Memo-SQL: Structured Decomposition and Experience-Driven Self-Correction for Training-Free NL2SQL

Zerui Yang[1], Weichuan Wang[1], Yanwei Xu[*2], Linqi Song[†1], Yudai Matsuda[1], Wei Han[2], and Bo Bai[2]

[1]City University of Hong Kong, Hong Kong, PR China
[2]Huawei Technologies Ltd.

Abstract

Existing NL2SQL systems face two critical limitations : (1) they rely on in-context learning with only correct examples, overlooking the rich signal in historical error–fix pairs that could guide more robust self-correction; and (2) test-time scaling (TTS) approaches often decompose questions arbitrarily, producing near-identical SQL candidates across runs and diminishing ensemble gains. Moreover, these methods suffer from a stark accuracy–efficiency trade-off: high performance demands excessive computation, while fast variants compromise quality. We present Memo-SQL, a training-free framework that addresses these issues through two simple ideas: structured decomposition and experience-aware self-correction. Instead of leaving decomposition to chance, we apply three clear strategies, entity-wise, hierarchical, and atomic sequential, to encourage diverse reasoning. For correction, we build a dynamic memory of both successful queries and historical error–fix pairs, and use retrieval-augmented prompting to bring relevant examples into context at inference time, no fine-tuning or external APIs required. On BIRD, Memo-SQL achieves 68.5% execution accuracy, setting a new state of the art among open, zero-fine-tuning methods, while using over 10× fewer resources than prior TTS approaches.

Contact: zeruiyang2-c@my.cityu.edu.hk

1 Introduction

Recent progress in NL2SQL has been heavily reliant on either fine-tuning large language models (LLMs) (Qin et al., 2024; Li et al., 2024b) or invoking powerful but closed-source APIs (Pourreza et al., 2024; Liu et al., 2025b). While these approaches achieve strong performance on standard benchmarks, they suffer from clear drawbacks, most notably, an inability to incorporate dynamic feedback and poor generalization beyond static training setups. This has spurred growing interest in TTS (Guan et al., 2025; Yang et al., 2025), a paradigm that enhances inference-time computation without modifying model parameters, thereby enabling fully open, training-free solutions based on publicly available LLMs. Despite its promise, current TTS methods (Yuan et al., 2025; Lee et al., 2024) for NL2SQL still suffer from several fundamental limitations that hinder both effectiveness and practicality.

First, they often struggle to balance accuracy and efficiency, with high-performing systems incurring prohibitive computational costs (Qin et al., 2024; Li et al., 2025b).

Second, many divide-and-conquer frameworks delegate the entire question decomposition process to the LLM itself (Wang et al., 2023; Pourreza et al., 2024). While seemingly flexible, this approach often produces highly similar or even identical sub-question sequences across different runs (Liao et al., 2025). The resulting SQL candidates are strongly correlated, which severely limits the utility of ensemble techniques like majority voting, precisely because meaningful diversity in reasoning paths is absent. Although some recent works combine divide-and-conquer with external planning modules or tool-augmented reasoning (Pourreza et al., 2024; Li et al., 2025a), our focus lies squarely on improving the core decomposition mechanism itself, without relying on auxiliary components, to unlock its full potential under a pure test-time scaling setting.

Third, existing self-correction mechanisms typically rely on static few-shot prompts that embed fixed examples of error-fix patterns (Pourreza and Rafiei, 2023; Xie et al., 2024). These static demonstrations primarily serve to familiarize the model with the task format rather than genuinely enhancing its ability to diagnose and correct novel er-

[*]Corresponding author
[†]Corresponding author

Findings of the Association for Computational Linguistics: ACL 2026, pages 5130–5148

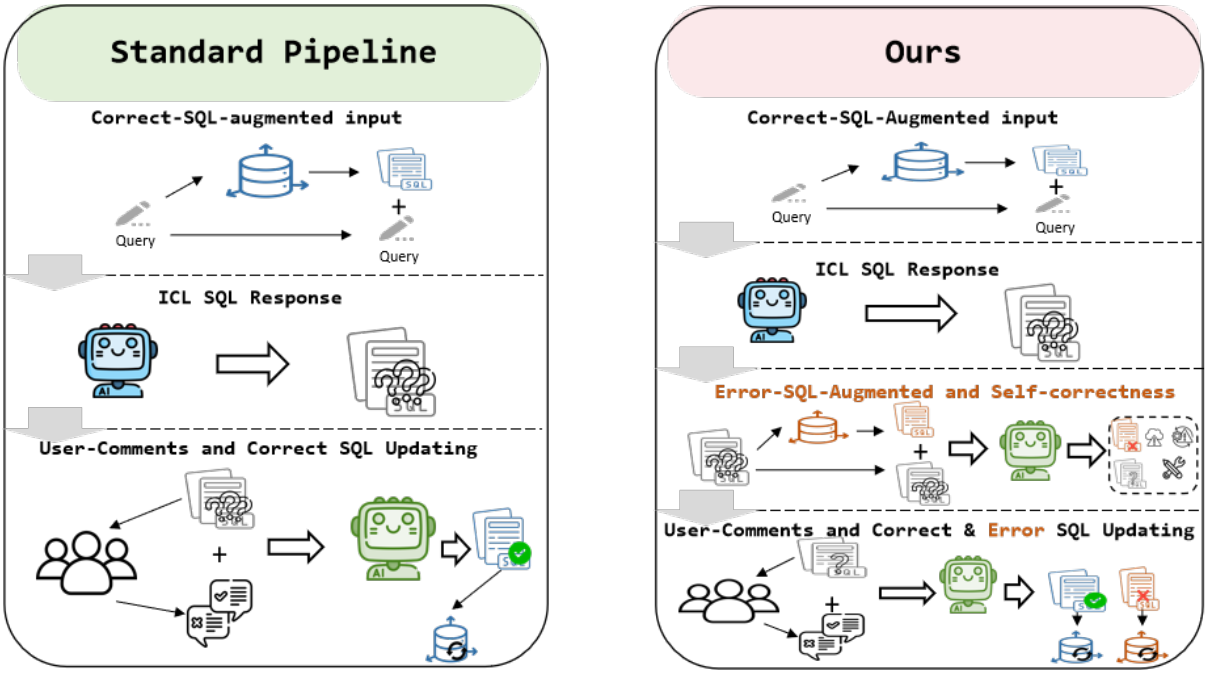

Figure 1: Conceptual comparison between a standard user-in-the-loop NL2SQL workflow and our proposed self-correction framework. In the standard approach, the model generates an initial SQL query, which is then revised through explicit user feedback, a process that typically stores only correct examples in the experience repository. In contrast, our framework introduces an automatic self-correction step after generation: it retrieves relevant error–correction pairs from a dynamic error-correction memory (including past failures) and refines the output without requiring user intervention, enabling fully training-free adaptation.

rors. Consequently, such designs cannot adapt to novel error types, evolving query formulations, or domain-specific schema quirks, often resulting in ineffective self-correction or requiring users to provide multiple rounds of clarification (Huang et al., 2023; Askari et al., 2025; Zhao et al., 2024). Moreover, they treat all past interactions as transient, discarding valuable feedback from user corrections, a rich source of supervision that could guide more robust and generalizable refinement.

To address these challenges, we propose Memo-SQL, a training-free TTS framework that operates exclusively with open-source LLMs and requires no external APIs or parameter updates. Memo-SQL delivers three key contributions:

(1) A principled divide-and-conquer framework for NL2SQL. Rather than leaving decomposition to the stochastic discretion of the LLM, an approach that often collapses into repetitive reasoning patterns, we formalize SQL generation as a structured divide-and-conquer process. Our framework explicitly decomposes natural language questions along multiple semantic axes (e.g., entity relations and operational sequencing), ensuring broad coverage of compositional structures and promoting diversity in the resulting candidate programs.

(2) Experience-guided self-correction via error-aware in-context learning. While retrieval-augmented in-context learning has been used to retrieve successful demonstrations (Li et al., 2025c; Talaei et al., 2024), we repurpose this mechanism for learning from failures. Specifically, we retrieve historical error-correction pairs, not just correct queries, and inject them as dynamic in-context examples during self-correction. This allows the model to explicitly reason about common failure modes and their fixes, enabling adaptive refinement that generalizes across error types and domains, without fine-tuning or handcrafted rules.

(3) State-of-the-art performance with high efficiency. Evaluated on the BIRD benchmark (Li et al., 2023a), Memo-SQL achieves 68.5% execution accuracy on the dev-new set, setting a new state of the art among all open, zero-fine-tuning methods. Remarkably, it reduces computational overhead by over an order of magnitude compared to leading TTS approaches, demonstrating that principled test-time reasoning can simultaneously achieve high accuracy, strong adaptability, and low latency.

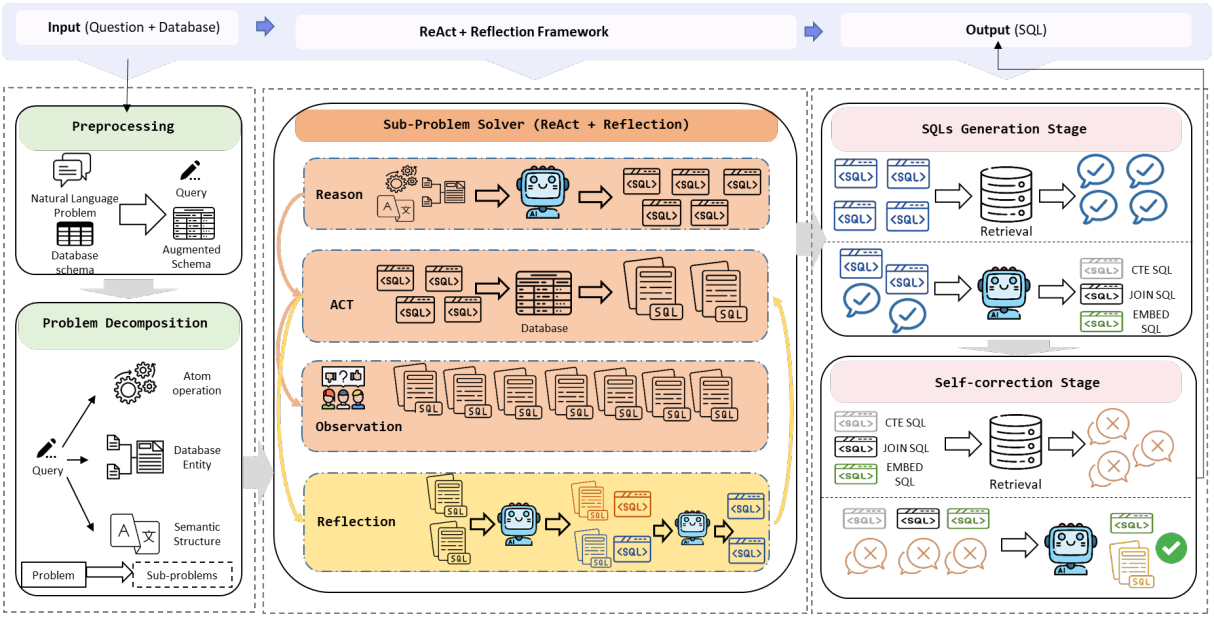

Figure 2: Overview of the Memo-SQL framework, which integrates problem decomposition, ReAct+Reflection reasoning, and self-correction to generate accurate SQL queries. The pipeline begins with preprocessing and multi-strategy question decomposition. Each sub-problem is solved via an iterative ReAct+Reflection loop: (1) reasoning about semantics, (2) generating a sub-SQL query, (3) observing execution results, and (4) reflecting on potential errors to enable one-step correction. Finally, multiple SQL candidates are synthesized using few-shot in-context learning across three syntactic styles (CTE, flat JOIN, nested), followed by error-aware refinement, guided by a error-correction memory, and selection via self-consistency scoring.

2 Preliminary

2.1 Divide-and-Conquer for NL2SQL

Given a natural language question q and a database schema $\mathcal{S}$, the goal of NL2SQL is to generate an executable SQL query y. In divide-and-conquer approaches, x is first decomposed into a sequence of sub-questions $\{x_1, x_2, \ldots, x_k\}$, each targeting a specific aspect of $\mathcal{S}$ (e.g., entities, joins, or aggregations). The corresponding SQL fragments $\{y_1, \ldots, y_k\}$ are then composed into the final query:

$$y = \texttt{Compose}(y_1, y_2, \ldots, y_k), \quad \text{where } y_i = \texttt{Gen}(x_i; \theta). \tag{1}$$

When decomposition is left entirely to the LLM (i.e., $\{x_i\}$ are sampled stochastically), the resulting $\{y_i\}$ often exhibit low structural diversity (Liao et al., 2025), limiting ensemble robustness.

2.2 In-Context Learning for Self-Correction

Current self-correction methods use static in-context learning (Pourreza and Rafiei, 2023; Xie et al., 2024). A fixed set of error–correction demonstrations $\mathcal{D}_{\text{static}} = \{(y_j^{\text{err}}, y_j^{\text{corr}})\}_{j=1}^{m}$ is embedded into the prompt:

$$p_{\text{prompt}} = [\mathcal{D}_{\text{static}}; y_{\text{new}}^{\text{err}}], \tag{2}$$

and the model generates a corrected version $y_{\text{new}}^{\text{corr}} = \texttt{LLM}(p_{\text{prompt}})$. Since $\mathcal{D}_{\text{static}}$ is handcrafted and static, it cannot adapt to unseen error patterns or domain shifts.

2.3 Experience-Aware Self-Correction via Retrieval-Augmented ICL

As outlined in Section A.1 and Figure 1, real-world BI systems log both user feedback and system revisions, forming a rich repository of error–correction experiences. Inspired by this practice, we maintain a dynamic experience repository $\mathcal{M} = \{(x^{(i)}, y_{(i)}^{\text{err}}, y_{(i)}^{\text{corr}})\}_{i=1}^{N}$. During inference, given a newly generated (possibly incorrect) query y^{err}, we retrieve the top-k most relevant error–correction pairs using a retriever $\mathcal{R}$:

$$\mathcal{D}_{\text{retrieved}} = \mathcal{R}(y^{\text{err}}; \mathcal{M}) = \underset{(y_{(i)}^{\text{err}}, y_{(i)}^{\text{corr}}) \in \mathcal{M}}{\arg\text{top-k}} \ \text{sim}(y^{\text{err}}, y_{(i)}^{\text{err}}), \tag{3}$$

where $\text{sim}(\cdot, \cdot)$ is a similarity function (e.g., embedding cosine similarity). These retrieved examples are then used as in-context demonstrations:

$$y^{\text{corr}} = \texttt{LLM}\big([\mathcal{D}_{\text{retrieved}}; y^{\text{err}}]\big). \tag{4}$$

This enables adaptive, experience-driven self-correction without fine-tuning or external APIs.

3 Methodology

3.1 Overview

The workflow of Memo-SQL comprises two phases: an offline preparation phase and an online inference phase.

In the offline phase, we construct an error-correction memory, a structured repository of quintuples $\langle q, s^+, s^-, \mathcal{E}, \delta \rangle$, each linking a natural language question q to a correct SQL query s^+, an erroneous counterpart s^-, its semantic error type(s) $\mathcal{E}$, and actionable correction suggestions δ.

During online inference, given a new natural language question, the system applies three complementary decomposition strategies, entity-wise, hierarchical, and atomic sequential, in parallel. For each decomposition path, the model independently executes a full ReAct+Reflect loop over its sub-questions: it (i) *reasons* about the sub-task semantics, (ii) *acts* by generating a sub-SQL query, (iii) *observes* the execution result from the database, and (iv) *reflects* on potential issues (e.g., data validity violations or semantic misalignment) based on the observed output. If reflection detects an error, the model revises the sub-query once and re-executes it to obtain a corrected intermediate result.

Because the three decomposition paths operate independently and produce complete end-to-end SQL candidates, our framework naturally implements a best-of-N ($N = 3$) selection scheme.

After resolving all sub-problems, the system compiles the full reasoning trace and synthesizes multiple candidate SQL queries for the original question via few-shot in-context learning. The prompt includes exemplars of (q, s^+) pairs, guiding the model to generate three syntactically diverse candidates: (1) CTE-based, (2) flat JOIN-based, and (3) nested subquery-based.

For each candidate, the system retrieves the most similar historical failure–fix instances from the error-correction memory, using a signature formed by concatenating the input question and the skeletal structure of the candidate SQL. These retrieved exemplars are injected into the prompt to enable error-aware refinement.

Finally, the system selects the best refined candidate based on a self-consistency score, which measures agreement across independently generated solutions, a proxy for robustness against decoding stochasticity.

3.2 Offline Phase: Constructing an Error-Correction Memory

To support error-aware refinement at inference time, we curate an error-correction memory: a structured knowledge repository that stores paired evidence of SQL failures and their fixes. An illustrative case is shown in Figure 7.

We first define a taxonomy of nine common semantic error types (excluding pure syntax issues), covering, e.g., incorrect join constraints, missing grouping keys, and mismatched aggregation scopes (Table 4). This taxonomy serves as a controlled vocabulary to represent failure modes and enables targeted correction.

Given a training corpus (BIRD training set; Li et al., 2023a), we elicit multiple diverse SQL candidates per question and retain instances exhibiting non-trivial semantic failures. For each such instance, we select a semantically plausible yet incorrect query as the representative erroneous query s^-, and pair it with the ground-truth SQL s^+. Conditioned on the natural language question q, gold SQL s^+, erroneous SQL s^-, and the database schema, a model is prompted to (i) assign one or more error types from the taxonomy and (ii) produce actionable correction suggestions. Each memory entry thus forms a structured quintuple $\langle q, s^+, s^-, \mathcal{E}, \delta \rangle$, explicitly linking an observed failure s^- to its correction rationale $\{\mathcal{E}, \delta\}$.

For efficient retrieval, each entry is encoded by concatenating the question with the skeletal structure of s^- and embedding the result using GTE-base (Li et al., 2023b). During inference, this representation enables retrieval of historically similar failure–fix patterns, which are injected as in-context evidence to guide the refinement of newly generated SQL candidates.

3.3 Question Decomposition

In complex natural NL2SQL tasks, accurate semantic parsing often benefits from decomposing a user question into simpler, executable sub-queries (Liao et al., 2025). We identify and formalize three complementary decomposition paradigms that reflect common reasoning patterns in multi-table relational querying:

Table-wise Decomposition. Identifies all relevant tables/entities mentioned or implied in the

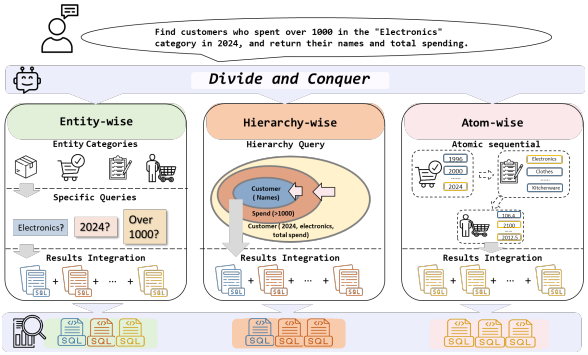

Figure 3: Illustration of the three complementary question decomposition strategies.

question and generates localized sub-queries per table (with table-specific filters and projections). These are later joined via foreign keys, ideal for multi-entity conjunctive queries (e.g., "customers who bought products in New York").

Hierarchical Decomposition. Handles nested linguistic constructs (e.g., "higher than average," "never ordered") by recursively decomposing the query from innermost to outermost semantics. Inner sub-queries compute aggregates or existence conditions that parameterize outer clauses, naturally aligning with SQL subquery patterns like `NOT EXISTS` or `IN`.

Atomic Operational Decomposition. Breaks the question into a sequence of atomic relational operations, selection, projection, join, grouping, and aggregation, guided by linguistic cues (e.g., "total," "per department"). Each step maps to a valid SQL fragment, enabling incremental composition and fine-grained error localization.

Figures 3 and 5 illustrate how these three decomposition strategies produce distinct reasoning paths for the same input question, highlighting their complementary roles in generating diverse and accurate SQL candidates.

Fallback Mechanism. Not all questions admit a meaningful decomposition under the above paradigms. Accordingly, we incorporate a fallback mechanism: if the model judges that a question is too simple to decompose, or that the targeted decomposition is ill-specified or would introduce unnecessary complexity, it falls back to either (i) a randomized decomposition, or (ii) no decomposition, preserving the original question for direct SQL generation.

3.4 SQL Generation

In this component, we adopt the ReAct+Reflect framework to generate SQL queries and proactively correct potential errors. Specifically, each sub-question is sequentially fed into the language model, which performs reasoning followed by generating a sub-SQL query (act). The generated sub-SQL is then executed to obtain intermediate results (observation). Subsequently, the model receives the current and all previously adopted strategies, and critically examines whether the current sub-SQL query contains potential issues, such as data validity violations or semantic misalignment. If errors are detected, the model revises the sub-SQL query accordingly and re-executes it. When resolving each sub-question, all prior strategies are passed to the model and incorporated into the prompt context. Finally, after all sub-problems are addressed, the model synthesizes the complete set of refined strategies, integrates them with few-shot examples, and generates the final SQL query using three distinct stylistic formulations.

Common Table Expressions. Intermediate results are materialized as named CTEs, promoting modular, stepwise reasoning and improving interpretability through explicit logical decomposition.

Flat JOIN-Based Formulation. All tables are joined in a single, flat structure using explicit `JOIN` clauses, with filters applied in `WHERE` or `ON`. This avoids nesting and aligns with classical relational algebra.

Nested Subqueries. Auxiliary logic is embedded directly within the outer query via `IN`, `EXISTS`, or scalar subqueries, enabling hierarchical representations that mirror nested natural language constructs.

3.4.1 In-Context Learning

In this work, ICL is employed twice to resolve ambiguities. The first instance occurs prior to final SQL generation: we concatenate the natural language question with the skeletal structure of the most recent sub-SQL query generated under the current reasoning strategy. From the database, we retrieve five semantically similar exemplars based on this combined representation and include them in the prompt as few-shot references to guide the model's generation.

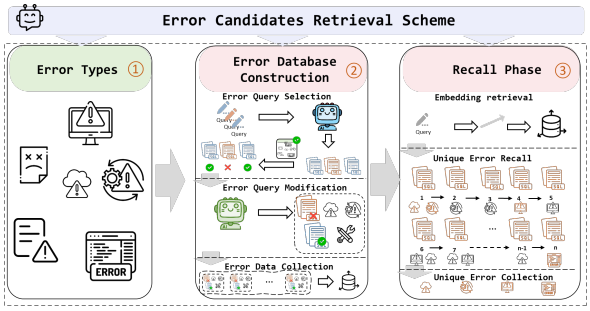

Figure 4: Construction pipeline of the error-correction memory.

3.5 SQL Refinement and Selection

From the preceding stages, we obtain a total of nine candidate SQL queries, derived from three decomposition strategies combined with three distinct syntactic styles. Each candidate is subsequently subjected to rigorous semantic validation and iterative correction.

3.5.1 Memory-Augmented Error Correction

We leverage historical correction experiences as contextual guidance to steer the model toward accurate revisions. Specifically, for a given query, we first combine the original question with the skeletal form of its current SQL candidate, embed this representation, and retrieve the top-40 most similar entries from the experience bank of previously corrected examples.

A second-stage filtering is then applied: we traverse the retrieved candidates in descending order of similarity and discard any example whose set of error types is fully subsumed by those of a previously retained example. This deduplication ensures diversity in error patterns and mitigates bias toward overrepresented error categories (e.g., `SELECT` clause errors are far more common than `GROUP BY` or multi-table `JOIN` errors as shown in Figure 6). After this two-stage selection, each problem is typically associated with 3–5 high-quality, error-diverse reference samples.

A critic agent utilizes these samples to assess whether the current SQL contains errors. If so, it identifies the error category and provides specific revision suggestions. A refine agent then modifies the query accordingly and resubmits it to the critic for re-evaluation. This iterative refinement loop continues until either the critic confirms correctness or a maximum iteration limit is reached.

Upon completing refinement for all nine candidates, we apply majority voting, selecting the SQL query that receives the most consensus across the refined outputs as the final prediction.

4 Experiments

Datasets and Evaluation Metrics. We build our error-correction memory using the BIRD training set (Li et al., 2023a) and evaluate our approach on four established NL2SQL benchmarks: the BIRD dev set, BIRD dev-new set (Li et al., 2023a), SPIDER development set (Yu et al., 2018), and CHESS-SDS dataset (Talaei et al., 2024). Further details are provided in Appendix A.6.

Implementation Details. Our implementation leverages instruction-tuned models from the Qwen-Coder family, specifically Qwen2.5-Coder-32B-Instruct and the recently released Qwen3-Coder-30B-Instruct (Hui et al., 2024; Team, 2025). To validate the robustness of our approach, we further conduct experiments using DeepSeek-Coder-V2-Lite (16B) (Zhu et al., 2024) and Llama-3.1-8B (Team, 2024), both instruction-tuned variants. In the error-correction phase, we retrieve up to 40 semantically similar historical error–correction instances from our experience bank to inform refinement. The iterative correction loop is capped at three rounds to balance effectiveness against computational overhead. All results reported are averaged over three independent runs to ensure reliability.

Baseline Methods. To ensure a rigorous and reproducible comparison, we not only incorporate results reported in prior work but also re-implement two recent open-source, training-free methods based on test-time scaling, namely ROUTE (Qin et al., 2024) and Alpha-SQL (Li et al., 2025b), using the four aforementioned models, enabling direct comparison of both accuracy and computational cost.

Additionally, we compare our SQL correction module against three representative correction strategies employing distinct technical paradigms, which are DIN-SQL (Pourreza and Rafiei, 2023), Solid-SQL (Liu et al., 2025a), and SHARE (Qu et al., 2025). More detailed descriptions of the baseline methods are provided in Appendix A.7.

5 Results

BIRD dataset. We evaluate Memo-SQL on the BIRD dev set and compare it against state-of-the-art NL2SQL methods across four categories: (1) approaches requiring fine-tuning with closed-source

Table 1: Comparison of training-free and open-source NL2SQL methods on the BIRD dev set.

Method	Inference Model	Selection Model	EX (%)
Distillery (Maamari et al., 2024)	Llama-3.1-405B	—	59.2
ROUTE (Multi-task only) (Qin et al., 2024)	Qwen2.5-Coder-32B	Iterative Refinement	59.8
ROUTE (Multi-task only) (Qin et al., 2024)	Qwen3-Coder-30B-A3B	Iterative Refinement	65.2
Alpha-SQL∗ (Li et al., 2025b)	Qwen2.5-Coder-32B	Majority Voting	64.4 (64.3)†
Alpha-SQL∗ (Li et al., 2025b)	Qwen3-Coder-30B-A3B	Majority Voting	68.2 (67.2)†
Memo-SQL	Qwen2.5-Coder-32B	Iterative Refinement	**64.9 (64.2)**†
Memo-SQL (Ours)	Qwen3-Coder-30B-A3B	Iterative Refinement	**67.6 (68.5)**†

***EX**: Execution accuracy on BIRD dev set. ∗We report results using our own evaluation script, which we argue offers a more principled and reliable assessment; see Appendix A.5 for a detailed justification. †The results in parentheses refer to those on the BIRD dev-new set.*

models, (2) training-free but closed-source methods, (3) fine-tuned open-source systems, and (4) training-free open-source frameworks.

As shown in Table 1 (Full results across all method categories are reported in Table 9), on the Qwen3-Coder-30B-A3B backbone, Memo-SQL achieves 67.6 on the original dev set and 68.5 on the corrected version—surpassing Alpha-SQL (Li et al., 2025b), which reports 68.2 (67.2) under identical evaluation conditions. While Alpha-SQL (Li et al., 2025b) slightly outperforms our method on the original split, Memo-SQL demonstrates superior robustness on the more reliable, updated benchmark, thereby claiming the new SOTA in the open, training-free regime. Moreover, our gains are consistent across model scales: with Qwen2.5-Coder-32B, we achieve 64.9 (64.2), edging out Alpha-SQL's (Li et al., 2025b) 64.4 (64.3), and Distillery (Maamari et al., 2024) with Llama-3.1-405B (59.2 despite its massive scale).

Crucially, as detailed in Section 5.1, Memo-SQL achieves this performance with significantly lower computational overhead than Alpha-SQL (Li et al., 2025b), striking a superior balance between effectiveness and efficiency, making it particularly suitable for real-world BI systems with latency or cost constraints. Furthermore, Memo-SQL not only dominates the training-free open-source category but also surpasses several fine-tuned baselines, including CHESS-SQL (Talaei et al., 2024) (61.5) and XiYan-SQL@DDL (Liu et al., 2025b) (62.3), despite requiring no parameter updates. This underscores the power of our experience-augmented, decomposition-driven inference framework. The consistent improvements across diverse model architectures are further validated in Table 6. Across all three models, ranging from 16B-scale to 32B, Memo-SQL consistently yields high execution accuracy, demonstrating strong compatibility and generalizability. In summary, Memo-SQL sets a new standard for open, training-free NL2SQL systems, delivering state-of-the-art accuracy on the most reliable version of BIRD, maintaining robustness across model scales, and offering favorable computational efficiency for practical deployment.

Cross-Dataset Generalization. To evaluate the robustness and transferability of our experience-augmented framework, we conduct experiments on the Spider dev set using an error-correction memory constructed exclusively from interactions on the BIRD training set, without any in-domain Spider data. As shown in Table 8, Memo-SQL achieves 86.5% execution accuracy with the Qwen3-Coder-30B-A3B model, slightly below Alpha-SQL (Li et al., 2025b) (87.8%) but substantially outperforming ROUTE (Qin et al., 2024) (80.0%).

When the SQL correction module is ablated (i.e., iterative refinement disabled), performance drops by 0.7% to 85.8%. While this degradation is less pronounced than the 2.1% drop observed on BIRD dev set, it remains consistent with expectations: Spider queries are generally simpler, allowing strong base models to achieve high accuracy even without sophisticated correction mechanisms. Consequently, the headroom for improvement is narrower. Nevertheless, the consistent gains, despite using only out-of-domain experiences from BIRD, demonstrate that our framework effectively transfers error-correction knowledge across datasets, highlighting its strong generalization and practical applicability in real-world settings where in-domain interaction history may be unavailable.

Table 2: Comparison of Baseline LLMs on the CHESS-SDS Set

Model	EX (%)	Tokens/Query	Time/Query (s)
Deepseek-R1	50.3	–	–
GPT-4o	53.7	–	–
Gemini-2.0-Flash-Thinking-Exp	60.8	–	–
Qwen2.5-Coder-32B	49.0	282	8
+ ROUTE	53.7	347	13
+ Alpha-SQL	58.5	150344	1839
+ **Memo-SQL**	57.6	7925	145
Llama-3.1-8B	21.1	332	4
+ ROUTE	44.2	414	9
+ Alpha-SQL	49.6	244775	1628
+ **Memo-SQL**	52.1	16563	135

5.1 Efficiency and Performance Trade-offs

We evaluate Memo-SQL on the CHESS-SDS benchmark (Talaei et al., 2024), a challenging BIRD dev subset (Implementation details in Section A.8). As shown in Table 2, our method achieves competitive execution accuracy (EX) with dramatically lower resource use.

With Qwen2.5-Coder-32B, Memo-SQL reaches 57.6% EX, just 0.9 points behind Alpha-SQL (58.5%), while using 19× fewer tokens (7,925 vs. 150,344) and running 12.7× faster (145s vs. 1,839s per query). On Llama-3.1-8B, it boosts EX from 21.1% (baseline) to 52.1%, surpassing both ROUTE and Alpha-SQL in efficiency, with 14.8× less token usage and 12.1× lower latency than Alpha-SQL. While ROUTE (Qin et al., 2024) also uses TTS for low latency, its EX (53.7%) lags behind ours by nearly 4 points, suggesting shallow reasoning limits its handling of complex queries.

Notably, Memo-SQL delivers strong results even on smaller models: with Llama-3.1-8B, it closes over half the gap to large proprietary systems while remaining fully open and deployable on commodity hardware. Overall, Memo-SQL offers the best balance of accuracy and efficiency, outperforming lightweight methods in correctness and heavyweight ones in speed.

5.2 Analysis of Self-Correction Strategies

As shown in Table 3, our retrieval-augmented self-correction framework achieves 68.5% execution accuracy, setting a new state of the art among training-free methods. In contrast, prompt engineering approaches (e.g., DIN-SQL (Pourreza and Rafiei, 2023) and direct correction) yield only marginal gains over the uncorrected baseline (66.5–66.6%), underscoring their limited capacity to handle complex semantic errors. SOLID-SQL (Liu et al., 2025a) improves upon this by incorporating positive examples via in-context learning, reaching 67.4%; however, it leverages only successful query demonstrations. Our method goes further by retrieving both positive and negative historical interactions, along with error-type annotations and correction rationales from the error-correction memory. This richer context enables the model to not only recognize what is correct, but also understand why a query is flawed and how to fix it, leading to more reliable and accurate revisions. Although SHARE (Qu et al., 2025) achieves slightly higher performance (69.2%), it relies on fine-tuning three specialized models and lacks dynamic upgradability, rendering it unsuitable for interactive BI environments where systems must continuously learn from user feedback. Memo-SQL, by contrast, is fully training-free, supports real-time experience integration, and maintains strong performance without compromising deployability or privacy.

6 Conclusion

We presented Memo-SQL, a training-free NL2SQL framework tailored for real-world BI deployment. By combining structured divide-and-conquer decomposition with experience-driven self-correction, Memo-SQL eliminates the need for closed-source models or fine-tuning, supports dynamic updates via user feedback, and ensures data privacy. Its experience bank, populated with both correct and corrected queries, enables retrieval-augmented in-context learning that steers the model away from common semantic errors. On the BIRD benchmark, Memo-SQL achieves state-of-the-art performance among open, zero-fine-tuning methods, while reducing token usage and latency by over an order of magnitude compared to leading TTS approaches like Alpha-SQL (Li et al., 2025b). These advantages make Memo-SQL highly suitable for scalable, efficient, and continuously improving enterprise NL2SQL systems.

7 Limitations

While Memo-SQL shows strong results, it has a few practical limits. First, our experience bank is built from model-generated errors on the BIRD training set. This means the quality of self-correction depends heavily on how well those synthetic errors reflect real user mistakes, if the error types or correction patterns don't match what users actually en-

Table 3: Comparison of different SQL error correction strategies on BIRD dev set. Methods are grouped into internal ablations (top) and external baselines (bottom).

Method	Correction Paradigm	Dynamic Update	Training-Free	EX (%)
	Internal Ablations			
Baseline (no correction)	—	✗	✓	66.4
+ Direct Correction	Prompt Engineering	✗	✓	66.6
+ RAG (top4, unfiltered)	ICL + RAG	✓	✓	67.5
+ Random RAG (unfiltered)	ICL + RAG (random)	✓	✓	67.6
	External SOTA Methods			
DIN-SQL (Pourreza and Rafiei, 2023)	Prompt Engineering	✗	✓	66.5
SOLID-SQL (Liu et al., 2025a)	ICL	✓	✓	67.4
SHARE (Qu et al., 2025)	SFT	✗	✗	69.2
Memo-SQL (Ours)	**ICL + RAG (filtered)**	✓	✓	**68.5**

counter, the benefit may drop. Second, the current decomposition strategies, though structured, still rely on the LLM to correctly interpret each sub-task. On very ambiguous or poorly phrased questions, this can lead to early missteps that even later correction can't fully fix. Finally, while much faster than alternatives like Alpha-SQL, our pipeline still involves multiple LLM calls and SQL executions per query, which may be too heavy for ultra-low-latency settings (e.g., interactive dashboards requiring sub-second responses).

Nevertheless, by demonstrating that error-aware, experience-driven self-correction can work without fine-tuning or closed APIs, our approach lays a practical foundation for future open and adaptive NL2SQL systems.

8 Ethical Statement

This work focuses on improving the reliability and efficiency of NL2SQL systems in business intelligence (BI) settings. Our method, Memo-SQL, operates on synthetic or publicly available benchmark datasets (e.g., BIRD) and does not involve human subjects, personal data, or sensitive information. The experience repository used for self-correction is constructed from model-generated queries and simulated or anonymized user feedback; no real user interactions or private database contents are collected or stored. While our approach enhances correctness and reduces computational cost, we acknowledge that any NL2SQL system could be misused to generate unauthorized database queries if deployed without proper access controls. Therefore, we emphasize that such systems should only be integrated into secure, permissioned environments where query execution is governed by strict authentication and authorization policies. The authors declare no conflicts of interest.

References

Arian Askari, Christian Poelitz, and Xinye Tang. 2025. Magic: Generating self-correction guideline for in-context text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 23433–23441.

Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. 2024. Rsl-sql: Robust schema linking in text-to-sql generation. *arXiv preprint arXiv:2411.00073*.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, page 253–262, New York, NY, USA. Association for Computing Machinery.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining small language models and large language models for zero-shot nl2sql. *Proceedings of the VLDB Endowment*, 17(11):2750–2763.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

JDCHO. 2025. Joyagent-jdgenie.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2024. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. *arXiv preprint arXiv:2405.07467*.

Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. 2025a. Deepeye-sql: A software-engineering-inspired text-to-sql framework. *arXiv preprint arXiv:2510.17586*.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024a. The dawn of natural language to sql: Are we fully ready? *arXiv preprint arXiv:2406.01265*.

Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025b. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. *arXiv preprint arXiv:2502.17248*.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024b. 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. 2023a. 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.

Xiuwen Li, Qifeng Cai, Yang Shu, Chenjuan Guo, and Bin Yang. 2025c. Aid-sql: Adaptive in-context learning of text-to-sql with difficulty-aware instruction and retrieval-augmented generation. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 3945–3957. IEEE.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023b. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Weibin Liao, Xin Gao, Tianyu Jia, Rihong Qiu, Yifan Zhu, Yang Lin, Xu Chu, Junfeng Zhao, and Yasha Wang. 2025. Learnat: Learning nl2sql with ast-guided task decomposition for large language models. *arXiv preprint arXiv:2504.02327*.

Geling Liu, Yunzhi Tan, Ruichao Zhong, Yuanzhen Xie, Lingchen Zhao, Qian Wang, Bo Hu, and Zang Li. 2025a. Solid-SQL: Enhanced schema-linking based in-context learning for robust text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 9793–9803, Abu Dhabi, UAE. Association for Computational Linguistics.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and 1 others. 2025b. Xiyan-sql: A novel multi-generator framework for text-to-sql. *arXiv preprint arXiv:2507.04701*.

Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-sql in the age of well-reasoned language models. *arXiv preprint arXiv:2408.07702*.

Solomon Negash and Paul Gray. 2008. Business intelligence. In *Handbook on Decision Support Systems 2: Variations*, pages 175–193. Springer.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems*, volume 36, pages 36339–36348. Curran Associates, Inc.

Mohammadreza Pourreza and Davood Rafiei. 2024. Dts-sql: Decomposed text-to-sql with small large language models. *arXiv preprint arXiv:2402.01117*.

Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, Sercan Arik, and 1 others. 2025. Reasoning-sql: Reinforcement learning with sql tailored partial rewards for reasoning-enhanced text-to-sql. *arXiv preprint arXiv:2503.23157*.

Yang Qin, Chao Chen, Zhihang Fu, Ze Chen, Dezhong Peng, Peng Hu, and Jieping Ye. 2024. Route: Robust multitask tuning and collaboration for text-to-sql. *arXiv preprint arXiv:2412.10138*.

Ge Qu, Jinyang Li, Bowen Qin, Xiaolong Li, Nan Huo, Chenhao Ma, and Reynold Cheng. 2025. Share: An slm-based hierarchical action correction assistant for text-to-sql. *arXiv preprint arXiv:2506.00391*.

Jie Shi, Bo Xu, Jiaqing Liang, Yanghua Xiao, Jia Chen, Chenhao Xie, Peng Wang, and Wei Wang. 2025. Gensql: 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*, pages 3794–3807.

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*.

Llama Team. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and 1 others. 2023. Mac-sql: A multi-agent collaborative framework for text-to-sql. *arXiv preprint arXiv:2312.11242*.

Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *Proceedings of the ACM on Management of Data*, 3(3):1–24.

Yuanzhen Xie, Xinzhou Jin, Tao Xie, Matrixmxlin Matrixmxlin, Liang Chen, Chenyun Yu, Cheng Lei, ChengXiang Zhuo, Bo Hu, and Zang Li. 2024. Decomposition for enhancing attention: Improving llm-based text-to-sql through workflow paradigm. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10796–10816.

Zerui Yang, Yuwei Wan, Siyu Yan, Yudai Matsuda, Tong Xie, Bram Hoex, and Linqi Song. 2025. Drugmcts: a drug repurposing framework combining multi-agent, rag and monte carlo tree search. *arXiv preprint arXiv:2507.07426*.

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*.

Shuozhi Yuan, Liming Chen, Miaomiao Yuan, Jin Zhao, Haoran Peng, and Wenming Guo. 2025. Mcts-sql: An effective framework for text-to-sql with monte carlo tree search. *arXiv preprint arXiv:2501.16607*.

Fuheng Zhao, Shaleen Deep, Fotis Psallidas, Avrilia Floratou, Divyakant Agrawal, and Amr El Abbadi. 2024. Sphinteract: Resolving ambiguities in nl2sql through user interaction. *Proceedings of the VLDB Endowment*, 18(4):1145–1158.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Appendix

A.1 Real-World BI Systems

The workflow of a real-world BI system typically involves four key components: (1) a model or a multi-agent system, (2) a generated answer, (3) user-provided feedback or revision requests, and (4) an external experience repository (Negash and Gray, 2008). After receiving the generated SQL, users often inspect the execution results and propose modifications. Upon receiving such feedback, the system revises the original output. This process forms a three-party iterative loop among the model, the model output, and the user, which continues until a correct answer is produced. The final question–answer pair is then stored in a database.

When a new question is posed, the system retrieves similar historical cases from the experience repository and concatenates them into the prompt, thereby guiding the LLM to generate an answer (JDCHO, 2025).

Building on this workflow, we propose an enhanced interaction paradigm. After each round of user feedback, the system not only stores the final correct query but also captures intermediate erroneous outputs and the model's reflections on them. Both correct and corrected–error examples are archived in the experience repository.

When a new question is posed, the system first generates an initial SQL query. A self-correction agent then retrieves historical error–correction cases whose erroneous queries are structurally or semantically similar to the current output. By analyzing these retrieved patterns, the agent assesses whether the generated SQL exhibits known failure modes. If potential issues are detected, it applies targeted edits to correct them; otherwise, the original query is retained. This transforms the conventional three-party interaction loop (model–output–user) into a four-component process: model $\rightarrow$ output $\rightarrow$ self-correction agent $\rightarrow$ user, enabling proactive error avoidance through experience-driven reasoning, without requiring retraining.

A.2 Database Schema Preprocessing

To facilitate accurate schema linking during SQL generation, we preprocess the structural metadata of each database, namely table names, column names, and foreign-key relationships, into a searchable semantic index.

Following established practices in NL2SQL systems (Talaei et al., 2024; Li et al., 2025b), we first tokenize each schema element (e.g., "orders.customer_id") and generate a compact fingerprint using MinHash (Datar et al., 2004). These fingerprints are precomputed for all tables and columns and stored in a local hash table.

At inference time, given a natural language question, we use a large language model to identify entity- or attribute-related keywords. We then employ locality-sensitive hashing (LSH) to efficiently retrieve candidate schema elements whose MinHash signatures are close to those of the extracted keywords. To ensure precision, retrieved candidates are further filtered using two criteria: (i) normalized edit distance below a fixed threshold, and (ii) semantic similarity above a preset cutoff, where similarity is computed as the cosine similarity between bert-large-uncased embeddings (Devlin et al., 2018) of the schema element and the keyword.

The final set of matched schema items is incorporated into the prompt as grounded context, enabling the model to resolve lexical mismatches and correctly reference database objects during query synthesis.

Table 4: Taxonomy of SQL generation errors used in our experience-augmented correction framework.

Error Type
E1: Join Logic Error
E2: Filter Condition Error
E3: Aggregation and Grouping Error
E4: Select Output Error
E5: Ordering and Limit Error
E6: Subquery Logical Error
E7: Null Handling Error
E8: Temporal Semantics Error
E9: Quantifier Intent Error

A.3 Ablation Studies

We conduct a systematic ablation study to assess the individual contribution of each component in Memo-SQL. All experiments are evaluated on the BIRD dev-new set, and results are summarized in Table 5.

Starting from a strong baseline that incorporates schema preprocessing and uses Qwen3-Coder-30B as the inference engine, our full pipeline achieves

an EX of 68.5%. Removing schema linking, which grounds natural language mentions to relevant database columns and tables, leads to a modest drop of 0.8 percentage points (to 67.7%), confirming that precise schema alignment remains beneficial even in large language model–based systems.

Replacing our structured decomposition strategy with random question decomposition reduces performance to 67.2% (-1.3%), highlighting the value of semantically meaningful sub-question partitioning. Similarly, ablating the ReAct+Reflect SQL generation module, a key component for enhancing query correctness through iterative reasoning and feedback loops, results in a decrease to 67.3% (-1.2%). This indicates that incorporating reactive adjustments and reflective refinements during SQL synthesis significantly boosts overall performance.

Disabling multi-style SQL generation (i.e., restricting output to a single syntactic form) yields 67.6% (-0.9%), demonstrating that generating diverse SQL formulations enhances coverage and robustness. Most notably, ablating the SQL refinement module, which performs memory-augmented error correction via iterative reflection, results in a 2.1-point drop (to 66.4%), underscoring the critical role of post-hoc validation and correction in handling subtle semantic errors.

An unexpected finding emerges when we add ICL during the final SQL generation phase: performance decreases to 67.1% (-1.4%). Upon analysis, we attribute this degradation to a mismatch between the stylistic patterns in the retrieved in-context examples and our three-style generation framework. Specifically, the few-shot exemplars predominantly exhibit flat or nested SQL structures, whereas our generator deliberately produces CTE-, JOIN-, and subquery-based variants in parallel. This discrepancy appears to introduce noise or bias during decoding, leading to suboptimal outputs. We observe this degradation only with the Qwen3-Coder-30B model; for all other evaluated models, incorporating ICL during the final SQL generation phase consistently improves performance, and thus we retain this component in their respective pipelines.

Collectively, these results validate the design choices in Memo-SQL and reveal nuanced interactions between retrieval-augmented prompting and structured generation strategies.

Table 5: Ablation study on BIRD-dev-new: progressive impact of each component in Memo-SQL. Reported values show absolute accuracy (%) and changes relative to the full model (in parentheses).

Variant	**EX (%)**
Memo-SQL (full)	68.5
w/o Schema Linking	67.7 (-0.8)
w/ Random Question Decomposition	67.2 (-1.3)
w/o ReAct+Reflect SQL Generation	67.3 (-1.2)
w/o Multi-Style Generation	67.6 (-0.9)
w/o SQL Refinement	66.4 (-2.1)
+ ICL in Final SQL Generation	67.1 (-1.4)

A.4 Per-Strategy Decomposition Analysis

To better understand the role of each decomposition strategy, we further analyze their standalone performance and the questions uniquely solved by each method. The results show that all three strategies contribute to overall performance and exhibit complementary strengths on different categories of SQL queries.

As shown in Table 7, each strategy solves a non-trivial set of questions that the others do not, indicating that the three decomposition strategies are complementary rather than redundant. We next provide representative examples and summarize the type of queries each strategy handles best.

A.4.1 Atomic Operational Decomposition

This strategy is most effective for queries with linear procedural logic, where reasoning can be decomposed into a sequence of local operations such as filtering, joining, and selecting.

Example.

```
SELECT T2.time
FROM drivers AS T1
JOIN results AS T2
  ON T2.driverId = T1.driverId
WHERE T2.raceId = 743
  AND T1.forename = 'Bruce'
  AND T1.surname = 'McLaren'
```

This query follows a simple linear process: identify the driver, join with the results table, and apply the race constraint. Such cases are well suited to atomic decomposition because each step can be solved incrementally.

A.4.2 Hierarchical Decomposition

Hierarchical decomposition is more suitable for queries with nested or dependent logic, especially those involving subqueries, superlatives, or aggregation-based comparisons.

Table 6: Execution accuracy of different models combined with various SQL generation strategies on BIRD dev set.

Model	Baseline	ROUTE	Alpha-SQL	Memo-SQL (Ours)
DeepSeek-Coder-V2-Lite	32.5	51.6	53.4	**53.8**
Qwen2.5-Coder-32B	52.8	59.8	64.4	**64.9**
Qwen3-Coder-30B-A3B	62.1	65.2	68.2	**67.6**

Example.

```
SELECT COUNT(t1.id)
FROM Player_Attributes AS t1
WHERE t1.preferred_foot = 'left'
  AND t1.crossing = (
    SELECT MAX(crossing)
    FROM Player_Attributes
  )
```

This query requires first solving the inner subquery and then using its result as a constraint in the outer query, making it a natural fit for hierarchical decomposition.

A.4.3 Table-wise Decomposition

Table-wise decomposition is particularly beneficial for multi-entity queries with complex join structures. In such cases, the main challenge is identifying the correct schema traversal path and grounding attributes to the proper tables.

Example.

```
SELECT T1.account_id
FROM account AS T1
JOIN disp AS T2
  ON T1.account_id = T2.account_id
JOIN client AS T3
  ON T2.client_id = T3.client_id
JOIN district AS T4
  ON T4.district_id = T1.district_id
WHERE T2.client_id = (
  SELECT client_id
  FROM client
  ORDER BY birth_date DESC
  LIMIT 1
)
GROUP BY T4.A11, T1.account_id
```

This example involves multiple tables with distinct semantic roles. Table-wise decomposition helps construct the schema interaction pattern more reliably and reduces join path errors.

Overall, the three strategies exhibit clear specialization: atomic decomposition is most effective for linear procedural queries, hierarchical decomposition for nested and dependency-intensive queries, and table-wise decomposition for multi-entity join-heavy queries.

A.5 On the Treatment of Empty Query Results in Evaluation

While attempting to replicate the results of AlphaSQL, we encountered a notable inconsistency between our initial evaluation and the numbers reported in their paper. After acquiring their official evaluation script, we were able to reproduce their published scores and identified a key difference in their protocol: during majority voting, any SQL prediction that yields an empty result set is automatically discarded, and the system defaults to the next most frequent non-empty candidate.

Although this heuristic allows exact replication of their reported performance, we contend that it rests on a questionable assumption, namely, that a correct SQL query should never return an empty result. In practice, however, semantically accurate queries often produce empty outputs, especially in realistic scenarios (e.g., "Find customers who spent more than 1000 on Electronics products in 2024" in a recently launched service with limited transaction history).

To assess how common this design choice is, we inspected the public codebases and evaluation procedures of several prominent NL2SQL systems, including the official BIRD benchmark implementation (Li et al., 2023a) and methods employing iterative refinement or majority voting (Xie et al., 2025; Pourreza and Rafiei, 2023; Wang et al., 2023; Li et al., 2024a). None of these adopt explicit filtering of empty, but executable results.

In light of this, we adopt a more principled and application-aligned evaluation protocol: we only exclude queries that fail to parse or execute (i.e., those that raise runtime errors), while retaining all successfully executed queries, even if they return empty result sets. For fair comparison under con-

Method	EX	Unique
Table-wise Decomposition	67.4	16
Hierarchical Decomposition	67.5	18
Atomic Operational Decomposition	67.9	18

Table 7: Standalone execution accuracy (EX, %) and the number of uniquely solved questions for each decomposition strategy.

sistent criteria, we re-evaluate AlphaSQL on the BIRD development set using this protocol and report the corresponding performance.

A.6 Detailed Information of Datasets

1. **BIRD training set** (Li et al., 2023a): Comprising 12,751 question–SQL pairs across 95 large-scale databases, this dataset serves as the source for constructing our experience memory bank.

2. **BIRD dev set**: Contains 1,534 question–SQL pairs and constitutes our primary evaluation benchmark for comparison with prior work.

3. **BIRD dev-new set**: On November 6, 2025, the BIRD team released a revised version of the development set, incorporating corrections to schema annotations and SQL labels from the original release. While this updated benchmark has not yet been widely adopted—and thus lacks published baselines—we report main results on the original BIRD dev set to ensure fair comparison. Nevertheless, we provide supplementary evaluations of several representative methods on the new split to assess robustness under refined data conditions.

4. **Spider dev set** (Yu et al., 2018): To test the generalizability of our SQL correction mechanism, we conduct cross-dataset experiments on SPIDER, which includes 1,034 question–SQL pairs drawn from heterogeneous databases spanning diverse domains.

5. **CHESS-SDS dataset** (Talaei et al., 2024): Given that our framework operates under the TTS paradigm, where system quality depends jointly on output correctness and computational cost—we adopt a multi-dimensional evaluation protocol. In addition to execution accuracy, we measure inference efficiency via total generated tokens and end-to-end latency. These metrics are evaluated on CHESS-SDS, a high-quality, manually verified subset of the BIRD dev set that emphasizes complex queries requiring precise schema grounding and deep semantic reasoning.

A.7 Baseline Methods

- **ROUTE**: A multi-agent framework that decomposes NL2SQL into four subtasks: schema linking, SQL generation, noise correction, and continuation writing. In our replication, we adopt only the agent-based orchestration without any supervised fine-tuning of the underlying LLMs.

- **Alpha-SQL**: Formulates NL2SQL as a Monte Carlo Tree Search problem, where each node corresponds to an atomic action, including question rephrasing, column-value identification, and SQL refinement, among seven predefined operations. To the best of our knowledge, Alpha-SQL represents the current state of the art among open-weight, training-free approaches.

- **DIN-SQL**: Relies on prompt engineering to elicit self-verification from the model, prompting it to critique and revise its own outputs.

- **Solid-SQL**: Similar to our approach, it performs post-hoc SQL correction by retrieving database-executed examples; however, it exclusively uses correct SQL queries for retrieval and does not leverage historical error–correction pairs.

- **SHARE**: Employs SFT on three specialized modules for schema linking, logical optimization, and SQL generation. To our knowledge, SHARE currently achieves the strongest reported performance in SQL correction and serves as a strong SFT-based baseline.

Although **MAGIC** (Askari et al., 2025) and **Sphinteract** (Zhao et al., 2024) also involve SQL correction, their primary focus lies in designing interactive prompting strategies where an external agent, assumed to have access to the ground-truth answer, guides the model toward correction. As explicitly noted in Zhao et al. (2024), such approaches do not address the model's inherent inability to perform self-correction in the absence of oracle feedback. In contrast, our work aims to enable correction using only past error–fix experiences without any knowledge of the current query's ground truth, making these methods fundamentally different in objective and thus outside the scope of our comparison.

A.8 Experimental Setup for Efficiency Evaluation

All efficiency measurements reported in Section 5.1 are obtained under identical hardware and software

Table 8: Comparison of state-of-the-art NL2SQL methods on the Spider dev set.

Method	Inference Model	Selection Model	Zero-shot	Open-Source	EX (%)
DAIL-SQL (Gao et al., 2023)	GPT-4	Majority Voting	Yes	No	83.6
ZeroNL2SQL (Fan et al., 2024)	GPT-4	-	Yes	No	84.0
MAC-SQL (Wang et al., 2023)	GPT-4	Majority Voting	Yes	No	86.8
SuperSQL (Li et al., 2024a)	GPT-4	Majority Voting	Yes	No	87.0
SFT CodeS (Li et al., 2024b)	CodeS-15B	-	No	Yes	84.9
ROUTE (Multi-task + FT) (Qin et al., 2024)	Qwen2.5-Coder-14B	Iterative Refinement	No	Yes	87.3
ROUTE (Multi-task only) (Qin et al., 2024)	Qwen2.5-Coder-14B	Iterative Refinement	Yes	Yes	80.0
Alpha-SQL (Li et al., 2025b)	Qwen3-Coder-30B-A3B-Instruct	Majority Voting	Yes	Yes	87.8
Memo-SQL (Ours)	Qwen3-Coder-30B-A3B-Instruct	Iterative Refinement	**Yes**	**Yes**	**86.5**
w/o SQL Correction	Qwen3-Coder-30B-A3B-Instruct	Iterative Refinement (disabled)	**Yes**	Yes	85.8

Table 9: Comparison of state-of-the-art NL2SQL methods on the BIRD dev set.

Method	Inference Model	Selection Model	Training Free	Open-Source	EX (%)
CHESS-SQL (Talaei et al., 2024)	Deepseek-Coder-33B	GPT-4-Turbo	No	No	65.0
Distillery (Maamari et al., 2024)	GPT-4o	—	No	No	67.2
CHASE-SQL (Pourreza et al., 2024)	Gemini-1.5-Pro	Gemini-1.5-Flash	No	No	73.0
XiYan-SQL (Liu et al., 2025b)	Unreported	Unreported	No	No	73.3
DAIL-SQL (Gao et al., 2023)	GPT-4	Majority Voting	Yes	No	55.9
SuperSQL (Li et al., 2024a)	GPT-4	Majority Voting	Yes	No	58.5
MAC-SQL (Wang et al., 2023)	GPT-4	Iterative Refinement	Yes	No	59.4
Gen-SQL (Shi et al., 2025)	GPT-4	Iterative Refinement	Yes	No	59.8
RSL-SQL (Cao et al., 2024)	GPT-4o	Iterative Refinement	Yes	No	67.2
DTS-SQL (Pourreza and Rafiei, 2024)	DeepSeek-7B	—	No	Yes	55.8
SFT CodeS (Li et al., 2024b)	CodeS-15B	—	No	Yes	58.5
ROUTE (Multi-task + FT) (Qin et al., 2024)	Qwen2.5-Coder-14B	Iterative Refinement	No	Yes	60.9
CHESS-SQL (Talaei et al., 2024)	Deepseek-Coder-33B	LLaMA3-70B	No	Yes	61.5
XiYan-SQL@DDL (Liu et al., 2025b)	Qwen2.5-Coder-32B	Qwen2.5-Coder-32B	No	Yes	62.3
XiYan-SQL@M-Schema (Liu et al., 2025b)	Qwen2.5-Coder-32B	Qwen2.5-Coder-32B	No	Yes	67.0
Reasoning-SQL (Pourreza et al., 2025)	Qwen2.5-Coder-14B	—	No	Yes	72.3
Distillery (Maamari et al., 2024)	Llama-3.1-405B	—	Yes	Yes	59.2
ROUTE (Multi-task only) (Qin et al., 2024)	Qwen2.5-Coder-32B	Iterative Refinement	Yes	Yes	59.8
ROUTE (Multi-task only) (Qin et al., 2024)	Qwen3-Coder-30B-A3B	Iterative Refinement	Yes	Yes	65.2
Alpha-SQL∗ (Li et al., 2025b)	Qwen2.5-Coder-32B	Majority Voting	Yes	Yes	64.4 (64.3)†
Alpha-SQL∗ (Li et al., 2025b)	Qwen3-Coder-30B-A3B	Majority Voting	Yes	Yes	68.2 (67.2)†
Memo-SQL	Qwen2.5-Coder-32B	Iterative Refinement	**Yes**	**Yes**	**64.9 (64.2)**†
Memo-SQL (Ours)	Qwen3-Coder-30B-A3B	Iterative Refinement	**Yes**	**Yes**	**67.6 (68.5)**†

conditions. We deploy both Memo-SQL and baseline systems using vLLM (Kwon et al., 2023) on a server equipped with four GPUs, each with 24 GB of memory. Each query is processed independently (i.e., we run inference on one question at a time), and no batching is applied to ensure fair latency measurement.

For each method, we record the total wall-clock time and total number of generated tokens across all queries in the CHESS-SDS benchmark. The per-query averages are then computed by dividing these totals by the number of questions.

Importantly, we adopt different stopping criteria for Alpha-SQL and Memo-SQL to reflect their respective design philosophies:

- For **Alpha-SQL**, we terminate execution immediately after all SQL candidates have been generated (as done in its original implementation), excluding the time required for majority voting or additional database executions used in final answer selection.

- For **Memo-SQL**, we include the full pipeline in our timing: this encompasses decomposition, multi-style candidate generation, retrieval-augmented refinement, database execution of refined candidates, and self-consistency–based majority voting. Thus, the reported latency and token counts for Memo-SQL reflect an end-to-end, deployable workflow.

A.9 Experimental Setup for Self-Correction Ablation

To ensure a fair comparison across correction strategies in Table 5.2, we adopt a unified evaluation protocol: for every input question, we first generate nine diverse SQL candidates using the same multi-style synthesis procedure (CTE-based, flat JOIN-based, and nested subquery-based, three each). Each candidate is then independently subjected to the respective correction method, and the final prediction is selected via self-consistency majority voting over the corrected outputs.

The specific configurations for each method are as follows:

- **Baseline (no correction)**: No refinement is applied; the final answer is selected by majority voting over the original nine uncorrected candidates.
- **Direct Correction**: We prompt the model to revise each candidate SQL directly, without providing any in-context examples. The prompt instructs the model to identify potential errors and output a corrected version in a single turn.
- **RAG (top4, unfiltered)**: For each candidate SQL, we retrieve the top-4 most similar entries from the error-correction memory based on the question and skeletal query structure. All retrieved entries (including erroneous queries, error types, and corrections) are concatenated into the prompt without filtering. This ablation evaluates the necessity of our two-stage retrieval-and-filtering pipeline.
- **Random RAG (unfiltered)**: Instead of similarity-based retrieval, we randomly sample 4 entries from the memory for each candidate and include them in the prompt. This tests whether performance gains stem from semantic relevance or merely from increased context length.
- **DIN-SQL**: We adapt the original prompt design of Pourreza and Rafiei (2023), slightly adjusting it to fit our two-agent critique–refine workflow: one agent critiques the candidate SQL, and another produces the revised version. Crucially, no historical examples are provided, only the current query and its execution feedback are used.
- **SOLID-SQL**: Following Liu et al. (2025a), we retrieve the top-4 most similar correct SQL examples (question + gold query pairs) and inject them into the prompt. Like DIN-SQL, we employ a critique–refine interaction between two agents, but grounded in positive demonstrations only.
- **SHARE**: We use the official implementation and pre-trained checkpoints provided by Qu et al. (2025) without modification.

A.10 Use of AI Tools

We used AI language models solely for proofreading and polishing the language of this paper (e.g., improving grammar, clarity, and fluency). All technical ideas, methodology, experiments, analysis, and writing content were conceived and produced entirely by the authors.

Original Question
Find customers who spent more than 1000 on products in the Electronics category in 2024, returning their names and total spending.

1) Table-wise / Entity-wise (by table/entity) — 4 sub-questions
- **Product entity**: Which products belong to the Electronics category?
- **Order entity**: Which orders occurred in 2024? Which customer does each order belong to?
- **Order detail entity**: Among these orders, which details involve purchases of "Electronics products"? What is the amount for each detail (quantity × unit price)?
- **Customer entity + Aggregation**: Aggregate the amounts by customer. Which customers have a total spending on Electronics > 1000? Output the names and total spending of these customers.

2) Hierarchical (layered/nested: compute inner layer first then outer layer) — 3 sub-questions
- **Inner layer measure definition**: For each customer, calculate their total spending on Electronics in 2024.
- **Outer layer condition**: Which customers have a total spending > 1000?
- **Outermost layer information retrieval**: Retrieve the names of these customers along with their total spending.

3) Atomic sequential (sequence of atomic operations: operational pipeline) — 5 sub-questions
- **Filter orders**: First, look at only those orders that occurred in 2024.
- **Expand purchase records**: Expand these orders into individual purchase details (what was bought, quantity, unit price).
- **Limit category**: From these details, retain only those records where the product category is Electronics.
- **Calculate and aggregate**: Calculate the amount for each record (quantity × unit price), then sum up the amounts by customer to get the total spending.
- **Filter and output**: Retain customers with a total spending > 1000, and output their names and total spending.

Figure 5: Illustration of three complementary question decomposition strategies in Memo-SQL, demonstrated on a sample query: "Find customers who spent more than 1000 on Electronics products in 2024." The methods include (1) *Table-wise/Entity-wise*, decomposing the query by relevant database entities; (2) *Hierarchical*, modeling nested logic through layered sub-questions; and (3) *Atomic Sequential*, breaking down the query into a pipeline of fundamental relational operations. Each strategy generates a unique set of sub-questions, enabling flexible and robust reasoning for complex multi-table SQL generation.

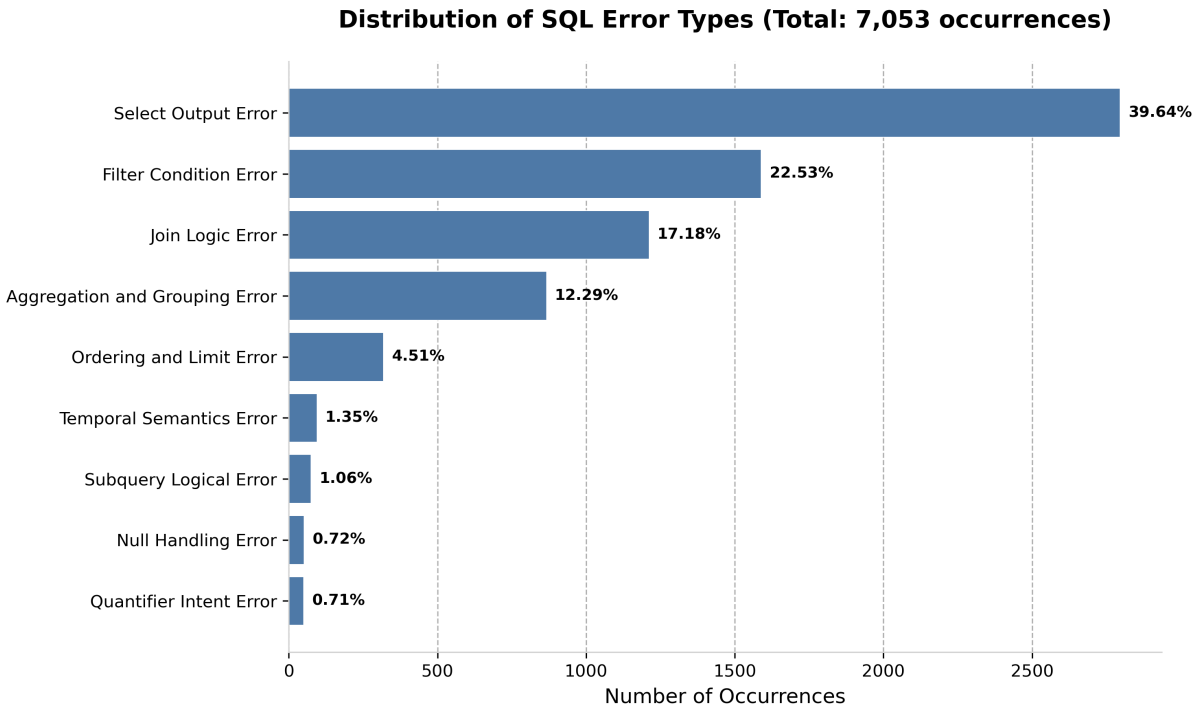

Figure 6: Distribution of SQL error types across 7,053 incorrect queries generated by the Qwen3-Coder-30B-A3B model. The most frequent error category is Select Output Error (39.64%), followed by Filter Condition Error (22.53%) and Join Logic Error (17.18%).

Question: List down the advised student IDs and IDs of employing professor in faculty.

GT SQL: SELECT T1.p_id, T2.p_id FROM advisedBy AS T1 INNER JOIN person AS T2 ON T1.p_id_dummy = T2.p_id WHERE hasPosition = 'Faculty_eme'

Erroneous SQL: SELECT A.p_id AS advised_student_ID, A.p_id_dummy AS employing_professor_ID FROM advisedBy A INNER JOIN person P ON A.p_id_dummy = P.p_id WHERE P.hasPosition = 'Faculty'

Error Types: ["filter_condition_error", "select_output_error"]

Correction Suggestions: The incorrect SQL uses WHERE P.hasPosition = 'Faculty' instead of the correct condition WHERE P.hasPosition = 'Faculty_eme'. This causes the query to return all professors with any 'Faculty' position rather than only those specifically identified as 'Faculty_eme', resulting in too many rows. Additionally, the erroneous SQL incorrectly selects A.p_id AS advised_student_IDand A.p_id_dummy AS employing_professor_ID. According to the gold query, it should select both columns as T1.p_idand T2.p_id(or with appropriate table aliases). The A.p_id_dummyis actually the foreign key used for joining, not the professor's ID.

Figure 7: An illustrative case of error memory in SQL generation.