TableCache: Primary Foreign Key Guided KV Cache Precomputation for Low Latency Text-to-SQL

Anonymous ACL submission

Abstract

In Text-to-SQL tasks, existing LLM-based methods often include extensive database schemas in prompts, leading to long context lengths and increased prefilling latency. While user queries typically focus on recurrent table sets—offering an opportunity for KV cache sharing across queries—current inference engines, such as SGLang and vLLM, generate redundant prefix cache copies when processing user queries with varying table orders. To address this inefficiency, we propose precomputing table representations as KV caches offline and querying the required ones online. A key aspect of our approach is the computation of table caches while preserving primary foreign key relationships between tables. Additionally, we construct a Table Trie structure to facilitate efficient KV cache lookups during inference. To enhance cache performance, we introduce a cache management module with a query reranking strategy to improve cache hit rates and a computation loading pipeline for parallelizing model inference and cache loading. Experimental results show that our proposed TableCache achieves up to a 3.62× speedup in Time to First Token (TTFT) with negligible performance degradation.[1]

1 Introduction

The Text-to-SQL task endeavors to translate natural language (NL) queries into precise structured query language (SQL) statements, enabling efficient database execution. By providing a natural language interface, it empowers non-technical users to interact with databases effortlessly, unlocking seamless access to targeted data. Recently, this task has garnered growing interest from both the NLP and database research communities.

Modern Text-to-SQL approaches primarily utilize LLM-based multi-agent systems, where agents process user queries and database schemas (i.e., table metadata) as input, coordinating a series of complex steps to generate the desired SQL statement. Although current LLM-based agents achieve state-of-the-art performance on various benchmarks, they face significant deployment challenges. A major bottleneck arises from embedding extensive table metadata into prompts, leading to longer sequences and substantially increased prefill latency.

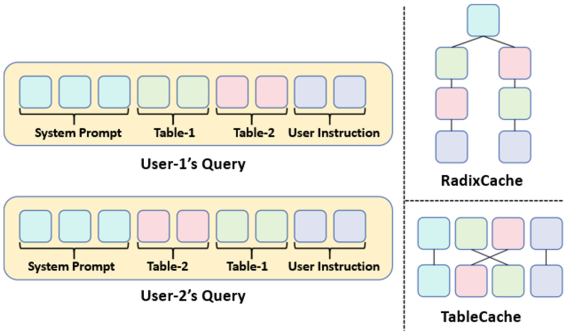

Figure 1: An example of TableCache

To address the high latency commonly associated with LLM inference, advanced inference engines like SGLang (Zheng et al., 2023) and vLLM (Kwon et al., 2023) have become essential tools for improving inference efficiency. These systems enhance LLM deployment mainly through efficient KV cache management and have shown strong performance across a variety of tasks. A key mechanism they rely on is the use of structures such as RadixCache or PrefixCache, which organize previously computed KV cache to enable reuse—though this requires exact prefix matches. However, as shown in Figure 1, the dynamic ordering of tables in user queries often leads to prefix matching failures, compelling the system to produce multiple independent KV cache paths and thereby reducing cache efficiency. Another limitation is that this strategy does not fully leverage distinctive attributes of Text-to-SQL workloads:

[1]The code is now available at https://anonymous.4open.science/r/TableCache

while the static structure of database schemas allows for table-level KV cache reuse, query trends often generate localized data hotspots. Together, these characteristics highlight the potential to accelerate the LLM prefill phase by exploiting shared table information.

Based on these observations, we propose **TableCache**, a method to accelerate LLM inference for Text-to-SQL tasks by precomputing KV caches for tables offline. Specifically, we proactively identify all tables in the database schemas that users may query, represent these tables as KV cache values offline, and store the results in CPU memory. The core idea is to represent tables while preserving primary foreign key relationships, preventing performance degradation caused by the loss of cross-attention when encoding tables individually. To achieve this, we construct a primary foreign key graph from the database, apply topological sorting to determine the sequence of tables, and use causal attention based encoding with LLMs to represent the tables. This approach restores essential relationships between tables, maintaining both efficiency and system performance.

To streamline the online query process, all tables are organized into a trie structure, referred to as the **Table Trie**. When a user query arrives, the system iteratively matches table positions within the Table Trie. Upon finding an exact match for a table, the corresponding KV cache is loaded from CPU memory to GPU memory. The offline KV caches are then concatenated and passed to the LLM for SQL generation.

To further optimize cache performance, we introduce a cache management system that implements CPU-GPU cache eviction and loading policies, such as FIFO and LRU. More importantly, we propose two strategies to enhance system efficiency: (1) a **query reranking strategy** that improves cache hit rates by leveraging user query similarities, and (2) a **computation loading pipeline** that parallelizes model inference with cache loading by asynchronously transferring the KV cache required for subsequent user queries from CPU to GPU during the computation of the current query.

In summary, the contributions of this paper are delineated as follows:

- We propose TableCache, a method that recomputes KV caches for tables while preserving primary foreign key relationships and builds a Table Trie for efficient table matching, ensuring high performance and efficiency for Text-to-SQL LLMs.
- We enhance cache performance with a query reranking strategy to improve hit rates and a computation loading pipeline to parallelize model inference and cache loading.
- Experimental results demonstrate that our proposed TableCache achieves a speedup of up to 3.62× in TTFT while maintaining the original model's performance.

2 Related Work

2.1 KV Cache Management

On the system side, KV cache optimization focuses on memory management and scheduling. vLLM improves memory utilization through paging mechanisms, while both vLLM and SGLang employ PrefixCache to reuse shared prefixes across sequences.

For retrieval-augmented generation(RAG) scenarios, methods like PromptCache (Gim et al., 2023), TurboRAG (Lu et al., 2024), and Block-Attention (Ma et al., 2024) utilize block-based pre-encoding but neglect inter-block attention relationships, causing performance degradation. In response, CacheBlend (Yao et al., 2024), KVLink (Yang et al., 2025), KVZip (Kim et al., 2025), and EPIC (Hu et al., 2024) introduce selective recomputation for more accurate cache reuse. Unlike RAG, Text-to-SQL provides explicit attention reconstruction guidance through primary foreign key, inspiring our primary foreign key guided attention reconstruction mechanism.

On the algorithmic side, research focuses on eliminating redundant or less useful KV cache entries to improve efficiency. Methods such as H2O (Zhang et al., 2023b), QUEST (Tang et al., 2024b), Ada-KV (Feng et al., 2024), RazorAttention (Tang et al., 2024a), snapKV (Li et al., 2024b), CORM (Dai et al., 2024), and PyramidKV (Cai et al., 2024) have designed various KV cache eviction and compensation schemes to accelerate inference. In contrast to them, our TableCache preserves all KV caches to suit the high-reuse setting of Text-to-SQL.

2.2 Text-to-SQL

Mainstream LLM-based approaches for Text-to-SQL fall into two paradigms: prompt engineering and supervised fine-tuning (SFT). Prompt engineering methods often leverage few-shot prompt-

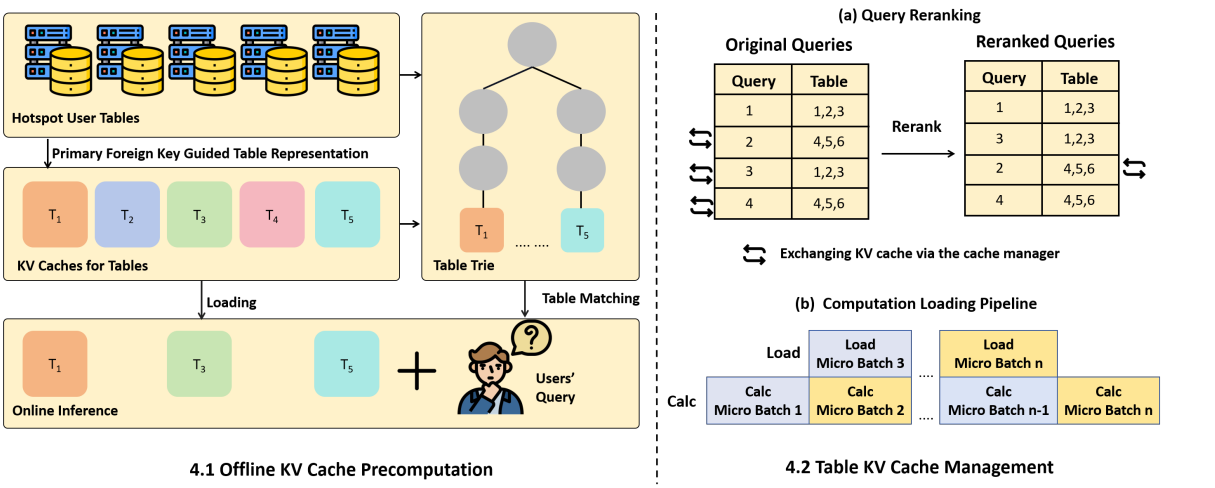

Figure 2: Overview of TableCache. KV caches are precomputed and organized into a hierarchical Table Trie during offline preprocessing. Inference utilizes cache management system, including query reranking for cache-friendly sequencing and a computation loading pipeline to reduce latency and improve efficiency.

ing strategies (Cheng et al., 2023; Nan et al., 2023; Sun et al.; Wang et al., 2025; Zhang et al., 2023a). Examples include DIN-SQL(Pourreza and Rafiei, 2023) and CHESS(Talaei et al., 2024), which decompose the task into subtasks like Schema Linking and SQL Generation with a self-correction mechanism. In contrast, SFT-based approaches (Gao et al., 2024b; Pourreza and Rafiei, 2024), such as CODES (Li et al., 2024a), DAIL-SQL (Gao et al., 2024a), and OmniSQL (Li et al., 2025), focus on fine-tuning the end-to-end Text-to-SQL capabilities of LLMs.

Unlike prior work, our research focuses on enhancing the inference speed of Text-to-SQL models in practical deployment scenarios where they serve end users.

3 Problem Setup

The Text-to-SQL problem can be formally defined as follows: given a set of databases, each consisting of multiple tables, a natural language query q issued on a database D requires retrieving a collection of m tables $T = \{T_1, ..., T_m\}$ to answer q. These tables may include all tables in D or a subset pre-retrieved as relevant to q. The goal is to enable an LLM to generate the corresponding SQL query. In online Text-to-SQL services, user queries are processed in batches to enhance throughput. Importantly, a significant number of queries often access a shared subset of tables—an observation exploited in this work to further optimize the efficiency of Text-to-SQL inference systems.

4 Method

The proposed TableCache framework reduces the latency of inference systems by maximizing the reuse of precomputed table representations across different queries. As illustrated in Figure 2, during the offline preprocessing phase, TableCache precomputes KV caches for all tables based on the primary foreign key graph and organizes them into a hierarchical Table Trie. During the inference phase, TableCache implements a meticulously designed cache management system. It first incorporates three distinct cache eviction strategies to manage the KV cache on the GPU. Simultaneously, to reduce cache switching overhead, the system performs query reranking upon receiving batched user requests, reordering the request sequence into a cache-friendly order. Furthermore, a computation loading pipeline is introduced to hide the latency of cache switching, thereby further enhancing overall efficiency.

4.1 Offline KV Cache Precomputation

During the offline KV cache precomputation phase, TableCache aims to construct table-level KV cache information for practical inference usage. To address the performance degradation caused by independent table encoding, inspired by database schemas, we designed a **Primary Foreign Key Guided Table Representation.** This approach leverages foreign keys as guidance to reconstruct inter-table attention relationships, thereby preserving model performance. Following this, to efficiently utilize preprocessed table information and

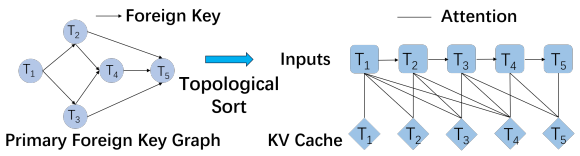

Figure 3: Overview of Primary Foreign Key Guided Table Representation.

KV caches during actual inference, TableCache organizes all table information into a **Table Trie** that remains loaded in memory. During inference, multiple matching operations are performed on this trie structure to retrieve all necessary KV caches.

4.1.1 Primary Foreign Key Guided Table Representation.

As shown in Figure 3, we leverage the inherent characteristics of Text-to-SQL tasks to facilitate primary foreign key guided table representation. Specifically, foreign keys often represent relationships between tables in a database. Therefore, if attention between tables linked by foreign keys is lost during attention modeling, it can increase model confusion. This insight inspired us to construct a primary foreign key graph of the original database. Specifically, we first abstract the tables in the database as nodes in the graph, and then for each table, we create edges from the tables referenced by its foreign keys to itself. By performing topological sorting on this graph, we ensure that nodes later in the topological order necessarily depend on preceding nodes. This dependency allows connected nodes to be jointly encoded offline. Through this joint encoding approach, we restore the most essential attention relationships between tables at minimal cost.

Moreover, it is noteworthy that, consistent with findings from prior studies (Lu et al., 2024; Ma et al., 2024), the proper implementation of positional encoding is crucial for maintaining the performance of independently encoded KV cache. Therefore, during the offline precomputation phase, TableCache first stores the KV cache information without positional encoding. During actual inference, it recalculates and applies positional encoding based on the concatenated KV cache.

4.1.2 Table Trie Construction

After constructing the primary foreign key guided table representation, all table information is organized into a trie structure called **Table Trie**. Specifically, the contents (i.e., table name, column names/descriptions) of each table are inserted into the Table Trie independently. The leaf nodes of the Table Trie store the table ID and storage path to its corresponding KV cache.

Algorithm 1: Table Trie Matching

Input: s: input token sequence
Input: $trie$: pre-built Table Trie
Input: $cache$: KV cache manager

1: **Function** `Match`(T, $trie$, $cache$)**:**
2: $p \leftarrow 0$; `// current start position`
3: $H \leftarrow \emptyset$;
4: **while** $p < l_s$ **do**
5: $(found, next, pos) \leftarrow trie.query(T, p)$;
6: **if** $found$ **then**
7: $kv \leftarrow cache.get(pos)$;
8: $H.add(kv)$;
9: $p \leftarrow next$; `// Next table's start position`
10: **else**
11: $p \leftarrow p + 1$;
12: **return** H;

After the Table Trie is built, during the inference stage, the system identifies all tables in a given query by performing continuous matching within the Table Trie. As shown in the Algorithm 1, the system finds the longest prefix match between the query and the Table Trie (Line-5) for each match and retrieves the corresponding KV cache from the matched node (Line-7). The matched prefix is then removed from the query (Line-9), and the process is repeated until no further matches can be found. All relevant tables associated with the query are collected through this iterative approach. Finally, the KV caches of these tables are concatenated and fed into the LLM for subsequent inference.

Below we analyze its corresponding time complexity. Assuming the sequence length is l_s and the length matched by the Trie each time is l_m, each matching operation requires $O(l_m)$ times. Assuming the starting position of the match is p, after each match, the pointer marking the end position jumps to $p + l_m$, meaning at most $O(\frac{l_s}{l_m})$ matches can occur. Therefore, the final amortized time complexity is $O(\frac{l_s}{l_m}) * O(l_m) = O(l_s)$. Therefore, by utilizing Table Trie, we achieve efficient table information matching, laying a solid foundation for subsequent inference.

4.2 Table KV Cache Management

Since TableCache performs offline precomputation and storage of the table's KV cache, the inference speed bottleneck during the actual prefilling phase shifts from computation to memory access. To address this bottleneck, TableCache meticulously designs a KV cache Management System. Inspired by traditional cache management algorithms, we have implemented various GPU cache eviction strategies—including First-In-First-Out (FIFO), Least Recently Used (LRU), and Least Frequently Used (LFU)—which effectively identify hotspots in user queries to enable efficient KV cache management. The specific implementations of these three types of caches are presented in the Appendix A.

However, in practical deployment scenarios, the input order of user queries significantly impacts the frequency of cache eviction algorithm switching. As illustrated in Figure 2(a), when the cache capacity is 3, the original sequence requires evicting Tables 1, 2, and 3 to load Tables 4, 5, and 6 while processing user query 2. Subsequently, when processing user query 3, it needs to evict Tables 4, 5, and 6 again, resulting in frequent cache replacements. In contrast, the reordered sequence only needs to load Tables 4, 5, and 6 once when processing user query 2. This demonstrates that cache-unfriendly input sequences can substantially increase cache switching overhead, ultimately leading to a sharp decline in system performance. To address this issue, we introduce a query reranking strategy into the cache management system, which reorganizes user input batches into memory-friendly sequences.

Meanwhile, to fully leverage GPU and CPU resources, TableCache designs a computation loading pipeline based on its cache precomputation mechanism. While the GPU performs computations, it prefetches the table KV caches required for future requests in parallel, ultimately hiding the additional latency caused by memory access within computation latency and achieving further reduction in overall latency.

4.2.1 Query Reranking

The objective of Query Reranking is to minimize the frequency of KV cache switches. To this end, we adopt a greedy approach. Specifically, the process begins by randomly selecting an initial query from the batch as a reference point. Then, from the remaining queries, we iteratively select the one whose associated tables exhibit the highest similarity to those of the current query, appending it to the sorted sequence and updating it as the new reference point. By repeating this process for the batch size number of times, we obtain a sequence that minimizes the number of switches.

To define the similarity between queries, we first match each query against the Table Trie to identify the associated set of table IDs, denoted as τ_i. Next, we define a binary array I_i of length n (where n represents the total number of tables across all given databases) for query i, where each

$$I_i[k] = \begin{cases} 1, & \text{if } k \in \tau_i \\ 0, & \text{if } k \notin \tau_i \end{cases}. \tag{1}$$

Next, we define the distance between two binary strings I_i and I_j as $d_{i,j}$.

$$d_{i,j} = \sum_{k=1}^{n} (I_i \oplus I_j)[k]. \tag{2}$$

where $\oplus$ denotes the binary XOR operation. This distance metric quantifies the difference in the sets of tables associated with query i and query j, thereby measuring their similarity. Based on this, at each step, we select the query with the minimal distance relative to the current reference point.

To ensure computational efficiency, we implement this module using a bitset in C++. The final time complexity of this procedure is $O(\frac{N^2M}{64})$, where N is the number of queries and M is the number of tables involved. Since for each pair (i, j), the time complexity of computing the $\oplus$ operation is $O(M)$, this can be optimized to $O(\frac{M}{64})$ on a 64-bit machine by utilizing a bitset. There are a total of N^2 (i,j) pairs, resulting in a final time complexity of $O(\frac{N^2M}{64})$.

4.2.2 Computation Loading Pipeline

In practice, the inference process of LLMs can primarily be decomposed into two stages: computation and loading. Since TableCache pre-computes the KV cache for each table, the computation and cache loading for the TableCache can proceed in parallel. Subsequently, TableCache divides the bs queries in a batch into smaller micro-batches of size b_c and b_m, where b_c represents the compute micro-batch and b_m the memory micro-batch. During inference, TableCache computes the current b_c requests while simultaneously prefetching the table information and KV cache required for the next b_m requests onto the GPU, thus achieving pipeline parallelism.

Compared to other KV cache management approaches, our solution fundamentally decouples computation from memory access during inference. In contrast, conventional methods rely on real-time GPU computation to obtain the KV cache, thereby making prefetching impossible and consequently failing to maximize memory performance. By leveraging this parallelization, we maximize the utilization of both GPU and CPU resources, achieving significant acceleration for the prefill phase.

5 Experiments

5.1 Experiment Setup

Datasets. To evaluate the effectiveness of our method, we conducted experiments on two text-to-SQL benchmarks: BIRD (Li et al., 2023)and Spider (Yu et al., 2018).

BIRD (Li et al., 2023) is a Text-to-SQL benchmark comprising a total of 12,751 questions, spanning 37 different specialized domains, emphasizing the model's ability to integrate and handle a wide range of external knowledge.

Spider (Li et al., 2023) is a large-scale, cross-domain Text-to-SQL dataset designed to assess the generalization capabilities across various databases and query structures.

To better simulate a realistic online Text-to-SQL scenario, we randomized the order of data samples in the dataset as well as the table order provided for each sample.

Baselines. We use the following models for comparison with our method. The term "baseline" refers to the approach where no inference acceleration methods are employed, and inference is conducted directly. For mainstream inference engines, we have chosen to compare with SGLang (Zheng et al., 2025) and vLLM (Kwon et al., 2023). For both vLLM and SGLang, we employed the radix-cache for a comparative analysis. Most existing block-cache methods are implemented exclusively on Llama (Touvron et al., 2023). For comparison, we selected PromptCache (Gim et al., 2024) and TurboRAG (Lu et al., 2024) because they provide native support for the Qwen model series. Both methods were adapted to the Text-to-SQL scenario to facilitate an experimental comparison with our approach.

Metrics. Following previous research (Lu et al., 2024; Chan et al., 2024; Yao et al., 2024), we primarily adopted **t**otal delay **t**ime to **f**irst **t**oken (TTFT/s) across the entire test set and the Execution Accuracy (**Acc.**) of the generated SQL statements as our metrics.

5.2 Implementation Details

We extracted the tables referenced in all SQL queries for each benchmark, computed their KV caches, and organized them into our offline Table Trie. For the backbone models, we used OmniSQL-7B (Li et al., 2025), a specialized fine-tuned Text-to-SQL model, and Qwen2.5-7B-Coder (Yang et al., 2024), a general-purpose code LLM, as representatives of mainstream Text-to-SQL backbones. In addition to directly inferring with the backbone models, we also created trained versions.

Offline KV caches introduce block-wise attention by preserving intra-table and inter-table token attention for tables connected via primary foreign key relationships. This contrasts with the original full attention KV caches, which span all input table tokens. To adapt backbone models previously trained on full attention to TableCache's block-wise KV caches, we introduced an attention mask in the training process to restrict attention to tokens within individual tables[2]. For fair comparison, the backbones used with vLLM and SGLang were also trained, but without the specialized attention mask.

As the goal of training was to adapt the models to TableCache, we trained the backbones exclusively on longer and more complex tables from the $\text{BIRD}_{\text{train}}$ dataset. Training was conducted over three epochs, with additional details provided in Appendix B. For the Computation Loading Pipeline, we set $b_c = 100$ and $b_m = 10$. All experiments were performed on a single NVIDIA A800 GPU. The prompt template used for model inference is detailed in Appendix C.

5.3 Main Results

Table 1 presents the following findings:

TableCache Achieves Outstanding Latency Optimization. In the experiments conducted on OmniSQL-7B and Qwen2.5-7B-coder, our proposed TableCache method significantly outperformed the existing baseline. The experimental

[2]For engineering convenience, inter-table attention based on primary foreign key relationships was omitted during training. The shift from full to block-level attention remains, with blocks representing either a single table or a group of connected tables.

Method	Model	$\text{Spider}_{\text{dev}}$		BIRD_{dev}	
		TTFT (s)	Acc. (%)	TTFT (s)	Acc. (%)
Baseline	OmniSQL-7B	97.51	78.3	291.26	61.2
vLLM		61.17	77.4	159.88	60.4
SGLang		53.16	77.9	128.49	61.0
PromptCache		55.47	66.2	85.01	49.7
TurboRAG		81.47	72.1	205.43	51.0
TableCache		**36.22**	76.9	**80.53**	58.1
Baseline	OmniSQL-7B *(trained)*	78.30	78.3	296.42	61.5
vLLM		68.72	77.5	177.36	61.5
SGLang		72.23	78.6	199.85	61.3
TableCache		**42.43**	77.1	**90.39**	59.9
Baseline	Qwen2.5-7B-coder	98.13	73.5	294.83	51.0
vLLM		75.38	73.0	233.74	48.2
SGLang		73.21	72.4	223.58	50.5
PromptCache		52.36	51.4	133.47	25.4
TurboRAG		77.40	56.0	203.11	29.2
TableCache		**42.01**	72.0	**115.36**	42.9
Baseline	Qwen2.5-7B-coder *(trained)*	74.81	73.2	301.35	54.3
vLLM		66.08	71.5	207.49	54.1
SGLang		62.16	72.3	177.43	54.1
TableCache		**41.27**	72.2	**117.22**	53.3

Table 1: The results of the comparison methods presented on two backbone models across two benchmarks. The baseline represents direct inference using the Transformers library, while the results for both vLLM and SGLang are obtained with radixcache enabled. Boldface indicates the best performance. "Trained" signifies that the backbone models were further fine-tuned on the $\mathbf{BIRD_{train}}$ dataset.

results demonstrate that, across two distinct benchmarks, our method achieves up to a 2.69× and 3.62× reduction in TTFT compared to directly using the transformers library for inferecne (i.e, Baseline in Table 1). Moreover, our method demonstrates an even higher acceleration ratio on BIRD, a longer and more challenging benchmark compared to Spider, surpassing inference engines like vLLM and SGLang that employ CUDA kernel level optimizations. This indicates that in Text-to-SQL scenarios involving multiple tables that may be shared across users, with potentially random table orders, the proposed TableCache is more adaptable to these scenarios, effectively reducing inference latency during the prefill phase.

TableCache Achieves Near-Lossless Performance in Text-to-SQL Models. We find that TableCache integrates seamlessly into OmniSQL-7B without requiring training and with minimal performance degradation. However, applying TableCache to Qwen2.5-7B-Coder, a model not pretrained on the Text-to-SQL task, leads to a performance decline due to its limited task understanding. As a result, the primary foreign key guided table representation fails to provide adequate guidance for accurate answers. Training on a Text-to-SQL dataset is therefore essential. Once fine-tuned with the block-wise attention introduced in Section 5.2, Qwen2.5-7B-Coder with TableCache achieves near-lossless performance (53.3% vs. 54.3%).

5.4 Ablation Study

To further analyze the effectiveness of the proposed modules, we conducted an ablation study on the individual components that constitute TableCache, and the results are presented in Table 2 and Table 3. From the ablation study results, it is evident that the integration of each module significantly contributes to the improvement of the overall TTFT and Accuracy metric.

Ablation of Primary Foreign Key Guided Table Representation. Table 3 shows that failing to reconstruct the relationships between tables with primary foreign key connections leads to significant performance degradation on both datasets, with an even more pronounced drop observed on the more challenging BIRD dataset. This result further supports our assertion that, in real-world database scenarios, inter-table relationships are primarily governed by foreign key joins—relationships that

Dataset	w/o Cache Mgmt.	w/o Query Rerank	w/o Comp. Load.	TableCache
$\text{Spider}_{\text{dev}}$	99.346	58.333	38.549	**36.229**
Bird_{dev}	212.626	114.529	86.842	**80.526**

Table 2: Ablation studies on cache management, query ranking, and compute loading pipelines on $\text{Spider}_{\text{dev}}$ and BIRD_{dev}, using OmniSQL-7B, with TTFT measured in seconds.

Dataset	w/o PFTR	TableCache
Spider	72.0	**76.9**
Bird	50.9	**58.1**

Table 3: Ablation study on the primary foreign key guided table representation, reported in terms of Accuracy. The backbone model is OmniSQL-7B. PFTR refers to Primary Foreign key guided Table Representation.

Cache Eviction Strategy	TTFT (s)
LFU	36.85
FIFO	36.41
LRU	**36.22**

Table 4: Ablation studies on cache eviction strategies on $\text{Spider}_{\text{dev}}$ using OmniSQL-7B.

are indispensable for solving complex Text-to-SQL tasks. A typical example illustrating the importance of primary foreign key guided table representation is provided in Appendix D.

Ablation of Table KV Cache Management. The results are presented in Table 2. As the experimental results demonstrate, the absence of this component leads to a significant increase in TTFT. This indicates that without the table KV cache management system, the direct transfer of table caches from the CPU to the GPU, followed by immediate offloading after inference, incurs substantial memory overhead, ultimately causing severe degradation in system efficiency.

To further validate the table KV cache management system, we conducted an ablation study on the $\text{Spider}_{\text{dev}}$ dataset using three caching strategies: LRU, LFU, and FIFO. As shown in Table 4, these strategies had minimal impact on prefill efficiency. This is because (1) clear hotspot patterns in user behavior are easily captured without complex policies, and (2) the query reranking module concentrates hot data in short time windows, minimizing unnecessary cache eviction.

Ablation of Query Reranking. The objective of Query Reranking is to minimize memory exchanges between the CPU and GPU. As shown in Table 2, disabling reranking significantly increases the system's TTFT. This occurs because disordered user inputs disrupt table access patterns, making it challenging for cache eviction policies to identify hot data. As a result, excessive swapping operations are required, leading to a notable decline in overall system efficiency.

Ablation of Computation Loading Pipeline. Unlike other modules designed to reduce cache swapping frequency, this module leverages TableCache's KV cache precomputation capability, which decouples computation from memory access, to optimize performance. As shown in Table 2, disabling the computation loading pipeline results in a slight increase in TTFT. This indicates that the module effectively balances the overhead of computation and memory access by masking the additional latency from memory operations within computation latency, thereby enhancing overall system performance.

6 Conclusion

This paper highlights that in text-to-SQL scenarios, current inference engines like SGLang and vLLM fail to efficiently utilize KV caches from frequently queried tables, resulting in substantial prefill overhead and increased request latency. To address this issue, we propose TableCache, which optimizes performance by reusing hot-table KV caches—precomputed table representations guided by primary foreign key graphs—and integrating a custom cache management system with a query ranking mechanism and computation loading pipeline. TableCache significantly reduces Time to First Token (TTFT) while preserving model accuracy. Experimental results demonstrate that TableCache achieves the lowest TTFT across multiple text-to-SQL benchmarks.

7 Limitations

While this work has focused on applying TableCache in the text-to-SQL scenario, future research could explore integrating TableCache with other graph retrieval methods to reconstruct attention edges, enabling faster reasoning in broader domains. Additionally, the modules developed for TableCache can be seamlessly incorporated into existing KV-cache management systems, such as vLLM and SGLang, to further enhance performance.

References

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*, abs/2406.02069.

Brian J Chan, Chao-Ting Chen, Jui-Hung Cheng, and Hen-Hsen Huang. 2024. Don't do rag: When cache-augmented generation is all you need for knowledge tasks. *Preprint*, arXiv:2412.15605.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, and 1 others. 2023. Binding language models in symbolic languages. In *International Conference on Learning Representations (ICLR 2023)(01/05/2023-05/05/2023, Kigali, Rwanda)*.

Jincheng Dai, Zhuowei Huang, Haiyun Jiang, Chen Chen, Deng Cai, Wei Bi, and Shuming Shi. 2024. Corm: Cache optimization with recent message for large language model inference. *arXiv preprint arXiv:2404.15949*.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. 2024. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. *arXiv preprint arXiv:2407.11550*.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024a. Text-to-sql empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145.

Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, and 1 others. 2024b. Xiyan-sql: A multi-generator ensemble framework for text-to-sql. *arXiv preprint arXiv:2411.08599*.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338.

In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2023. Prompt cache: Modular attention reuse for low-latency inference. *ArXiv*, abs/2311.04934.

Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. 2024. Epic: Efficient position-independent caching for serving large language models. *arXiv preprint arXiv:2410.15332*.

Jang-Hyun Kim, Jinuk Kim, Sangwoo Kwon, Jae W Lee, Sangdoo Yun, and Hyun Oh Song. 2025. Kvzip: Query-agnostic kv cache compression with context reconstruction. *arXiv preprint arXiv:2505.23416*.

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 *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, and 1 others. 2025. Omnisql: Synthesizing high-quality text-to-sql data at scale. *arXiv preprint arXiv:2503.02240*.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024a. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357.

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

Songshuo Lu, Hua Wang, Yutian Rong, Zhi Chen, and Yaohua Tang. 2024. Turborag: Accelerating retrieval-augmented generation with precomputed kv caches for chunked text. *arXiv preprint arXiv:2410.07590*.

Dongyang Ma, Yan Wang, and Lan Tian. 2024. Block-attention for efficient prefilling. *arXiv preprint arXiv:2409.15355*.

Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. 2023. Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies. *arXiv preprint arXiv:2305.12586*.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: decomposed in-context learning of text-to-sql with self-correction. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pages 36339–36348.

Mohammadreza Pourreza and Davood Rafiei. 2024. Dts-sql: Decomposed text-to-sql with small large language models. In *EMNLP (Findings)*.

Ruoxi Sun, Sercan O Arik, Alexandre Muzio, Lesly Miculicich, Satya Kesav Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and 1 others. Sql-palm: Improved large language model adaptation for text-to-sql. *Transactions on Machine Learning Research*.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*.

Hanlin Tang, Yang Lin, Jing Lin, Qingsen Han, Shikuan Hong, Yiwu Yao, and Gongyi Wang. 2024a. Razorattention: Efficient kv cache compression through retrieval heads. *arXiv preprint arXiv:2407.15891*.

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

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and 1 others. 2025. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. 2025. Kvlink: Accelerating large language models via efficient kv cache reuse. *arXiv preprint arXiv:2502.16002*.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2024. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. *Preprint*, arXiv:2405.16444.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023a. 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*, pages 3501–3532.

Zhenyu (Allen) 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. 2023b. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *ArXiv*, abs/2306.14048.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody_Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, and 1 others. 2023. Efficiently programming large language models using sglang.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, and 1 others. 2025. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583.

A Details Of Cache Manager

Algorithm 2: LRU Cache Replacement

Input: $cache$: the cache with a fixed capacity C
Input: $request$: the requested table
Function `Update`($request$)**:**
  if $request \in cache$ **then**
    Move $request$ to the **front**;
  else
    if $|cache| \geq C$ **then**
      Evict the item at the **end**;
    Insert $request$ at the **front**;

Algorithm 3: FIFO Cache Replacement

Input: $cache$: a queue with a fixed capacity C
Input: $request$: the requested item
Function `Update`($request$)**:**
  if $request \notin cache$ **then**
    if $|cache| \geq C$ **then**
      Evict the item at the **head** of the queue;
    Enqueue $request$ to the **tail** of the queue;

Algorithm 4: LFU Cache Replacement

Input: $cache$: a dictionary mapping item to (frequency, timestamp)
Input: $request$: the requested item
Input: Current $timestamp$
Function `Update`($request$)**:**
  if $request \in cache$ **then**
    $cache[request]$.frequency $\leftarrow$ $cache[request]$.frequency $+ 1$;
    $cache[request]$.timestamp $\leftarrow$ $timestamp$;
  else
    if $|cache| \geq C$ **then**
      Find the item x with the **smallest** $frequency$;
      (Break ties by selecting the smallest $timestamp$**)**;
      Evict x from the $cache$;
    Insert $cache[request] \leftarrow (1, timestamp)$;

B Training Details

During training, the TableCache mechanism utilized the entire $\text{BIRD}_{\text{train}}$ dataset. The model was trained for 3 epochs with a learning rate of 1e-6, a warmup ratio of 0.03, and the Adam optimizer. For the training of TableCache, we aimed to enhance the model's adaptation to the Primary Foreign Key Guided Table Representation by familiarizing it with table-level KV cache behavior. To achieve this, for each data sample in the training set, we first extracted all involved tables and segmented the original input based on table positions. For each individual table, we designed a dedicated attention mask to ensure that during computation, it could only attend to its own content and not access information from other tables. Through this training approach, we further enabled the model to adapt to table-level KV cache, thereby improving model performance.

C Prompt Template for Model Inference

Prompt Template for model inference

<tables>
Task Overview:
You are a data science expert. Below, you are provided with a database schema and a natural language question. Your task is to understand the schema and generate a valid SQL query to answer the question.
Database Engine:
SQLite
This schema describes the database's structure, including tables, columns, primary keys, foreign keys, and any relevant relationships or constraints.
Question:
<Question>
Instructions:
- Make sure you only output the information that is asked in the question. If the question asks for a specific column, make sure to only include that column in the SELECT clause, nothing more.
- The generated query should return all of the information asked in the question without any missing or extra information.

- Before generating the final SQL query, please think through the steps of how to write the query.
Output Format:
In your answer, please enclose the generated SQL query in a code block:
```sql
– Your SQL query
```
Take a deep breath and think step by step to find the correct SQL query.

D Case Study

In this section, we analyze the limitations of pure block-wise attention—which processes tables independently without primary foreign key guided table representation—through a representative case study, and further demonstrate the effectiveness of our proposed primary-foreign-key-guided table representation method.

Question: Exclusively virtual refers to Virtual = 'F'. How many schools with an average score in Math greater than 400 in the SAT test are exclusively virtual?

TableCache:

```
SELECT COUNT(s.CDSCode)
FROM schools AS s
INNER JOIN satscores AS ss ON s.
    CDSCode = ss.cds
WHERE s.Virtual = 'F' AND ss.
    AvgScrMath > 400;
```

Without primary foreign key graph:

```
SELECT COUNT(*)
FROM frpm
WHERE Virtual = 'F' AND AvgScrMath >
     400;
```

Ground truth:

```
SELECT COUNT(DISTINCT T2.School)
FROM satscores AS T1
INNER JOIN schools AS T2 ON T1.cds =
     T2.CDSCode
WHERE T2.Virtual = 'F' AND T1.
    AvgScrMath > 400;
```

From the above example, we can observe that without the primary foreign key guided table representation, although the model correctly comprehends the query content, it fails to capture the relationship between the satscores and schools tables. These two tables are in fact connected by the CDSCode foreign key, but this critical information is entirely lost when using independent block-wise encoding. Consequently, the model erroneously attempts to retrieve required information from the frpm table.

Our dataset analysis reveals that in the absence of primary foreign key guided table representation, a substantial proportion of erroneous model inferences exhibit precisely this type of failure, constituting the predominant category of discrepancy cases. This finding further validates both the motivation behind and effectiveness of our proposed approach.