

TierCache: A Multi-Granularity Semantic Caching Framework for Structured Query Generation

Myeonghwa Lee, Hyunjun Yang, Min-Soo Kim[†]

School of Computing, KAIST

Republic of Korea

{myeon9h, hy30n80, minsoo.k}@kaist.ac.kr

Abstract

Semantic caching is a promising solution to reduce LLM serving costs by reusing cached responses. However, existing methods are predominantly designed for single-level, verbatim response reuse. This limits their effectiveness in structured query generation, where queries often share output structures. To address this, we propose **TierCache**, a multi-granularity semantic caching framework that introduces two-stage retrieval and two-tier reuse: Tier 1 provides the fastest path via semantic retrieval and verbatim reuse, while Tier 2 enables the reuse of structurally similar queries through structure-aware retrieval and lightweight slot filling. We evaluate our approach on two structured query generation tasks: Text-to-SQL and Text-to-Cypher. To this end, we construct **SQStream**, a semantic caching benchmark comprising ten query-stream workloads—six for SQL and four for Cypher—that preserve realistic cache-reuse opportunities. Across all workloads, TierCache consistently establishes the most favorable quality–cost Pareto frontier. Under a strict quality-preserving constraint ($> 95\%$ retention), it achieves $1.28\times$ – $4.44\times$ the cache hit rates of state-of-the-art semantic caching baselines, delivering $1.48\times$ – $2.99\times$ end-to-end speedups. We release our code and benchmark at <https://github.com/myeon9h/TierCache>.

1 Introduction

Large language models (LLMs) increasingly serve as the foundational inference engines for interactive applications (Agrawal et al., 2024; Zhong et al., 2024; Kwon et al., 2023), making response latency and deployment costs critical factors for usability. To mitigate these overheads, recent studies have proposed **semantic caching** (Bang, 2023; Schroeder et al., 2026; Cheema et al., 2025). By retrieving and reusing previous responses for semantically similar requests, this approach significantly reduces the need for costly LLM invocations.

However, the effectiveness of semantic caching diminishes significantly when applied to **structured query generation** (e.g., Text-to-SQL, Text-to-Cypher), a key component for data-driven LLM applications (Zheng et al., 2024b; Chen et al., 2025; Dong et al., 2024; Lee et al., 2024). In structured query generation, LLMs generate outputs with structural patterns under strict syntax. Despite the prevalence of reusable output structures across queries (Xie et al., 2018; Luo et al., 2002; Huang et al., 2024), existing methods fail to fully exploit these reuse opportunities due to two key properties of structured query generation. First, **(P1)** there are two types of reusable queries: (i) *semantically similar queries*, and (ii) *structurally similar queries*, where the correct outputs share the same underlying structure but differ in only a few spans. Second, **(P2)** correctness is critical: unlike open-ended text tasks where mere plausibility often suffices, a reused output must be valid and execution-correct.

These properties impose additional design challenges on semantic caching. Under **P1**, existing methods struggle to capture structural similarity, often producing false hits under small perturbations (Schroeder et al., 2026; Gill et al., 2025; Afify et al., 2026) that lead to incorrect answers (Figure 1(a)). Moreover, they are predominantly designed around a

[†]Corresponding author

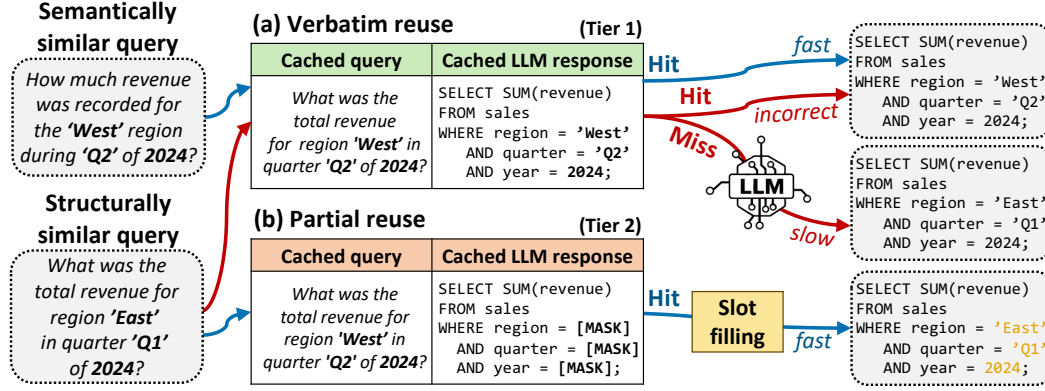


Figure 1: Comparison of two distinct reuse granularities in structured query generation.

single-level architecture restricted to *verbatim response granularity*. Under **P2**, this design forces conservative hit decisions to preserve correctness, significantly increasing cache misses and reducing the overall caching benefit.

To address these limitations, we first reformulate structured query generation as a semantic caching problem over query streams, formalizing both verbatim reuse for semantically similar queries and partial reuse for structurally similar queries. Driven by this formulation, we propose **TierCache**, a *multi-granularity semantic caching framework* designed to satisfy two requirements: **(R1)** capturing both semantic and structural similarities, and **(R2)** enabling output reuse at finer granularities. TierCache features a *two-tier caching architecture* coupled with *two-stage retrieval*. The first stage ensures fast verbatim reuse by retrieving cached outputs from Tier 1 (T1) based on conservative semantic matching. If T1 misses, the second stage performs structure-aware retrieval from Tier 2 (T2), which stores masked outputs to capture structural similarities. On a T2 hit, TierCache achieves *partial reuse* by employing a lightweight model to fill only the masked spans of the retrieved structure, bypassing expensive full generation (Figure 1(b)). This separates verbatim reuse from partial reuse, expanding cache coverage while maintaining execution-level correctness.

We instantiate our study on *Text-to-SQL* and *Text-to-Cypher*. Existing benchmarks for these tasks primarily support instance-level evaluation rather than semantic caching, which requires reuse opportunities across queries. To address this gap, we construct **SQStream**, a semantic caching benchmark for both tasks. It comprises six Text-to-SQL workloads derived from BIRD (Li et al., 2023), EHRSQL (Lee et al., 2022), and ScienceBenchmark (Zhang et al., 2023), and four Text-to-Cypher workloads derived from CypherBench (Feng et al., 2025). Each workload is organized as a sequential query stream that preserves cache-reuse opportunities through recurring query structures and skewed query frequencies.

We evaluate TierCache with the following three research questions (RQs): **RQ1** asks whether it improves the overall quality–cost trade-off and practical serving efficiency under a strict quality-preserving constraint. **RQ2** asks whether its two-tier design reliably expands cache coverage while delivering end-to-end efficiency gains. **RQ3** asks how effectively it serves structurally similar queries that existing methods fail to exploit.

Our contributions are summarized as follows:

- We formulate the semantic caching problem in structured query generation, incorporating two distinct reuse opportunities for semantically and structurally similar queries.
- We propose **TierCache**, a multi-granularity semantic caching framework that introduces two-stage retrieval and two-tier reuse for verbatim and partial query-structure reuse.
- We construct **SQStream**, a semantic caching benchmark with six Text-to-SQL and four Text-to-Cypher query-stream workloads that preserve realistic cache-reuse opportunities.
- We demonstrate that TierCache outperforms prior semantic caching methods in quality–cost trade-offs, practical serving efficiency, and hit reliability.

Framework	Store & Reuse Granularity	Retrieval	Hit Decision	Action on Hit	Supports Partial Reuse?
GPTCache	Verbatim	Semantic	Similarity-based hit decision	Return cached response	No
vCache	Verbatim	Semantic	Refined hit decision	Return cached response	No
MeanCache	Verbatim	Semantic	Calibrated hit decision	Return cached response	No
TweakLLM	Verbatim	Semantic	Similarity-based hit decision	Regenerate from cached response via auxiliary model	No
TierCache (ours)	T1: Verbatim T2: Partial	T1: Semantic T2: Structure-aware	Hierarchical hit decision	T1: Return cached response T2: Fill masked slots and return	Yes

Table 1: Comparison of semantic caching frameworks. TierCache is the only method that supports multi-granularity reuse.

2 Related Work

Semantic Caching for LLM Response Reuse. Early methods like GPTCache (Bang, 2023) store query–response pairs and directly return a cached response upon a semantic hit. Subsequent work improves this pipeline either by enhancing cache-hit reliability through verification mechanisms (e.g., vCache (Schroeder et al., 2026), MeanCache (Gill et al., 2025)), or by utilizing auxiliary models to regenerate outputs from retrieved cached responses (e.g., TweakLLM (Cheema et al., 2025)). Despite these advances, existing methods share a fundamental limitation: they rely on *single-level verbatim reuse* governed by semantic retrieval, as summarized in Table 1. As a result, these methods offer limited flexibility in balancing broader cache coverage and output validity when accommodating heterogeneous reuse opportunities. In contrast, TierCache introduces *two-stage retrieval* and *two-tier reuse*, enabling both verbatim reuse and partial query-structure reuse.

Efficient Structured Query Serving. Recent work has explored efficient techniques for reducing structured query serving costs. For query generation, EllieSQL (Zhu et al., 2025) routes requests to generation pipelines of different complexities, while other approaches reduce input-processing overhead through efficient schema retrieval and pruning (Shi et al., 2025; Piao et al., 2026; Ozsoy, 2025). TableCache (Su et al., 2026), a structured-data-specific approach within the broader family of KV-cache techniques (Kwon et al., 2023; Zheng et al., 2024b), precomputes reusable table-level states to reduce redundant prefill computation. After a structured query has been generated, result-caching systems such as LLMSigCache (Bindschaedler, 2026) reduce database-execution costs by canonicalizing queries and reusing previously materialized analytical results. However, these techniques do not bypass the costly LLM invocation required to generate a structured query. Instead, they complement semantic caching pipelines by reducing generation and LLM-serving costs on cache misses or downstream database-execution costs incurred after query generation.

Benchmarks and Evaluation. Prior semantic caching studies largely focus on open-ended generation, employing binary classification metrics (e.g., Precision, Recall, F1) or subjective proxies (e.g., LLM-as-a-judge) to assess reuse quality (Bang, 2023; Schroeder et al., 2026; Gill et al., 2025; Cheema et al., 2025). These settings are less suitable for structured query generation, where the primary issue is not perceived plausibility but execution-level correctness. Furthermore, existing semantic caching benchmarks typically evaluate queries in isolation, and therefore do not capture the properties of the streaming workload, including recurring reusable structures and skewed distributions (Luo et al., 2002; Huang et al., 2024; Ma et al., 2018; Arampatzis & Kamps, 2008). To bridge these gaps, we construct SQStream as a semantic caching benchmark comprising ten query-stream workloads that preserve such reuse opportunities across Text-to-SQL and Text-to-Cypher.

3 Method

3.1 Problem Definition

Structured query generation is the task of converting a natural language query (NLQ) x into an executable structured query (SQ) y over a schema M (Scholak et al., 2021; Dong et al., 2024; Liu et al., 2024). The generation process is defined as $y = \mathcal{G}(x, M)$, where $\mathcal{G}$ denotes an LLM-based generator. The generated query y must be executable on a structured database

$\mathcal{B}$ that follows M . For Text-to-SQL, $(y, \mathcal{B})$ is an SQL query and relational database (RDB); for Text-to-Cypher, it is a Cypher query and graph database (GDB).

In semantic caching, this task is reformulated as a combination of *retrieval* and *generation* over a query stream, $(x_t)_{t \geq 1}$. Given an incoming query x_t , the cache searches for a semantically similar past query x_k . If retrieval succeeds (a cache hit), it directly returns the cached output y_k (*verbatim reuse*); otherwise (a cache miss), it invokes $\mathcal{G}(x_t, M)$ and updates $\mathcal{C}(M)$. The objective is to reduce the serving overhead (e.g., latency) by maximizing reuse of cache entries—replacing costly generator calls with retrieval—while returning valid y_t .

A key challenge is that natural language queries for structured query generation exhibit two forms of similarity: (i) *semantic similarity*, where they admit the same structured query, and (ii) *structural similarity*, where their structured queries share the same structure while differing in only a few spans. These two forms of similarity imply two levels of granularity for reuse. Semantic similarity enables *verbatim reuse* of the cached structured query, while structural similarity allows only *partial reuse* through an abstracted structure. However, existing semantic caching methods are designed primarily for verbatim reuse and therefore fall short in effectively supporting structured query generation.

To address this, we formulate a multi-granularity caching objective for structured query generation by introducing an abstraction operator $\mathcal{A}$ and a slot-filling operator $\mathcal{F}$. The operator $\mathcal{A}$ masks target spans in a given structured query y to derive a reusable structure $\hat{y}$, i.e., $\mathcal{A} : y \rightarrow \hat{y}$. Specifically, in this work, we focus on literal values (e.g., strings and numbers) as the masking targets. If an incoming query x_t maps to a cached structure $\hat{y}_k$, we achieve *partial reuse* by instantiating the masked slots using the operator $\mathcal{F}$, such that $y_t = \mathcal{F}(x_t, \hat{y}_k)$. This formulation broadens the scope of semantic caching by considering semantic similarity for verbatim reuse and structural similarity for partial reuse.

3.2 TierCache: A Multi-Granularity Semantic Caching Framework

TierCache is designed to satisfy two requirements: **(R1)** capturing both semantic and structural similarity in retrieval, and **(R2)** supporting structured query reuse at two granularities, verbatim and partial. It comprises four components: (i) *Cache*, (ii) *Structure reuse*, (iii) *Generation*, and (iv) *Execution*. Figure 2 illustrates its architecture and three serving paths.

3.2.1 Cache Component: Two-Tier Architecture and Routing

The cache component manages a schema-specific two-tier store $\mathcal{C}^{(M)} = (T_1, T_2)$ and routes incoming queries to the appropriate serving path. Tier 1 ($T_1 = \langle \mathcal{V}_1, \mathcal{D}_1 \rangle$) handles verbatim reuse, where $\mathcal{V}_1$ is a vector index and $\mathcal{D}_1$ stores (x, y) pairs. Tier 2 ($T_2 = \langle \mathcal{V}_2, \mathcal{D}_2 \rangle$) is dedicated to partial reuse, where $\mathcal{V}_2$ is a vector index and $\mathcal{D}_2$ stores abstracted $(x, \hat{y})$ pairs.

Given an incoming NLQ x_t , the cache manager executes a two-stage retrieval process:

1. **Tier 1 (Semantic Retrieval):** A semantic encoder f_1 (e.g., BGE-M3 (Chen et al., 2024)) computes the query embedding $e_{1,t} = f_1(x_t)$. The cache manager then searches $\mathcal{V}_1$ with $e_{1,t}$; if the maximum similarity (e.g., cosine similarity) exceeds a predefined threshold τ_1 , a **T1 hit** is triggered, and the corresponding $y_k \in \mathcal{D}_1$ is directly reused.
2. **Tier 2 (Structure-Aware Retrieval):** If T1 misses, a structure-aware encoder f_2 computes $e_{2,t} = f_2(x_t)$. If the similarity to $\mathcal{V}_2$ exceeds τ_2 , a **T2 hit** is triggered. The cache then retrieves the abstracted structure $\hat{y}_k \in \mathcal{D}_2$ and passes it to the structure reuse component.
3. **Fallback Generation:** If both tiers miss, the query is routed to the **T1&T2 miss** path, invoking the generation component for full LLM inference.

3.2.2 Structure-Aware Representation Learning

To satisfy requirement **(R1)**, the Tier-2 encoder f_2 should cluster NLQs sharing the same abstracted structure $\hat{y}$ in the latent space, ensuring that their representations are closely aligned despite perturbations in the masked spans. A straightforward approach is to apply supervised contrastive learning by treating examples with the same structure as positives and those with different structures as negatives. However, due to the many-to-many relationship between NLQs and structured queries, examples with different structures may still express related intents and thus be incorrectly sampled as hard negatives.

We address this issue by annotating each training example with two labels: (i) a *structure label* w_x , which identifies the abstracted structure $\hat{y}$, and (ii) an *intent-group label* g_x , which captures the broader semantic connectivity induced by the many-to-many relationship. We optimize f_2 using a dual-level supervised contrastive loss: $\mathcal{L} = \mathcal{L}_{\text{struct}} + \lambda \mathcal{L}_{\text{group}}$. Within

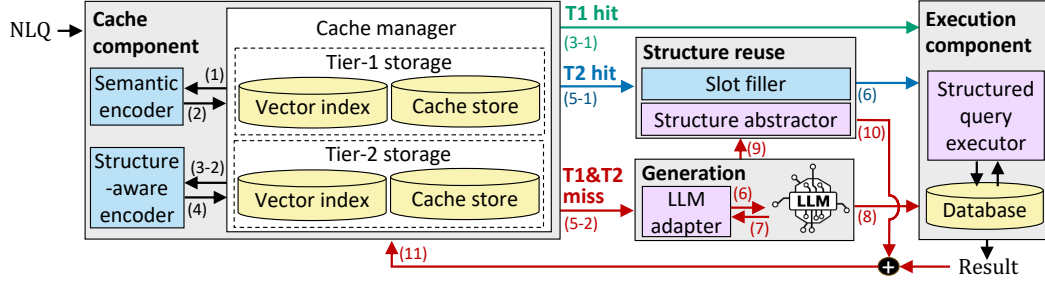


Figure 2: Overview of **TierCache**. Given an NLQ, the cache manager routes it through one of three serving paths: (1) **T1 hit** for verbatim reuse, (2) **T2 hit** for partial reuse, or (3) **T1&T2 miss** for full LLM generation. Numbers in parentheses indicate the processing order. and denote non-parametric and lightweight parametric modules, respectively.

this objective, $\mathcal{L}_{\text{struct}}$ excludes instances sharing the same intent group g_x from the negative denominator. This prevents conflicting supervision signals, ensuring f_2 learns structural alignments while preserving related semantic intents; see Appendix A.4.2 for details.

3.2.3 Structure Reuse Component

To fulfill requirement **(R2)**, this component handles both the abstraction of structured queries and the instantiation of masked spans.

Structure Abstraction ($\mathcal{A}$). On a miss path, $\mathcal{A}$ derives $\hat{y}_t$ from the generated y_t : $\hat{y}_t = \mathcal{A}(x_t, y_t)$. It uses a query-language-specific parser (e.g., SQLGlot¹) to parse y_t into an abstract syntax tree (AST) and identify literal spans. To prevent semantic distortion from overmasking, $\mathcal{A}$ enforces lexical grounding validation: a literal is replaced with a [MASK] token only if its specific value is explicitly supported by the input x_t ; otherwise, it is retained in place. Algorithm details are provided in Appendix A.3.

Lightweight Slot Filling ($\mathcal{F}$). On a T2 hit, the slot filler $\mathcal{F}$ employs a lightweight sequence-to-sequence model p_ϕ (e.g., T5 (Raffel et al., 2020)) to resolve the [MASK] tokens in the retrieved $\hat{y}_k$. To minimize inference latency, p_ϕ is trained using a selective cross-entropy objective. Instead of autoregressively modeling the entire sequence y , the loss is computed exclusively over the masked target positions. This localized supervision avoids reconstructing structural tokens that are already present in the retrieved abstraction, reducing slot-filling overhead; see Appendix A.4.3 for details.

3.2.4 Generation, Execution, and Write-Back

Generation and Execution. On a miss path, the LLM-based generator constructs the structured query $y_t = \mathcal{G}(x_t, M)$. Across all serving paths, the final query y_t is forwarded to the execution component. The executor runs y_t on the database $\mathcal{B}$, returning the execution result along with a signal $\delta_{\text{exec}} \in \{0, 1\}$, where 1 indicates that the query is executable.

Conditional Write-Back. To mitigate the risk of cache pollution from invalid outputs, TierCache employs a conditional write-back policy. Let $\delta_{\text{mask}} \in \{0, 1\}$ indicate whether the abstracted structure $\hat{y}_t$ contains at least one masked slot. The cache manager updates the tiered stores on the following conditions: the (x_t, y_t) pair and its embedding $e_{1,t}$ are cached in T_1 if $\delta_{\text{exec}} = 1$, while the abstracted pair $(x_t, \hat{y}_t)$ and its embedding $e_{2,t}$ are stored in T_2 only if $\delta_{\text{exec}} \cdot \delta_{\text{mask}} = 1$. This policy allows only executable outputs to be written to T_1 , while preventing T_2 from accumulating entries without reusable masked spans.

4 SQStream: A Benchmark for Structured Query Generation Caching

To evaluate semantic caching in structured query generation, we introduce SQStream, an execution-based benchmark over query streams with recurring structures. It comprises ten workloads across two structured query generation tasks: six Text-to-SQL workloads and four Text-to-Cypher workloads. The workloads are constructed from existing structured query generation benchmarks covering both relational and graph databases.

4.1 Goal 1: Enabling Verifiable Evaluation

Structured query generation provides an objective notion of correctness: whether a reused output executes correctly and returns the intended result. This makes Text-to-SQL and

¹<https://sqlglot.com/sqlglot.html>

Task	Source	Workload	Domain	$ \mathcal{P} $	Average Length (Tokens)		Unseen DB
					NLQ	SQ	
Text-to-SQL	EHRSQL (Lee et al., 2022)	EICU	Clinical	207	17.24	90.90	No
		MIMIC	Clinical	190	15.13	99.55	
	ScienceBenchmark (Zhang et al., 2023)	CORDIS	Scientific	70	11.82	40.17	No
		ONCOMX	Scientific	67	11.82	62.96	
	BIRD (Li et al., 2023)	CODE	Software	170	15.26	40.15	Yes
		STUDENT	University	148	13.51	52.93	
Text-to-Cypher	CypherBench (Feng et al., 2025)	COMPANY	Business	251	14.08	15.21	Yes
		ACCIDENT	Aviation	149	13.15	14.95	
		MOVIE	Entertainment	293	13.77	15.73	
		NBA	Sports	185	11.60	13.64	

Table 2: Statistics of the ten workloads in **SQStream**. $|\mathcal{P}|$ denotes the number of distinct structural pairs used to generate each workload. Each workload consists of 1,000 queries generated from its corresponding structural pair pool.

Text-to-Cypher suitable testbeds for semantic caching in structured query generation, since reuse quality can be evaluated by execution rather than plausibility. To instantiate this setting across different query languages, we construct six Text-to-SQL workloads from the dev splits of BIRD, EHRSQL, and ScienceBenchmark, and four Text-to-Cypher workloads from the test split of CypherBench. For each database, we construct a separate workload, enabling evaluation across diverse relational and graph schemas.

4.2 Goal 2: Preserving Recurring Query Structures

A benchmark for evaluating semantic caching in structured query generation should preserve recurring query structures, where queries differ in literals but share the same structured query pattern (Huang et al., 2024; Ma et al., 2018). To capture this, we construct verified structural pairs $(\hat{x}, \hat{y})$ by masking only literals that are explicitly grounded in both the natural language query x and its corresponding structured query y . The resulting structural pair pool $\mathcal{P}$ preserves reusable query structure while maintaining semantic validity and serves as the basis for subsequent workload construction.

4.3 Goal 3: Evaluating Cache Utility under Query Streams

Cache utility depends on reuse opportunities that arise across a sequence of queries rather than over isolated instances. Accordingly, each workload in **SQStream** is constructed as a query stream in two stages. First, structural pairs $(\hat{x}, \hat{y})$ are sampled from $\mathcal{P}$ under a length-based power-law distribution to induce recurrence and skew (Arampatzis & Kamps, 2008; Petersen et al., 2016). Second, masked literals are instantiated with values grounded in the corresponding database, retaining only executable, non-trivial instances. Table 2 summarizes the workload statistics of **SQStream**. Additional benchmark construction details and workload statistics are provided in Appendix A.5.

5 Experiments

This section reports experiments on the six Text-to-SQL workloads of **SQStream**; the complete Text-to-Cypher setup and results are provided in Appendix A.7.

5.1 Experimental Setup

Hardware. All experiments run on a single NVIDIA A100 GPU (80GB) in a dual AMD EPYC 7302 server. Under a unified budget, GPU memory prioritizes lightweight models, with the remainder allocated to the base LLM served via vLLM (Kwon et al., 2023).

Workloads. The six Text-to-SQL workloads each contain 1,000 NLQs over a corresponding database, issued sequentially to the serving pipeline. CODE and STUDENT comprise queries over databases unseen during lightweight-model training.

Baselines and settings. We compare TierCache with three methodologically distinct semantic caching methods: **GPTCache** (Bang, 2023), **vCache** (Schroeder et al., 2026), and **TweakLLM**² (Cheema et al., 2025). Across all methods, we use OmniSQL-32B³ (Li et al., 2025), a strong Text-to-SQL model, as the base LLM and serve it with vLLM.

To ensure a fair comparison, we standardize the underlying storage and retrieval infrastructure across all methods: SQLite as the cache store and Milvus (Wang et al., 2021) as the vector index. For Milvus, we use cosine similarity with a FLAT index. All experiments are conducted under a cold-start setting with an empty initial cache. We set a sufficiently large cache capacity to isolate the effects of eviction policy.

All baselines use the same encoder, BGE-M3⁴ (Chen et al., 2024). In TierCache, it serves as the Tier-1 semantic encoder, while Tier 2 uses a unified structure-aware encoder—initialized from the same backbone—across all six workloads. For slot-filling, we use two T5-based models specialized for SQLite and PostgreSQL. Finally, we configure each baseline following its original paper: Llama-3.1-8B-Instruct⁵ as the auxiliary model for TweakLLM, and an error-tolerance threshold of 0.02 for vCache.

Research questions and metrics. We evaluate TierCache to answer the following three research questions:

- **RQ1:** Does TierCache improve the overall quality–cost trade-off, and how effectively does it enhance practical serving efficiency under a strict quality-preserving constraint?
- **RQ2:** Does the two-tier design of TierCache reliably expand cache coverage while delivering end-to-end efficiency gains?
- **RQ3:** How effectively does TierCache serve structurally similar queries that existing methods fail to exploit?

To answer RQ1, we measure *quality* by **execution accuracy (EX)**—whether the served structured query returns the correct execution result—and *cost* by the **average latency** across the entire query stream, excluding database-execution time. We vary the similarity threshold $\tau \in \{0.75, 0.8, 0.85, 0.9, 0.95, 0.99\}$ to plot the quality–cost Pareto frontier. For TierCache, we apply a uniform threshold across both retrieval stages, i.e., $\tau = \tau_1 = \tau_2$. Furthermore, we evaluate practical efficiency under a *strict quality-preserving constraint*, defined as $EX > 0.95 \cdot EX_{\text{no-cache}}$. Within this constraint, we report the **cache hit rate** and **saved input and output token rates**. The token-based metrics comprehensively account for tokens from the base generator, lightweight models, and auxiliary models.

Next, to address RQ2, we analyze the **false-hit rate (FHR)** across similarity thresholds that control cache coverage. To evaluate cache-hit reliability independently of base-LLM generation performance, we compute FHR by tracking cache behavior and query history to determine whether each hit reuses the correct cache entry. We fix the Tier-1 threshold of TierCache at $\tau_1 = 0.95$ and vary its Tier-2 threshold $\tau_2 \in \{0.75, 0.8, 0.85, 0.9, 0.95\}$, while varying the single threshold of each baseline over the same range. This setting evaluates the flexibility of the two-tier design: conservative verbatim reuse in Tier 1 is decoupled from partial-reuse coverage in Tier 2, rather than having both reuse modes governed by a single threshold. We further report the **hit-rate gain (ΔHR)** and **end-to-end speedup** relative to the baselines to assess whether this reliable coverage expansion sustains efficiency gains.

Finally, to answer RQ3, we classify each query into three mutually exclusive categories based on its stream order: (i) *semantic*, if an identical gold SQL query exists in the history; (ii) *structural*, if only the abstracted gold structure is shared with a prior query; and (iii) *novel*, otherwise. We report the **cache hit rate** and the **execution accuracy conditioned on a cache hit (H-EX)** for both semantic and structural query types.

5.2 Experimental Results

Overall quality–cost trade-off (RQ1). Figure 3 plots the quality–cost Pareto frontiers across the six workloads. At some operating points, caching methods achieve higher EX than the no-cache baseline: independent LLM invocations on semantically or structurally

²TweakLLM is not publicly released. We therefore re-implement it on top of GPTCache based on the paper description; see Appendix A.6.2 for details.

³<https://huggingface.co/seeklhy/OmniSQL-32B>

⁴<https://huggingface.co/BAAI/bge-m3>

⁵<https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct>

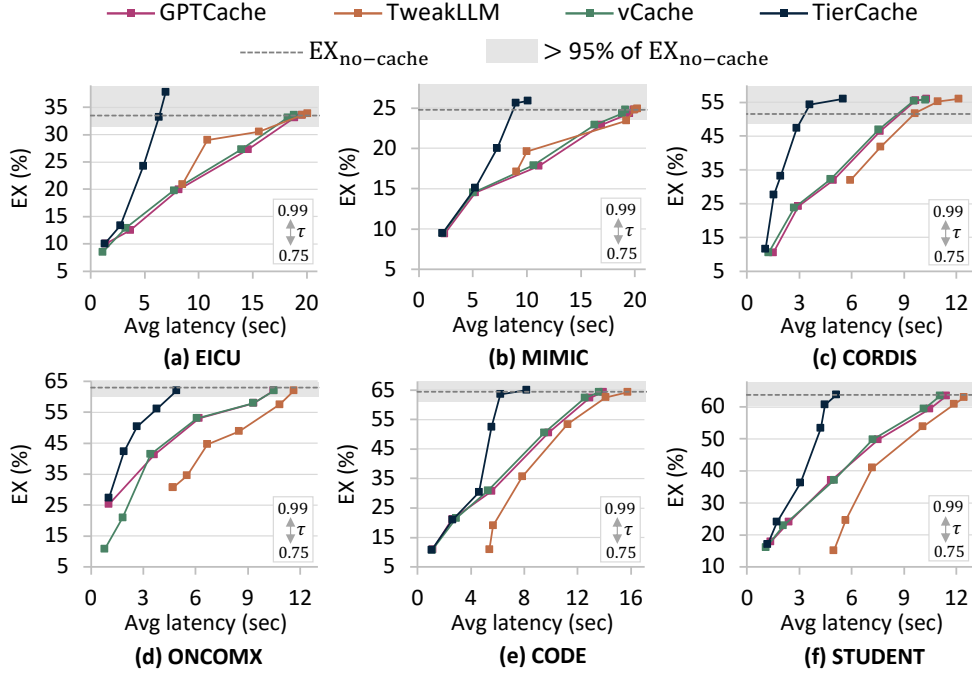


Figure 3: Quality-cost Pareto frontiers of execution accuracy (EX; %) versus average latency (sec). Along each curve, lower-left and upper-right points correspond to thresholds near 0.75 and 0.99, respectively. The shaded region denotes the strict quality-preserving zone, where EX exceeds 95% of the no-cache baseline. Higher EX and lower latency are better.

Method	Hit Rate (%) $\uparrow$			Saved Input Tokens (%) $\uparrow$			Saved Output Tokens (%) $\uparrow$		
	Min	Average	Max	Min	Average	Max	Min	Average	Max
GPTCache	27.05	44.34	66.30	26.49	42.96	65.68	28.61	45.41	66.50
vCache	27.05	44.30	66.10	26.49	42.99	65.79	28.61	45.33	66.27
TweakLLM	27.15	45.42	66.50	6.70	22.11	40.71	10.72	26.78	45.12
TierCache	65.46	76.57	85.40	67.89	75.68	85.47	70.11	79.00	86.62

Table 3: Summary of hit rates (%) and saved token rates (%) over the six workloads at the lowest-latency operating point satisfying the quality-preserving constraint.

similar NLQs do not guarantee consistency in query generation (Chang et al., 2023; Gan et al., 2021), whereas caching can reuse an earlier correct query or its structure for subsequent similar NLQs. Overall, TierCache establishes the most favorable frontier in the upper-left region, indicating high execution accuracy and low latency. This advantage persists on the unseen databases (CODE and STUDENT), demonstrating generalization to unseen schemas; Appendix A.4.4 further evaluates lightweight models across schemas, domains, and SQL dialects. Notably, to enter the strict quality-preserving zone (shaded area, $>95\%$ of $EX_{no-cache}$), TierCache incurs an average latency of only 3.59–9.01 seconds, while the baselines require 9.58–19.57 seconds to achieve comparable execution accuracy. The cumulative latency trajectory for each workload is provided in Appendix A.6.5.

Table 3 reports the efficiency gains at the lowest-latency operating point satisfying the quality-preserving constraint for each method; the corresponding parameters and per-workload results are provided in Appendices A.6.3 and A.6.4, respectively. At this setting, TierCache achieves a cache hit rate ranging from 65.46% to 85.40% (averaging 76.57%), substantially outperforming the baselines, which range between 27.05% and 66.50%. Despite the additional lightweight models, TierCache achieves substantial end-to-end token reductions. Its saved input token rate ranges from 67.89% to 85.47% (75.68% on average), and its saved output token rate reaches 86.62% (79.00% on average). In contrast, the best baselines average only 42.99% and 45.41% in input and output savings, respectively. These results demonstrate that TierCache substantially improves the overall quality-cost trade-off and enhances end-to-end efficiency under the strict quality-preserving constraint.

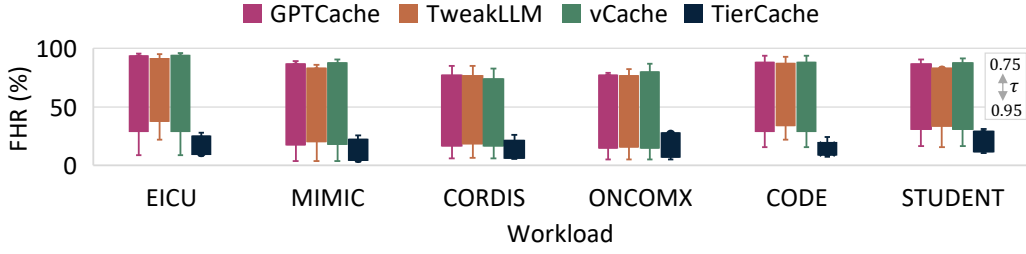


Figure 4: False-hit rate (FHR; %) across similarity thresholds τ ranging from 0.75 to 0.95 with 0.05 increments. TierCache fixes $\tau_1 = 0.95$ and varies τ_2 according to τ . Lower is better.

Workload	GPTCache		TweakLLM		vCache	
	Δ HR	SpUP	Δ HR	SpUP	Δ HR	SpUP
EICU	+34.3	2.16	+31.3	2.29	+34.3	2.06
MIMIC	+38.3	1.75	+37.3	1.87	+38.4	1.68
CORDIS	+23.0	2.22	+22.0	2.99	+23.0	2.19
ONCOMX	+11.2	1.95	+10.0	3.11	+11.3	1.91
CODE	+26.7	1.66	+25.0	2.23	+26.7	1.61
STUDENT	+23.9	1.70	+21.4	2.29	+24.0	1.63

Table 4: Hit-rate gain (Δ HR; percentage points) and speedup (SpUP; $\times$) of TierCache ($\tau_1 = 0.95$, $\tau_2 = 0.9$) relative to each baseline ($\tau = 0.9$).

Cache Hit	Latency (s)
GPTCache, vCache	0.28–0.30
TweakLLM	6.61
TierCache T1	0.29
TierCache T2	0.83
Cache Miss	Latency (s)
GPTCache, vCache	24.92–25.05
TweakLLM	25.18
TierCache	25.42

Table 5: Average latency (sec) by serving path.

Cache reliability and efficiency (RQ2). Figure 4 illustrates a key limitation of the baselines: as the similarity threshold (τ) is relaxed to expand cache coverage, their false-hit rates (FHR) increase substantially. In contrast, TierCache maintains consistently low FHR across thresholds. By fixing the Tier-1 threshold at $\tau_1 = 0.95$, it effectively limits false verbatim hits while reliably expanding coverage through structure-aware Tier-2 hits.

As shown in Table 4, at the same threshold ($\tau = \tau_2 = 0.9$), TierCache increases hit rates by 10.0–38.4 percentage points over the baselines. Although a Tier-2 hit takes 0.83s versus 0.28–0.30s for a baseline hit, it remains substantially below the cache-miss penalty of approximately 25 seconds (Table 5). As a result, the expanded coverage more than offsets the Tier-2 overhead, yielding $1.61\times$ – $3.11\times$ end-to-end speedups. These results show that the two-tier design of TierCache reliably expands cache coverage while delivering efficiency gains.

Performance breakdown by query type (RQ3). Table 6 and Table 7 report the cache hit rate (HR) and execution accuracy on hits (H-EX) for *semantic-type* and *structural-type* queries, respectively. For semantic-type queries, all methods exhibit comparable HR and H-EX, indicating that TierCache preserves the verbatim-reuse capability of prior semantic caching methods. The main difference appears for structural-type queries, for which the baselines show substantially lower HR and H-EX. Although TweakLLM obtains modest improvements through full regeneration with an auxiliary LLM, its H-EX remains relatively low. In contrast, TierCache effectively reuses shared query structures, reaching up to 50.1% HR and 63.5% H-EX. These results demonstrate that TierCache captures reuse opportunities that the baselines otherwise miss while maintaining higher accuracy on hits.

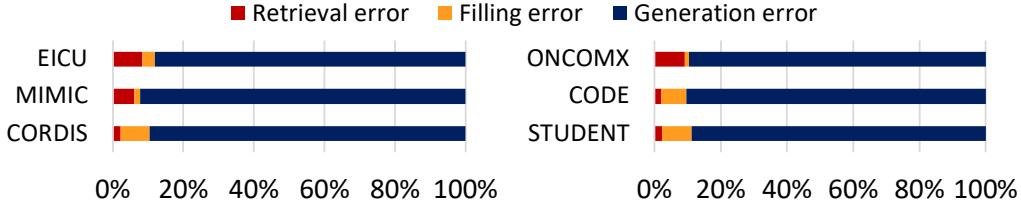
Tier-2 failure analysis. We categorize Tier-2 failures into three types: (i) *retrieval errors*, where an incorrect cached structure is retrieved; (ii) *filling errors*, where the correct structure is retrieved but incorrectly instantiated; and (iii) *generation errors*, where the appropriate cache entry is retrieved but its stored structure was derived from an incorrect base-LLM output. Figure 5 reports their proportions at $\tau = 0.95$.

Generation errors dominate across the six workloads, accounting for 88.11%–92.31% of Tier-2 failures, while retrieval and filling errors each remain below 10%. This distribution indicates that the primary source of the observed Tier-2 failures is cache pollution caused by incorrect base-LLM outputs, rather than errors in structure retrieval or slot filling. The same issue can arise whenever semantic caching reuses generated structured queries, since an incorrect cached output may affect subsequent hits even under reliable retrieval. Although TierCache mitigates this issue through its conditional write-back policy, the observed error distribution motivates stronger verification and cache-management policies.

Semantic Method	EICU		MIMIC		CORDIS		ONCOMX		CODE		STUDENT	
	HR	H-EX	HR	H-EX	HR	H-EX	HR	H-EX	HR	H-EX	HR	H-EX
GPTCache	26.2	36.2	31.1	21.5	60.4	38.4	70.4	60.6	36.9	57.1	44.7	48.1
TweakLLM	26.2	31.9	31.1	23.4	60.5	38.4	70.7	45.1	36.9	38.2	44.6	44.4
vCache	26.3	36.3	31.0	21.6	60.6	38.9	70.3	60.5	36.9	57.2	45.0	48.2
TierCache	26.3	36.5	31.1	29.3	60.6	38.9	70.8	60.5	36.5	62.6	44.6	51.5

Table 6: Cache hit rate (HR; %) and EX on hits (H-EX; %) for *semantic-type* queries at $\tau = 0.9$.

Structural Method	EICU		MIMIC		CORDIS		ONCOMX		CODE		STUDENT	
	HR	H-EX	HR	H-EX	HR	H-EX	HR	H-EX	HR	H-EX	HR	H-EX
GPTCache	17.4	5.1	9.7	13.4	7.1	1.4	10.0	3.0	18.1	9.3	16.8	14.2
TweakLLM	20.3	22.6	10.7	24.9	7.9	32.9	11.0	22.7	20.1	22.8	18.9	32.8
vCache	17.4	5.1	9.7	13.4	7.0	1.4	10.0	3.0	18.1	9.4	16.8	14.3
TierCache	50.1	33.9	46.9	31.1	29.3	63.5	20.4	33.4	42.8	49.5	37.8	61.4

Table 7: Cache hit rate (HR; %) and EX on hits (H-EX; %) for *structural-type* queries at $\tau = 0.9$.Figure 5: Distribution of Tier-2 failure cases by error source at $\tau = 0.95$.

Variant	EICU		MIMIC		CORDIS		ONCOMX		CODE		STUDENT	
	EX	AL	EX	AL	EX	AL	EX	AL	EX	AL	EX	AL
T1 only	33.10	18.59	23.50	21.71	55.50	10.05	58.90	10.33	62.40	13.71	59.50	11.07
T2 only	37.80	12.10	26.20	13.71	53.60	13.62	56.20	8.27	62.40	11.49	61.20	6.33
TierCache	37.80	6.74	<u>26.00</u>	9.64	<u>55.20</u>	3.42	<u>58.60</u>	3.17	62.40	5.88	61.20	4.41

Table 8: Execution accuracy (EX; %) and average latency (AL; sec) for the ablation study of isolated tiers ($\tau_1 = 0.95$, $\tau_2 = 0.9$). Underlines indicate comparable EX (difference $< 1\%$).

Ablation study. Finally, we conduct an ablation study to validate the two-tier design of TierCache. Table 8 compares TierCache against its isolated variants: Tier 1 only (T1 only) and Tier 2 only (T2 only). The results indicate that relying on an isolated tier leads to inconsistent execution accuracy across different workloads. For instance, T1 alone underperforms on EICU, while T2 exhibits a performance drop on ONCOMX. In contrast, TierCache (T1+T2) achieves highly competitive EX within a 1% relative margin of the better-performing isolated tier while consistently attaining the lowest average latency. These results demonstrate that Tier 1 and Tier 2 are complementary: their combination retains near-best execution accuracy while substantially reducing latency relative to either isolated tier.

6 Conclusions and Future Work

We presented **TierCache**, a multi-granularity semantic caching framework that introduces two-stage retrieval and two-tier reuse for structured query generation. We further constructed **SQStream**, a semantic caching benchmark that preserves realistic cache-reuse opportunities for Text-to-SQL and Text-to-Cypher. Across all workloads, TierCache consistently establishes the most favorable quality–cost Pareto frontier, achieving higher cache hit rates while maintaining greater reliability than state-of-the-art semantic caching baselines. Future work will extend TierCache to other structured query generation tasks such as Text-to-SPARQL and broaden abstraction targets beyond literals to other query components, such as aggregation functions and operators. We will also investigate stronger verification and cache-management policies to mitigate cache pollution, together with stronger lightweight models and auxiliary re-rankers for more accurate partial reuse.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00347471).

References

- M. Afify, M. W. Fakh, and F. A. Maghraby. Enhancing adversarial resilience in semantic caching for secure retrieval augmented generation systems. *Scientific Reports*, 16(1):5936, 2026. doi: 10.1038/s41598-026-36721-w. URL <https://doi.org/10.1038/s41598-026-36721-w>.
- Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL <https://www.usenix.org/conference/osdi24/presentation/agrawal>.
- Avi Arampatzis and Jaap Kamps. A study of query length. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pp. 811–812, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581644. doi: 10.1145/1390334.1390517. URL <https://doi.org/10.1145/1390334.1390517>.
- Fu Bang. GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings. In Liling Tan, Dmitrijs Milajevs, Geeticka Chauhan, Jeremy Gwinnup, and Elijah Rippeth (eds.), *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pp. 212–218, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.nlpss-1.24. URL <https://aclanthology.org/2023.nlpss-1.24/>.
- Laurent Bindschaedler. Semantic caching for OLAP via LLM-based query canonicalization. In *Proceedings of the 28th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data*, volume 4186 of *CEUR Workshop Proceedings*, pp. 125–132. CEUR-WS.org, 2026. URL <https://ceur-ws.org/Vol-4186/short5.pdf>.
- Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. Dr.Spider: A diagnostic evaluation benchmark towards text-to-SQL robustness. In *The Eleventh International Conference on Learning Representations*, 2023. URL <https://openreview.net/forum?id=Wc5bmZZU9cy>.
- Muhammad Taha Cheema, Abeer Aamir, Khawaja Gul Muhammad, Naveed Anwar Bhatti, Ihsan Ayyub Qazi, and Zafar Ayyub Qazi. TweakLLM: A routing architecture for dynamic tailoring of cached responses, 2025. URL <https://arxiv.org/abs/2507.23674>.
- Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2318–2335, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.137. URL <https://aclanthology.org/2024.findings-acl.137/>.
- Junyi Chen, Shihao Bai, Zaijun Wang, Siyu Wu, Chuheng Du, Hailong Yang, Ruihao Gong, Shengzhong Liu, Fan Wu, and Guihai Chen. Pre³: Enabling deterministic pushdown automata for faster structured LLM generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11253–11267, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.551. URL <https://aclanthology.org/2025.acl-long.551/>.

- Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. XGrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100*, 2024. doi: 10.48550/arXiv.2411.15100. URL <https://arxiv.org/abs/2411.15100>.
- Yanlin Feng, Simone Papicchio, and Sajjadur Rahman. CypherBench: Towards precise retrieval over full-scale modern knowledge graphs in the LLM era. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8934–8958, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.438. URL <https://aclanthology.org/2025.acl-long.438/>.
- Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. Towards robustness of text-to-SQL models against synonym substitution. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 2505–2515, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.195. URL <https://aclanthology.org/2021.acl-long.195/>.
- Waris Gill, Mohamed Elidrisi, Pallavi Kalapatapu, Ammar Ahmed, Ali Anwar, and Muhammad Ali Gulzar. MeanCache: User-centric semantic caching for LLM web services. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1298–1310, 2025. doi: 10.1109/IPDPS64566.2025.00117.
- Hanxian Huang, Tarique Siddiqui, Rana Alotaibi, Carlo Curino, Jyoti Leeka, Alekh Jindal, Jishen Zhao, Jesús Camacho-Rodríguez, and Yuanyuan Tian. Sibyl: Forecasting time-evolving query workloads. *Proc. ACM Manag. Data*, 2(1), March 2024. doi: 10.1145/3639308. URL <https://doi.org/10.1145/3639308>.
- Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschiot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. In *Advances in Neural Information Processing Systems*, volume 33, pp. 18661–18673, 2020.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23*, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL <https://doi.org/10.1145/3600006.3613165>.
- Gyubok Lee, Hyeonji Hwang, Seongsu Bae, Yeonsu Kwon, Woncheol Shin, Seongjun Yang, Minjoon Seo, Jongyeup Kim, and Edward Choi. EHRSQL: A practical Text-to-SQL benchmark for electronic health records. In *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS '22*, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.
- Myeonghwa Lee, Seonho An, and Min-Soo Kim. PlanRAG: A plan-then-retrieval augmented generation for generative large language models as decision makers. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 6537–6555, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-long.364. URL <https://aclanthology.org/2024.naacl-long.364/>.
- Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. OmniSQL: Synthesizing high-quality Text-to-SQL data at scale. *Proc. VLDB Endow.*, 18(11):4695–4709, July 2025. ISSN 2150-8097. doi: 10.14778/3749646.3749723. URL <https://doi.org/10.14778/3749646.3749723>.

- Jinyang Li, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM already serve as A database interface? A Big bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems*, volume 36, pp. 42330–42357, 2023.
- Michael Xieyang Liu, Frederick Liu, Alexander J. Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J. Cai. “We Need Structured Output”: Towards User-centered Constraints on Large Language Model Output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA ’24, New York, NY, USA, 2024. Association for Computing Machinery. doi: 10.1145/3613905.3650756.
- Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’02, pp. 600–611, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134975. doi: 10.1145/564691.564763. URL <https://doi.org/10.1145/564691.564763>.
- Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD ’18, pp. 631–645, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450354701. doi: 10.1145/3183713.3196908.
- Makbule Gulcin Ozsoy. Enhancing Text2Cypher with schema filtering. In *Joint Proceedings of the Fourth International Workshop on LLM-Integrated Knowledge Graph Generation From Text and the Second International BiKE Challenge*, volume 4020 of CEUR Workshop Proceedings, pp. 9–17. CEUR-WS.org, 2025. URL https://ceur-ws.org/Vol-4020/Paper_ID.5.pdf.
- Casper Petersen, Jakob Grue Simonsen, and Christina Lioma. Power law distributions in information retrieval. *ACM Transactions on Information Systems*, 34(2), 2016. ISSN 1046-8188. doi: 10.1145/2816815. URL <https://doi.org/10.1145/2816815>.
- Shengmin Piao, Jieun Lee, and Sanghyun Park. LitE-SQL: A lightweight and efficient text-to-SQL framework with vector-based schema linking and execution-guided self-correction. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 3593–3608, Rabat, Morocco, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-eacl.186. URL <https://aclanthology.org/2026.findings-eacl.186/>.
- Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1), January 2020. ISSN 1532-4435.
- Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9895–9901, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.779. URL <https://aclanthology.org/2021.emnlp-main.779/>.
- Luis Gaspar Schroeder, Aditya Desai, Alejandro Cuadron, Kyle Chu, Shu Liu, Mark Zhao, Stephan Krusche, Alfons Kemper, Matei Zaharia, and Joseph E. Gonzalez. vCache: Verified semantic prompt caching. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=zF0A0xw3HZ>.
- Jie Shi, Bo Xu, Jiaqing Liang, Yanghua Xiao, Jia Chen, Chenhao Xie, Peng Wang, and Wei Wang. Gen-SQL: Efficient text-to-SQL by bridging natural language question and database schema with pseudo-schema. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 3794–3807, Abu Dhabi, UAE, 2025. Association for Computational Linguistics. URL <https://aclanthology.org/2025.coling-main.256/>.

- Jinbo Su, Yuxuan Hu, Cuiping Li, Hong Chen, Jia Li, Lintao Ma, and Jing Zhang. TableCache: Primary foreign key guided KV cache precomputation for low latency text-to-SQL. *arXiv preprint arXiv:2601.08743*, 2026. URL <https://arxiv.org/abs/2601.08743>.
- Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment*, 17(11):3694–3706, 2024. doi: 10.14778/3681954.3682031.
- Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21*, pp. 2614–2627, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457550. URL <https://doi.org/10.1145/3448016.3457550>.
- Ting Xie, Varun Chandola, and Oliver Kennedy. Query log compression for workload analytics. *Proceedings of the VLDB Endowment*, 12(3):183–196, 2018. doi: 10.14778/3291264.3291265.
- Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. ScienceBenchmark: A complex real-world benchmark for evaluating natural language to SQL systems. *Proc. VLDB Endow.*, 17(4):685–698, December 2023. ISSN 2150-8097. doi: 10.14778/3636218.3636225. URL <https://doi.org/10.14778/3636218.3636225>.
- Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. Wild-Chat: 1M ChatGPT interaction logs in the wild. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=Bl8u7ZRlbM>.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. LMSYS-Chat-1M: A large-scale real-world LLM conversation dataset. In *The Twelfth International Conference on Learning Representations*, 2024a. URL <https://openreview.net/forum?id=B0fDKxfwt0>.
- Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Thirty-Eighth Conference on Neural Information Processing Systems*, 2024b. URL <https://openreview.net/forum?id=VqkAKQibpq>.
- Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL <https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin>.
- Yizhang Zhu, Runzhi Jiang, Boyan Li, Nan Tang, and Yuyu Luo. EllieSQL: Cost-efficient text-to-SQL with complexity-aware routing. In *Second Conference on Language Modeling*, 2025. URL <https://openreview.net/forum?id=80qGNXKwo8>.

A Appendix

A.1 Notations

Notation	Description
Structured query generation	
x, x_t, x_k	Natural language query (NLQ) (general, incoming at time t , cached at time k)
y, y_t, y_k	Executable structured query (SQ)
$\hat{y}, \hat{y}_k$	Abstracted structured query with masked spans
$M, \mathcal{B}$	Database schema and the corresponding structured database
$\mathcal{G}$	LLM-based structured query generator
Cache storage	
$\mathcal{C}^{(M)}$	Schema-specific two-tier cache store, defined as (T_1, T_2)
T_1, T_2	Tier 1 (verbatim reuse) and Tier 2 (partial reuse)
$\mathcal{V}_1, \mathcal{V}_2$	Vector indices for T_1 and T_2
$\mathcal{D}_1, \mathcal{D}_2$	Cache stores for verbatim pairs (x, y) and abstracted pairs $(x, \hat{y})$
Models & Operators	
$\mathcal{A}, \mathcal{F}$	Structure abstraction operator and lightweight slot-filling operator
f_1, f_2	Semantic encoder (for T_1) and structure-aware encoder (for T_2)
$e_{1,t}, e_{2,t}$	Query embeddings computed by f_1 and f_2 for incoming query x_t
p_ϕ	Lightweight sequence-to-sequence model used for slot filling
Retrieval, Training, & Constraints	
τ_1, τ_2	Similarity thresholds for T1 and T2 cache hits
w_x, g_x	Structure label and intent-group label for training f_2
$\mathcal{L}_{\text{struct}}, \mathcal{L}_{\text{group}}$	Supervised contrastive losses for structure and intent-group alignment
λ	Weight balancing the dual-level supervised contrastive loss
$\delta_{\text{exec}}, \delta_{\text{mask}}$	Binary indicators for successful execution and presence of masked slots
Benchmark construction (SQStream)	
$\mathcal{P}$	Structural pair pool containing verified $(\hat{x}, \hat{y})$ pairs

Table 9: Summary of notations

A.2 TierCache Inference

Algorithm 1 TierCache Inference

Input: Incoming NLQ x_t , Schema M , Target database $\mathcal{B}$, Thresholds τ_1, τ_2
Output: Served structured query y_t

```

1: % Step 1: Tier 1 (Verbatim retrieval)
2:  $e_{1,t} \leftarrow f_1(x_t)$ 
3:  $y_k, s_1 \leftarrow \text{SEARCH}(\mathcal{V}_1, \mathcal{D}_1, e_{1,t})$ 
4: if  $s_1 \geq \tau_1$  then
5:    $y_t \leftarrow y_k$  ▷ T1 hit
6:    $\text{EXECUTE}(y_t, \mathcal{B})$  ▷ Forward to execution component
7:   return  $y_t$ 
8: end if
9: % Step 2: Tier 2 (Structure-aware retrieval)
10:  $e_{2,t} \leftarrow f_2(x_t)$ 
11:  $\hat{y}_k, s_2 \leftarrow \text{SEARCH}(\mathcal{V}_2, \mathcal{D}_2, e_{2,t})$ 
12: if  $s_2 \geq \tau_2$  then
13:    $y_t \leftarrow \mathcal{F}(x_t, \hat{y}_k)$  ▷ T2 hit: Lightweight slot filling
14:    $\delta_{\text{mask}} \leftarrow 0$  ▷ Skip T2 write-back for partial reuse
15: else
16:   % Step 3: Fallback generation
17:    $y_t \leftarrow \mathcal{G}(x_t, M)$  ▷ T1&T2 miss: LLM-based generation
18:    $\hat{y}_t \leftarrow \mathcal{A}(x_t, y_t)$  ▷ Structure abstraction
19:    $\delta_{\text{mask}} \leftarrow (\hat{y}_t \neq y_t)$  ▷ 1 if mask exists, 0 otherwise
20: end if
21: % Step 4: Execution & Conditional write-back
22:  $\delta_{\text{exec}} \leftarrow \text{EXECUTE}(y_t, \mathcal{B})$  ▷ Returns 1 if executable
23: if  $\delta_{\text{exec}} = 1$  then
24:    $\text{UPDATE}(T_1, x_t, y_t, e_{1,t})$  ▷ Store verbatim pair in  $T_1$ 
25:   if  $\delta_{\text{mask}} = 1$  then
26:      $\text{UPDATE}(T_2, x_t, \hat{y}_t, e_{2,t})$  ▷ Store abstracted pair in  $T_2$ 
27:   end if
28: end if
29: return  $y_t$ 

```

A.3 Structure Abtractor

The structure abtractor $\mathcal{A}$ implements the abstraction in two steps, as outlined in Algorithm 2. First, it parses a structured query into an abstract syntax tree with a query-language-specific parser and identifies literal spans from the tree structure. Second, it applies grounding validation against the natural language query and masks a literal only when that literal is supported by the input. Unsupported literals remain unmasked. This procedure preserves alignment between the abstraction and the source query while retaining as much reusable structure as possible.

Algorithm 2 Structure Abstraction Operator $\mathcal{A}$

Input: Natural language query x_t , Structured query y_t
Output: Abstracted structure $\hat{y}_t = \mathcal{A}(x_t, y_t)$

```

1: Parse  $y_t$  into an abstract syntax tree  $A$  using a query-language-specific parser
2:  $\mathcal{L} \leftarrow \text{ExtractLiteralSpans}(A)$ 
3:  $\hat{y}_t \leftarrow y_t$ 
4: for all  $\ell \in \mathcal{L}$  do
5:   if  $\text{LexicallyGrounded}(\ell, x_t)$  then
6:      $\hat{y}_t \leftarrow \text{Replace}(\hat{y}_t, \ell, [\text{MASK}])$ 
7:   end if ▷ Otherwise, retain the original literal
8: end for
9: return  $\hat{y}_t$ 

```

A.4 Lightweight Model Training

We train separate structure-aware encoders and slot-fillers for Text-to-SQL and Text-to-Cypher. The two tasks share the training objectives and backbone architectures described below, while using task-specific training data and hyperparameters. All models are trained on three NVIDIA A100 GPUs (80 GB each).

A.4.1 Training Data Construction

For both tasks, we construct training data from the training splits of the respective source benchmarks. To improve the robustness of structure-aware retrieval, we first augment the training data with multiple query variants for each structure extracted by the structure abstraction in Algorithm 2, thereby increasing within-structure diversity for contrastive learning. Next, we annotate each training example i with a structure label w_i and an intent-group label g_i . The structure label w_i identifies the abstracted query $\hat{y}$, while the intent-group label g_i connects $\hat{x}$ and $\hat{y}$. Finally, we sample hard negatives that are lexically similar but belong to different intent groups, enabling the structure-aware encoder f_2 to effectively discriminate among structurally confusable candidates.

Text-to-SQL. We use the training splits of EHRSQL, ScienceBenchmark, and selected BIRD databases, summarized in Table 10. The augmented dataset contains 86,247 examples derived from 15,432 seed instances (Table 11), with 15,432 structure labels and 15,295 intent-group labels. It contains an average of 5.59 examples per structure label, 5.64 examples per intent-group label, and 1.43 masked literals per example. We retain eight mined hard-negative candidates per example.

Source	Database	Train Size	Domain	Dialect	Evidence
EHRSQL	eicu mimic_iii	5,352 5,321	Clinical	SQLite	No
ScienceBenchmark	cordis oncomx sdss	499 466 1,890	Scientific	PostgreSQL	No
BIRD	codebase_comments	112	Software	SQLite	Yes
	software_company	74			
	talkingdata	186			
	social_media	74			
	college_completion	76	University		
	computer_student	71			
	cs_semester	107			
	university	128			
	european_football_1	57	Sports		
	hockey	207			
	ice_hockey_draft	83			
	olympics	131			
professional_basketball	156				
soccer_2016	256				
student_loan	186	Finance			

Table 10: Statistics of the datasets used to construct the training data for the lightweight models in the Text-to-SQL experiments. Each dataset represents either the full or a partial training split from its respective source benchmark.

Source	Database	Original	Augmented	Total	Avg. #Masks
EHRSQL	eicu	5,352	28,372	33,724	1.448
	mimic_iii	5,321	27,488	32,809	1.523
ScienceBenchmark	cordis	499	2,423	2,922	1.116
	oncomx	466	1,802	2,268	1.097
	sdss	1,890	2,936	4,826	1.571
BIRD	codebase_comments	112	462	574	1.035
	college_completion	76	413	489	1.843
	computer_student	71	105	176	0.727
	cs_semester	107	410	517	1.520
	european_football.1	57	322	379	1.694
	hockey	207	939	1,146	1.359
	ice_hockey_draft	83	420	503	1.537
	olympics	131	650	781	1.152
	professional_basketball	156	704	860	1.331
	soccer_2016	256	903	1,159	1.045
	social_media	74	342	416	1.192
	software_company	74	321	395	1.995
	student_loan	186	346	532	0.829
	talkingdata	186	753	939	1.204
	university	128	704	832	1.674
Overall Total		15,432	70,815	86,247	1.430

Table 11: Statistics of the augmented training dataset for the lightweight models in the Text-to-SQL experiments, generated using the datasets in Table 10 as seeds. Original and Augmented denote the numbers of seed instances (aug=False) and augmented instances (aug=True), respectively. Avg. #MASK denotes the average number of masked spans per instance.

Text-to-Cypher. We apply the same construction procedure to the training splits of four CypherBench databases, summarized in Table 12. The augmented dataset contains 23,366 examples derived from 3,612 seed instances (Table 13), with 3,612 structure labels and 3,541 intent-group labels. It contains an average of 6.47 examples per structure label, 6.60 examples per intent-group label, and 1.35 masked literals per example. We retain eight mined hard-negative candidates per example.

Source	Database	Train Size	Domain	Dialect	Evidence
CypherBench	art	1,136	N/A	Cypher	No
	biology	1,251	N/A		
	soccer	732	N/A		
	terrorist_attack	493	N/A		

Table 12: Statistics of the datasets used to construct the training data for the lightweight models in the Text-to-Cypher experiments. Each dataset represents the training split of its corresponding CypherBench database.

Source	Database	Original	Augmented	Total	Avg. #Masks
CypherBench	art	1,136	6,769	7,905	1.36
	biology	1,251	5,832	7,083	1.31
	soccer	732	4,370	5,102	1.37
	terrorist_attack	493	2,783	3,276	1.38
Overall Total		3,612	19,754	23,366	1.35

Table 13: Statistics of the augmented training dataset for the lightweight models in the Text-to-Cypher experiments, generated using the datasets in Table 12 as seeds. Original and Augmented denote the numbers of seed instances (aug=False) and augmented instances (aug=True), respectively. Avg. #MASK denotes the average number of masked spans per instance.

A.4.2 Structure-Aware Encoder

We train the structure-aware encoder f_2 so that NLQs sharing the same abstracted structure are close, while structurally different but semantically related NLQs are not treated as harmful negatives.

Each training example i has a structure label w_i and an intent-group label g_i . The first identifies the abstracted query $\hat{y}$, while the second captures coarse semantic connectivity induced by the many-to-many relation between natural language queries and structured queries.

We use a two-level supervised contrastive objective:

$$\mathcal{L} = \mathcal{L}_{\text{struct}} + \lambda \mathcal{L}_{\text{group}}.$$

Let $e_i = f_2(x_i)$ be the normalized embedding of example i , and let $\gamma > 0$ be the temperature. Using the standard supervised contrastive form (Khosla et al., 2020),

$$\mathcal{L}_{\text{sup}}(\mathcal{P}, \mathcal{A}) = - \sum_i \frac{1}{|\mathcal{P}(i)|} \sum_{p \in \mathcal{P}(i)} \log \frac{\exp(e_i^\top e_p / \gamma)}{\sum_{a \in \mathcal{A}(i)} \exp(e_i^\top e_a / \gamma)},$$

we define

$$\mathcal{P}_{\text{struct}}(i) = \{p \neq i \mid w_p = w_i\}, \quad \mathcal{A}_{\text{struct}}(i) = \{a \neq i \mid g_a \neq g_i\} \cup \mathcal{P}_{\text{struct}}(i),$$

and

$$\mathcal{P}_{\text{group}}(i) = \{p \neq i \mid g_p = g_i\}, \quad \mathcal{A}_{\text{group}}(i) = \{a \neq i\}.$$

We then set

$$\mathcal{L}_{\text{struct}} = \mathcal{L}_{\text{sup}}(\mathcal{P}_{\text{struct}}, \mathcal{A}_{\text{struct}}), \quad \mathcal{L}_{\text{group}} = \mathcal{L}_{\text{sup}}(\mathcal{P}_{\text{group}}, \mathcal{A}_{\text{group}}).$$

Training configuration. For both tasks, training converges within 60 minutes in our experimental environment.

- Backbone model: BAAI/BGE-M3⁶
- Embedding dimension: 1,024
- Maximum training epochs: (Text-to-SQL) 6, (Text-to-Cypher) 10
- Batch size per device: (Text-to-SQL) 36, (Text-to-Cypher) 48
- Pooling strategy: CLS pooling
- Maximum input lengths for queries: 256
- Number of samples per label: (Text-to-SQL) 3, (Text-to-Cypher) 2
- Number of hard negatives: 4
- Optimizer: AdamW
- Learning rate: (Text-to-SQL) 1e-5, (Text-to-Cypher) 2e-5
- Loss temperature: 0.05
- Intent-group loss weight λ : (Text-to-SQL) 0.3, (Text-to-Cypher) 0.2
- Embedding normalization: L2
- Seed: 42

⁶<https://huggingface.co/BAAI/bge-m3>

A.4.3 Slot Filler

Each training example i consists of a natural language query x , an abstracted structured query $\hat{y}$, and the gold literal values corresponding to the masked spans $\mathcal{M}(\hat{y})$. We train the slot filler by supervising only the masked spans:

$$\mathcal{L}_{\text{fill}} = - \sum_{i=1}^N \sum_{t \in \mathcal{M}(\hat{y}_i)} \log p_{\phi}(y_{i,t} \mid y_{i,<t}, x_i, \hat{y}_i).$$

This objective focuses learning on slot completion and avoids redundant reconstruction of tokens that are already present in the retrieved abstraction.

For Text-to-SQL, the source benchmarks natively employ different SQL dialects: SQLite for EHRSQL and BIRD, and PostgreSQL for ScienceBenchmark. To train the slot filler robustly with abundant data for each dialect, we use SQLGlot⁷ to cross-translate SQLite queries into PostgreSQL and vice versa. This process allows us to construct two comprehensive, dialect-specific training datasets. For Text-to-Cypher, we directly use the Cypher-specific examples in Table 13.

Training configuration. For both tasks, training converges within 2.5 hours in our experimental environment.

- Backbone: google/t5-v1.1-large⁸
- Encoder input format: "fill literals in template <SEP> question:" question "<SEP> template:" template "<SEP> slots:" slots
- Decoder target format: "<extra_id_0>" literal_0 "<extra_id_1>" literal_1 ...
- Maximum source length: 512
- Maximum target length: 128
- Maximum training epochs: (Text-to-SQL) 8, (Text-to-Cypher) 12
- Batch size per device: 16
- Optimizer: AdamW
- Learning rate: (Text-to-SQL) 1e-4, (Text-to-Cypher) 3e-5
- Weight decay: 0.01
- Seed: 42

A.4.4 Cross-Benchmark Generalization and Maintenance

To assess maintenance under changes in schema, domain, and SQL dialect, we train each lightweight model separately on EHRSQL, ScienceBenchmark, and BIRD using the objectives and procedures described above, and evaluate them on EICU, ONCOMX, and CODE. These workloads span distinct schemas and domains; EICU and CODE use SQLite, whereas ONCOMX uses PostgreSQL.

As shown in Tables 14 and 15, source-matched models perform best, but cross-source drops are limited to 0.006–0.016 in Precision@1 and 0.2–1.2 percentage points in exact match, respectively. These small gaps indicate effective transfer across the evaluated schemas, domains, and SQL dialects, suggesting that TierCache can be maintained without per-schema retraining in these settings.

Training Benchmark	Evaluation Workload		
	EICU (EHRSQL)	ONCOMX (ScienceBenchmark)	CODE (BIRD)
EHRSQL	0.982	0.971	0.982
ScienceBenchmark	0.975	0.987	0.983
BIRD	0.976	0.971	0.996

Table 14: Cross-benchmark Precision@1 of the structure-aware encoders. Bold indicates source-matched evaluation. Higher is better.

⁷<https://sqlglot.com/sqlglot.html>

⁸<https://huggingface.co/google/t5-v1.1-large>

Training Benchmark	Evaluation Workload		
	EICU (EHRSQL)	ONCOMX (ScienceBenchmark)	CODE (BIRD)
EHRSQL	97.6	97.9	95.2
ScienceBenchmark	97.2	98.2	95.5
BIRD	97.4	97.9	96.4

Table 15: Cross-benchmark exact match (%) of the slot-filling models. Bold indicates source-matched evaluation. Higher is better.

A.5 SQStream Details

This section provides additional details on the construction of SQStream, a benchmark designed to evaluate semantic caching under query streams with recurring reusable structures. SQStream comprises ten workloads across two structured query generation tasks: six Text-to-SQL workloads and four Text-to-Cypher workloads. Despite differences in query language and database type, all workloads follow the same benchmark design principles and construction procedure.

A.5.1 Source Benchmark Splits

Table 16 summarizes the source benchmark splits used to construct SQStream. The Text-to-SQL workloads are constructed from the dev splits of EHRSQL (Lee et al., 2022), ScienceBenchmark (Zhang et al., 2023), and BIRD (Li et al., 2023), while the Text-to-Cypher workloads are constructed from the test split of CypherBench (Feng et al., 2025). Each workload is derived from a single underlying database, enabling evaluation across diverse schemas, domains, database systems, and query languages. Abbreviated database names used in the main text and experiments are shown in parentheses.

Task	Benchmark	Database	Source Size	Domain	Dialect	Evidence	Unseen DB
Text-to-SQL	EHRSQL	eicu (EICU)	1,117	Clinical	SQLite	No	No
		mimic_iii (MIMIC)	1,122	Clinical			
	Science Benchmark	cordis (CORDIS)	100	Scientific	PostgreSQL	No	No
		oncomx (ONCOMX)	99	Scientific			
	BIRD	codebase community (CODE)	186	Software	SQLite	Yes	Yes
		student_club (STUDENT)	158	University			
Text-to-Cypher	Cypher Bench	company (COMPANY)	251	Business	Cypher	No	Yes
		flight_accident (ACCIDENT)	149	Aviation			
		movie (MOVIE)	293	Entertainment			
		nba (NBA)	185	Sports			

Table 16: Statistics of the source benchmark splits used to construct SQStream. Source Size denotes the number of source instances for each database before workload construction. Evidence indicates whether the source benchmark provides additional information beyond the database schema. Unseen DB indicates whether the corresponding database is excluded from lightweight-model training.

A.5.2 SQStream Construction Procedure

Starting from the source benchmark instances, SQStream constructs a structural pair pool $\mathcal{P}$ by abstracting each natural language query and structured query pair (x, y) into $(\hat{x}, \hat{y})$. The abstraction masks only literals that are lexically grounded in the natural language query, preserving both structural reuse and semantic validity.

To generate evaluation workloads, SQStream samples structural pairs from $\mathcal{P}$ under a target distribution, instantiates the masked literals with values grounded in the corresponding target database, and retains only executable and non-trivial query instances. Here, non-trivial results exclude failed instantiations and degenerate cases such as empty or uninformative

Algorithm 3 SQStream Construction

Input: Seed set $S = \{(x_i, y_i)\}$, schema M , target database $\mathcal{B}$, target size N , distribution π_{target} , retry limit R

Output: Query stream W

```

1:  $\mathcal{P} \leftarrow \emptyset$ 
2: for all  $(x, y) \in S$  do
3:   Parse  $y$  into AST  $A$  using a query-language-specific parser
4:    $L \leftarrow \text{EXTRACTLITERALSPANS}(A)$ 
5:    $L_g \leftarrow \emptyset$ 
6:    $\hat{y} \leftarrow y$ 
7:   for all  $\ell \in L$  do
8:     if  $\text{LEXICALLYGROUNDED}(\ell, x)$  then
9:        $\hat{y} \leftarrow \text{REPLACE}(\hat{y}, \ell, [\text{MASK}])$ 
10:       $L_g \leftarrow L_g \cup \{\ell\}$ 
11:     end if
12:   end for
13:    $\hat{x} \leftarrow \text{MASK}(x, L_g)$ 
14:   Add  $(\hat{x}, \hat{y})$  to  $\mathcal{P}$ 
15: end for
16: Merge entries with identical  $\hat{y}$  in  $\mathcal{P}$ 
17:  $W \leftarrow \emptyset$ 
18: while  $|W| < N$  do
19:   Sample  $(\hat{x}, \hat{y}) \in \mathcal{P}$  according to  $\pi_{\text{target}}$ 
20:    $\text{attempt} \leftarrow 0$ 
21:   while  $\text{attempt} < R$  do
22:     Sample literals from target database  $\mathcal{B}$ 
23:     Instantiate  $(x', y')$ 
24:     if  $y'$  executes successfully and yields non-trivial results then
25:       Add  $(x', y')$  to  $W$ 
26:       break
27:     end if
28:      $\text{attempt} \leftarrow \text{attempt} + 1$ 
29:   end while
30: end while
31: return  $W$ 

```

execution outputs. The same construction procedure is applied to both Text-to-SQL and Text-to-Cypher, using query-language-specific parsing and execution. The full construction pipeline is described in Algorithm 3.

A.5.3 Evaluation Workloads

Each SQStream workload is constructed by sampling structural pairs under a predefined target distribution. In real-world workloads, query distributions are often skewed, with a small number of recurring query patterns accounting for a large fraction of requests (Luo et al., 2002; van Renen et al., 2024). To approximate this property in a simple and controllable manner, we adopt a length-based power-law distribution over structural pairs, motivated by prior observations that query length exhibits power-law behavior (Arampatzis & Kamps, 2008; Petersen et al., 2016).

Formally, the sampling probability is defined as:

$$\tilde{p}_i \propto \ell_i^{-s},$$

where ℓ_i denotes the NLQ length and s is the scaling parameter. We fix $s = 2.0$ for all workloads. Figures 6 and 7 visualize the resulting structural-pair frequency distributions for the Text-to-SQL and Text-to-Cypher workloads, respectively.

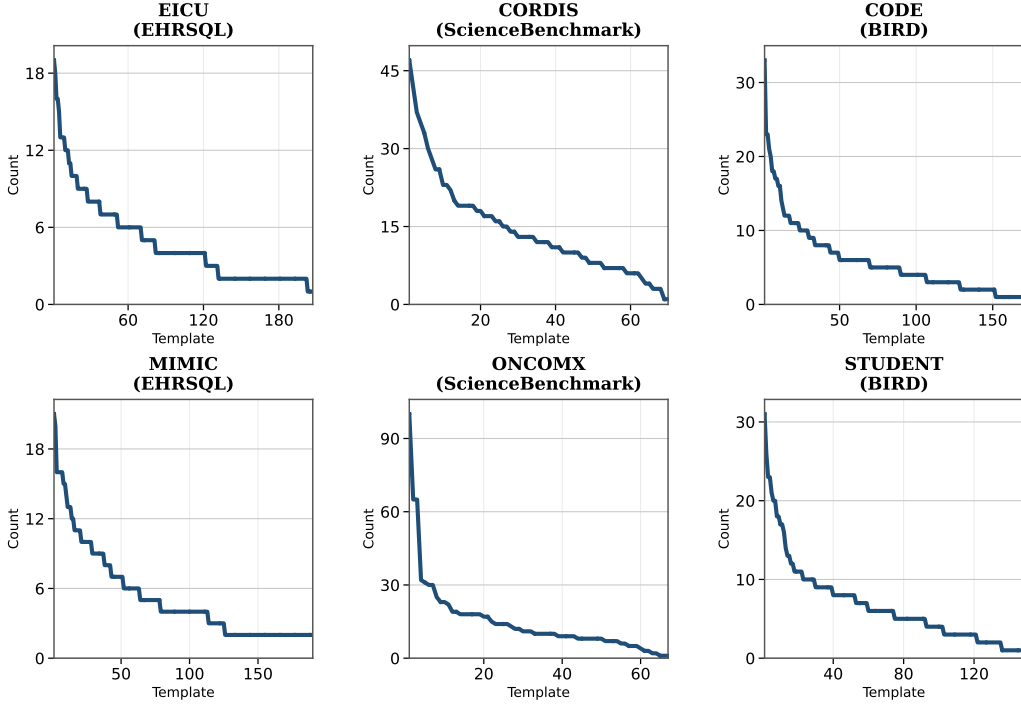


Figure 6: Structural-pair frequency distributions of the Text-to-SQL workloads in SQStream under length-based power-law sampling, with structural pairs sorted in descending order of frequency.

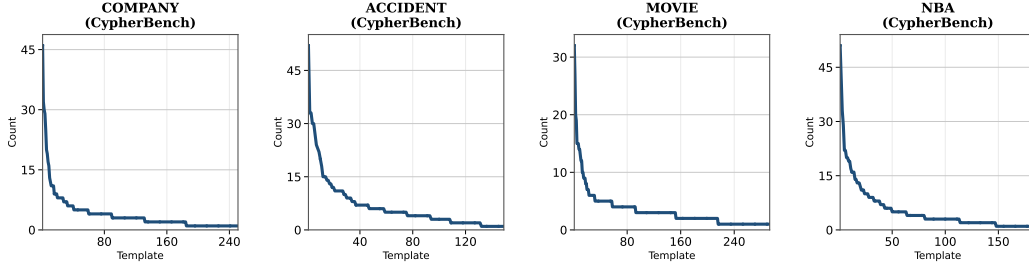


Figure 7: Structural-pair frequency distributions of the Text-to-Cypher workloads in SQStream under length-based power-law sampling, with structural pairs sorted in descending order of frequency.

Table 17 reports the query composition of all ten workloads. Queries are categorized into three mutually exclusive types based on their preceding stream history: *semantic*, if an identical gold structured query has previously appeared; *structural*, if no identical gold structured query exists but the same abstracted gold structure has appeared; and *novel*, otherwise. Each workload contains 1,000 queries.

Task	Source	Workload	Query Composition			
			Novel	Semantic	Structural	Total
Text-to-SQL	EHRSQL	EICU	207	264	529	1,000
		MIMIC	190	312	498	1,000
	ScienceBenchmark	CORDIS	70	611	319	1,000
		ONCOMX	67	710	223	1,000
	BIRD	CODE	170	369	461	1,000
		STUDENT	148	462	390	1,000
Text-to-Cypher	CypherBench	COMPANY	251	136	613	1,000
		ACCIDENT	149	174	677	1,000
		MOVIE	293	129	578	1,000
		NBA	185	339	476	1,000

Table 17: Query composition of the SQStream workloads used in experiments. Semantic, structural, and novel denote mutually exclusive query types determined by the preceding query-stream history.

A.6 Additional Results for Text-to-SQL

A.6.1 Base LLM Performance

For Text-to-SQL, we use OmniSQL-32B⁹ as the base LLM. Table 18 reports its execution accuracy (EX) and average end-to-end latency (sec), which serve as the no-cache baselines in Section 5. We use the official prompt template shown in Figure 8.

Workload	EHRSQL		ScienceBenchmark		BIRD	
	EICU	MIMIC	CORDIS	ONCOMX	CODE	STUDENT
Execution Accuracy (EX; %)	33.60	24.80	51.50	63.00	64.50	63.70
Average Latency (sec)	24.99	27.12	23.23	25.20	20.95	19.04

Table 18: OmniSQL-32B performance on the Text-to-SQL workloads of SQStream.

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 Database Schema: {db_schema} 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: <pre> ''' -- Your SQL query ''' </pre> Take a deep breath and think step by step to find the correct SQL query.
--

Figure 8: OmniSQL prompt template.

⁹<https://huggingface.co/seeklhy/OmniSQL-32B>

A.6.2 TweakLLM Re-implementation and Prompt Adaptation

As the official TweakLLM implementation is currently unavailable, we re-implement its retrieval-and-regeneration pipeline following the paper description and use the recommended auxiliary model (Meta-Llama-3.1-8B-Instruct¹⁰). We first validate our implementation under the original TweakLLM configuration, using its original prompt and sentence-transformers/all-MiniLM-L6-v2¹¹ encoder. Table 19 compares cache hit rates at $\tau = 0.8$, the setting for which the original paper reports quantitative results. Our implementation differs from the reported results by at most 0.4 percentage points.

For structured query generation, the original general-purpose response-adaptation prompt occasionally produces invalid outputs, including responses containing no SQL or Cypher query. We therefore use a task-specific prompt adapted from the respective base-LLM prompt templates, each incorporating the database schema and explicit constraints on structured query generation. Figure 9 shows an example of the resulting Text-to-SQL prompt used in our experiments, adapted from the OmniSQL prompt shown in Figure 8. As shown in Table 20, this adaptation improves EX by 2.2–8.3 percentage points across all Text-to-SQL workloads of SQStream at $\tau = 0.9$.

Dataset	Reported (%)	Re-implementation (%)
LMSYS (Zheng et al., 2024a)	68.0	67.6
WildChat (Zhao et al., 2024)	40.0	39.7

Table 19: Cache hit rates (%) reported in TweakLLM and obtained from our re-implementation.

Prompt	EICU	MIMIC	CORDIS	ONCOMX	CODE	STUDENT
Original	28.0	21.2	49.5	44.7	45.1	50.7
Task-specific	30.5	23.4	51.8	48.8	53.4	53.9

Table 20: Execution accuracy (EX; %) of the TweakLLM re-implementation with original and task-specific prompts across the Text-to-SQL workloads of SQStream ($\tau = 0.9$).

¹⁰<https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct>

¹¹<https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>

Task Overview:

You are a data science expert. You are given a database schema, a new natural language question, and a cached SQL query that was previously written for a similar question.

Your task is to adapt the cached SQL query so that it correctly answers the new question.

Database Engine:

SQLite

Database Schema:

{db_schema}

This schema describes the database's structure, including tables, columns, primary keys, foreign keys, and any relevant relationships or constraints.

New Question:

{question}

Similar Cached Question:

{cached_question}

Cached SQL:

{cached_sql}

Instructions:

- Reuse the cached SQL as much as possible, but modify it wherever necessary so that it answers the new question correctly.
- Pay careful attention to changed entities, polarity, comparators, aggregation, filters, sort order, limits, and selected columns.
- Make sure you only output the information that is asked in the new question.
- If the cached SQL is not useful, write a new SQL query from scratch.
- Before generating the final SQL query, please think through the steps of how to adapt the cached SQL.

Output Format:

In your answer, please enclose the generated SQL query in a code block:

```
'''
```

```
-- Your SQL query
```

```
'''
```

Take a deep breath and think step by step to find the correct SQL query.

Figure 9: Prompt template for TweakLLM’s auxiliary model, adapted for Text-to-SQL.

A.6.3 Strict Quality-Preserving Operating Points

We define a *strict quality-preserving constraint* as $EX > 0.95 \cdot EX_{\text{no-cache}}$ in Section 5. Table 21 reports the threshold settings (τ) that satisfy this constraint for each method across the Text-to-SQL workloads.

Method	EHRSQL		ScienceBenchmark		BIRD	
	EICU	MIMIC	CORDIS	ONCOMX	CODE	STUDENT
GPTCache	0.95 , 0.99	0.95 , 0.99	0.95 , 0.99	0.99	0.95 , 0.99	0.99
TweakLLM	0.95 , 0.99	0.95 , 0.99	0.90 , 0.95, 0.99	0.99	0.95 , 0.99	0.95 , 0.99
vCache	0.95 , 0.99	0.95 , 0.99	0.95 , 0.99	0.99	0.95 , 0.99	0.99
TierCache	0.90 , 0.95, 0.99	0.90 , 0.95, 0.99	0.95 , 0.99	0.99	0.95 , 0.99	0.95 , 0.99

Table 21: Threshold settings (τ) that satisfy the strict quality-preserving constraint for each method across the Text-to-SQL workloads. For TierCache, $\tau = \tau_1 = \tau_2$. Bold indicates the lowest-latency setting that satisfies this constraint.

A.6.4 Cache Hit Rates and Token Saving Rates

Tables 22, 23, and 24 detail the cache hit rates, saved input token rates, and saved output token rates, respectively, for each Text-to-SQL workload at the lowest-latency operating point satisfying the strict quality-preserving constraint. TierCache improves cache hit rates by 18.90–47.81 percentage points ($1.28 \times$ – $2.76 \times$ the best-baseline rates). These gains translate into substantial token reductions, improving saved input and output token rates by 18.66–49.25 percentage points ($1.28 \times$ – $2.86 \times$) and 19.25–49.11 percentage points ($1.29 \times$ – $2.72 \times$), respectively.

Method	EHRSQL		ScienceBenchmark		BIRD	
	EICU	MIMIC	CORDIS	ONCOMX	CODE	STUDENT
GPTCache	27.05	31.30	58.80	66.30	39.30	43.30
TweakLLM	27.15	31.20	62.46	66.50	39.35	45.85
vCache	27.05	31.35	58.80	66.10	39.30	43.20
TierCache	74.96	65.46	83.15	85.40	70.80	79.65

Table 22: Cache hit rates (%) across the Text-to-SQL workloads at the lowest-latency operating point that satisfies the quality constraint for each method.

Method	EHRSQL		ScienceBenchmark		BIRD	
	EICU	MIMIC	CORDIS	ONCOMX	CODE	STUDENT
GPTCache	26.49	30.91	59.00	65.68	37.67	37.98
TweakLLM	6.70	12.43	35.89	40.71	18.03	18.90
vCache	26.49	30.81	59.31	65.79	37.67	37.88
TierCache	75.74	68.19	85.47	84.45	67.89	72.36

Table 23: Saved input token rates (%) across the Text-to-SQL workloads at the lowest-latency operating point that satisfies the quality constraint for each method.

Method	EHRSQL		ScienceBenchmark		BIRD	
	EICU	MIMIC	CORDIS	ONCOMX	CODE	STUDENT
GPTCache	28.61	31.93	58.91	66.50	42.70	43.80
TweakLLM	10.72	16.95	39.55	45.12	24.60	23.78
vCache	28.61	31.81	58.91	66.27	42.70	43.70
TierCache	77.72	70.11	86.62	85.75	73.68	80.11

Table 24: Saved output token rates (%) across the Text-to-SQL workloads at the lowest-latency operating point that satisfies the quality constraint for each method.

A.6.5 Cumulative Latency Under the Strict Quality-Preserving Constraint

Figure 10 illustrates the cumulative latency trajectories for each method at the lowest-latency operating point satisfying the quality-preserving constraint. TierCache maintains the lowest cumulative latency across all six workloads, achieving $2.01\times$ – $2.89\times$ speedups over the best baselines by the end of each query stream. Furthermore, evaluations of TierCache variants (T1-only and T2-only) show that both single-tier configurations incur higher latency than the full framework. This demonstrates that either tier alone is less efficient under the quality constraint, whereas combining Tier 1 and Tier 2 further reduces end-to-end latency while maintaining execution accuracy.

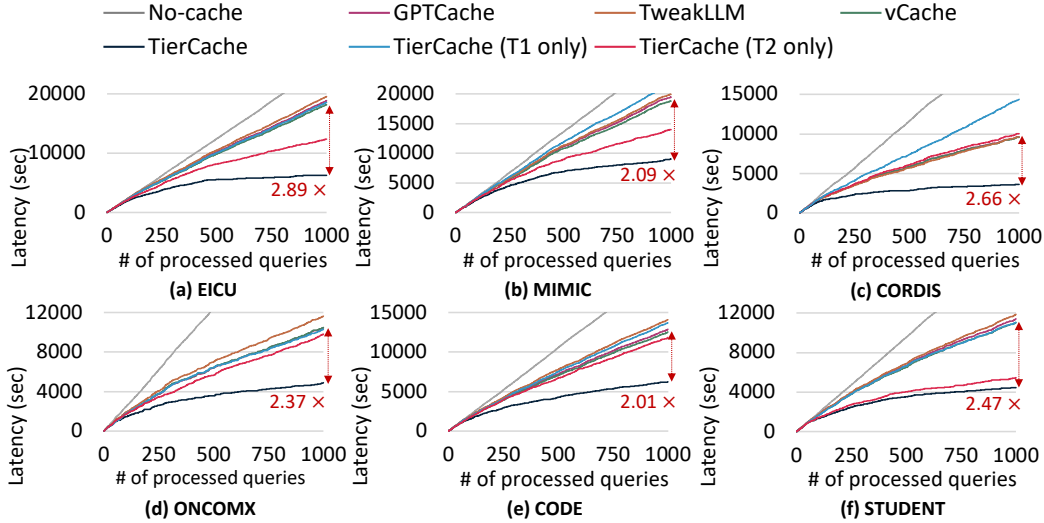


Figure 10: Cumulative latency (sec) for each Text-to-SQL workload under the strict quality-preserving constraint, defined as $EX > 0.95 \cdot EX_{\text{no-cache}}$. Red annotations indicate the end-to-end speedup of TierCache over the best baselines, excluding our internal ablation variants (T1 only and T2 only).

A.7 Text-to-Cypher Experiments

These experiments run on two NVIDIA A100 GPUs (80 GB each) in the same server used for the main-text experiments (Section 5). We use Qwen2.5-72B-Instruct as the base LLM and serve it with vLLM across the two GPUs. The structure-aware encoder and slot filler are trained using the Cypher-specific data described in Appendix A.4.1. Because the training data are drawn from databases disjoint from those used by the Text-to-Cypher workloads of SQStream, all four workloads follow the unseen-database setting. The baseline configurations, underlying storage, retrieval infrastructure, cold-start protocol, and cache-capacity setting follow Section 5. We evaluate $\tau \in \{0.9, 0.95, 0.99\}$ and select the lowest-latency operating point satisfying the strict quality-preserving constraint, $EX > 0.95 \cdot EX_{\text{no-cache}}$, for each method and workload.

A.7.1 Base LLM Performance

We use Qwen2.5-72B-Instruct¹² as the base LLM. Table 25 reports its execution accuracy (EX) and average end-to-end latency (sec), which serve as the no-cache baselines in this section. We use the CypherBench prompt template¹³, which is detailed in Figure 11.

¹²<https://huggingface.co/Qwen/Qwen2.5-72B-Instruct>

¹³<https://github.com/megagonlabs/cypherbench>

Workload	CypherBench			
	COMPANY	ACCIDENT	MOVIE	NBA
Execution Accuracy (EX; %)	50.5	55.3	49.7	70.1
Average Latency (sec)	2.64	2.81	2.83	2.39

Table 25: Qwen2.5-72B-Instruct performance on the Text-to-Cypher workloads of SQStream.

Translate the question to Cypher query based on the schema of a Neo4j knowledge graph. - Output the Cypher query in a single line, without any additional output or explanation. Do not wrap the query with any formatting like ``.` - Perform graph pattern matching in the 'MATCH' clause if possible. - Avoid listing the same entity multiple times in the results. However, if multiple distinct entities share the same name, their names should be repeated as separate entries. - Do not return node objects. Instead, return entity names or properties. Graph Schema: {db_schema} Question: {question} Cypher:

Figure 11: CypherBench prompt template.

A.7.2 Strict Quality-Preserving Operating Points

Table 26 reports the threshold settings (τ) that satisfy the strict quality-preserving constraint, $EX > 0.95 \cdot EX_{\text{no-cache}}$, for each method across workloads.

Method	CypherBench			
	COMPANY	ACCIDENT	MOVIE	NBA
GPTCache	0.95 , 0.99	0.99	0.95 , 0.99	0.99
TweakLLM	0.90 , 0.95, 0.99	0.95 , 0.99	0.90 , 0.95, 0.99	0.95 , 0.99
vCache	0.95 , 0.99	0.99	0.95 , 0.99	0.99
TierCache	0.90 , 0.95, 0.99	0.95 , 0.99	0.90 , 0.95, 0.99	0.95 , 0.99

Table 26: Threshold settings (τ) that satisfy the strict quality-preserving constraint for each method across the Text-to-Cypher workloads. For TierCache, $\tau = \tau_1 = \tau_2$. Bold indicates the lowest-latency setting that satisfies this constraint.

A.7.3 Experimental Results

Table 27 and Table 28 report the latency and cache hit rate at the lowest-latency operating point satisfying the strict quality-preserving constraint. No-cache latency is only 2.39–2.83s, substantially lower than for Text-to-SQL. This difference stems from the query-length gap between the workloads for the two tasks: Cypher queries in SQStream average 13.64–15.73 tokens, whereas SQL queries average 40.15–99.55 tokens. This leaves less generation cost for caching to eliminate: the strongest caching baseline achieves only $1.06\times$ – $1.17\times$ speedups over the no-cache baseline, with cache hit rates of 15.4%–38.6%.

Even in this low-latency setting, TierCache reduces latency to 0.88–1.68s and achieves cache hit rates of 68.3%–94.1%. These results correspond to $1.48\times$ – $2.99\times$ speedups and $2.02\times$ – $4.44\times$ the cache hit rates of the strongest baselines for each workload. The gains are particularly notable on ACCIDENT and MOVIE, where TierCache achieves hit rates of

94.1% and 90.0% and speedups of $2.99\times$ and $2.81\times$, respectively, highlighting the additional coverage provided by partial reuse of recurring query structures. Moreover, because all four workloads use graph databases unseen during lightweight-model training, the consistent gains demonstrate generalization to unseen graph schemas and extend the effectiveness of multi-granularity reuse beyond Text-to-SQL. As Text-to-Cypher workloads evolve toward longer and more complex queries, their greater generation costs may provide further opportunities for caching gains.

Method	CypherBench			
	COMPANY	ACCIDENT	MOVIE	NBA
No-cache	2.64	2.81	2.83	2.39
GPTCache	2.60	2.78	2.79	2.21
TweakLLM	2.78	2.63	2.73	2.24
vCache	2.49	2.65	2.67	2.04
TierCache	1.68	0.88	0.95	1.29

Table 27: Average latency (sec) across the Text-to-Cypher workloads under the strict quality-preserving constraint. Lower is better.

Method	CypherBench			
	COMPANY	ACCIDENT	MOVIE	NBA
GPTCache	15.4	17.8	14.6	33.4
TweakLLM	15.4	25.1	21.2	38.6
vCache	15.4	17.8	14.5	33.3
TierCache	68.3	94.1	90.0	77.9

Table 28: Cache hit rates (%) across the Text-to-Cypher workloads at the operating points reported in Table 27. Higher is better.