# 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 precompute table representations as offline KV caches and query only the required caches online, while preserving primary–foreign key relationships during cache construction and using a Table Trie to enable efficient cache lookup at inference time. To enhance cache performance, we introduce a cache management system 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.

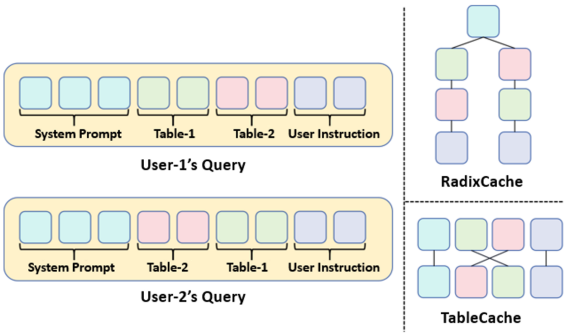


Figure 1: An example of TableCache

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: while the static structure of database schemas allows for table-level KV cache reuse, query trends

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

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. 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., 2023), QUEST (Tang et al., 2024), Ada-KV (Feng et al., 2024), snapKV (Li et al., 2024b), 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 prompting strategies (Cheng et al., 2023; Nan et al., 2023; Zhang et al., 2026). Examples include DIN-SQL(Pourreza and Rafiei, 2023) and CHESS(Talaei et al., 2024), which decompose the

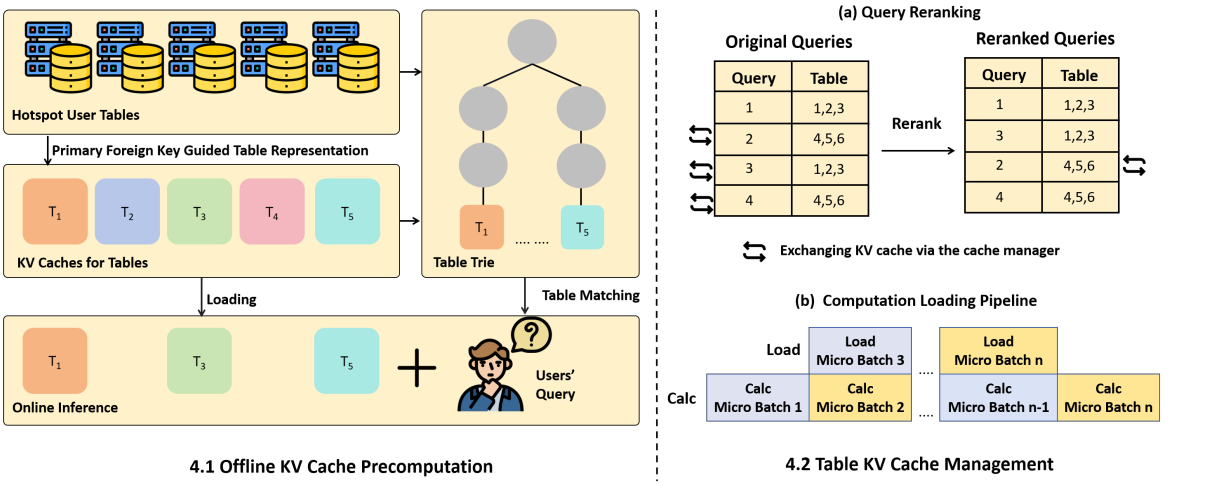


Figure 2: Overview of TableCache. KV caches are precomputed and organized into a hierarchical Table Trie during offline preprocessing. Inference utilizes cache management system, including query reranking for cache-friendly sequencing and a computation loading pipeline to reduce latency and improve efficiency.

task into subtasks like Schema Linking and SQL Generation with a self-correction mechanism.

Unlike SFT approaches (Gao et al., 2024b; Pourreza and Rafiei, 2024; Li et al., 2024a; Gao et al., 2024a; Li et al., 2025) that optimize Text-to-SQL capabilities, we focus on accelerating inference speed to efficiently serve end users in practice.

## 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 systems, batched query processing is standard for achieving high throughput. This work exploits the frequent access to shared tables within these batches to further optimize LLM inference efficiency.

## 4 Method

TableCache accelerates inference through extensive reuse of precomputed KV caches. As shown in Figure 2, it features an offline hierarchical Table Trie built upon primary-foreign key graphs. For online inference, it integrates query reranking with efficient cache management to minimize switching overhead, alongside a computation pipeline that overlaps memory access to optimize latency.

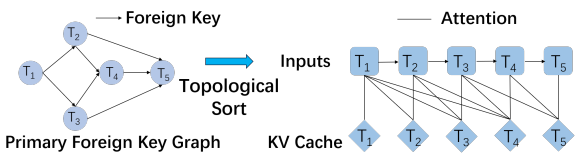


Figure 3: Primary Foreign Key Guided Table Representation.

### 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. Specifically, we independently compute and store KV caches for individual tables, which are subsequently concatenated during inference. This process transforms the standard global causal attention mask into a series of isolated lower-triangular blocks, inevitably severing inter-block attention dependencies and leading to performance degradation. To address this, we introduce a **Primary Foreign Key Guided Table Representation** that leverages schema links to reconstruct inter-table attention. For efficient inference, TableCache organizes these precomputed caches into a memory-resident **Table Trie**, enabling rapid retrieval of necessary KV caches through structural matching.

#### 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 relation-

**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** $H$;

ships 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 (treating isolated ones as zero-degree) in the graph, and then for each table, we create edges from the tables referenced by its foreign keys to itself. Leveraging the inherent acyclic nature of standard schemas, we apply topological sorting on this graph, ensuring 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. Subsequently, during the inference phase, TableCache dynamically reapplies positional encodings to each token based on its global position within the concatenated sequence.

#### 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.

After the Table Trie is built, during the inference stage, the system identifies all tables required by 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.

### 4.2 Table KV Cache Management

Since TableCache precomputes KV caches offline, the prefilling bottleneck shifts from computation to memory access. To address this, we designed a KV cache management system using GPU eviction strategies (FIFO, LRU, and LFU) to efficiently identify query hotspots. This system triggers updates exclusively when a request targets a non-cached table, necessitating a swap-in operation where the eviction candidate is selected by the cache management module. Specific implementations are detailed in Appendix A.

However, in practice, the order of user queries significantly impacts cache eviction frequency. As shown in Figure 2(a), a cache-unfriendly sequence causes frequent, redundant replacements (e.g., repeatedly evicting and reloading Tables 1–6). In contrast, reordering the sequence loads these tables only once. Since high switching overhead severely degrades performance, we introduce a query reranking strategy to reorganize input batches into memory-friendly sequences. As shown in Table 5, the choice of cache eviction policy has a negligible impact on overall performance. This minimal variance indicates that our reranking mechanism successfully clusters queries

accessing the same "hot tables" together. Because these hotspots are grouped in close temporal proximity, even simple cache management strategies can easily identify and retain them.

To maximize resource efficiency, TableCache employs a computation-loading pipeline that parallelizes memory I/O and processing across different batches. By prefetching KV caches for subsequent batches during the GPU computation of the current batch, this strategy effectively hides latency. The details of the query reranking and this pipeline are introduced below.

#### 4.2.1 Query Reranking

The objective of Query Reranking is to minimize the frequency of KV cache switches. To this end, we employ a greedy strategy that iteratively selects from the remaining queries based on maximal table similarity to the immediate predecessor, starting from a random anchor.

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 $\tau_i$. 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.

#### 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.

### 4.3 System Time Complexity

In practical deployment, TableCache's latency is divided into offline and online phases.

The one-time offline computation involves graph construction to extract foreign key dependencies, followed by topological sorting. Since both operations visit each of the $m$ tables at most once, the total offline time complexity is strictly bounded by $O(m)$.

During the online phase, we first rerank the queries within a batch. To maximize efficiency, this module is implemented via a C++ bitset. While natively computing the XOR ($\oplus$) operation for any query pair $(i, j)$ requires $O(m)$ time (where $m$ is the number of tables), utilizing bitset operations on a 64-bit architecture reduces this pair-wise complexity to $O(\frac{m}{64})$. With $O(N^2)$ possible pairs across $N$ queries, the total time complexity for reranking is strictly bounded by $O(\frac{N^2 m}{64})$.

Following this, the latency for each online request can be broken down into three components:

**Table Trie matching overhead:** Assuming the total sequence length is $n$ and the length matched by the Trie at each step is $l_m$, a single matching operation requires $O(l_m)$ time. Because the pointer advances from the starting position $p$ to $p + l_m$ after each successful match, at most $O(\frac{n}{l_m})$ matches will occur. Consequently, the amortized time complexity for this process is bounded by $O(\frac{n}{l_m}) \times O(l_m) = O(n)$.

**Table loading overhead:** Thanks to our proposed computation-loading pipeline, we proactively prefetch the KV caches of upcoming tables during model computation. As a result, this loading overhead is effectively hidden behind the compu-

tation time and can be considered negligible in the final latency breakdown.

**Computation overhead:** This component incurs a time complexity of $O(qn)$, where $q$ represents the length of the user query. This efficiency stems from the fact that the model only needs to compute the KV states for the user's query, which subsequently attends to all preceding cached table content.Summing these components, the total time overhead for a single request is $O(n + qn) = O(qn)$.

In contrast, the standard baseline incurs a time overhead of $O(n^2)$. Because the query length is vastly smaller than the table sequence length ($q \ll n$), TableCache achieves a substantial theoretical improvement in time complexity. Finally, for an overall system processing $N$ requests, the base complexity scales to $O(Nqn)$. When incorporating the query reranking module, the total online time complexity is bounded by $O(Nqn + \frac{N^2 m}{64})$.

# 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 compare our method against a standard Baseline (direct inference without acceleration) and two categories of optimized systems. First, for mainstream inference engines, we evaluate SGLang (Zheng et al., 2025) and vLLM (Kwon et al., 2023), both configured with RadixCache. Second, for training-free block-cache methods, we select PromptCache (Gim et al., 2024) and TurboRAG (Lu et al., 2024), as they natively support the Qwen model family.

**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

For each benchmark, we extracted referenced tables to pre-compute KV caches and organized them into the offline Table Trie. Using OmniSQL-7B (Li et al., 2025) and Qwen2.5-7B-Coder (Yang et al., 2024) as representative backbones, we implemented an adaptive training strategy to align them with the Primary Foreign Key Guided Table Representation.

Offline KV caches replace standard full attention with block-wise attention, restricting computations exclusively to intra-table tokens and inter-table primary-foreign key relationships. 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 the other baselines were also trained, but without the specialized attention mask.

To adapt the backbones, we fine-tuned them on complex tables from $\text{BIRD}_{\text{train}}$ for three epochs (see Appendix B). Experiments were conducted on a single NVIDIA A800 GPU with pipeline parameters $b_c = 100$ and $b_m = 10$. The inference prompt is provided in Appendix C. details regarding the offline overhead are provided in the Appendix D.

## 5.3 Main Results

Table 1 presents the following findings: TableCache significantly outperforms standard baselines on OmniSQL-7B and Qwen2.5-7B-Coder (Table 1), reducing Time to First Token (TTFT) by up to $2.69\times$ and $3.62\times$, respectively, compared to the standard Transformers library. This demonstrates TableCache's effectiveness in minimizing

[2]As KV caching is unavailable during training, we omitted PFK-based inter-table attention to streamline implementation. Both independent table chunking and PFK-guided chunking manifest as series of lower-triangular attention blocks—differing only in block size. Due to this formal identity, the simplified training mask is sufficient to adapt the model to TableCache's inference-time attention patterns.

| Method | Model | $\text{Spider}_{\text{dev}}$ | | $\text{BIRD}_{\text{dev}}$ | |
|---|---|---|---|---|---|
| | | TTFT↓ (s) | Acc.↑ (%) | TTFT↓ (s) | Acc.↑ (%) |
| Baseline | OmniSQL-7B | 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 | 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.

| 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** |
| $\text{Bird}_{\text{dev}}$ | 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 $\text{BIRD}_{\text{dev}}$, using OmniSQL-7B, with TTFT measured in seconds.

prefill latency for multi-user Text-to-SQL scenarios with shared, randomly ordered tables. Notably, on the more challenging BIRD benchmark, TableCache achieves even higher acceleration, surpassing CUDA-optimized engines like vLLM and SGLang. This is primarily because the tables in BIRD are inherently more complex and lengthier; consequently, the disparity between the user query length $q$ and the table length $n$ is more pronounced, yielding a higher theoretical speedup ratio of $\frac{n}{q}$. Crucially, this high acceleration is achieved with near-lossless performance (accuracy gaps within $1\%$). This robust retention of accuracy stems from our Primary Foreign Key (PFK) guided table representation, which effectively recovers and stores inter-block dependencies that are typically compromised by block-wise encoding.

### 5.4 Ablation Study

**Ablation of Primary Foreign Key Guided Table Representation.** Performance drops significantly without PFK-guided table representation (Table 4), with the effect more pronounced on the challenging BIRD dataset. This confirms that capturing foreign key joins is vital for complex Text-to-SQL scenarios; Appendix E provides a detailed case study.

**Ablation of Table KV Cache Management.** As shown in Table 2, removing the loading pipeline sharply increases TTFT, confirming that naive CPU-GPU cache swapping creates a severe I/O bottleneck. Furthermore, Table 5 reveals that specific cache eviction policies minimally impact prefill efficiency; because our query reranking concentrates hot data within short temporal windows, hotspots are easily captured, minimizing evictions.

**Ablation of Query Reranking.** Query reranking minimizes CPU-GPU memory exchanges. As Table 2 shows, disabling it sharply increases TTFT. Without reranking, disordered inputs disrupt table access patterns, obscuring hot data from eviction policies and triggering excessive, inefficient cache swapping.

**Ablation of Computation Loading Pipeline.** Unlike methods focused solely on swap frequency, this module leverages TableCache's KV precomputation to decouple computation from memory access. Table 2 shows that disabling this pipeline increases TTFT, confirming its effectiveness in masking memory latency within computation.

**Ablation of Model Training** As shown in Table 3, TableCache demonstrates robust performance in training-free settings, achieving lossless accuracy on pre-trained Text-to-SQL models like OmniSQL alongside SOTA efficiency in TTFT reduction. This is primarily because models specialized in Text-to-SQL tasks possess a more profound understanding of primary and foreign key dependencies,

| Method | Model | $\text{Spider}_{\text{dev}}$ | | $\text{BIRD}_{\text{dev}}$ | |
|---|---|---|---|---|---|
| | | 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 | 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 |

Table 3: Ablation study on the training process and comparison with additional training-free and block-wise methods. Boldface indicates the best performance.

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

Table 4: Ablation study on the primary foreign key guided table representation, reported in terms of Accuracy. The backbone model is OmniSQL-7B(zero-shot). PFTR refers to Primary Foreign key guided Table Representation.

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

Table 5: Ablation studies on cache eviction strategies on $\text{Spider}_{\text{dev}}$ using OmniSQL-7B.

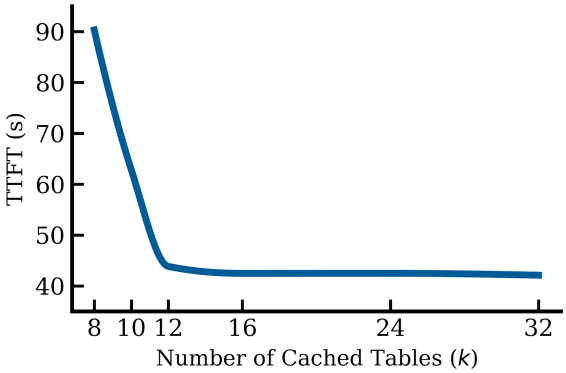


Figure 4: TTFT vs. Cache Budget ($k$)

enabling them to maintain overall performance without requiring further fine-tuning. Although exhibiting degradation on non-specialized models (e.g., Qwen2.5-7B-Coder), it still significantly outperforms existing training-free, block-wise encoding methods such as TurboRAG and PromptCache. We attribute this superior efficacy to the exploitation of two distinct domain characteristics. First, we leverage static, relation-rich schemas to precompute PFK representations offline. Second, we exploit query temporal locality via reranking and pipelining to resolve critical Text-to-SQL bottlenecks unaddressed by generic caching.

**Ablation of the Number of Cached Tables.** As shown in Figure 4, varying the cache budget reveals that TTFT drops rapidly and stabilizes once $k \geq 12$. By leveraging the synergy between our reranking and cache management modules, TableCache successfully overlaps the cache loading overhead with computation. Unlike other approaches, this allows our system to deliver near-optimal latency with a significantly smaller cache footprint.

## 6 Conclusion

This paper highlights that current inference engines inefficiently manage KV caches for frequently accessed tables in Text-to-SQL tasks, causing severe prefill overhead and latency. To address this, we propose TableCache. It optimizes performance by reusing hot-table KV caches precomputed via primary foreign key graphs and integrates a custom cache manager featuring query reranking and computation pipelining. Experimental results demonstrate that TableCache achieves the lowest TTFT across multiple text-to-SQL benchmarks.

## 7 Limitations

While TableCache is theoretically extensible to broader tasks like textual QA and KBQA, we confined this work to the Text-to-SQL scenario. This choice is motivated by the domain's unique advantages: explicit foreign key constraints provide stronger structural signals than the implicit relationships found in general text, and table granularity offers a well-defined retrieval boundary. Extending to unstructured domains requires addressing these complex, implicit dependencies, which we leave for future research—potentially by integrating with graph retrieval methods. Additionally, the modules developed herein are designed for high interoperability and 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)*.

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. A preview of 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)*.

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*.

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

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*.

Jiajun Zhang, Yuying Li, Zhixun Li, Xingyu Guo, Jingzhuo Wu, Leqi Zheng, Yiran Yang, Jianke Zhang, Qingbin Li, Shannan Yan, and 1 others. 2026. Realchart2code: Advancing chart-to-code generation with real data and multi-task evaluation. *arXiv preprint arXiv:2603.25804*.

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. 2023. 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 Cache Manager Details

**Algorithm 2:** LRU Cache Replacement

```
Input: cache: the cache with a fixed capacity C
Input: request: the requested table
Function Update(request):
    if request ∈ cache then
        Move request to the front;
    else
        if |cache| ≥ 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 ∉ cache then
        if |cache| ≥ 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 ∈ cache then
        cache[request].frequency ← cache[request].frequency + 1;
        cache[request].timestamp ← timestamp;
    else
        if |cache| ≥ 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] ← (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.

| Dataset | Max KV Cache | Min KV Cache | Avg. KV Cache | Preprocessing Time |
|---|---|---|---|---|
| BIRD | 248.0 MB | 2.1 MB | 44.38 MB | 33.01 s |
| Spider | 77.0 MB | 3.5 MB | 34.49 MB | 49.59 s |

Table 6: Statistics of KV cache sizes and offline preprocessing times.

## C Prompt Template

**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 Analysis of Metrics and Offline Overhead

To assess the system's offline requirements and robustness, we conducted a statistical analysis using the Qwen2.5-7B-Coder model. We randomly sampled 50 tables from the Spider and BIRD datasets, along with their associated KV caches. The data characteristics and preprocessing times are detailed in Table 6. From the results, both the offline storage overhead of the KV cache and the preprocessing time cost are within acceptable ranges, while also yielding a substantial improvement in TTFT.

## E Case Study

In this section, we analyze the limitations of pure block-wise attention, which processes tables independently without primary–foreign key guidance, via a case study, and 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.