Database Perspective on LLM Inference Systems

James Pan
jpan@tsinghua.edu.cn
Tsinghua University
Beijing, China

Guoliang Li
liguoliang@tsinghua.edu.cn
Tsinghua University
Beijing, China

ABSTRACT

Large language models (LLMs) are powering a new wave of language-based applications, including database applications, leading to new techniques and systems for dealing with the enormous compute and memory needs of LLMs, coupled with advances in computing hardware. In this tutorial, we review how these techniques lower inference costs by managing uncertain request lifecycles, exploiting specialized hardware, and scaling over distributed inference devices and machines. We present these techniques from the database perspective of request processing, model execution and optimization, and memory management. Following these discussion, we review how inference systems combine these techniques in diverse architectures to achieve application or performance objectives.

PVLDB Reference Format:
James Pan and Guoliang Li. Database Perspective on LLM Inference Systems. PVLDB, 18(12): 5504-5507, 2025.
doi:10.14778/3750601.3750703

1 INTRODUCTION

Large language models (LLMs) have revolutionized natural language processing and are now powering a new wave of language-based applications, including database applications such as Text-to-SQL, lake analytics, database maintenance, and visualization [14]. But LLMs come at a high cost. They require huge amounts of compute and memory to process a single request and additionally suffer confabulations that are disastrous for mission-critical applications. Model execution is made complicated by the unpredictable resource needs of each request. Meanwhile, the desire to use specialized hardware, along with distributed inference devices and machines, adds to system design complexity.

The original transformer-based LLM relies on a series of dense attention mechanisms, combined with large feed-forward networks (FFNs), to turn natural-language prompts into high-dimensional contextualized embedding vectors that are then used to generate semantically consistent textual sequence continuations through a number of model execution rounds [17]. Recently developed operator designs, such as sparse, grouped, and shared attention [2, 4, 20], and sequence generation techniques, such as beam search, graph of thoughts, and self-consistency [3, 7, 18], aim to reduce the fundamental cost of model execution while increasing the quality of generated sequences. Meanwhile, specialized kernel implementations,

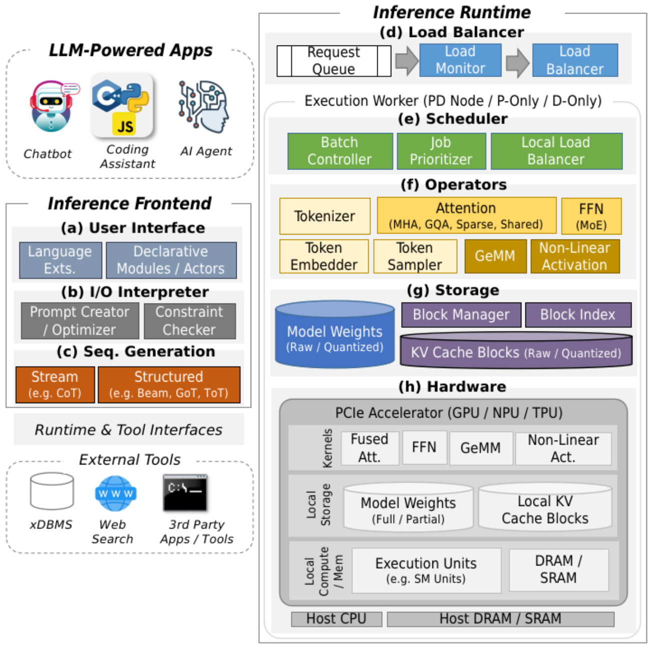

Figure 1: The LLM inference stack.

such as FlashAttention [6] and PagedAttention [12], combined with techniques for request batching, scheduling, and memory management, have been developed to take advantage of hardware accelerators while managing dynamic resource needs. Alongside these techniques are various application objectives that have led to a variety of inference systems, such as frontend-runtime co-designed systems like SGLang [20] that aim to provide low-latency inference for structured output applications, scalable disaggregated systems like Mooncake [16] that aim to provide high throughput inference, and serverless systems like DeepFlow [9] that are designed to fully leverage existing AI infrastructure.

In this tutorial, we review LLM inference from a database perspective, as shown in Figure 1, examining the new challenges for request processing, optimization and execution, and memory management that these techniques and systems aim to solve. We conclude with remaining challenges and open problems.

Tutorial Outline. This tutorial is intended to last 1.5 hours, divided into five parts.

(1) Request Processing (10 min.) The first part introduces the request processing workflow and discusses operator design along with sequence generation. Executing an LLM consists of executing several operators one after another. The compute-intensive prefill phase and memory-intensive decode phase motivates more efficient operator designs [2, 4, 20], while the risk of confabulations motivates new approaches to sequence generation [3, 7, 18].

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750703

(2) Model Optimization and Execution (30 min.) The second part describes techniques for optimizing model execution. Along with large resource needs, fundamental uncertainty regarding decoding rounds motivates techniques for batching and scheduling [1, 19], including job prioritization and load balancing [11]. At the same time, the desire to exploit GPUs and other hardware devices motivate efficient operator kernels [6, 12].

(3) Memory Management (30 min.) The third part discusses techniques for memory management. The dynamic nature of the key-value (KV) vector byproducts generated during request processing require flexible memory management frameworks, such as PagedAttention [12], to avoid waste and overhead from under and overallocation. Memory eviction and offloading techniques aim to support large contexts that stretch beyond device memory, while quantization and durable cache persistence techniques [8, 15] aim to reduce the overall memory burden.

(4) Inference Systems (15 min.) The fourth part discusses inference systems. An inference system comprises a frontend, which allows programmatic prompting, in addition to a backend runtime, which manages the various aspects of LLM execution. Centralized systems, such as vLLM [12] and SGLang [20], are aimed at deploying on a single server, although they can be deployed across clusters but require an external load balancer. These systems are aimed at providing fast inference while consuming low memory. Distributed systems, such as Mooncake [16] and DeepFlow [9], aim to increase scalability, achieved through hardware disaggregation which allows flexibly adapting to the different needs of prefill and decode.

(5) Open Problems (5 min.) We conclude the tutorial with a discussion of open problems.

Intended Audience. This tutorial is intended for researchers and practitioners that are interested in the techniques and designs of LLM inference systems. The topics will be presented in a manner that requires no prior background on LLMs.

Related Tutorials. A recently proposed tutorial aims to discuss techniques for improving LLM trustworthiness and quality, with applications to database systems [10], while also discussing LLM efficiency at a high level. To complement this effort, here we elaborate on the fundamental techniques for speeding up inference as well as provide a more comprehensive analysis of LLM inference system design from a database perspective.

2 TUTORIAL

2.1 Request Processing

Transformer-based LLMs introduce new resource-intensive operators compared to relational operators. The fundamental costs depend on the logical designs of the operators. Parallelism, sparse sampling, and speculative techniques have been used to develop faster attention, FFN, and token sampling mechanisms, while various probabalistic and structured generation techniques have been proposed to improve output quality.

Operator Design. The complexity of the attention operator depends on the amount of cached KV vectors, used for contextualizing input embeddings. Physical execution time can be reduced via parallelized multi-head attention [17], while the fundamental memory and compute costs can be reduced via grouped, sparse, and shared attention [2, 4, 20], which aim to reduce the number or size of matrix multiplications. Likewise, the fundamental compute costs for the FFN can be reduced by adopting a mixture of experts (MoE) that replaces the large FFN with several smaller ones, only some of which are used during model execution. For the token sampler, speculative decoding [13] increases token throughput by using smaller models to quickly generate potential output tokens while using the full model to verify them in a single execution cycle.

Sequence Generation. To increase output quality, various token sampling strategies, including temperature-based probabalistic sampling, top-k sampling, and nucleus sampling can be used to increase the exploration of the token space, while structured generation techniques, such as beam search [7], graph-of-thoughts [3], and self consistency [18], can be used to expand and refine the set of output candidates, thereby increasing the chance of producing a high-quality output.

2.2 Model Optimization and Execution

Like GPU-based DBMSs, inference systems take advantage of hardware accelerators to reduce execution latency. These devices require specialized kernels to fully exploit the expanded processing capabilities. Likewise, the extreme data parallelism motivates batching techniques that aim to serve multiple requests within a single execution cycle. At the same time, inference machines often contain multiple accelerator devices, moreover distributed inference systems increase throughput by exploiting multiple inference machines, similar to distributed DBMSs. Effectively using these resources while dealing with unpredictable request lifecycles requires specialized scheduling techniques for job ordering and load balancing.

Kernels. Hardware kernels are carefully designed to provide parallelized operators that maximize core utilization while minimizing overhead, including memory I/O costs on the device as well as kernel invocation overhead. Blockwise attention kernels, such as FlashAttention[1] [6], FlashDecoding[2] and Lean Attention[3], rely on tiled matrix multiplication and online softmax to avoid I/O and storage costs related to intermediate product materialization as well as reducing kernel invocations via operator fusion. Distributed attention kernels, such as Ring Attention[4], take advantage of multiple devices to increase parallelism farther, useful for supporting large contexts that require memory capacity beyond a single device. Specialized kernels have also been developed for sparse attention in addition to non-GeMM operators. Automatic kernel compilation via computation graph optimization has also been explored.

Request Batching. Multiple requests can be served at once by concatenating their token embeddings and processing the batched input matric in a single execution cycle. But due to differences in input lengths as well as KV cache sizes, batching can lead to ragged tensors during the attention computation. Batch formation techniques, such as TurboTransformers[5], aim to minimize tensor sparsity in order to maximize core utilization, while byte packing

[1]http://arxiv.org/abs/2407.08608
[2]http://arxiv.org/abs/2311.01282
[3]http://arxiv.org/abs/2405.10480
[4]http://arxiv.org/abs/2310.01889
[5]http://arxiv.org/pdf/2010.05680

techniques, such as ByteTransformer[6], try to repack ragged tensors in order to reduce wasted computation due to tensor sparsity. Meanwhile, the autoregressive nature of LLM sequence generation allows requests to be periodically rebatched as a means of balancing TTFT and TBT. This has led to techniques for continuous batching [19], including chunked prefills [1].

Job Prioritization. The order that requests are served can affect mean latency and total throughput across the workload. Requests can be prioritized by completion time, so that long-running requests do not stall short requests. This requires estimating the completion time, which can be based on various cost models [11]. On the other hand, SGLang prioritizes requests by amount of reusable cache in order to avoid cache thrashing [20]. Dynamic promotion and demotion mechanisms can also be used to avoid request starvation.

Load Balancing. Systems like Mooncake [16] and DeepFlow [9] are designed to support multiple inference machines, requiring load balancing mechanisms to evenly distribute request processing costs across the machines. But uncertain request lifetimes, combined with uncertainty about future loads, makes it difficult to select the latency-minimizing worker for a given request. As a result, most systems resort to greedily assigning requests to workers based on local least-load heuristics [11] or based on cache availability in the case of persisted caches.

2.3 Memory Management

Processing a request creates volatile KV memory byproducts, resembling in-memory OLTP workloads. Except for length-constrained generation, the final size of the KV cache is not knowable, posing a challenge for memory management. Static preallocation risks issues caused by under or over allocation, leading to techniques for dynamic paged-based blockwise allocation. Eviction and offloading techniques, in addition to quantization techniques, aim to reduce the overall memory burden. In some scenarios, it may be possible to share certain KV cache entries across multiple requests, leading to techniques for durable cache persistence and reuse.

Paged-Based Memory Allocation. Dynamic paged-based blockwise memory allocation can lead to non-contiguous cache storage, requiring a memory manager to coordinate logical memory pages and their physical blocks. Both vLLM [12] and vAttention[7] introduce memory managers targeted at GPU systems for handling page creation, deletion, and lookup, but with different implementation approaches. The vLLM approach performs memory management on the host machine and requires a specialized page-aware kernel to operate over non-contiguous cache storage whereas vAttention exploits native CUDA GPU memory management capabilities, allowing the use of non-paged kernels. Blockwise memory allows for block sharing, in which the same physical block is mapped to multiple LLM inference processes, when these processes share a common prefix. These situations arise under *e.g.* shared system prompts, or when one request yields multiple output sequences via beam search or tree-of-thoughts.

Eviction and Offloading. Unneeded cache entries can be permanently removed via eviction or temporarily removed via offloading onto a secondary memory container (*e.g.* from GPU memory to host DRAM). Both eviction and offloading can be used to support longer contexts beyond primary memory capacity and as a way to handle preempted requests, but require techniques for identifying which entries to remove, taking into account potential recovery costs. For long contexts, eviction techniques identify least important cache entries via an importance score, which can be position-based, attention-based, or usage-based [20]. For handling preempted requests, the cache entries of a preempted request can be directly evicted or offloaded and reloaded upon resumption, depending on the recovery cost [12]. For offloaded requests, techniques include layer-wise and model-wise offloading, with pre-fetch recovery.

Quantization. Model weights, layer activations, and KV cache entries are typically represented by floating-point numbers, but their byte sizes can be reduced by quantizing them into low-bit integers [15]. For weights and activations, the quantizer aims to minimize compression error, and techniques include whole-tensor quantization as well as group-based quantization. For KV cache entries, the quantizer aims to be similarity-preserving, and techniques include importance weighting and outlier tracking.

Cache Persistence. Persisting KV cache entries instead of discarding them after request completion allows them to be potentially reused for future requests, avoiding the need for recomputation. But under standard attention, KV values in the inner transformer layers are position-dependent. Prefix sharing aims to quickly identify the longest shareable token sequence between a request and the persisted entries using techniques such as brute-force scanning and radix tree lookup [20]. On the other hand, selective reconstruction techniques aim to reuse all entries for matching tokens, even if the token positions are non-matching, by selectively recomputing KV values for entries that cause greatest quality loss [8].

2.4 Inference Systems

Various inference systems have been developed to support diverse needs, adopting different architectures and techniques. Some focus on particular application scenarios, leading to frontend and runtime co-design, while others focus on providing high-performance inference over general LLM workloads. Centralized systems focus on low-latency inference while distributed systems aim to provide high throughput via scalable disaggregated and serverless architectures. Additionally, frontend-only systems aim to facilitate development of LLM-based programs.

Centralized Systems. SGLang [20] offers a co-designed frontend and runtime and is aimed at fast structured output generation. It exploits frequent prefix sharing opportunities by using a radix tree to quickly retrieve shareable KV entries, along with a cache-aware scheduler and a prefill interleaving technique for fast template completion. As another example, vLLM [12] is aimed at providing low latency but for generic workloads. It adopts chunked prefills, continuous batching, and distributed attention across GPUs while minimizing memory usage via prefix sharing and paged attention. As it targets single server deployment, it lacks an internal load balancer but can be scaled via third-party tools.

Distributed Systems. Mooncake [16] is aimed at low latency and high throughput via scalable P/D disaggregated architecture. To minimize memory usage, it adopts a distributed blocked KV cache.

[6]http://arxiv.org/abs/2210.03052
[7]http://arxiv.org/abs/2405.04437

To reduce latency, it uses a greedy load balancer based on a latency estimation model that considers cache availability, cache transfer time, worker load, and other factors. It also supports hot cache block replicas, prefill and decode prefetch, continuous batching, and chunked prefills. DeepFlow [9] is another example of a distributed system but aimed at deployment over shared hardware via serverless architecture. Model inference is decomposed into logical task executors that perform P/D-only, mixed, expert-only, or attention-only operations, allowing for fine-grained load balancing and resource scaling based on the workload.

Frontends. Frontends allow for programmatic prompting, where the full prompting logic is provided upfront, in a form resembling a traditional program. This makes it easier to implement complex prompting workflows, such as beam search, tree-of-thoughts, RAG, tool use, and so on. An LLM frontend typically also supports automatic prompt generation, taking advantage of prompt engineering techniques to reduce latency (e.g. prefill interleaving) or improve output quality (e.g. chain-of-thought, few-shot prompting). Most frontends also include mechanisms for constrained generation based on simple rules (e.g. output value) or complex patterns (e.g. regular expressions), giving users more control over sequence generation termination conditions as well as allowing for structured outputs. These features are achieved by extending common programming languages with new imperative commands (e.g. SGLang [20], Guidance[8], LangChain[9]) or declarative modules (e.g. DSPy[10], LMQL[11] that serve as an interface to the underlying LLM.

2.5 Open Problems

Existing techniques for batching and scheduling rely on heuristics, such as simple load equations and request lifetime estimates, to inform batch formation, job prioritization, and load balancing. Developing more accurate cost estimates is needed to make these techniques more effective. On the other hand, developing adaptive techniques that can respond to uncertain request lifetimes, such as more sophisticated load balancing techniques, can improve system performance in the absence of accurate cost estimates. Finally, the rapidly expanding LLM ecosystem demands new benchmarks, like [5], for systematically evaluating inference systems.

3 PRESENTERS

James Pan is an assistant researcher at Tsinghua University. His research interest is data management for artificial intelligence applications, including LLM inference systems and vector databases. He received his Ph.D. in Computer Science from Tsinghua University.

Guoliang Li is a professor in the Department of Computer Science at Tsinghua University in Beijing, China. He is an ACM and IEEE Fellow. His research interests include machine learning for databases, database systems, and data cleaning and integration. His research has received numerous accolades, including the VLDB 2017 Early Research Contribution Award, TCDE 2014 Early Career Award, VLDB 2023 Industry Best Paper Runner-Up Award, Best of SIGMOD 2023, SIGMOD 2024 Research Highlight Award, DASFAA 2023 Best Paper Award, and CIKM 2017 Best Paper Award.

[8]http://github.com/guidance-ai/guidance
[9]http://langchain.com
[10]http://dspy.ai
[11]http://lmql.ai

ACKNOWLEDGMENTS

This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (62525202, 62232009), Shenzhen Project (CJGJZD20230724093403007), Zhongguancun Lab, Huawei, and Beijing National Research Center for Information Science and Technology (BNRist). Guoliang Li is the corresponding author.

REFERENCES

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *OSDI'24*. 117–134.

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *EMNLP'23*. 4895–4901.

[3] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of thoughts: Solving elaborate problems with large language models. *AAAI'24* 38, 16 (2024), 17682–17690.

[4] 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] https://arxiv.org/abs/2410.16179

[5] Krishna Teja Chitty-Venkata, Siddhisanket Raskar, Bharat Kale, Farah Ferdaus, Aditya Tanikanti, Ken Raffenetti, Valerie Taylor, Murali Emani, and Venkatram Vishwanath. 2024. LLM-Inference-Bench: Inference benchmarking of large language models on AI accelerators. arXiv:2411.00136 [cs.LG]

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *NeurIPS'22*.

[7] Alex Graves. 2012. Sequence transduction with recurrent neural networks. arXiv:1211.3711 [cs.NE] https://arxiv.org/abs/1211.3711

[8] Junhao Hu, Wenrui Huang, Haoyi Wang, Weidong Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. 2025. EPIC: Efficient position-independent context caching for serving large language models. arXiv:2410.15332 [cs.LG] https://arxiv.org/abs/2410.15332

[9] Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Baoquan Zhang, Shining Wan, Gengyuan Dan, Zhiyu Dong, Zhihao Ren, Jie Meng, Chao He, Changhong Liu, Tao Xie, Dayun Lin, Qin Zhang, Yue Yu, Hao Feng, Xusheng Chen, and Yizhou Shan. 2025. DeepFlow: Serverless large language model serving at scale. arXiv:2501.14417 [cs.DC]

[10] Kyoungmin Kim and Anastasia Ailamaki. 2024. Trustworthy and efficient LLMs meet databases. arXiv:2412.18022 [cs.DB] https://arxiv.org/abs/2412.18022

[11] Ferdi Kossmann, Bruce Fontaine, Daya Khudia, Michael Cafarella, and Samuel Madden. 2025. Is the GPU half-empty or half-full? Practical scheduling techniques for LLMs. arXiv:2410.17840 [cs.LG] https://arxiv.org/abs/2410.17840

[12] 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'23*. 611–626.

[13] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *ICML'23*.

[14] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. LLM for data management. *Proc. VLDB Endow.* 17, 12 (2024), 4213–4216.

[15] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Guangxuan Xiao, and Song Han. 2025. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. *GetMobile* 28, 4 (2025), 12–17.

[16] 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]

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS'17*. 6000–6010.

[18] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *ICLR'23*.

[19] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for transformer-based generative models. In *OSDI'22*. 521–538.

[20] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient execution of structured language model programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104