attention score for efficient and effective long-context LLM inferences upon various LLM models. Figure 4 shows an illustration example of AlayaDB with HuggingFace transformers, which only changes few lines of code. In particular, Figure 4(a) is the original code. The `inference` function is the common implementation of using an LLM model (offered by HuggingFace transformers). For a model and a list of prompts, it creates a new `DynamicCache` to manage the KV cache as the `past_key_values`. The prompts and `past_key_value` are inputs of LLM model to generate the next tokens. `LlamaAttention.forward` is the implementation of an attention layer in HuggingFace transformers. It first updates the `past_key_value` with the newly generated key and value matrix, which is now a `DynamicCache` with the full KV cache. Then, it invokes `flash_attn_func` attention operator on the newly generated query matrix and the full KV cache.

```python
from transformers.cache_utils import DynamicCache
from flash_attn import flash_attn_func

def inference(model, prompts):
    past_key_values = DynamicCache()
    output = model(prompts, past_key_values)
    ...
class LlamaAttention:
def forward(self, ...):
    ...
    k, v = past_key_values.update(k, v, self.layer_idx)
    o = flash_attn_func(q, k, v)
    ...
```

(a) Original code using flash-attention and transformers

```python
from AlayaDB.LLM import DB

def inference(model, prompts):
    session, prompts = DB.CreateSession(prompts)
    past_key_values = session
    output = model(prompts, past_key_values)
    ...
class LlamaAttention:
def forward(self, ...):
    ...
    past_key_values.update(q, k, v, self.layer_idx)
    o = past_key_values.attention(q, self.layer_idx)
    ...
```

(b) Modified code using AlayaDB with transformers

Figure 4: Using AlayaDB APIs for LLM inference

Figure 4(b) shows how to use AlayaDB APIs for the above LLM inference procedure. From the application side, users can enjoy the ability to manage and reuse the contexts in AlayaDB by simply replacing `DynamicCache` with `Session`, as the pink-colored lines show. Specifically, it calls `DB.CreateSession` to initialize a session and the truncated prompts for the input prompts. The non-reusable parts (truncated prompts) are input to the LLM model together with the Session for further generation. To further leverage the native attention computation from AlayaDB, users only need to modify the `LlamaAttention.forward` to replace the flash-attention with the `Session.attention`, see the last highlighted line in Figure 4(b).

6 Query Processing in AlayaDB

In this section, we introduce the query processing procedure in AlayaDB. In particular, we propose a novel query type in Section 6.1, which captures the dynamic nature of sparse attention in LLM inference. In Section 6.2, we introduce the query optimizer of AlayaDB.

6.1 Dynamic Inner Product Range Query (DIPR)

6.1.1 Limitations of Top-k Query. In sparse attention, a subset of critical key and value vectors are retrieved to approximately compute the attention output. Thus, the effectiveness of the computed attention output is determined by the number of retrieved critical tokens. The efficiency of LLM inference also depends on the cost to retrieve these critical tokens. Almost all (if not all) existing work [26, 36, 45, 55, 63, 64, 68] utilize top-k query for critical token retrieval. The value of k is a pre-defined hyper-parameter, and it is applied to all attention heads among all layers. It means the top-k query assumes the number of critical tokens is the same among *different tasks* and *different heads*. However, this assumption is probably not true in various LLM applications. We summarize two crucial observations as follows, which contradict this assumption. These observations are summarized from the user experiences of our product's industry customers, and we reproduce them in widely-used LLM benchmarks to follow the DeWitt Clause.

Observation I: the number of critical tokens significantly varies in different *heads*. The transformer-based LLM model

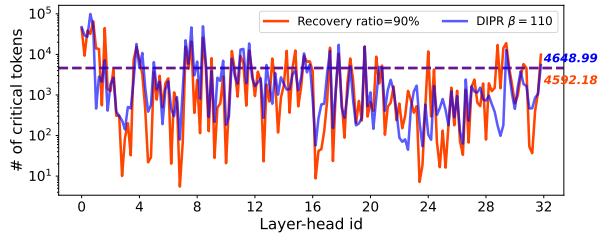


Figure 5: The number of selected tokens in different heads

Table 3: The number k of required tokens in different tasks

Task	k	proportion	Task	k	proportion
Qasper	350	9.67%	LCC	65	5.26%
Passage R.	250	2.69%	HotpotQA	200	2.19%
QMSum	150	1.41%	TriviaQA	20	0.24%

includes multiple layers and every layer has multiple heads. We conduct an experiment with Llama-3-8B-Instruct-262k model [12] on the KV retrieval dataset in ∞-Bench [67] to investigate how many critical tokens are needed to result in a good approximation of the full attention scores. We measure the accuracy of this approximation with the recovery ratio [45], which represents the proportion of the total attention scores accounted for by the attention scores of the selected critical tokens. The red curve in Figure 5 shows the number of tokens needed to achieve a recovery ratio of 90% for each head (randomly sampled five heads per layer), which significantly varies among different heads. For example, it needs on average 42,979.85 tokens in layer 0 head 5, which is much larger than the 53.36 tokens in layer 31 head 5.

Observation II: different *tasks* require different number of critical tokens. We conducted experiments on various tasks in LongBench [23] to explore the number of critical tokens for LLM inference in different tasks. These tasks cover key long-text applications including single-doc QA (Qasper), synthetic tasks (Passage Retrieval), multi-doc QA (HotpotQA), summarization (QMSum), code completion (LCC), and fewshot learning (TriviaQA). Table 3 lists the number of tokens k (resp. its proportion to the context length) required for the top-k query based sparse attention to achieve the same accuracy as full attention in these tasks. Obviously, the number of critical tokens varies widely across tasks, ranging from 20 (0.24%) to 350 (9.67%). For the simple tasks (e.g., TriviaQA), they only need a few tokens as the answer can be obtained from a short paragraph of context. In contrast, complex tasks (e.g., Qasper) require a large amount of tokens to understand the whole context then return correct answers.

Take-away message. The nature of sparse attention is to use a dynamic set of critical tokens to generate high-quality responses (w.r.t. full attention) in different tasks and different heads of the transformer-based LLM models. The traditional top-k query fails to capture the dynamic nature of sparse attention as it uses a fixed and static k, which always results in either low generation quality (i.e., retrieving too few critical tokens) or high computation cost (i.e., retrieving too many critical tokens).

6.1.2 From Attention to DIPR. To overcome the limitation of the traditional top-k query with static and fixed k for different heads and tasks, we propose Dynamic Inner-Product Range query (DIPR) to capture the dynamic nature of sparse attention. In particular, DIPR adaptively determines the number of *critical tokens* in different tasks and heads. We first formally define the critical token considered by DIPR in Definition 1.

Definition 1 (Critical token). *Give the definition of attention score in Equation (1), considering all key vectors in the key matrix $\boldsymbol{K} = [\boldsymbol{k}_1, \cdots, \boldsymbol{k}_n]$, the key $\boldsymbol{k}_j$ is a critical token for query vector $\boldsymbol{q}_i$ if and only if $a_{ij} \geq \alpha \times \max_{s\in[1,n]}(a_{is})$, where α is a proportion threshold and ranges in $[0, 1]$.*

The intuition of DIPR query is finding all tokens which are larger than a given proportion α of the token with maximum inner product as all these tokens are critical. We next transform the critical token in Definition 1 to an inner product-based version in Definition 2. Theorem 1 guarantees the correctness of the definition transformation. Interestingly, Definition 2 means the DIPR query explicitly considers the attention computation in Equation (1).

Definition 2 (Inner Product-based Critical token). *Considering all key vectors in the key matrix $\boldsymbol{K} = [\boldsymbol{k}_1, \cdots, \boldsymbol{k}_n]$, the key $\boldsymbol{k}_j$ is a critical token for query vector $\boldsymbol{q}_i$ if and only if $\boldsymbol{q}_i \cdot \boldsymbol{k}_j^T \geq \max_{s\in[1,n]}(\boldsymbol{q}_i \cdot \boldsymbol{k}_s^T) - \beta$, where $\beta = -\sqrt{d} \times ln(\alpha)$.*

Theorem 1. *The critical token in Definition 1 is equivalent to the inner product-based critical token in Definition 2.*

Proof.

$$a_{ij} \geq \alpha \times \max_{s\in[1,n]}(a_{is}) \Leftrightarrow \frac{\exp(z_{ij})}{\sum_{t=1}^{n}\exp(z_{it})} \geq \alpha \times \max_{s\in[1,n]}\left(\frac{\exp(z_{is})}{\sum_{t=1}^{n}\exp(z_{it})}\right)$$

$$\Leftrightarrow \exp(z_{ij}) \geq \alpha \times \max_{s\in[1,n]}(\exp(z_{is})) \Leftrightarrow z_{ij} \geq \ln(\alpha) + \max_{s\in[1,n]}(z_{is})$$

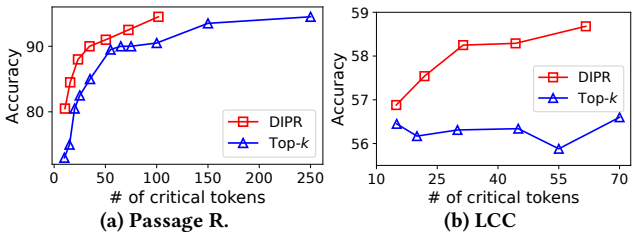
$$\Leftrightarrow \boldsymbol{q}_i \cdot \boldsymbol{k}_j^T \geq \sqrt{d} \times \ln(\alpha) + \max_{s\in[1,n]}(\boldsymbol{q}_i \cdot \boldsymbol{k}_s^T)$$

The proof completes by setting β as $-\sqrt{d} \times ln(\alpha)$. □

Last, we formally define the novel DIPR query in Definition 3.

Definition 3 (Dynamic Inner-Product Range Query, DIPR($\boldsymbol{q}, \beta$)). *Given a key matrix $\boldsymbol{K} = [\boldsymbol{k}_1, \cdots, \boldsymbol{k}_n]$, a query vector $\boldsymbol{q}_i$ and a parameter $\beta \geq 0$, the DIPR query returns a subset $\boldsymbol{cK}$ of $\boldsymbol{K}$, which includes all inner product-based critical tokens.*

The advantages of our novel DIPR query are three-fold: (i) For a given β, different numbers of critical tokens will be retrieved by different tasks and heads in DIPR query. Thus, it explicitly considers the dynamic nature of sparse attention; (ii) the input parameter β of DIPR query directly considers the critical tokens by the attention score of every key, however, the top-k query utilizes the absolute rank of every key's attention score; and (iii) the core computation of DIPR is the inner product $\boldsymbol{q}_i \cdot \boldsymbol{k}_j^T$, which does not introduce extra overhead and the optimizations for inner product-based top-k query can be directly adopted. We demonstrate the effectiveness of our novel DIRP query by the experiments in Figures 5 and 6. In particular, the blue curve in Figure 5 shows the number of retrieved critical tokens of DIPR query by setting β to 110, which is very close to the number of tokens required to achieve 90% recovery ratio. In Figure 6, we present the obtained results by varying β and k in DIPR query and top-k query for Passage R. and LCC tasks, respectively. It confirms the DIPR query achieves higher accuracy with fewer retrieved tokens when compared with top-k query.



(a) Passage R. (b) LCC

Figure 6: The number of critical tokens in different tasks

6.1.3 DIPR Query Processing. The top-k query processing algorithms efficiently return a sized-k set of critical tokens for every query vector by exploiting widely used graph indices on key vectors, e.g., HNSW [48], NSG [37] and RoarGraph [28]. However, they cannot be directly used to process DIPR query as DIPR query returns a variable length of critical tokens w.r.t. the maximum inner product value of the query $\boldsymbol{q}_i$ and key matrix $\boldsymbol{K}$ for different tasks and different heads. In this section, we devise the first approximate DIPR query processing algorithm DIPRS. There are two principles of DIPRS algorithm design: (i) it should explore more points to find the larger inner product value quickly; and (ii) it should reduce non-critical point explorations.

Algorithm 1 shows the pseudocode of the DIPRS algorithm, which follows the above two principles. Specifically, it utilizes the widely used graph-based indices as the fundamental building block as they offer high recall and good efficiency for inner product-based vector similarity search. Given an input β, the number of returned tokens in the critical token set $\boldsymbol{cK}$ is *dynamic* and *unknown* in advance, until the token with maximum inner product value is found. The core ideas of DIPRS algorithm are (1) maintaining an unordered candidate list with variable capacity, and (2) progressively reducing the search space with the best-so-far inner product value. The subroutine tryAppend (Line 10) decides whether the given point $\boldsymbol{k}$ should be appended into candidate list or not.

We next briefly present how Algorithm 1 achieves both above intuitions with the illustration example in Figure 7. To achieve (i), we set a capacity threshold l_0. When the list capacity is lower than l_0, it explores all points without pruning (see Line 13). As shown in Figure 7(a), 2 is added to the list even though it is not critical. For (ii), after reaching the capacity threshold, it does not append the non-critical points to the list to reduce the search space. Figures 7(b) and (c) show that 3 is pruned and 7 is appended w.r.t. the current maximum inner product value, respectively.

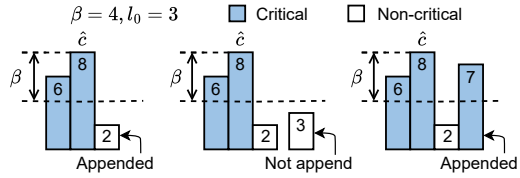
6.2 Query Optimizer in AlayaDB

Except for the traditional top-k query and our novel proposed DIPR query, we believe other auxiliary queries can be defined to achieve sparse attention, i.e., retrieving a subset of critical keys and values for high-quality generation. However, the processing performance of these queries significantly varies among different hardware settings and workload characteristics. Thus, it is crucial to provide a query optimizer in AlayaDB, which assists the LLM application developer in choosing the best query type with its underlying index structure. In AlayaDB, we consider three query types (e.g., top-k, DIPR and filter query) and three index types (e.g., coarse-grained index, fine-grained index, and flat index). Interestingly, both query

Algorithm 1: DIPRS(G, $\boldsymbol{q}$, $\boldsymbol{k}_0$, l_0, β)

Input: Graph G, query $\boldsymbol{q}$, start key $\boldsymbol{k}_0$, capacity threshold l_0, and β
Output: Critical token set $\boldsymbol{cK}$

1 Initialize a list C with start key vector $\boldsymbol{k}_0$
2 $i \leftarrow 0$
3 **while** $i < C$.capacity() **do**
4 $\boldsymbol{c}_i \leftarrow$ the $(i+1)$-th key vector in C
5 $i \leftarrow i + 1$
6 **foreach** unvisited neighbor $\boldsymbol{k}$ of $\boldsymbol{c}_i$ in G **do**
7 tryAppend($\boldsymbol{q}, \boldsymbol{k}, \beta, C, l_0$)
8 $\hat{\boldsymbol{c}} \leftarrow$ the closest point to $\boldsymbol{q}$ in C
9 **return** $\boldsymbol{cK} \leftarrow \{\boldsymbol{c} | \boldsymbol{c} \in C, \boldsymbol{q} \cdot \boldsymbol{c}^T \geq \boldsymbol{q} \cdot \hat{\boldsymbol{c}}^T - \beta\}$
10 **Procedure** tryAppend($\boldsymbol{q}, \boldsymbol{k}, \beta, C, l_0$)**:**
11 $\hat{\boldsymbol{c}} \leftarrow$ the closest point to $\boldsymbol{q}$ in C
12 Mark $\boldsymbol{k}$ as visited
13 **if** C.capacity() $\leq l_0$ or $\boldsymbol{q} \cdot \boldsymbol{k}^T \geq \boldsymbol{q} \cdot \hat{\boldsymbol{c}}^T - \beta$ **then**
14 C.append($\boldsymbol{k}$)



(a) $|C| \leq l_0$ (b) Point pruning (c) $\boldsymbol{q} \cdot \boldsymbol{k}^T \geq \boldsymbol{q} \cdot \hat{\boldsymbol{c}}^T - \beta$

Figure 7: Three cases of tryAppend in DIPRS

and index types can be extended in AlayaDB for efficient and effective sparse attention. We next introduce the core idea of each index type and analyze their characteristics in Table 4.

- **Coarse-grained index**. It groups the adjacent tokens into blocks, where each block is represented by several vectors. It only computes the inner products between query and representative vectors during the retrieval and selects the critical blocks for attention computation. This kind of algorithms includes InfLLM [63], Quest [55] and PQCache [66]. These methods usually require a large GPU memory to cache the blocks and they can provide a very low latency for LLM inference.
- **Fine-grained index**. It builds the traditional vector search indexes on the key-level, e.g., indexing all key vectors by a graph (a.k.a., graph indices). It quickly and accurately locates a small number of critical tokens in the index, which can be efficiently computed on CPU. However, due to the expensive random memory access during index traversal, it can be slow when the number of used critical tokens is large, e.g., k is large in top-k queries.
- **Flat index**. It scans all the keys to find the critical tokens on CPU. Compared to fine-grained indices, it is less efficient when the number of critical tokens are small due to redundant scans. However, it can be more efficient when the number of critical tokens is large due to the sequential memory access.

Inspired by the rule-based query optimizer in database systems, AlayaDB implements a unified and extensible optimizer to select an optimized query plan (including specified query type and index type) for attention computation. The workflow of the rule-based query optimizer in AlayaDB is shown in Figure 8. It identifies the context length at first. Query to the short contexts will be processed directly with full attention. For the long contexts, if the context involves partial reuse, an attribute filtering predicate containing

Table 4: Characteristics of index types

Index type	Supported query type	GPU memory comsumption	Latency small k	Latency large k
Coarse	Top-k, Filter	Large	Low	Low
Fine	Top-k, Filter, DIPR	Small	Low	High
Flat	Top-k, Filter, DIPR	Small	Medium	Medium

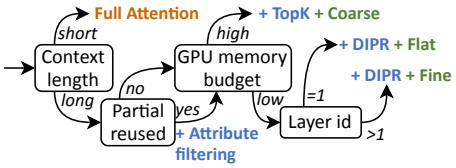


Figure 8: Rule-based query optimizer in AlayaDB

the length of the reused prefix is applied to the query, as we will introduce in Section 7.1. Then the optimizer identifies GPU memory budget, which is set to the available GPU memory by default and can be manually set by users. If the budget is enough, the query will be processed as top-k queries with coarse-grained indices, i.e., InfLLM [63] in AlayaDB. If the GPU memory budget is limited, the optimizer will choose DIPR query and select the index type based on the layer id. From production environments of LLM inference and experimental evaluation benchmarks (see Figure 5), we observed that the first layer requires a large number of tokens to maintain the generation quality. Thus, the optimizer of AlayaDB chooses flat indices for the first layer and uses graph-based DIPRS for the other layers. It is still an open problem in optimizing the sparse attention with different query types and index types. However, query optimization is widely studied in our database community, we hope the researchers in our community can solve it together.

7 Performance Optimization in AlayaDB

7.1 Query Processing Optimization

Window Caching Enhanced DIPR. Window caching retains a window of initial and last tokens during LLM inference, which is a standard technique in existing sparse attention algorithms [29, 43, 45, 63, 65]. The intuition of window cache is that those tokens usually contribute large attention weights. AlayaDB adopts the window cache mechanism and caches the window in GPU memory. Interestingly, the cached window can be further utilized to further enhance the quality of DIPR query results. Recall that the core challenge of DIPR queries is to correctly identify the key vector with the maximum inner product value. Interestingly, our engineers observed that the key vector with maximum inner product value has a large probability in the cached windows. For example, for dataset math_find on the model Llama-3-8B-Instruct-262k, a window of 32 (initial) + 32 (last) tokens can cover almost 98% of the key vectors with the maximum inner product values. Motivated by this observation, we enhance DIPRS by taking the maximum inner product values in both the candidate list and the cached window into consideration. It improves the performance of DIPRS by reducing the number of unnecessary tokens explored.

Flexible Context Reuse By Attribute Filtering. When a new session containing a full context is stored in AlayaDB, its vector index can be reused for efficient generation via sparse attention. However, when a new session contains only a partial prefix of a stored context, the index cannot be reused. This is a common case in practice. For example, a stored context contains a book and user A's conversation, while the incoming session of user B contains the same book but with new questions. The new session only reuses the book, which is a partial prefix of the stored context. In these cases, the session has to either be processed with expensive full attention or wait until a new index is built on the partial prefix. To address the limitations, AlayaDB supports *flexible context reuse*, which enables reusing the index of a stored context for efficient LLM inference when only a *prefix* of the stored context is reused. The challenge is to retrieve critical tokens only among the subset of tokens that are reused during searching within the full index. Interestingly, the problem can be transformed into a well-studied problem in the database community called *attribute filtering query* by considering the token id as an attribute. The naive approach of attribute filtering is pruning those nodes that do not satisfy the attribute predicate. However, this approach severely disrupts the connectivity of the graph index structure and leads to a significant decline in accuracy. We improve DIPRS with a similar idea to [49] to solve this problem. During each node exploration, the algorithm traverses both its neighbors and its neighbors' neighbors (2-hop neighbors). Subsequently, the candidates that do not meet the filtering predicate are excluded. This strategy enables AlayaDB to achieve a broader search scope during the retrieval process, which enjoys high efficiency and good accuracy.

7.2 Computation Optimization

Index Construction Acceleration. In AlayaDB, the index of a long context is constructed when a context is imported by `DB.store()` and `DB.import()`. Although this procedure is usually offline, e.g., the book is pre-loaded before the service is launched, the cost is still not negligible. We first analyze the overhead of index construction and then show how to optimize it. The fine-grained index used in AlayaDB is RoarGraph [28], a state-of-the-art index for sparse attention [45] due to its ability to handle vector search on Out-Of-Distribution (OOD) data. Following RetrievalAttention [45], a RoarGraph is constructed for each query head, and the procedure can be divided into two stages: (i) $\boldsymbol{q}$ to $\boldsymbol{k}$ kNN construction, which constructs a graph that links each query vector to its exact nearest key vectors, and (ii) connectivity enhancement, which links each vector to its approximate nearest vectors that are produced by an ANNS search on the graph. We observe that the overhead comes from the large number of indices and the slow kNN construction. We devise the following optimizations to reduce the overhead.

***GQA-based index sharing*:** GQA is commonly used in state-of-the-art LLMs [56] to reduce KV cache size. It splits the h_q query heads into h_{kv} groups, where h_{kv} is the number of key-value heads and $h_{kv} < h_q$. Queries heads within the same group will query the same key head, making one copy of KV cache able to be shared among a query group. In AlayaDB, we share a RoarGraph among a query group by sampling query vectors from each query head and merging them into one RoarGraph in the stage of kNN construction. In this way, the graph can still capture the distribution of all query heads while enjoying a speedup of h_q/h_{kv} times by the reduction in the number of indices. Our experiments show that index sharing

only results in $\leq$ 3% loss in top-k recall, and does not affect the generation quality of end-to-end LLM inference.

***GPU-based kNN construction*:** The kNN construction [58] can be highly parallel, making it suitable for GPU. We directly use the NVIDIA cuVS library [16] to accelerate its construction on GPU. To reduce the overhead of KV cache transfer, we process one layer at a time, which is compute-bound, and overlap it with the asynchronous CPU-GPU transmission in a pipeline manner.

Late Materialization for Index Updating. For each session, there are new KV caches generated from user inputs and model outputs. It raises a design choice about when to physically update them into the context. A straightforward solution is inserting the new KV cache to the existing index immediately after a new token is generated or input by users. However, it significantly (i) increases the TPOT by the blocked index updating, and (ii) occupies two memory copies by maintaining a physical index for every session. To address this problem, AlayaDB adopts a late-materialization strategy for index updating that does not affect the SLO. By default, the newly generated KV cache is appended to the local window for retrieval. The session will only be materialized into a new physical index when the `DB.store()` API is explicitly called. This is based on two practical observations that the user prompt and LLM generation following the long context are (i) often short and (ii) often not reused across sessions. Therefore, there is no need to early materialize the newly generated KV cache to the physical index in most cases.

Data-centric Attention Engine. AlayaDB is integrated with a native attention engine, which is optimized with data-centric computation. Instead of computing attention after gathering the retrieved vectors [63, 66], AlayaDB directly applies attention to the vectors where they reside, and then aggregates the attention results. This data-centric mechanism can reduce the overhead of moving the large KV cache across different computing devices. For example, when most of the context is on CPU and a window is cached on the GPU, partial attention of the two parts is computed independently in parallel and aggregated into a final attention output. We use the same algorithm as FlashAttention [32] and RetrievalAttention [45] to compute and aggregate the partial attention outputs.

7.3 Storage Optimization

During LLM inference, AlayaDB retrieves a specific portion of vector data from each attention head of different attention layers to generate the next token. However, storing all the data in the limited CPU is not practical due to the large KV cache size. To efficiently manage and reuse these vector data, we devise a vector file system and a purpose-built buffer manager within AlayaDB.

Vector File Systems. The vector file system in AlayaDB is built upon SPDK (Storage Performance Development Kit) to manage multiple vector files on disk in user space. Specifically, each vector file stores the vectors of an attention head in a specific layer. These stored vectors are organized into blocks, where vector indices and vector data are stored separately in different types of blocks, and vector index blocks are linked together in a graph structure. The benefits of our layout are two-fold: (i) the graph-based structure allows for quick traversal and access to related vectors, and (ii) the vector data can be inserted or deleted without the need for restructuring the entire file. Furthermore, the system can bypass traditional kernel I/O paths by leveraging SPDK, which significantly reduces latency and improves throughput.

Purpose-built Buffer Manager. AlayaDB has a purpose-built buffer manager built upon the underlying vector file system, which is designed to efficiently process the frequently used data in memory. It employs the eviction strategy based on the corresponding block types. For example, blocks storing the vector indices for attention heads are more likely to be kept in memory, as these vectors are frequently accessed during inference. In contrast, blocks storing the vector data are only fetched once to calculate the attention score for each token. The specific designs of it minimize redundant I/O operations by avoiding the need to retrieve them from secondary storage repeatedly. Additionally, the buffer manager supports parallel access, enabling efficient processing in a multi-threaded environment.

8 Use Cases of AlayaDB

AlayaDB provides easy-to-use interfaces and good performance for long context management and inference. In this section, we present two LLM applications to demonstrate the use cases of AlayaDB.

Financial Document Analysis. AlayaDB can be used by financial companies to assist in their financial document analysis. These documents are long, including financial statements, audit reports, business plans, etc. Data analysts in the financial company leverage domain-specific LLMs with AlayaDB to analyze a large number of financial documents and generate summarizations for their purposes, e.g., the top-10 news of Hong Kong stock market in 2024. The cost and latency of the document analysis service are reduced.

Legal Assistant for Question Answering. Law firms can utilize AlayaDB to enhance their intelligent legal assistant service. The major difference between the legal assistant and other LLM applications is that answers to users' questions must be precise and accurate, e.g., comply with the rules of the government. The legal documents can be stored as context in AlayaDB. Their domain-specific LLM answers user questions by the stored context to achieve low costs while guaranteeing result accuracy.

9 Empirical Evaluation

In this section, we conduct experiments to evaluate the end-to-end performance of AlayaDB in long-context LLM serving. In particular, we aim to answer the following two questions:

- **Q1: Can AlayaDB achieve low latency, high quality, and low resource consumption at the same time for long-context LLM serving?** (Section 9.1)
- **Q2: How is the effectiveness of our proposed performance optimizations in AlayaDB?** (Section 9.2)

Hardware Configuration. We conduct our experiments on a server with one NVIDIA L20 GPU (48GB memory) and two Intel XEON GOLD 6542Y CPUs with 48 cores, 96 threads and 512 GB DRAM in total. We use AlayaDB together with HuggingFace Transformers [61] to support LLM inference. We use the bfloat16 version of Llama-3-8B-Instruct-262k [12], the long context variant of a state-of-the-art LLM model Llama [56] for inference. The model has 32 layers. Each layer includes 32 query heads and 8 key value heads. Its weights occupy 15.4 GB GPU memory during inference.

Table 5: Generation quality of different sparse attention algorithms in ∞-Bench. Each method used the number of [*initial*+*last*]+*retrieved* tokens for attention computation.

Methods	Setting	SLO	Retr.KV	Retr.P	Retr.N	Code.D	En.MC	En.QA	En.Sum	Math.F	Avg.
Full Attention	–	✗	15.8	100.0	100.0	27.4	55.9	31.0	15.1	19.1	45.6
InfLLM	[128+4K]+4K tokens	✓	**25.0**	100.0	100.0	28.2	39.7	18.7	15.3	23.4	43.8
StreamingLLM	[128]+8K tokens	✓	3.8	8.5	8.5	27.7	41.5	14.5	14.3	16.3	16.9
Top100	[128+512]+100 tokens	✓	6.6	100.0	100.0	30.0	56.3	29.7	15.2	24.6	45.3
Top2000	[128+512]+2K tokens	✗	14.6	100.0	100.0	29.7	58.1	31.2	16.0	24.3	46.7
DIPRS	[128+512] tokens, $\beta = 50$	✓	14.0	**100.0**	**100.0**	**30.7**	**58.1**	**32.1**	**16.4**	**24.9**	**47.0**

9.1 End-to-end Performance Evaluation

9.1.1 TPOT, Quality and GPU Memory Consumption. We compare our proposed DIPR query (see Section 6.1) with existing sparse attention algorithms and full attention algorithm w.r.t. *Time-Per-Output-Token* (a.k.a., TPOT, the inference latency per token generation), quality, and GPU memory consumption in various LLM inference workloads.

Tested Workloads. We adopt a widely-used long-context benchmark ∞-Bench [67] for overall performance evaluation. Specifically, we use 8 tasks in ∞-Bench including Retr.KV, Retr.P, Retr.N, Code.D, En.MC, En.QA, En.Sum, Math.F. The average input context length of different tasks ranges from 43.9K to 192.6K tokens. In the experiments, the index of the input context is built in advance and we only measure the latency of each token generation (TPOT). We set the SLO of TPOT ≤ 0.24s, which is the reading speed of human [70],

Compared Methods. We compare the following methods:

- Full Attention, which stores the KV cache of full context and computes the full attention on GPU.
- InfLLM [63], it is a coarse-grained algorithm which selects critical tokens in blocks and computes their attention on GPU.
- StreamingLLM [65], it is an algorithm that keeps a window of tokens in GPU memory for attention computation and simply drops the other tokens.
- Top-k, it is a fine-grained algorithm which processes the top-k similarity search with graph-based index on CPU. We follow the RetrievalAttention [45] to use RoarGraph [28] as the index and align the window size. In particular, the parameter k is set as 100 and 2000 to study the performance of difference retrieved critical tokens in our experiments.
- DIPRS, our proposed DIPR query processing algorithm for sparse attention. It also uses RoarGraph as the index. The window size of DIPRS is the same as that of the top-k query.

Result Analysis. Table 5 shows the generation quality of different methods in all 8 tasks of ∞-Bench. The quality score is measured by ∞-Bench. First of all, our proposed DIPRS not only guarantees the SLO, but also achieves the best average generation quality among all the compared methods, as the last column in Table 5 shows. Moreover, it is the overall winner in 7 tasks out of the 8 tested tasks. For full attention, the SLO of TPOT is violated due to the expensive O(n) computation cost even with the KV cache in GPU memory. Compared with full attention, DIPRS can achieve a near or even higher quality in all tasks. The result also confirms the superiority of DIPRS against traditional top-k query. Top-k requires retrieving

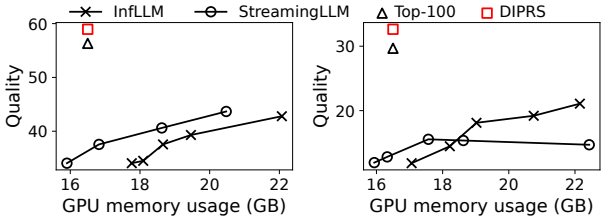


Figure 9: Generation quality and GPU memory consumption with SLO guarantees

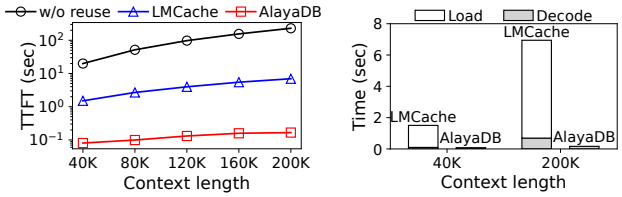
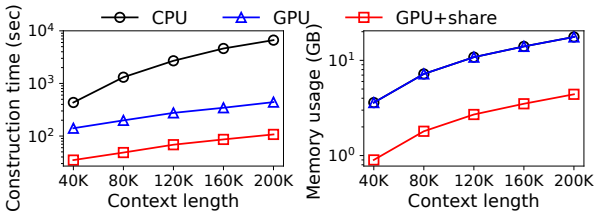


Figure 10: TTFT of long context reusing

2000 tokens to achieve a similar quality to DIPRS, but fails to meet the SLO because of retrieving too many tokens. The generation quality of top-k = 100 is worse than DIPRS in 6 tasks. In Retr.P and Retr.N, both DIPRS and top-k = 100 have the same performance.

To answer Q1, we perform in-depth analysis on two tasks (i.e., EN.MC and EN.QA) w.r.t. the generation quality and GPU memory consumption with user specified SLO. We vary the number of cached tokens for InfLLM and StreamingLLM to investigate the relationship of generation quality and GPU memory consumption. For top-k = 100 and DIPRS, we use the same settings in Table 5. As Figure 9 shows, compared to all the other methods, DIPRS achieves the best generation quality and lowest GPU memory consumption while guaranteeing the SLO of TPOT. Regarding the coarse-grained methods InfLLM and StreamingLLM, a large GPU memory is required to achieve higher accuracy, which limits the throughput of online serving and makes them impractical to run on the consumer-grade GPU, e.g., NVIDIA GTX4090 (24GB memory). Compared to top-k, the generation quality of DIPRS surpasses top-k due to its ability to identify the dynamic number of critical tokens for efficient sparse attention.

9.1.2 Time-To-First-Token: TTFT. We compare AlayaDB with the state-of-the-art disaggregated KV cache service LMCache [15, 46] to evaluate its ability to reduce the TTFT by efficiently reusing the stored long context in Figure 10. LMCache stores the compressed KV cache of the full context, and supports context reusing by loading the KV cache into GPU. In this experiment, we store the context



(a) Construction time (b) Memory consumption

Figure 11: Index construction optimization

in CPU memory in advance and measure the time of decoding the first token on this offloaded context as TTFT. Figure 10(a) depicts the experimental results. Firstly, reusing the KV cache is faster than recomputing the expensive prefill stage without reuse. For example, our AlayaDB outperforms the w/o reuse by 2 to 3 orders of magnitude, see the red and black curves in Figure 10(a). Secondly, the TTFT of AlayaDB is 19 to 42 times faster than LMCache, see the red and blue curves in Figure 10(a). By analyzing the breakdown of latency of LMCache and AlayaDB in Figure 10(b), LMCache suffers from the slow KV cache loading, including decompressing and transferring from CPU to GPU. The loading time increases linearly with the context length. Instead of loading the KV cache, AlayaDB can directly decode on the offloaded KV cache with an extremely low latency, thus, resulting to a low TTFT for context reuse. This experiment also confirms the analyzed limitation of the existing KV cache disaggregation architecture. In other words, decoupling both attention computation and KV cache from the LLM inference engine as our proposal AlayaDB is a new opportunity for our community to develop fast and accurate LLM inference systems, which provides a huge optimization space.

9.2 Effectiveness of Optimization Techniques

9.2.1 Index construction. We conduct an ablation study of our proposed optimizations for the RoarGraph construction in Section 7.2. In particular, we set the ratio of sampled queries for each index to 40%, which means when building an index, the number of used query vectors is 40% of the key vectors. Figure 11(a) shows the index construction time under different context lengths. The baseline method follows RetrievalAttention, which builds the index on CPU and builds one index for each query head, see the black curve. Introducing GPU to build kNN and employing CPU-GPU pipeline can gain a speedup from 3× to 15×, see the blue curve. Then, by sharing the index in the same query group, index construction time can be further reduced from 12× to 62× compared to pure CPU baseline, as the red curve shows. Moreover, index sharing also significantly reduces memory consumption by reducing the number of indexes. As depicted in Figure 11(b), the index size can be 4× smaller than the GPU and CPU baseline without index sharing.

9.2.2 Filter-based DIPRS. In Section 7.1, we introduce that AlayaDB leverages the attribute filtering with DIPRS algorithm to support partial context reuse. In this experiment, we study the effect of this optimization to the generation quality and inference latency. In particular, we conduct a micro-benchmark to evaluate the recall and latency of filter-based DIPRS search in the case of partial context reuse. We fix the reused prefix length to 40K, and range the reuse ratio from 100% to 20% by varying the length of the stored

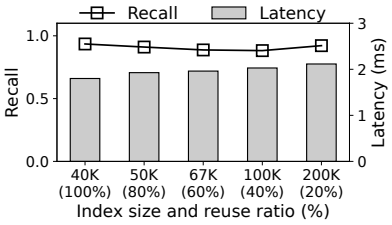


Figure 12: Micro-benchmark of filter-based DIPRS

context, i.e., the index size. This micro-benchmark uses the KV cache generated by all heads in layer 1 during the En.QA task. The 100% reuse ratio means the stored context is fully reused, in other words, the filter-based DIPRS is the same as the original DIPRS without attribute filtering. Figure 12 shows the measured recall and latency. Firstly, the recall of filter-based DIPRS remains high with different reuse ratios, which guarantees the generation quality with partial context reuse in AlayaDB. Secondly, when searching in a larger context with the same prefix length, the latency of filter-based DIPRS increases only slightly . For example, the latency to search in 200K long context is only 1.13 ms higher than it is of 40K long context. Thus, AlayaDB guarantees the inference latency with good generation quality when partial context reuse is enabled.

10 Conclusion

At AlayaDB.AI, we built AlayaDB for efficient and effective long-context inference in LLM era. From the architecture perspective, AlayaDB decouples the KV cache and attention computation from the LLM inference systems, and encapsulates them into a novel vector database system. It optimizes the overall performance by co-optimizing attention computation and KV cache management in a monolithic manner. Collaborating with the inference engine, AlayaDB is able to guarantee the SLO while enjoying low resource consumption and high generation quality for long-context LLM inference. The novel architecture poses new challenges and opportunities, including (i) implementing different parallelism strategies to enable distributed inference, (ii) supporting more LLM inference engines like vLLM and SGLang, (iii) improving the query processing methods (or sparse attention algorithms) and query optimizer, (iv) leveraging various storage tiers to store the KV cache of contexts, (v) utilizing heterogeneous hardware to accelerate the attention computation, and (vi) designing attention-hybrid architecture for general-purpose vector databases. We hope the researchers from different communities (e.g., database, machine learning, system) could tackle them together in the future.

Acknowledgments

Bo Tang was supported by National Science Foundation of China (NSFC No. 62422206). Huan Li was supported by National Science Foundation of China (NSFC No. 62402420). Man Lung Yiu was supported by Hong Kong Research Grants Council (GRF 152043/23E). Kyriakos Mouratidis was supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award No. MOE-T2EP20121-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the Ministry of Education, Singapore.

References

[1] 2024. *AlloyDB AI.* https://cloud.google.com/alloydb/ai
[2] 2024. *Amazon Kendra.* https://aws.amazon.com/cn/kendra
[3] 2024. *Bing.* https://www.microsoft.com/en-us/bing/apis/llm
[4] 2024. *ChatGPT.* https://chatgpt.com
[5] 2024. *Cursor.* https://www.cursor.com
[6] 2024. *Deepseek.* https://chat.deepseek.com
[7] 2024. *Deepseek Coder.* https://chat.deepseek.com/coder
[8] 2024. *Explainpaper.* https://www.explainpaper.com/
[9] 2024. *Gemini.* https://gemini.google.com
[10] 2024. *Generative AI in Search: Let Google do the searching for you.* https://blog.google/products/search/generative-ai-google-search-may-2024
[11] 2024. *Github Copilot.* https://github.com/features/copilot
[12] 2024. *Gradient AI. Llama-3-8b-instruct-262k.* https://huggingface.co/gradientai/Llama-3-8B-Instruct-262k
[13] 2024. *Gradient AI. Llama-3-8B-Instruct-Gradient-1048k.* https://huggingface.co/gradientai/Llama-3-8B-Instruct-Gradient-1048k
[14] 2024. *Kimi.* https://kimi.moonshot.cn
[15] 2024. *LMCache.* https://lmcache.ai/
[16] 2024. *NVIDIA cuVS.* https://github.com/rapidsai/cuvs
[17] 2024. *Perplexity AI.* https://www.perplexity.ai
[18] 2024. *Pinecone.* http://pinecone.io
[19] 2024. *weaviate: The AI-native database for a new generation of software.* http://weaviate.io
[20] 01. AI, :, Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Guoyin Wang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yanpeng Li, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. 2025. Yi: Open Foundation Models by 01.AI. arXiv:2403.04652 [cs.CL]
[21] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. 1976. System R: Relational Approach to Database Management. *TODS* 1, 2 (1976), 97–137.
[22] Saleem Ayesha. 2023. LLM for Lawyers, Enrich Your Precedents with the Use of AI. In *Data Science Dojo.* https://datasciencedojo.com/blog/llm-for-lawyers/
[23] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding. In *ACL*. 3119–3137.
[24] Zheng Bian, Xiao Yan, Jiahao Zhang, Man Lung Yiu, and Bo Tang. 2024. QSRP: Efficient Reverse k-Ranks Query Processing on High-Dimensional Embeddings. In *ICDE*. 4614–4627.
[25] Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, Xiaoyi Dong, Haodong Duan, Qi Fan, Zhaoye Fei, Yang Gao, Jiaye Ge, Chenya Gu, Yuzhe Gu, Tao Gui, Aijia Guo, Qipeng Guo, Conghui He, Yingfan Hu, Ting Huang, Tao Jiang, Penglong Jiao, Zhenjiang Jin, Zhikai Lei, Jiaxing Li, Jingwen Li, Linyang Li, Shuaibin Li, Wei Li, Yining Li, Hongwei Liu, Jiangning Liu, Jiawei Hong, Kaiwen Liu, Kuikun Liu, Xiaoran Liu, Chengqi Lv, Haijun Lv, Kai Lv, Li Ma, Runyuan Ma, Zerun Ma, Wenchang Ning, Linke Ouyang, Jiantao Qiu, Yuan Qu, Fukai Shang, Yunfan Shao, Demin Song, Zifan Song, Zhihao Sui, Peng Sun, Yu Sun, Huanze Tang, Bin Wang, Guoteng Wang, Jiaqi Wang, Jiayu Wang, Rui Wang, Yudong Wang, Ziyi Wang, Xingjian Wei, Qizhen Weng, Fan Wu, Yingtong Xiong, Chao Xu, Ruiliang Xu, Hang Yan, Yirong Yan, Xiaogui Yang, Haochen Ye, Huaiyuan Ying, Jia Yu, Jing Yu, Yuhang Zang, Chuyu Zhang, Li Zhang, Pan Zhang, Peng Zhang, Ruijie Zhang, Shuo Zhang, Songyang Zhang, Wenjian Zhang, Wenwei Zhang, Xingcheng Zhang, Xinyue Zhang, Hui Zhao, Qian Zhao, Xiaomeng Zhao, Fengzhe Zhou, Zaida Zhou, Jingming Zhuo, Yicheng Zou, Xipeng Qiu, Yu Qiao, and Dahua Lin. 2024. InternLM2 Technical Report. arXiv:2403.17297 [cs.CL]
[26] Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and Wen Xiao. 2024. PyramidKV: Dynamic KV Cache Compression based on Pyramidal Information Funneling. arXiv:2406.02069 [cs.CL]
[27] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proc. VLDB Endow.* 17, 12 (2024), 3772–3785.
[28] Meng Chen, Kai Zhang, Zhenying He, Yinan Jing, and X. Sean Wang. 2024. RoarGraph: A Projected Bipartite Graph for Efficient Cross-Modal Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 17, 11 (2024), 2735–2749.
[29] Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, and Beidi Chen. 2024. MagicPIG: LSH Sampling for Efficient LLM Generation. arXiv:2410.16179 [cs.CL]
[30] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.
[31] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *ICLR*.
[32] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *NIPS*.
[33] Gunika Dhingra. 2023. *LLMs in Finance: BloombergGPT and FinGPT — What You Need to Know.* https://12gunika.medium.com/llms-in-finance-bloomberggpt-and-fingpt-what-you-need-to-know-2fdf3af29217
[34] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. The Faiss library. arXiv:2401.08281 [cs.LG]
[35] Zhiwei Fei, Xiaoyu Shen, Dawei Zhu, Fengzhe Zhou, Zhuo Han, Alan Huang, Songyang Zhang, Kai Chen, Zhixin Yin, Zongwen Shen, Jidong Ge, and Vincent Ng. 2024. LawBench: Benchmarking Legal Knowledge of Large Language Models. In *EMNLP*. 7933–7962.
[36] Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. 2025. Ada-KV: Optimizing KV Cache Eviction by Adaptive Budget Allocation for Efficient LLM Inference. arXiv:2407.11550 [cs.CL]
[37] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.
[38] Wensheng Gan, Shicheng Wan, and Philip S. Yu. 2023. Model-as-a-Service (MaaS): A Survey. In *BigData*. 4636–4645.
[39] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: A Cloud Native Vector Database Management System. *Proc. VLDB Endow.* 15, 12 (2022), 3548–3561.
[40] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proc. VLDB Endow.* 13, 12 (2020), 3072–3084.
[41] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL]
[42] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*.
[43] 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. In *NIPS*.
[44] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G. Parameswaran, and Eugene Wu. 2024. Towards Accurate and Efficient Document Analytics with Large Language Models. arXiv:2405.04674 [cs.DB]
[45] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. 2024. RetrievalAttention: Accelerating Long-Context LLM Inference via Vector Retrieval. arXiv:2409.10516 [cs.LG]
[46] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *SIGCOMM*. 38–56.
[47] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD*. 2530–2542.
[48] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. In *IEEE transactions on pattern analysis and machine intelligence*. 824–836.
[49] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proc. ACM Manag. Data* 2, 3 (2024), 120.
[50] Sundar Pichai and Demis Hassabis. 2024. *Our next-generation model: Gemini 1.5.* https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#context-window
[51] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. arXiv:2407.00079 [cs.DC]

[52] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition.*
[53] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD*. 340–355.
[54] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. DiskANN: fast accurate billion-point nearest neighbor search on a single node. In *NIPS*.
[55] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. QUEST: Query-Aware Sparsity for Efficient Long-Context LLM Inference. In *ICML*.
[56] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]
[57] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. 1041–1052.
[58] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. 2021. Fast k-NN Graph Construction by GPU based NN-Descent. In *CIKM*. 1929–1938.
[59] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD*. 2614–2627.
[60] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165.
[61] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *EMNLP Demos*. 38–45.
[62] Long Xiang, Xiao Yan, Lan Lu, and Bo Tang. 2021. GAIPS: Accelerating maximum inner product search with GPU. In *SIGIR*. 1920–1924.
[63] Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. 2024. InfLLM: Training-Free Long-Context Extrapolation for LLMs with an Efficient Context Memory. In *NIPS*.
[64] Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. 2024. DuoAttention: Efficient Long-Context LLM Inference with Retrieval and Streaming Heads. arXiv:2410.10819 [cs.CL]
[65] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient Streaming Language Models with Attention Sinks. In *ICLR*.
[66] Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. 2025. PQCache: Product Quantization-based KVCache for Long Context LLM Inference. arXiv:2407.12820 [cs.CL]
[67] Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Hao, Xu Han, Zhen Thai, Shuo Wang, Zhiyuan Liu, and Maosong Sun. 2024. ∞Bench: Extending Long Context Evaluation Beyond 100K Tokens. In *ACL*. 15262–15277.
[68] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *NIPS*.
[69] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *NIPS*.
[70] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *OSDI*. 193–210.