MatSQL: Accelerating Text-to-SQL via Database Schema Materialization

Kyong-Shik Lee*
Graduate School of Data Science
Seoul National University
Seoul, Republic of Korea
kyongshikl@snu.ac.kr

Dongseob Kim*
Graduate School of Data Science
Seoul National University
Seoul, Republic of Korea
eastskim48@snu.ac.kr

Sang-Won Lee
Graduate School of Data Science
Seoul National University
Seoul, Republic of Korea
swlee69@snu.ac.kr

***Abstract*—LLM-driven methods have become the dominant paradigm for Text-to-SQL, driving a shift from specialized parsers to prompting-based generation. In production settings, however, these systems must satisfy latency and cost constraints while operating over large and heterogeneous schemas. Modern Text-to-SQL workflows typically invoke the model multiple times, repeatedly re-encoding static prompt prefixes that include system instructions and schema descriptions. Prefill computation thus becomes a major source of redundancy. We characterize Text-to-SQL serving as a context-intensive workload in which large, low-volatility static prefixes dominate prefill latency. Existing prefix caching mechanisms that rely on GPU or DRAM residency do not scale in multi-tenant environments, where the aggregate working set of schema contexts exceeds available memory capacity and frequent eviction negates reuse.**

To address this limitation, we present MatSQL, a KV caching framework for Text-to-SQL serving. MatSQL treats high-reuse prompt segments as persistent database assets and manages them in a hierarchical cache spanning GPU memory and SSDs. This design decouples effective cache capacity from volatile memory limits and enables scalable prefix reuse across workflows and tenants. We further provide mechanisms to maintain cache correctness under schema evolution and to manage replacement under changing workloads. Integrated into a multi-call Text-to-SQL pipeline, MatSQL reduces prefill latency by up to $6.98\times$ and improves end-to-end latency by $1.06\times$ while preserving generation accuracy.

***Index Terms*—Text-to-SQL, Large Language Models, Key-Value Caching, Materialization, Storage Systems**

I. Introduction

Text-to-SQL translates a user's natural-language question (NLQ) into an executable SQL query over a relational database, enabling non-experts to effectively access structured data. The advent of large language models (LLMs) has shifted the prevailing paradigm from supervised, parser-centric pipelines [1]–[5] to in-context learning and prompt-based inference [6]–[16], yielding substantial accuracy gains on standard benchmarks such as Spider [19] and BIRD [18]. These advances make LLM-based Text-to-SQL increasingly viable beyond academic leaderboards. However, production deployments must satisfy service-level objective (SLOs) that couple correctness with strict latency and operational cost constraints.

In practice, meeting these constraints is increasingly difficult because state-of-the-art systems rely on multi-stage, multi-call inference [17]. For robustness, modern approaches compose workflows that generate multiple candidates and iteratively refine them with execution feedback [6], [24]. Each call reprocesses the same prompt prefix, including system instructions and database schema context, re-encoding the identical schema-induced KV states from scratch.

TABLE I: Scale Summary of Text-to-SQL Benchmarks.

Dataset	Year	Tbl/DB	Col/Tbl	Rec/Tbl	Size/DB*
WikiSQL [21]	2017	1.00	6.34	17	4.37×10^{-6}
Spider [19]†	2018	5.13	5.01	1,752	0.003
KaggleDBQA [22]	2021	2.12	10.53	280,035	0.049
BIRD [18]	2023	7.64	7.14	600,371	0.352
Spider 2.0 [23]‡	2024	23.71	35.61	54,734	0.056
BEAVER [20]§	2024	77.50	9.15	95,526	2.33

* *Size/DB* is the average size of executable database files (e.g., `.sqlite`, `.db`); annotation and metadata files are excluded.
† Statistics for *Spider* are computed from the development split.
‡ Statistics for *Spider 2.0* are computed on the *Spider 2.0 Lite* release (reported as *Spider 2.0* for brevity) and include only SQLite databases.
§ *BEAVER* statistics correspond to the Data Warehouse (DW) domain.

This computational burden is further exacerbated in enterprise environments, where schemas are substantially larger and more heterogeneous. Table I illustrates this shift from early academic datasets to enterprise-grade benchmarks. While predecessors such as Spider [19] average fewer than six tables per database, recent benchmarks like BEAVER [20] average 77.5 tables, with average database size increasing from 0.003 GB (Spider) to 2.33 GB (BEAVER DW). This aligns with production data warehouses, where systems operate over large, evolving schemas that are frequently modified as tables, columns, and metadata evolve over time. To maintain high recall in such expansive search spaces, systems increasingly abandon brittle pruning heuristics and instead condition schema grounding on the model by placing the full schema and associated metadata directly in the prompt context [12], [13].

The combination of iterative inference cycles and large schema contexts creates a significant performance bottleneck. While decoding latency is inherent to autoregressive generation, the prefill phase accounts for a substantial fraction of redundant computation. The static schema prefix remains invariant across all queries targeting a given database, yet is

*These authors contributed equally.

re-encoded from scratch for every request. As a result, the prefill phase emerges as the primary bottleneck and a clear target for system-level optimization.

Ideally, this structural invariance suggests prefix caching as a natural solution. General-purpose serving systems such as vLLM [25] and RadixAttention [26] mitigate prefill overhead by retaining KV states of repeated prefixes in GPU memory. However, the effectiveness of memory-resident caching is bounded by memory capacity constraints in enterprise Text-to-SQL settings. A typical multi-tenant service hosts thousands of distinct databases, each requiring a large, static schema context to ensure reasoning accuracy. The aggregate working set of schema prefixes often exceeds the physical capacity of HBM and DRAM. Under such contention, memory-based systems experience frequent eviction, negating the benefits of caching and reverting to repeated re-computation [27]. This limitation motivates a storage-centric hierarchy that extends cache capacity beyond volatile memory using high-bandwidth SSDs.

To address this, we propose **MatSQL (Materialized SQL Generation)**. Building on the persistent KV caching principle of MatKV [28], MatSQL redefines high-reuse prompt segments in Text-to-SQL as database assets, treating them as persistent views whose prefill states are materialized rather than repeatedly recomputed. By identifying low-volatility prompt segments and offloading their materialized KV states to a hierarchical pool spanning GPU memory and SSDs, MatSQL decouples cache capacity from DRAM limitations. We integrate MatSQL into CHESS-SQL [6] to demonstrate its efficacy in representative multi-call workflows. Our evaluation shows that MatSQL achieves an end-to-end speedup of $1.06\times$ while maintaining generation quality. For the most repetitive generation stage, MatSQL accelerates prefill computation by up to $6.98\times$.

To the best of our knowledge, MatSQL is the first system to systematically address prefill redundancy in enterprise-scale Text-to-SQL. Our contributions are summarized as follows:

- **Workload Characterization.** We characterize modern Text-to-SQL serving as a context-intensive, multi-invocation workload where large schema prefixes dominate prefill latency and induce redundant computation.
- **Storage-Centric KV Materialization.** We design MatSQL, a hierarchical caching system that materializes KV states of reusable prompt segments onto SSDs. This design enables scalable prefix reuse that transcends traditional DRAM/HBM capacity walls.
- **Operational Mechanisms for Deployment.** We introduce essential mechanisms for persistent caching in real-world databases, including lifecycle management for schema evolution and cost-aware replacement policies to handle workload churn.
- **Evaluation.** We implement MatSQL within CHESS-SQL and evaluate it on large-schema benchmarks, demonstrating significant latency reduction ($6.98\times$ prefill speedup) without compromising Text-to-SQL accuracy.

II. Background and Motivation

A. LLM Inference and the Cost of Context

To understand the performance bottlenecks LLM-based in Text-to-SQL, we first examine the underlying mechanics of LLM inference, which proceeds in two phases, **Prefill** and **Decode**. During the Prefill phase, the model processes the entire input prompt sequence in parallel to compute Key-Value (KV) tensors for every token. These tensors, which encapsulate the semantic context, are stored in the KV cache to facilitate subsequent attention computations. The Decode phase then generates output tokens autoregressively, attending to this cached context without re-processing the input history.

The prefill phase becomes increasingly expensive with longer inputs. In context-intensive tasks like Text-to-SQL, this phase dominates the GPU compute budget. In standard stateless serving, this intermediate state is discarded after request completion, requiring full re-computation for subsequent queries. While prior research has largely optimized memory management by mitigating fragmentation during autoregressive generation (e.g., PagedAttention [25]), the expanding context windows of modern applications have shifted the bottleneck back to the Prefill phase. Recent works like MatKV [28] have begun to address this by persisting pre-computed states, highlighting the potential of storage-backed acceleration.

B. Anatomy of Text-to-SQL Prompts

While Section I introduced the scale of enterprise schemas, we quantify how this scale translates into inference overhead. We analyze the prompt structure of CHESS-SQL [6], a representative multi-agent Text-to-SQL framework.

Workflow Composition. CHESS-SQL executes a composite workflow that maps an NLQ to an executable SQL query via multiple stages. This process comprises Information Retrieval (IR) for gathering information related to the input, Candidate Generation (CG) for SQL synthesis, and Unit Testing (UT) for validation. Such a design results in repeated LLM invocations, where a single logical request triggers multiple sequential inference calls across workflow stages. In particular, the Candidate Generation phase involves several iterative invocations to explore alternative reasoning paths.

Prompt Decomposition. For a user query q targeting a database schema S, we model the prompt as

$$P(q, S) = \underbrace{T_{\text{sys}} \oplus T_{\text{schema}}(S)}_{\textbf{Static Prefix}} \oplus \underbrace{T_{\text{query}}(q)}_{\textbf{Dynamic Suffix}}, \qquad (1)$$

where $\oplus$ denotes concatenation. The static prefix contains system instructions and schema context, and the dynamic suffix contains query specific information. Since the static prefix is fixed for a given database S, iterative execution of the IR→CG→UT pipeline repeatedly re-encodes it during prefill.

Empirical Prefix Dominance. We quantify this redundancy by measuring the prefill token composition of each CHESS-SQL stage on a BIRD benchmark [18]. Figure 1 shows that static tokens ($T_{\text{sys}} \oplus T_{\text{schema}}$) dominate prefill across all stages. This effect is strongest in *Candidate Generation (CG)*,

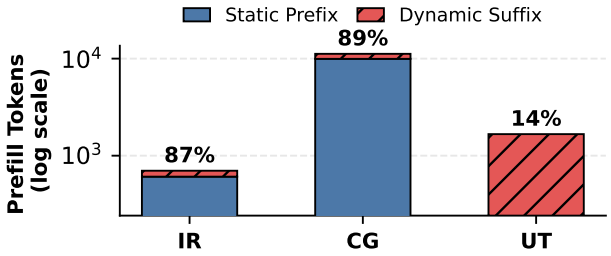


Fig. 1: Prefill token breakdown by workflow stage.

which both incurs the largest prefill cost and exhibits a static-prefix ratio of 89%. Because CG is executed iteratively, the same static prefix is re-processed multiple times, indicating substantial potential for prefill reduction through prefix reuse.

C. The Memory Capacity Wall and a Storage-Centric Solution

The observations above suggest a straightforward optimization: cache the KV states of the static prefix in fast memory and reuse them across invocations. Existing serving systems already exploit in-memory prefix caching to reduce redundant prefill. However, this approach fundamentally hits a *memory capacity wall* in the target Text-to-SQL setting.

The Scalability Gap. Enterprise Text-to-SQL platforms operate in multi-tenant environments where a single service instance hosts thousands of distinct databases. In these environments, caching a large schema prefix for a single tenant consumes substantial HBM/DRAM, and extending this allocation to thousands of tenants exceeds physical memory capacities. Memory-resident caches are thus compelled to evict schema prefixes aggressively. Upon the reactivation of an evicted tenant under time-varying access patterns, the system reverts to recomputing the static prefix. This recurrence incurs the full prefill overhead and negates the performance gains of caching.

SSD-Backed Cache Hierarchy. For large-scale models like Qwen-32B, the KV cache requires approximately 0.25 MB per token, creating substantial memory overhead. Using the California Schools database (5,197 tokens) from the BIRD benchmark [18] as a case study, retaining its 1.3 GB cache in DRAM costs roughly $20, whereas offloading to an NVMe SSD reduces this expenditure to just $0.16.[1] Furthermore, modern SSDs provide sufficient bandwidth to mitigate the latency penalty of this storage tier, enabling cached states to be retrieved with sub-100 ms latency per stage. This overhead is negligible compared to the latency of the standard LLM inference, confirming the efficacy of SSD offloading.

Implication. This motivates a storage-centric caching design that (i) materializes KV states for reusable static prefixes, (ii) keeps hot prefixes in HBM/DRAM, and (iii) swaps cold prefixes to SSD without discarding them. The next section details how MatSQL implements this hierarchy and how it manages cache updates under schema evolution.

III. Method

A. System Overview

MatSQL eliminates redundant prefill computation by persisting and reusing KV states for static prompt prefixes in agentic Text-to-SQL workflows. Fig. 2 shows the system architecture. The system consists of five components: (i) an **Agent Workflow Orchestrator** that executes a CHESS-style IR→CG→UT workflow, (ii) an **LLM Engine** that performs prefill and decoding, (iii) a **Cache Orchestrator** that manages KV states between GPU memory and SSD, (iv) a **Schema Synchronizer** that maintains cache correctness as schemas evolve, and (v) a **Cache Store** that persists materialized KV blocks and raw texts. The orchestrator implements the CHESS-SQL execution model using LangChain [32]. In the remainder of this section, we describe the core mechanisms of MatSQL.

B. Structural Prompt Formalization for Granular Caching

MatSQL adopts the prompt decomposition from Section II (Eq. 1). Given a database S, the static prefix $T_{\text{sys}} \oplus T_{\text{schema}}(S)$ is reused across invocations, whereas the dynamic suffix $T_{\text{query}}(q)$ varies across queries and workflow stages. This separation enables granular cache management. MatSQL materializes and manages KV states for T_{sys} and $T_{\text{schema}}(S)$ separately, reflecting their distinct reuse patterns.

Template KV Cache. The system-instruction prefix T_{sys} is shared across all requests and thus exhibits the highest reuse. MatSQL materializes it once and keeps the resulting KV states resident in GPU memory.

Schema KV Cache. The schema prefix $T_{\text{schema}}(S)$ is database-specific and typically large. In multi-tenant settings, access often exhibits temporal locality, where a small subset of schemas is active at any time while others remain idle. MatSQL treats schema KV blocks as a swappable working set, retaining hot schemas' KV blocks in GPU memory and spilling cold ones to SSD based on observed access patterns.

C. LLM Inference with Cache Orchestration

The Cache Orchestrator manages the lifecycle and placement of KV caches across the memory hierarchy. It coordinates with the LLM Engine to minimize redundant computation under constrained GPU memory.

Upon receiving a user request, the engine queries the Cache Store to locate reusable prefix KV states corresponding to the prompt template and database schema. All template and schema KV caches are pre-materialized and maintained in the Cache Store. Inference begins by loading these prefixes into the execution context.

Due to limited GPU memory, not all schema KV blocks can remain resident. Template KV caches are kept in GPU memory, while schema KV caches are placed in GPU memory or SSD according to the cache replacement policy. The Cache Orchestrator determines the location of the required KV blocks and retrieves them before executing accelerated prefill, avoiding recomputation of static contexts.

[1]Cost estimates are based on server-grade memory and NVMe SSD market prices as of January 2026.

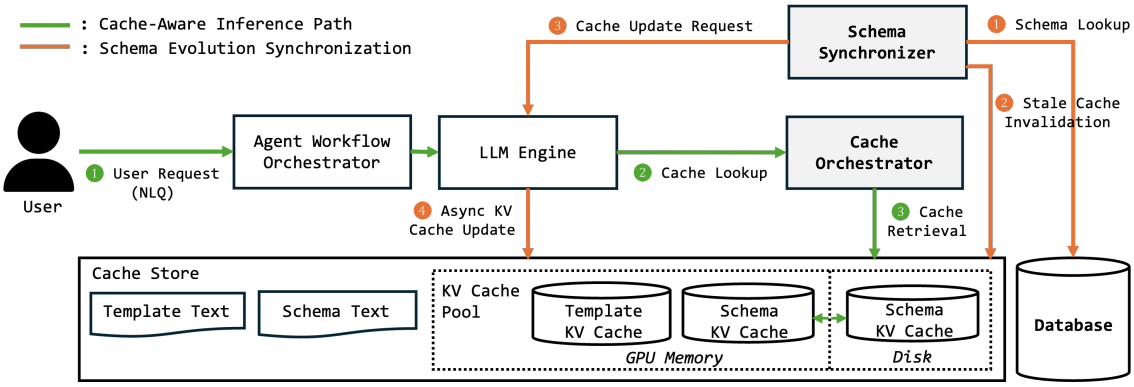


Fig. 2: Overall Architecture of MatSQL

D. Hierarchical Cache Management

To address the physical limitations of GPU memory capacity, the system adopts a hierarchical storage architecture spanning both GPU Memory and SSD. Unlike traditional caching strategies that permanently evict cold data, the Cache Orchestrator governs cache residency using a swapping-only strategy driven by a Least Recently Used (LRU) policy.

Recognizing that Database Schema contexts hold high reusability value, the system avoids cache eviction entirely. When the GPU memory pool reaches saturation, the LRU policy identifies the least recently accessed Schema KV blocks. These ”cold” blocks are swapped out to the SSDs, which acts as a backing store. Conversely, when a query targeting a swapped-out schema arrives, the system triggers a swap-in operation, seamlessly reloading the KV cache from the SSD to the GPU memory. This mechanism, analogous to demand paging in operating systems and paged attention [25], enables the system to support a vast number of databases exceeding physical GPU limits by optimizing the trade-off between storage I/O latency and re-computation costs.

E. Schema Synchronization

Database schemas in production are subject to continuous evolution. Relying on stale schema contexts leads to critical SQL generation errors. To maintain strict consistency between the database and the LLM context, the system employs an event-driven Schema Synchronizer that automates the cache lifecycle as depicted in the red workflow of Fig. 2.

Real-time Invalidation and Fallback Strategy. Unlike passive periodic polling, the Schema Synchronizer reacts to real-time schema change events (e.g., DDL triggers) directly from the database. Upon receiving a change signal, the synchronizer immediately executes Stale Cache Invalidation, marking the outdated KV blocks in the pool as unusable.

Crucially, the system ensures high availability during the cache miss window—the interim period between invalidation and the completion of the update. If a user query targets a schema that is currently invalidated, the LLM Engine falls back to on-demand recomputation and immediate cache update. It processes the raw schema text from scratch for that specific request, trading a transient latency spike for guaranteed correctness, rather than serving stale information or failing the request.

Asynchronous Prefill-only Materialization. To restore optimal performance, the system triggers an Async KV Cache Update as a background process. This update is strictly executed as a prefill-only inference task, leveraging idle GPU cycles to encode the new schema definition into KV tensors without performing subsequent token generation. Once this prefill is complete, the new materialized KV cache is stored in the hierarchical storage (Memory/Disk), and the system reverts to serving low-latency queries from the cache.

IV. Evaluation

A. Experimental Setup

1) Hardware: All experiments were conducted on a server equipped with an NVIDIA RTX A6000 GPU (48GB VRAM) and an Intel Xeon Gold 6338 CPU. For persistent storage, the KV cache system is backed by a 1TB Samsung 990 EVO Plus NVMe SSD, which supports a sequential read bandwidth of 7,450 MB/s via the PCIe Gen 4.0.

2) Implementation Details: We implemented the serving engine using the Hugging Face Transformers library [31] integrated with Flash Attention-2 [30]. For the target model, we utilized Qwen 2.5 Coder 32B Instruct [29] with 4-bit NF4 quantization, employing the 0.5B Instruct variant as the speculative drafter. We developed a custom serving stack because existing high-throughput systems such as vLLM [25] do not natively support the hierarchical, storage-resident KV cache management required by our disaggregated architecture.

3) Benchmark & Workload Configuration: We integrated MatSQL into CHESS-SQL [6], a representative multi-turn Text-to-SQL framework, and evaluated it on the BIRD benchmark [18]. To mitigate the prohibitive computational costs of self-hosted inference, all experiments operated on the Sub-sampled Development Set (SDS) defined in CHESS-SQL [6]. We adopted the standard IR–CG–UT workflow but disabled the Schema Selection (SS) module to enforce full-schema contexts, thereby explicitly isolating the prefill bottleneck. While the default configuration allows up to 20 invocations,

TABLE II: End-to-End Performance Comparison.

System	CG Latency (s)	End-to-End Latency (s)	Exec. Acc. (EX)
Vanilla	273.4	496.7	53.74%
MatSQL	**249.3**	**468.2**	**53.06%**
Speedup	*1.10×*	*1.06×*	–

TABLE III: End-to-End Latency Composition.

Pipeline Stage	% of End-to-End Latency (%)	Speedup
Information Retrieval (IR)	18.7	1.07×
Candidate Generation (CG)	77.8	1.10×
Unit Testing (UT)	3.5	1.01×

we fixed the candidate generation stage to 4 iterations for this evaluation to ensure controlled comparison.

4) Baseline: We compared MatSQL against a vanilla baseline executing the identical CHESS-SQL workflow without KV materialization. This baseline employs the same model, prompt templates, and workflow configurations but performs a full re-computation of the prompt prefix at every LLM invocation using standard in-memory KV caching.

5) Metrics: We evaluate MatSQL using following metrics.

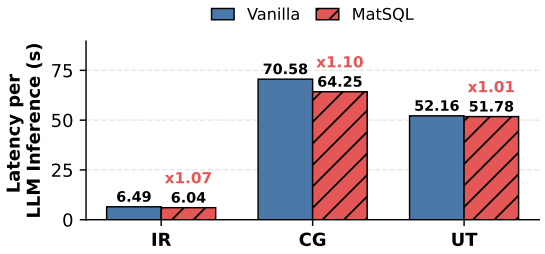
Latency. We measure wall-clock latency and report both *End-to-End Latency* (IR+CG+UT) and *Candidate Generation Latency (CG)*, which constitutes the dominant portion of the total execution time. We also report stage-wise latency per LLM inference for IR/CG/UT, and decompose it into *prefill* and *decode* components to attribute performance gains. Latency speedup is computed relative to the vanilla baseline.

Execution Accuracy (EX). We report execution accuracy following prior Text-to-SQL benchmarks [18], [19], where a generated SQL query is considered correct if its execution result matches the ground-truth result set. This metric captures functional correctness regardless of syntactic variations.

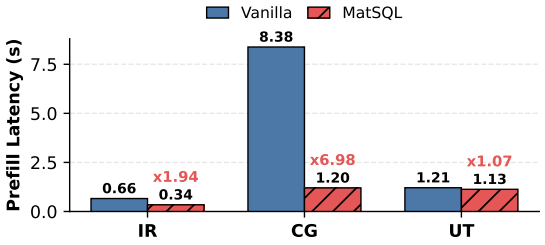
B. Performance Evaluation

1) End-to-End Latency: We measure the average end-to-end latency on the SDS split under the IR→CG→UT workflow, as summarized in Table II. MatSQL reduces the total execution time from 496.7s to 468.2s. This corresponds to a 1.06× end-to-end latency improvement achieved solely through system-level optimization, without altering the logical workflow.

Table III details the latency contribution of each workflow stage. The performance gain is primarily concentrated in the Candidate Generation (CG) phase, which accounts for 77.8% of the total runtime. In this phase, MatSQL achieves a 1.10× speedup. This improvement stems from the architectural characteristics of the CG stage, where the system repeatedly invokes the model with an identical context to generate diverse SQL candidates. Unlike the vanilla baseline which re-encodes these static prefixes at every invocation, MatSQL eliminates this redundant computation by reloading persistent states.



(a) Latency per LLM Inference



(b) Prefill latency

Fig. 3: Step-wise Latency Breakdown.

2) Accuracy Preservation: Table II reports the execution accuracy (EX) on the SDS workload. MatSQL achieves an accuracy of 53.06%, comparable to the 53.74% of the vanilla baseline. The marginal difference falls within the expected variance of stochastic decoding processes. This result confirms that reusing materialized KV states preserves the generation quality. MatSQL neither modifies the prompt text nor relies on approximate retrieval techniques. The system loads exact KV tensors computed by the same model for the identical prefix, ensuring the generation continues from a mathematically equivalent state.

3) Latency Breakdown: To isolate the source of the performance gains, we analyze the latency at the granularity of a single LLM inference. While Table III presents the cumulative latency aggregated over all model calls within a stage, Figure 3 reports the average latency for a single inference operation.

Figure 3(a) depicts the total latency per LLM inference, encompassing both the prefill and decode phases. The Candidate Generation (CG) stage exhibits a 1.10× reduction per inference. Note that the speedup appears modest in the total inference time because the generation (decode) phase accounts for a significant portion of the latency, which remains unaffected by our prefill optimization.

Figure 3(b) further dissects the latency to focus exclusively on the prefill phase. This breakdown reveals that the speedup is driven by the elimination of redundant prefix processing. The CG stage achieves a 6.98× speedup in the prefill phase (8.38s to 1.20s), and the IR stage sees a 1.94× improvement (0.66s to 0.34s). The Unit Testing (UT) stage shows limited gains as its input context is dominated by dynamic feedback rather than static schemas. Note that the reported latency for MatSQL excludes the I/O overhead for fetching KV blocks from the SSD. This fetch latency remains below 100ms per stage, proving negligible compared to the computational savings.

V. Conclusion

This paper presented MatSQL, a system that mitigates prefill redundancy in Text-to-SQL pipelines where large invariant prompt segments are redundantly processed across multi-step inference workflows. MatSQL materializes reusable prefix KV states in a storage-backed hierarchy, decoupling effective cache capacity from volatile memory limits, and incorporates operational mechanisms for schema evolution and cache lifecycle management. When integrated into CHESS-SQL, MatSQL achieves up to a $6.98\times$ prefill speedup in the most iteration-intensive stage and reduces end-to-end latency by $1.06\times$ without compromising accuracy, demonstrating storage-backed KV materialization as a practical solution for context-intensive LLM-based Text-to-SQL systems.

Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2024-00414981). This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00454666,Developing a Vector DB for Long-Term Memory Storage of Hyperscale AI Models)

References

[1] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, "RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2020, pp. 7567–7578.

[2] D. Jha, L. Ward, Z. Yang, C. Wolverton, I. Foster, W.-k. Liao, A. Choudhary, and A. Agrawal, "IRNet: A general purpose deep residual regression framework for materials discovery," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min. (KDD)*, 2019, pp. 2385–2393.

[3] X. V. Lin, R. Socher, and C. Xiong, "Bridging Textual and Tabular Data for Cross-Domain Text-to-SQL Semantic Parsing," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 4870–4888. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.438/

[4] T. Scholak, N. Schucher, and D. Bahdanau, "PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2021, pp. 9895–9901. [Online]. Available: https://aclanthology.org/2021.emnlp-main.779/

[5] J. Li, B. Hui, R. Cheng, B. Qin, C. Ma, N. Huo, F. Huang, W. Du, L. Si, and Y. Li, "Graphix-T5: Mixing pre-trained transformers with graph-aware layers for text-to-SQL parsing," in *Proc. AAAI Conf. Artif. Intell.*, vol. 37, no. 11, 2023, pp. 13076–13084.

[6] S. Talaei, M. Pourreza, Y.-C. Chang, A. Mirhoseini, and A. Saberi, "CHESS: Contextual Harnessing for Efficient SQL Synthesis," in ICML 2025 Workshop on Multi-Agent Systems in the Era of Foundation Models, 2025. [Online]. Available: https://arxiv.org/abs/2405.16755

[7] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, "Text-to-SQL empowered by large language models: A benchmark evaluation," *Proc. VLDB Endow.*, vol. 17, no. 5, pp. 1132–1145, 2024, doi:10.14778/3641204.3641221.

[8] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction," Adv. Neural Inf. Process. Syst., vol. 36, pp. 36339–36348, 2023.

[9] H. Zhang, R. Cao, L. Chen, H. Xu, and K. Yu, "ACT-SQL: In-Context Learning for Text-to-SQL with Automatically-Generated Chain-of-Thought," in Findings of the Association for Computational Linguistics: EMNLP 2023, pp. 3501–3532, 2023.

[10] R. Sun, S. O. Arik, A. Muzio, L. Miculicich, S. K. Gundabathula, P. Yin, H. Dai, H. Nakhost, R. Sinha, Z. Wang, and T. Pfister, "SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL," *Trans. Mach. Learn. Res. (TMLR)*, 2024. [Online]. Available: https://openreview.net/forum?id=rlloVZoKrX

[11] X. Dong, C. Zhang, Y. Ge, Y. Mao, Y. Gao, J. Lin, D. Lou, *et al.*, "C3: Zero-shot text-to-SQL with ChatGPT," arXiv:2307.07306, 2023. [Online]. Available: https://arxiv.org/abs/2307.07306

[12] K. Maamari, F. Abubaker, D. Jaroslawicz, and A. Mhedhbi, "The Death of Schema Linking? Text-to-SQL in the Age of Well-Reasoned Language Models," in NeurIPS 2024 Workshop on Table Representation Learning, 2024. [Online]. Available: https://arxiv.org/abs/2408.07702

[13] Y. Chung, G. T. Kakkar, Y. Gan, B. Milne, and F. Özcan, "Is long context all you need? Leveraging LLM's extended context for NL2SQL," *Proc. VLDB Endow.*, vol. 18, no. 8, pp. 2735–2747, 2025.

[14] Y. Xie, X. Jin, T. Xie, M. Matrixmxlin, L. Chen, C. Yu, C. Lei, C. Zhuo, B. Hu, and Z. Li, "Decomposition for enhancing attention: Improving LLM-based text-to-SQL through workflow paradigm," in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 10796–10816.

[15] M. Pourreza, H. Li, R. Sun, Y. Chung, S. Talaei, G. T. Kakkar, Y. Gan, A. Saberi, F. Ozcan, and S. O. Arik, "CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2025.

[16] Y. Gao, Y. Liu, X. Li, X. Shi, Y. Zhu, Y. Wang, S. Li, W. Li, Y. Hong, Z. Luo, J. Gao, L. Mou, and Y. Li, "A Preview of XiYan-SQL: A Multi-Generator Ensemble Framework for Text-to-SQL," arXiv preprint arXiv:2411.08599, 2024. [Online]. Available: https://arxiv.org/abs/2411.08599

[17] V. Shkapenyuk, D. Srivastava, P. Ghane, and T. Johnson, "Automatic Metadata Extraction for Text-to-SQL," arXiv preprint arXiv:2505.19988, 2025. [Online]. Available: https://arxiv.org/abs/2505.19988

[18] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, *et al.*, "Can LLM Already Serve as a Database Interface? A big bench for large-scale database grounded text-to-SQLs," Adv. Neural Inf. Process. Syst., vol. 36, pp. 42330–42357, 2023.

[19] T. Yu *et al.*, "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2018, pp. 3911–3921.

[20] P. B. Chen *et al.*, "BEAVER: An Enterprise Benchmark for Text-to-SQL," in ACL 2025 Workshop on Table Representation Learning (TRL), 2025. [Online]. Available:https://arxiv.org/abs/2409.02038

[21] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating structured queries from natural language using reinforcement learning," arXiv:1709.00103, 2017. [Online]. Available: https://arxiv.org/abs/1709.00103

[22] C.-H. Lee, O. Polozov, and M. Richardson, "KaggleDBQA: Realistic Evaluation of Text-to-SQL Parsers," in Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pp. 2261–2273, 2021. [Online]. Available: https://aclanthology.org/2021.acl-long.176

[23] F. Lei, *et al.*, "Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows," in International Conference on Learning Representations (ICLR), 2025. [Online]. Available: https://arxiv.org/abs/2411.07763

[24] X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Zhang, J. Fan, G. Li, N. Tang, and Y. Luo, "A Survey of Text-to-SQL in the Era of LLMs: Where Are We, and Where Are We Going?" IEEE Trans. Knowl. Data Eng., vol. 37, no. 10, pp. 5735–5754, 2025.

[25] W. Kwon *et al.*, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proc. 29th Symp. Operating Systems Principles (SOSP)*, 2023, pp. 611–626.

[26] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, *et al.*, "SGLang: Efficient execution of structured language model programs," Adv. Neural Inf. Process. Syst., vol. 37, pp. 62557–62583, 2024.

[27] Y. Sheng *et al.*, "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU," in *Proc. 40th Int. Conf. Mach. Learn. (ICML)*, 2023, pp. 31094–31116.

[28] K.-W. Shin, J. H. Park, M. Oh, Y. Jo, J. Do, and S.-W. Lee, "MatKV: Trading compute for flash storage in LLM infer-

ence," arXiv preprint arXiv:2512.22195, 2025. [Online]. Available: https://arxiv.org/abs/2512.22195

[29] Qwen Team, "Qwen2.5 Technical Report," *arXiv preprint arXiv:2412.15115*, 2024. [Online]. Available: https://arxiv.org/abs/2412.15115

[30] T. Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2024 (Poster). [Online]. Available: https://openreview.net/forum?id=mZn2Xyh9Ec

[31] T. Wolf *et al.*, "Transformers: State-of-the-Art Natural Language Processing," in *Proc. EMNLP: Syst. Demonstrations*, 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6/

[32] H. Chase, "LangChain," *GitHub repository*, 2022. [Online]. Available: https://github.com/langchain-ai/langchain