QBridge: Bridging Natural Language and SQL via Gold Query Rewriting with Agentic Refinement

Zhensheng Luo[1,2], Sai Wu[1,2], Yuan Qiu[3], Chang Yao[1,2], Gang Chen[1,2], Xiu Tang[1,2]*
[1]Zhejiang University
[2]Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute
[3]Hong Kong University of Science and Technology
{zhenshengluo,wusai,changy,cg,tangxiu}@zju.edu.cn
yqiubo@connect.ust.hk

Abstract

Natural language to SQL (NL2SQL) provides an intuitive interface for querying structured data, yet real user questions are often noisy, ambiguous, and weakly grounded to database semantics. As a result, token-level schema linking and single-pass SQL decoding can be brittle: small misunderstandings in language or schema grounding may propagate into incorrect generation. We present *QBridge*, an agentic, feedback-driven NL2SQL framework based on a *Refined Gold Query Paradigm*, which bridges natural language and SQL via *Gold Query*—a structured, SQL-aligned intermediate representation. A core insight of *QBridge* is *Distilled Back-Translation (DBT)* for SL-independent rewriting. DBT converts SQL-grounded supervision into *execution-verified* Gold-Query-style rewrites from a teacher model, and distills a lightweight, plug-and-play rewriter that generates schema-aware rewrites without requiring explicit schema linking at inference. *QBridge* then (i) verifies and conservatively refines the rewrite into a high-fidelity *Refined Gold Query*, and (ii) refines the generated SQL with dual feedback from execution validity and semantic consistency, enabling interpretable self-correction while remaining compatible with diverse SQL backbones. Extensive experiments on Spider, BIRD, and three robustness variants demonstrate that *QBridge* consistently improves zero-shot NL2SQL, outperforming strong prompting and agentic baselines while showing strong robustness and generalization. Code and data are available at https://github.com/WannaBSteve/QBridge.

1 Introduction

Natural language to SQL translation (NL2SQL) enables users to query relational databases without writing SQL, and has been a long-standing goal spanning NLP and data management. Recent large language models (LLMs) substantially improve text-to-SQL generation on cross-domain benchmarks such as Spider (Yu et al., 2018), but practical deployment is still bottlenecked by the *natural language interface* itself: real user questions are often noisy, underspecified, and weakly aligned with database semantics.

Most neural and LLM-based NL2SQL systems remain largely *decoder-driven*, they ask a model to simultaneously interpret the question, ground it to a schema, and synthesize an executable query in one shot (or with lightweight post-hoc repairs). When the input question is imperfect, small language-level misunderstandings can propagate into incorrect joins, conditions, and aggregations, making schema linking and single-pass decoding brittle. Despite extensive efforts on schema linking, prompting, and execution-based correction, comparatively fewer approaches explicitly address the *linguistic sources* of misalignment before SQL decoding. A natural response is to introduce intermediate representations (IRs) such as SQL sketches, SemQL, NatSQL, and related templates (Lee and Baik, 1999; Gan et al., 2021c; Fan et al., 2024). While IRs can reduce the search space for decoding, they typically depart from natural language and may discard useful linguistic cues; once an incorrect template is selected, errors can become difficult to recover. More importantly, many IR-based pipelines still leave the core ambiguity of the original question unresolved, delegating schema alignment to the decoder.

Figure 5 in Appendix A.2 illustrates recurring language-level issues that make NL2SQL brittle under noisy, user-style questions. We observe four patterns: (i) *unnecessary reasoning burden*, where intent is expressed indirectly and must be normalized into explicit operators or predicates (e.g., "from the oldest to the youngest" ⇒ `ORDER BY age DESC`, "in year 2014 or after" ⇒ `year ≥ 2014`); (ii) *ambiguity*, where a mention does not uniquely resolve to a schema field or is easily

*Corresponding author

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 8915–8933
July 2-7, 2026

misinterpreted (e.g., "nation" → `singer.country`, "average" as an attribute vs. an `AVG` operator); (iii) *redundancy*, where verbose wording increases surface length and distracts downstream reasoning without adding constraints; and (iv) *schema unawareness*, where user-friendly values or predicates do not match the database encoding or attribute naming (e.g., "female" vs. `sex` $\in \{F, M\}$, "opened" vs. `Open_Year`). These mismatches systematically induce intent–schema misalignment, weakening surface-form schema linking and making single-pass decoding error-prone.

To address this gap, we propose **QBridge**, a feedback-driven NL2SQL framework that bridges natural language and SQL through an explicit, schema-aware *linguistic* interface. At its core is **Gold Query**, a structured reformulation of the user question that follows a canonical *Gold Query Norm* and makes clause-level intent explicit while remaining readable natural language. Building on this interface, *QBridge* decomposes NL2SQL into three cooperating stages: (i) *SL-Independent NLQ Rewriting* produces a Gold-Query-style rewrite; (ii) *Feedback-Driven Query Refinement* verifies and conservatively edits the rewrite into a high-fidelity *Refined Gold Query*; and (iii) *Feedback-Driven SQL Refinement* iteratively improves the generated SQL using dual signals from execution validity and semantic consistency (via reverse deduction). This decomposition yields interpretable intermediate artifacts and enables plug-and-play integration with diverse SQL backbones without fine-tuning the decoder.

Our main **contributions** are as follows.

- *Gold Query Paradigm.* We introduce the *Gold Query Norm* as a novel intermediate representation that bridges natural language and SQL. This is the first time such a paradigm has been proposed, and through extensive experiments, we demonstrate its effectiveness in improving the accuracy and reliability of NL2SQL systems.

- *Distilled Back-Translation (DBT) Workflow.* We propose a new DBT workflow that includes high-quality data construction and distillation processes. This workflow ensures the quality of the training data (particularly the rewritten data) and is reusable across different tasks and models, making it a versatile approach for improving model performance in various scenarios.

- *QBridge framework.* We propose a novel framework, **QBridge**, an agentic NL2SQL framework that decomposes generation into *SL-Independent NLQ Rewriting*, *Feedback-Driven Query Refinement*, and *Feedback-Driven SQL Refinement* to bridge NLQ and SQL through the *Refined Gold Query Paradigm* with fine-grained feedback-driven refinement mechanism.

- *Comprehensive zero-shot evaluation and analysis.* We conduct comprehensive evaluation on Spider, BIRD, and three Spider-derived robustness variants, together with analyses of cost/latency, rewriter generalization, robustness without explicit schema linking, structure-stratified performance, and BIRD-oriented NLQ issue taxonomy. Results show state-of-the-art zero-shot performance on Spider, strong EX/VES on BIRD-dev, and consistent robustness and cost-efficiency across settings.

2 Related Work

NL2SQL. Traditional NL2SQL was mainly treated as semantic parsing, evolving from template- or grammar-based systems to neural encoder–decoder models that jointly represent the question and schema (Gan et al., 2021c; Lee and Baik, 1999; Cao et al., 2024). Recent advances emphasize schema-aware encoding and constrained decoding for cross-domain generalization on benchmarks such as Spider (Yu et al., 2018), including graph-based encoders and structured decoding (Yin et al., 2020). A common ingredient is schema linking, which reduces schema ambiguity by aligning mentions in the question to tables/columns, but can be brittle under paraphrases, implicit mentions, and noisy user phrasing. With large language models, NL2SQL has shifted toward prompting, in-context learning, and lightweight adaptation (Liu et al., 2023; Dong et al., 2023; Wang et al., 2020; Xie et al., 2024). To improve reliability beyond single-pass decoding, many LLM-based systems introduce decomposition and verification loops (Pourreza and Rafiei, 2023; Gao et al., 2024a; Cen et al., 2025; Fu et al., 2023): they break the task into schema understanding, draft SQL generation, and iterative repair, often guided by execution feedback or LLM-based critique. Agentic designs further separate these roles into cooperative modules to reduce hallucinations and improve interpretability (Wang et al., 2025; Shao et al., 2025; Chaturvedi et al., 2025; Gao et al., 2024b). QBridge follows this trend of verifiable, multi-stage generation, but focuses

on stabilizing the *natural-language interface* via a structured rewrite that is schema-aware yet does not require token-level SL at inference time.

Distilled Back-Translation. Back-translation was originally proposed in neural machine translation (Sennrich et al., 2016; Edunov et al., 2018) to synthesize training pairs by translating monolingual data with a reverse model, and distillation can further compress such synthetic supervision into a smaller student (Gou et al., 2021; Fujita and Takada, 2025). In our setting, we use SQL as the pivot supervision: a teacher LLM generates candidate structured rewrites, and we retain only those that are *SQL-grounded and behaviorally consistent* under verification (Andersen, 1990; He et al., 2025; Uhlig et al., 2025). We then distill these verified rewrites into a lightweight, plug-and-play rewriter. This differs from conventional back-translation in that the supervision signal is not merely a reverse-model output, but is filtered by executable, SQL-grounded consistency, aligning the rewriter with downstream SQL decoding.

3 QBridge

QBridge is a three-stage NL2SQL pipeline that bridges natural language and SQL via an interpretable, schema-aware natural-language intermediate representation. Given a question q and schema S, it (i) rewrites q into a structured *Gold Query* $\tilde{r}$, (ii) verifies it into a *Refined Gold Query* $\hat{r}$ with conservative edits, and (iii) generates SQL and refines it with execution- and semantics-based feedback until convergence. Figure 1 provides an overview. We first introduce the *Refined Gold Query Paradigm* (§3.1), then describe how DBT trains an SL-independent rewriter (§3.2), and finally present the feedback-driven refinement stages for $\hat{r}$ and SQL (§3.3–§3.4).

3.1 Refined Gold Query Paradigm

We reformulate NL2SQL with an explicit natural-language intermediate representation: $(q, S) \rightarrow \tilde{r} \rightarrow \hat{r} \rightarrow \hat{s}$, where $\tilde{r}$ is a structured rewrite (*Gold Query*) and $\hat{r}$ is its verified version (*Refined Gold Query*). This decouples linguistic normalization from schema reasoning and symbolic SQL generation, and provides an interpretable interface for verification and correction.

Definition 1 (Gold Query Norm)**.** *A canonical, schema-grounded rewriting scheme that specifies an SQL-aligned slot order and phrasing rules for natural-language queries.*

Definition 2 (Gold Query)**.** *A Gold Query is a schema-aware reformulation r^* of a user question q over schema S that follows the Gold Query Norm, verbalizing clause semantics in the fixed slot order* `[SELECT, FROM, JOIN, WHERE, GROUPBY, ORDERBY, LIMIT]`.

Gold Queries reduce ambiguity by making clause-level intent explicit while remaining natural-language, which stabilizes downstream decoding and mitigates error propagation before SQL generation.

End-to-end inference. Given (q, S), QBridge first produces a structured rewrite $\tilde{r}$ (Stage I), verifies it into $\hat{r}$ (Stage II), and then generates and refines SQL into $\hat{s}$ (Stage III). Formally, $\hat{s} = \mathcal{R}_{\text{SQL}}(\mathcal{G}_{\text{SQL}}(\hat{r}, S), f_{\text{exec}}, f_{\text{sem}})$, where the refiner uses execution validity and semantic alignment to guide corrections. Figure 2 illustrates the three stages: Stage I produces a clause-explicit rewrite, Stage II removes unsupported or drifting parts to obtain $\hat{r}$, and Stage III corrects execution and semantic mismatches to output the final SQL.

3.2 Distilled Back-Translation for SL-Independent Rewriting

Teacher generation and verification. During training, we assume access to triples (q, S, s^*), where s^* is the Gold SQL. A teacher LLM $\mathcal{M}_T$ samples a set of candidate rewrites that aim to follow the *Gold Query Norm*:

$$\mathcal{R} = \{r_i \sim \mathcal{M}_T(q, S, s^*)\}_{i=1}^{k}. \quad (1)$$

We then retain only *verified* candidates whose induced SQL is executable and execution-consistent with s^* under a fixed SQL generator $\mathcal{G}_{\text{SQL}}$. The surviving rewrite is denoted as r^*, yielding a rewriting corpus $\mathcal{D} = \{(q, S, r^*)\}$ that filters hallucinated schema elements and logical drift, and aligns rewriting behavior with the downstream generator.

Student distillation objective. We fine-tune a lightweight student rewriter $\mathcal{M}_S$ on $\mathcal{D}$ with supervised learning to predict the verified rewrite given only the question and schema:

$$\mathcal{L}_{\text{SFT}} = -\sum_{i=1}^{N} \log P_{\mathcal{M}_S}(r_i^* \mid q_i, S_i). \quad (2)$$

At inference time, the student directly generates $\tilde{r}$ from (q, S), removing any dependence on Gold

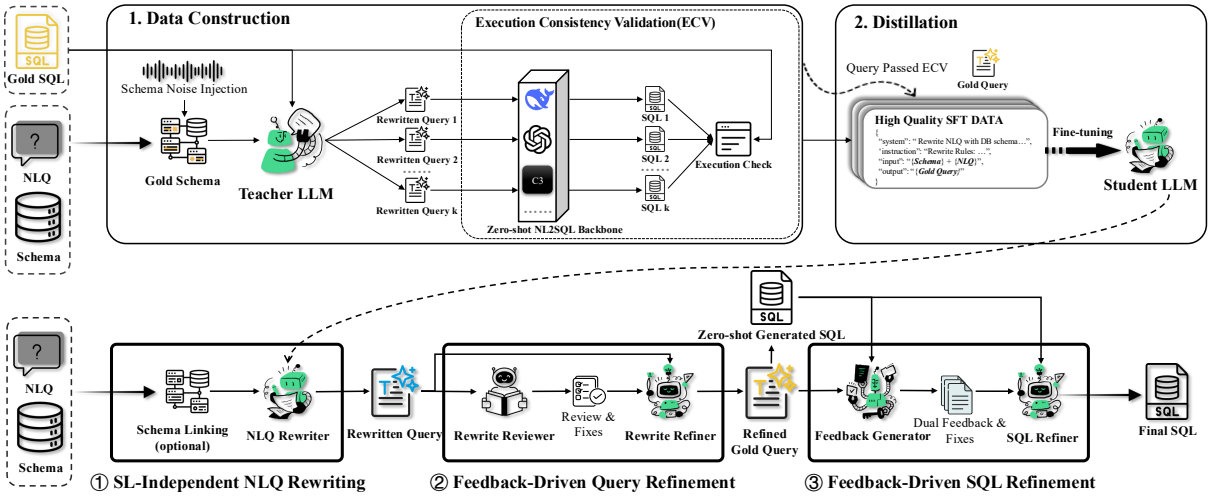

Figure 1: Overview of *QBridge*. Stage I rewrites NLQ into a structured *Gold Query* via distilled back-translation; Stage II verifies and minimally edits it into a *Refined Gold Query*; Stage III generates SQL and refines it with execution and semantic feedback.

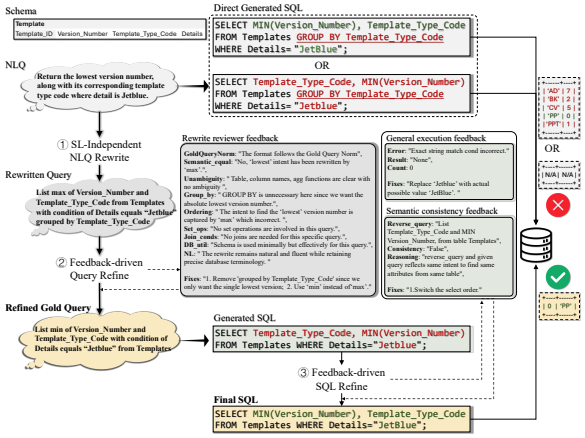

Figure 2: An end-to-end example of *QBridge*. Correct is marked green and incorrect is marked red.

SQL and explicit schema linking while inheriting SQL-grounded rewriting behavior from the teacher. We provide the complete protocol for constructing $\mathcal{D}$, including the verification procedure and robustness augmentations, in Appendix A.3.1.

3.3 Feedback-Driven Query Refinement

While Stage I produces structured rewrites $\tilde{r}$ using a rewriter learned via DBT, the output may still contain hallucinated schema elements, over-reformulations, or subtle logical deviations that mislead SQL decoding. Stage II therefore verifies $\tilde{r}$ with clause-level constraints and performs conservative, targeted edits to obtain the final *Refined Gold Query* $\hat{r}$. The refinement stage consists of two cooperative agents. A *Rewrite Reviewer* diagnoses violations and proposes minimal edit actions, and a *Rewrite Refiner* applies only these actions under a constrained-edit policy, yielding $\hat{r}$ as the verified input to downstream SQL generation.

Multi-Dimensional Rewrite Reviewer. Given schema S, the original question q, and the rewrite $\tilde{r}$, the reviewer audits $\tilde{r}$ along a fixed checklist that covers semantic equivalence, norm adherence (slot order), ordering and grouping logic, join validity, condition soundness, schema coverage, and conciseness. It returns a set of detected issues together with minimal, actionable edits:

$$\left[f_{\text{review}},\ \texttt{fixes}_{\text{rewrite}}\right] \leftarrow \mathcal{R}_{\text{Review}}(\tilde{r}, q, S). \quad (3)$$

We enforce a conservative optimization principle: edits are suggested only when necessary to restore faithfulness and structural consistency, and the reviewer avoids introducing new assumptions or speculative schema elements. The concrete checklist and the reviewer prompt are provided in Appendix A.4.

Feedback-Driven Rewrite Refiner. The refiner applies $\texttt{fixes}_{\text{rewrite}}$ to update $\tilde{r}$ and outputs the verified rewrite $\hat{r}$:

$$\hat{r} \leftarrow \mathcal{R}_{\text{Refine}}(\tilde{r}, \texttt{fixes}_{\text{rewrite}}, q, S). \quad (4)$$

It follows a constrained-edit policy that preserves user intent, adheres to the *Gold Query Norm*, and avoids adding unsupported joins, attributes, or conditions. This design reduces over-editing and prevents new hallucinations, producing a faithful and SQL-ready *Refined Gold Query*.

3.4 Feedback-Driven SQL Refinement

Given the verified *Refined Gold Query* $\hat{r}$ from Stage II and schema S, Stage III closes the loop between SQL generation, validation, and correction. A zero-shot SQL generator $\mathcal{G}_{\text{SQL}}$ first produces candidate SQLs $\{\tilde{s}^{(k)}\}_{k=1}^{K} = \mathcal{G}_{\text{SQL}}(\hat{r}, S)$. When $K > 1$, we optionally apply self-consistency sampling (Wang et al., 2022) to select an initial hypothesis $\tilde{s}_0$. Although $\tilde{s}_0$ is often executable, it may still deviate from the intent of $\hat{r}$, especially on joins, aggregations, and conditions. We therefore refine it with dual feedback, details are shown in Figure 6 in Appendix A.3.3.

Dual feedback and refinement. At iteration t, we derive execution-level feedback by running the query:

$$\phi_t^e = f_{\text{exec}}(\tilde{s}_t, S). \tag{5}$$

We also derive semantic-level feedback through reverse deduction:

$$(q_r, \Delta_t^s) \leftarrow \mathcal{D}_{\leftarrow}(\tilde{s}_t; \hat{r}, S). \tag{6}$$

where with reverse-deduction operator as $\mathcal{D}_{\leftarrow}$, q_r is a reverse-deduced query representation induced from $\tilde{s}_t$ that makes mismatches against $\hat{r}$ explicit, and Δ_t^s summarizes minimal semantic edit actions (e.g., aggregation, filtering, and join logic).

We convert execution feedback into edit actions $\Delta_t^e = \texttt{fixes}_{\text{exec}}(\phi_t^e)$ and update the SQL as:

$$\tilde{s}_{t+1} = \mathcal{R}_{\text{SQL}}(\tilde{s}_t, \Delta_t^e, \Delta_t^s, S). \tag{7}$$

We stop when the query is executable and no semantic edits remain, or when reaching the maximum iteration $t_{\max}$. The final output $\hat{s}$ satisfies both executability and semantic faithfulness to $\hat{r}$.

Real database-grounded case canonicalization. To handle casing mismatches between user value mentions and database entries, we canonicalize string literals via case-insensitive database lookup and replace them with the database form when a case-folded exact match exists. Details are provided in Appendix A.3.3.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate *QBridge* on two primary benchmarks, Spider (Yu et al., 2018) and BIRD (Li et al., 2023b), together with three Spider-derived robustness variants: Spider-DK (Gan et al., 2021b), Spider-SYN (Gan et al., 2021a), and Spider-Realistic (Deng et al., 2021). Spider is the standard cross-domain benchmark for compositional text-to-SQL, spanning diverse schemas and complex multi-table queries. BIRD-dev serves as a complementary benchmark for evaluating transfer beyond Spider-style data, and we report EX and VES on it. The three Spider variants probe complementary robustness challenges aligned with our goal: Spider-DK tests implicit domain knowledge, Spider-SYN replaces schema terms with synonyms to reduce lexical overlap, and Spider-Realistic rewrites NLQs into less schema-explicit, more user-like phrasing. Statistics are summarized in Table 14 in Appendix A.5.

Evaluation Metrics. We report Execution Accuracy (EX), Exact Match (EM), and Test-Suite Accuracy (TS) under the standard Spider protocol. EX compares execution results, EM matches SQL structure after value normalization, and TS mitigates false positives in EX via a curated test suite. For BIRD, we report EX and VES following the standard BIRD evaluation protocol. To characterize practical inference overhead, we additionally report Avg. Tokens / Q, Avg. Time / Q, Avg. Cost / Q, and EX / Avg. Cost on Spider. Avg. Tokens / Q measures the realized token usage aggregated over the full inference pipeline, including all LLM calls used for rewriting, review/refinement, SQL generation, repair, and optional self-consistency sampling. Avg. Time / Q is the end-to-end wall-clock latency from input (q, S) to the final SQL under a unified deployment setting. Avg. Cost / Q is estimated from the realized token usage based on the provider pricing in our experiments. EX / Avg. Cost summarizes cost-efficiency by normalizing execution accuracy with respect to per-query monetary cost. Unlike static prompt-budget proxies, these metrics reflect the actual realized overhead of the full multi-stage inference pipeline.

Baselines. We include strong **zero-shot** LLM baselines, including ChatGPT-SQL + ChatGPT (Liu et al., 2023), zero-shot GPT-4 (Pourreza and Rafiei, 2023), and C3 (Dong et al., 2023) instantiated with ChatGPT or GPT-4. They require no task-specific fine-tuning and serve as the main references for assessing the gain from our rewriting–refinement pipeline under a zero-shot SQL backbone. We also compare against **few-shot** prompting baselines (few-shot GPT-4 and SQL-PaLM (Sun et al., 2023)), which often improve accuracy at the cost of longer prompts and thus pro-

Table 1: Overall comparison on Spider and BIRD. *QBridge* keeps the SQL backbone in *zero-shot* mode. The best result is bolded and second-best is underlined. ZS, FS, FT denotes Zero-Shot, Few-Shot and Fine-Tuned.

Approach	Paradigm			Spider Dev			Spider Test		Spider Mean		BIRD Dev	
	ZS	FS	FT	EX%	EM%	TS%	EX%	EM%	EX%	EM%	EX%	VES
Zero-Shot NL2SQL												
ChatGPT-SQL + ChatGPT (Liu et al., 2023)	✓	–	–	70.1	37.9	60.1	–	–	–	–	43.8	37.2
Zero-Shot + GPT-4	✓	–	–	72.9	42.4	64.9	–	–	–	–	46.4	49.8
C3 + ChatGPT (Dong et al., 2023)	✓	–	–	81.8	43.1	72.1	82.3	–	82.1	–	50.2	46.3
C3 + GPT-4	✓	–	–	82.1	50.7	–	–	–	–	–	–	–
Few-Shot Prompting												
Few-Shot + GPT-4	–	✓	–	76.8	54.3	67.4	–	–	–	–	–	–
Few-Shot SQL-PaLM (Sun et al., 2023)	–	✓	–	82.8	78.2	77.3	–	–	–	–	–	–
Fine-Tuned / Agentic Systems												
Fine-Tuned SQL-PaLM (Sun et al., 2023)	✓	–	✓	82.8	–	78.2	–	–	–	–	53.6	–
RESDSQL-3B + NatSQL (Li et al., 2023a)	✓	–	✓	84.1	**80.5**	79.9	79.9	**72**	82.0	**76.3**	–	–
SFT CodeS-7B (Li et al., 2024)	✓	–	✓	85.4	–	80.3	–	–	–	–	57.2	58.8
COG-SQL + GPT-4 (Yuan et al., 2025)	✓	–	✓	85.4	–	–	86.4	–	85.9	–	59.6	64.3
DAIL-SQL + GPT-4 (Gao et al., 2024a)	–	✓	✓	83.6	68.7	76.2	86.2	–	84.9	–	54.8	56.1
DAIL-SQL + DeepSeek-V3	–	✓	✓	82.3	56.0	74.4	–	–	–	–	57.0	59.1
DIN-SQL + GPT-4 (Pourreza and Rafiei, 2023)	–	✓	✓	82.8	60.1	74.2	85.3	60.0	84.1	60.1	50.7	58.8
DIN-SQL + DeepSeek-V3	–	✓	✓	66.7	57.4	61.5	79.9	60.9	73.3	59.2	52.9	54.0
MAC-SQL + GPT-4 (Wang et al., 2025)	–	✓	✓	86.8	63.2	–	82.8	–	84.8	–	57.6	59.5
MAC-SQL + DeepSeek-V3	–	✓	✓	80.1	58.5	67.5	77.9	19.2	79	38.9	62.7	66.9
Zero-Shot + Lightweight Adapter (Ours)												
QBridge + GPT-4o	✓	–	✓(†)	85.5	56.0	79.6	82.3	61.6	83.9	58.8	61.2	71.4
QBridge + DeepSeek-V3	✓	–	✓(†)	86.4	57.4	80.8	85.9	38.7	86.2	48.1	63.4	70.7
QBridge + DeepSeek-V3 + SC	✓	–	✓(†)	**86.9**	65.7	**81.5**	**86.7**	39.1	**86.8**	52.4	**64.9**	**71.9**

(†) Only the 7B rewriter adapter is fine-tuned; "SC" = Self-Consistency; "–" = not reported/open-source in performance.

vide a natural contrast to our prompt-budget analysis. Finally, we include representative **fine-tuned** and **agentic** systems—RESDSQL (Li et al., 2023a), DAIL-SQL (Gao et al., 2024a), DIN-SQL (Pourreza and Rafiei, 2023), COG-SQL (Yuan et al., 2025), and MAC-SQL (Wang et al., 2025)—to contextualize performance relative to methods that rely on specialized training or multi-step coordination.

Implementation Details. We use DeepSeek-V3 as the teacher LLM for DBT data construction (with ECV) and as the zero-shot SQL backbone at inference. This aligns DBT verification with downstream decoding and is cost-critical for our multi-stage pipeline. We additionally evaluate *QBridge* with GPT-4o for extended validation.

For generation steps (DBT sampling, SQL generation, and refinement), we use the provider default temperature, while setting temperature to 0 for verification modules (rewrite review and reverse deduction) for stable checking. We distill the rewriter by fine-tuning Qwen2.5-7B-Instruct (Team et al., 2024) on NVIDIA A800 GPUs.

4.2 Overall Performance

We evaluate *QBridge* against representative NL2SQL systems on Spider and BIRD-dev. Table 1 summarizes the main results. On Spider, QBridge + DeepSeek-V3 achieves 86.2% mean EX across dev/test and 80.8% TS on Spider dev, and further reaches 86.8% mean EX and 81.5% TS with self-consistency. On BIRD-dev, QBridge + DeepSeek-V3 reaches 63.4% EX and 70.7 VES, and further improves to 64.9% EX and 71.9 VES with self-consistency. These results outperform zero-shot baselines, remain competitive with strong fine-tuned or agentic systems, and show that the rewriting–refinement pipeline transfers beyond Spider.

Beyond accuracy, we further evaluate practical deployment overhead on Spider. Since QBridge uses a multi-stage rewrite–review–repair pipeline and optional self-consistency, static prompt proxies do not fully reflect realized inference overhead. We therefore report measured tokens, latency, monetary cost, and cost-efficiency in Table 2.

We further evaluate on Spider-derived robustness variants. As shown in Figure 3, *QBridge* achieves the highest execution accuracy across Spider-DK, Spider-SYN, and Spider-Realistic, demonstrating robustness to domain-knowledge requirements and less schema-explicit phrasing.

Finally, we observe a modest drop when pairing the distilled rewriter with a different backbone than the one used during DBT verification. This is expected because DBT implicitly aligns the rewrite distribution with the downstream SQL generator used in verification; we provide further analysis in Appendix A.5.

Table 2: Measured cost and latency on Spider. Avg. Tokens / Q, Avg. Time / Q, and Avg. Cost / Q reflect realized end-to-end inference overhead. EX / Avg. Cost measures cost-efficiency. Higher is better for EX and EX / Avg. Cost; lower is better for the other metrics.

Method	Avg. Tokens / Q	Avg. Time / Q	Avg. Cost / Q	EX (%)	EX / Avg. Cost
C3SQL + GPT-3.5	5702	–	0.0114	82.0	7190
SuperSQL + GPT-4	942	–	0.0283	87.0	3079
DTS-SQL + GPT-4	1081	7.7s	0.0324	71.0	2189
DEA-SQL + GPT-4	12038	100.2s	0.3611	19.0	53
DIN-SQL + GPT-4	10453	16.2s	0.3136	55.3	176
DAIL-SQL + GPT-4	932	2.4s	0.0280	74.8	2675
DAC + GPT-4	11741	7.4s	0.3522	75.7	215
QBridge + DeepSeek-V3	3328.26	23.71s	**0.0010**	86.4	**86636**

4.3 Effectiveness of the Distilled Rewriter and Gold Query Norm

To isolate the effect of rewriting, we compare different rewriters under the same DeepSeek-V3 SQL backbone. All settings are kept identical including zero-shot SQL prompt (Liu et al., 2023), M-Schema representation (Gao et al., 2024b), and the same schema linking module. Table 3 reports EX/EM on Spider dev. Two observations stand out. First, the DBT-distilled Qwen2.5-7B rewriter is the strongest rewriter overall. Compared with *Rewriter-Free*, our final configuration improves by +5.4 EX and +28.9 EM, indicating that DBT successfully distills *verified*, SQL-aligned rewriting behavior into a lightweight student model. Second, rewriting is not always beneficial: naive rewriting can drift away from the original intent and propagate errors to SQL generation. This is reflected in Table 3, where several off-the-shelf rewriters degrade once forced into a rigid *Gold Query Norm*. In contrast, our DBT-distilled rewriter benefits from the *Gold Query Norm* (+2.4 EX and +11.4 EM over its unconstrained rewriting), because it is explicitly trained to produce rewrites that remain semantically equivalent to original NLQs. Together,

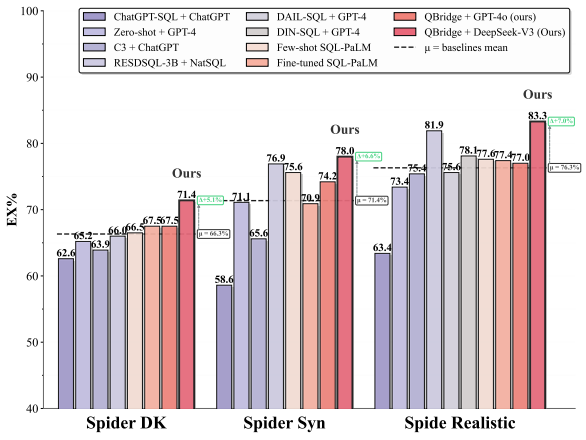

Figure 3: Comparison of EX accuracy on Spider-DK, Spider-SYN and Spider-Realistic.

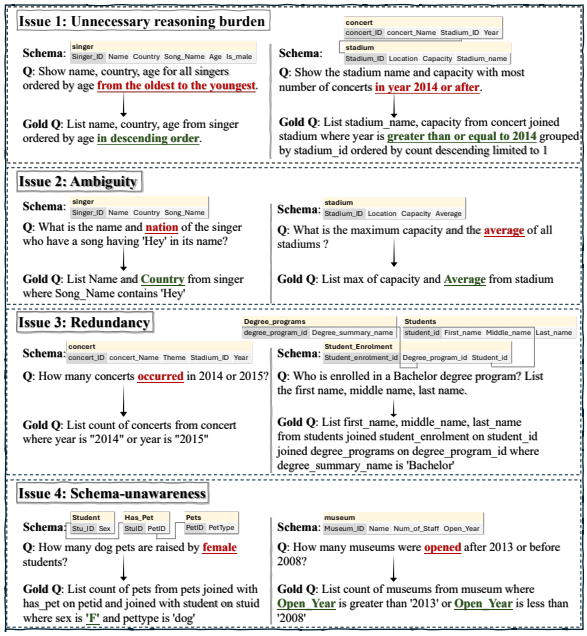

Figure 4: Demonstration of typical NLQ representation issues rewriting solution.

Table 3: Comparison of EX/EM across different rewrite strategies. "+ *Gold Query Norm*" denotes the use of our structured *Gold Query* representation.

Rewrite Strategy	EX %	EM %
Rewriter-Free (Zero-Shot NL2SQL)	79.5	36.3
DeepSeek-V3 Rewriter	78.7	39.3
+ *Gold Query Norm*	75.6 (-3.1)	39.8 (+0.5)
GPT-4o Rewriter	79.3	42.5
+ *Gold Query Norm*	78.4 (-0.9)	41.8 (-0.7)
Gemini-2.5-flash Rewriter	58.9	33.5
+ *Gold Query Norm*	46.4 (-12.5)	20.9 (-12.6)
Llama3-8b-Instruct Rewriter	68.7	27.9
+ *Gold Query Norm*	41.1 (-27.6)	17.8 (-10.1)
Qwen2.5-7B-Instruct Rewriter	76.7	35.4
+ *Gold Query Norm*	61.0 (-15.7)	38.6 (+1.2)
Distilled Qwen2.5-7B-Instruct Rewriter	82.5	53.8
+ *Gold Query Norm* (Ours)	**84.9** (+2.4)	**65.2** (+11.4)

DBT and the norm turn rewriting from a potentially harmful transformation into an effective, reliable intermediate representation for downstream SQL decoding.

We further analyze performance on structurally complex SQL by grouping sampled queries according to three common structural patterns: nested queries, set operations, and multi-table joins. Table 4 reports execution accuracy on 500 sampled queries from Spider-dev and 500 sampled queries from BIRD-dev.

On Spider, QBridge achieves the highest accuracy on nested queries and set operations, while remaining competitive on multi-table joins. On BIRD, all methods become more challenged, but QBridge still attains the strongest accuracy on

Table 4: Structure-stratified execution accuracy (%) on sampled Spider-dev and BIRD-dev queries. Each dataset uses 500 sampled queries. Higher is better.

Dataset	Method	Nested Query	Set Operations	Multi-table JOIN
Spider	QBridge	**71.7**	**75.0**	68.9
	C3-SQL	50.3	42.5	52.7
	DAIL-SQL	67.9	65.0	64.9
	MAC-SQL	66.0	56.2	**77.0**
	DIN-SQL	6.3	7.5	36.5
BIRD	QBridge	39.7	**33.3**	**61.8**
	C3-SQL	36.5	**33.3**	43.8
	DIN-SQL	37.3	**33.3**	43.0
	MAC-SQL	**49.2**	**33.3**	56.2

multi-table joins and remains competitive on the other structures. These results indicate that the Gold Query serves as an intent-stabilizing intermediate representation without preventing the framework from handling complex SQL composition. We also provide a targeted BIRD NLQ-issue analysis in Appendix A.1. On a paired 500-query subset, adding the NLQ rewriter improves EX from 69.4% to 77.0%, with the largest gains on unnecessary reasoning and ambiguity. This directly supports our motivation that stabilizing the natural-language interface is especially beneficial when the original question is linguistically noisy or weakly aligned with schema terms.

Beyond quantitative gains, Figure 4 illustrates how *Gold Query* reformulation resolves common NLQ issues. It normalizes vague or compositional expressions (e.g., "*from the oldest to the youngest*" → "*in descending order*"), disambiguates underspecified mentions, and adds missing schema-grounded constraints, yielding a more compositional intermediate form that supports stable SQL decoding.

4.4 Generalization Ability of the Rewriter

We assess the generalization ability of our distilled rewriter by applying it as a plug-and-play preprocessing module to a range of zero-shot NL2SQL LLMs and representative baseline architectures. In all experiments, the downstream SQL generator remains unchanged; only the NLQ is rewritten. Table 5 shows that our rewriter consistently improves zero-shot NL2SQL across diverse backbones (ChatGPT, GPT-4o, Gemini-2.5-Pro (Team et al., 2023), Qwen3 (Yang et al., 2025), and DeepSeek-V3). EX improves by +1.6 to +5.4 points, and EM gains reach up to +28.9 points. The largest gains appear on DeepSeek-V3, indicating that the distilled rewriter is especially effective when paired with the backbone used during DBT verification. Table 6 evaluates our rewriter as a plug-in preprocessing module for representative NL2SQL systems (ChatGPT-SQL, DIN-SQL, MAC-SQL, and DAIL-SQL) under the same DeepSeek-V3 backbone. The rewriter improves all baselines, with particularly large gains on DIN-SQL (+14.0 EX and +12.0 EM), and also improves *QBridge* itself (+1.4 EX and +14.9 EM). Overall, the rewriter acts as a model-agnostic enhancement that complements downstream decoding and verification.

Table 5: Comparison of EX/EM across different zero-shot NL2SQL backbones LLMs with and without rewriter. "vanilla" denotes models without rewriting.

Backbone Zero-Shot LLM	Strategy	EX %	EM %
ChatGPT	vanilla	76.3	44.8
	w/ our rewriter	79.5 (+3.2)	58.0 (+13.2)
GPT-4o	vanilla	80.9	44.8
	w/ our rewriter	82.5 (+1.6)	56.2 (+11.4)
Gemini-2.5-Pro	vanilla	78.5	60.4
	w/ our rewriter	81.8 (+3.3)	59.6 (-0.8)
Qwen3-235B-A22B	vanilla	79.4	38.2
	w/ our rewriter	82.2 (+2.8)	47.0 (+8.8)
DeepSeek-V3	vanilla	79.5	36.3
	w/ our rewriter	**84.9 (+5.4)**	**65.2 (+28.9)**

Table 6: Comparison of EX/EM across different baselines with and without the NLQ rewriter. "vanilla" denotes no rewriting.

Approach	Strategy	EX %	EM %
ChatGPT-SQL + ChatGPT	vanilla	70.1	37.9
	w/ our rewriter	79 (+8.9)	51.0 (+13.1)
DIN-SQL + DeepSeek-V3	vanilla	66.7	57.4
	w/ our rewriter	80.7 **(+14.0)**	**69.4** (+12.0)
MAC-SQL + DeepSeek-V3	vanilla	80.1	58.5
	w/ our rewriter	82.1 (+2.0)	60.3 (+1.8)
DAIL-SQL + DeepSeek-V3	vanilla	82.3	56
	w/ our rewriter	83.8 (+1.5)	61.6 (+5.6)
QBridge + DeepSeek-V3 (Ours)	vanilla	85.0	42.5
	w/ our rewriter	**86.4** (+1.4)	57.4 **(+14.9)**

4.5 Robustness of the SL-Independent Rewriter

To test whether *QBridge* remains effective without explicit schema linking, we remove each method's declared linking module (e.g., the *Selector* in MAC-SQL) while keeping all other settings unchanged. As shown in Table 7, conventional systems degrade substantially without linking (e.g., RESDSQL-Base and DAIL-SQL drop by 7.8 and 4.7 EX points), indicating a strong reliance on token-level alignment. In contrast, *QBridge* drops by only 0.3 EX and 1.7 EM. This robustness comes from two designs: (i) Gold Query rewriting explicitly grounds tables, attributes, and joins in the intermediate representation, and (ii) DBT training injects schema noise, encouraging the rewriter to identify the essential schema elements under noisy metadata.

Table 7: Comparison of models with and without schema linking (SL). "w/o SL" denotes removing each model's declared schema linking component.

Approach	Strategy	EX %	EM %
C3 + ChatGPT	w/ SL w/o SL	81.8 79.5 (-2.3)	43.1 –
COG-SQL + GPT-4o-mini	w/ SL w/o SL	84.2 82.2 (-2.0)	– –
RESDSQL-Base	w/ SL w/o SL	77.9 70.1 (-7.8)	71.7 –
DIN-SQL + DeepSeek-V3	w/ SL w/o SL	66.7 62.3 (-4.4)	57.4 52.5 (-4.9)
MAC-SQL + DeepSeek-V3	w/ SL w/o SL	80.1 79.8 (-0.3)	58.5 44.9 (-13.6)
DAIL-SQL + DeepSeek-V3	w/ SL w/o SL	82.3 77.6 (-4.7)	56.0 43.3 (-12.7)
QBridge + DeepSeek-V3 (Ours)	w/ SL w/o SL	**86.4** **86.1 (-0.3)**	**57.4** **55.7 (-1.7)**

4.6 Ablation Study

To evaluate the contribution of each component in *QBridge*, we conduct ablations on the Spider dev set. Starting with *QBridge*+DeepSeek-V3 (without self-consistency), we sequentially remove the NLQ Rewriter, Query Refiner, SQL Refiner, and their combinations, and also assess the effect of adding self-consistency during SQL generation. Table 8 reports EX by difficulty level and overall. Removing the NLQ Rewriter causes the largest drop (–1.4% overall EX), particularly on medium queries (90.6% $\rightarrow$ 88.1%), highlighting the importance of rewriting ambiguous NLQs into structured *Gold Queries*. Removing either the *Query Refiner* or *SQL Refiner* results in slight accuracy reductions of 0.3% and 0.2%, respectively, with a larger drop (1.0%) when both are removed. This confirms their complementary roles: the *Query Refiner* improves query clarity, while the *SQL Refiner* ensures semantic alignment and execution correctness. Introducing self-consistency during SQL generation yields a stable +0.5 EX gain, by generating multiple SQL candidates and selecting the final one based on execution outcomes, thus mitigating stochastic decoding variance.

Table 8: Ablation study of *QBridge* on Spider Dev across four difficulty levels.

Strategy	Easy	Medium	Hard	Extra	Overall
QBridge (DeepSeek-V3)	94.4	90.6	79.9	**69.9**	86.4
– NLQ Rewriter	92.3	88.1	**82.8**	68.1	85.0(**-1.4**)
– Query Refiner	**95.2**	90.6	80.5	66.3	86.1(**-0.3**)
– SQL Refiner	94.8	91.5	78.7	66.9	86.2(**-0.2**)
– Query Refiner & SQL Refiner	94.4	90.8	78.7	64.5	85.4(**-1.0**)
+ Self-Consistency	94.4	**92.2**	80.5	68.7	**86.9(+0.5)**

5 Conclusion

We introduced *QBridge*, a feedback-driven NL2SQL framework that bridges natural language and SQL through a structured, verifiable intermediate representation. *QBridge* decomposes the process into three stages: SL-independent rewriting using DBT-distilled Gold Queries, feedback-driven query refinement to prevent semantic drift, and dual-feedback SQL refinement to ensure both executability and semantic alignment. This approach enables schema-aware reasoning with less reliance on token-level schema linking, while maintaining performance. Experiments on Spider, BIRD, and three Spider variants show that *QBridge* achieves state-of-the-art zero-shot performance with strong robustness and generalization. Future work will explore scaling for larger, noisier schemas and applying the *Refined Gold Query Paradigm* to other structured prediction tasks.

6 Limitations

Although our design shows significant performance, it involves some practical trade-offs. We use a lightweight 7B rewriter for plug-and-play deployment; scaling it up (or distilling from stronger teachers) is a straightforward path for harder settings with larger and noisier schemas. The multi-stage verification and refinement in *QBridge* improves reliability but adds inference latency compared to single-pass decoding. Finally, since DBT verifies rewrites with a specific SQL backbone during data construction, the distilled rewriter may be mildly coupled to that backbone's decoding style. We leave these extensions to future work.

Acknowledgments

This paper is supported by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No.JYB2025XDXM103), the National Natural Science Foundation of China (NSFC) under Grant No.(62572422, 62502439). The authors are supported by Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

References

Poul Andersen. 1990. How close can we get to the ideal of simple transfer in multi-lingual machine translation (MT)? In *Proceedings of the 7th Nordic Conference of Computational Linguistics (NODALIDA 1989)*, pages 103–113, Reykjavík, Iceland. Institute of Lexicography, Institute of Linguistics, University of Iceland, Iceland.

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

Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. 2025. Sqlfixagent: Towards semantic-accurate text-to-sql parsing via consistency-enhanced multi-agent collaboration. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 49–57.

Saumya Chaturvedi, Aman Chadha, and Laurent Bindschaedler. 2025. Sql-of-thought: Multi-agentic text-to-sql with guided error correction. *arXiv preprint arXiv:2509.00581*.

Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-grounded pretraining for text-to-SQL. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1337–1350, Online.

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, and 1 others. 2023. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*.

Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. 2018. Understanding back-translation at scale. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 489–500, Brussels, Belgium.

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.

Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. Catsql: Towards real world natural language to sql applications. *Proceedings of the VLDB Endowment*, 16(6):1534–1547.

Felipe Fujita and Hideyuki Takada. 2025. Improving low-resource Japanese translation with fine-tuning and backtranslation for the WMT 25 general translation task. In *Proceedings of the Tenth Conference on Machine Translation*, pages 765–768, Suzhou, China.

Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. 2021a. 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)*, pages 2505–2515, Online.

Yujian Gan, Xinyun Chen, and Matthew Purver. 2021b. Exploring underexplored limitations of cross-domain text-to-SQL generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8926–8931, Online and Punta Cana, Dominican Republic.

Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. 2021c. Natural SQL: Making SQL easier to infer from natural language specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2030–2042, Punta Cana, Dominican Republic.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024a. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*

Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, and 1 others. 2024b. Xiyan-sql: A multi-generator ensemble framework for text-to-sql. *arXiv e-prints*, pages arXiv–2411.

Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International journal of computer vision*, 129(6):1789–1819.

Xuan He, Da Yin, and Nanyun Peng. 2025. Guiding through complexity: What makes good supervision for hard reasoning tasks? In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 11006–11046, Albuquerque, New Mexico.

Jeong-Oog Lee and Doo-Kwon Baik. 1999. Semql: a semantic query language for multidatabase systems. In *Proceedings of the eighth international conference on Information and knowledge management*, pages 259–266.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 13067–13075.

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

Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S Yu. 2023. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability. *arXiv preprint arXiv:2303.13547*.

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 36 (NeurIPS 2023)*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 86–96, Berlin, Germany.

Zhihui Shao, Shubin Cai, Rongsheng Lin, and Zhong Ming. 2025. Enhancing text-to-SQL with question classification and multi-agent collaboration. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 4340–4349, Albuquerque, New Mexico.

Ruoxi Sun, Sercan Ö Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. Sql-palm: Improved large language model adaptation for text-to-sql. *CoRR*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Qwen Team and 1 others. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2(8).

Kaden Uhlig, Joern Wuebker, Raphael Reinauer, and John Denero. 2025. Cross-lingual human-preference alignment for neural machine translation with direct quality optimization. In *Proceedings of the Tenth Conference on Machine Translation*, pages 31–51, Suzhou, China.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A multi-agent collaborative framework for text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557, Abu Dhabi, UAE.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Wenxuan Xie, Gaochen Wu, and Bowen Zhou. 2024. Mag-sql: Multi-agent generative approach with soft schema linking and iterative sub-sql refinement for text-to-sql. *arXiv preprint arXiv:2408.07930*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426, Online.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium.

Hongwei Yuan, Xiu Tang, Ke Chen, Lidan Shou, Gang Chen, and Huan Li. 2025. Cogsql: A cognitive framework for enhancing large language models in text-to-sql translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25778–25786.

A Appendix

A.1 NLQ Issue Taxonomy and Targeted Ablation on BIRD

As discussed in Figure 5, many NL2SQL failures originate from mismatches between the surface form of the natural-language question and the schema-grounded query logic. We consider four common issue types: unnecessary reasoning burden, ambiguity, redundancy, and schema-unawareness. To quantify their effect on a more realistic benchmark, we analyze 500 sampled queries from BIRD-dev and compare a variant without the NLQ rewriter against the full QBridge pipeline under the same DeepSeek-V3 backbone.

Table 9 shows that adding the NLQ rewriter improves execution accuracy from 69.4% to 77.0% on this paired subset. Table 10 further breaks the results down by issue type. The largest gains appear on unnecessary reasoning and ambiguity, showing that rewriting is especially beneficial when the original question requires extra inference or uses expressions that do not align well with schema terms. Table 11 shows that the improvement persists across all issue counts, and becomes larger

Table 9: Overall EX on a paired 500-query BIRD-dev subset, with and without the NLQ rewriter.

Method	Correct	Total	EX (%)
w/o NLQ rewriter	347	500	69.4
QBridge + DeepSeek-V3	**385**	500	**77.0**

Table 10: Performance breakdown by NLQ issue type on the paired 500-query BIRD-dev subset.

Issue Type	Samples (n)	w/o NLQ rewriter	QBridge	Improv.
Unnecessary Reasoning	147	53.1	**66.0**	+12.9
Ambiguity	124	59.7	**72.6**	+12.9
Redundancy	67	59.7	**68.7**	+9.0
Schema-unawareness	88	65.9	**69.3**	+3.4

when multiple issue types co-occur in the same question. These results support our formulation of NLQ stabilization as a primary component of robust Text-to-SQL.

A.2 Issues Demonstrations

Figure 5 in Appendix A.2 illustrates typical cases of semantic inconsistency in natural language expressions. Such inconsistencies arise when the linguistic form fails to directly reflect the underlying query logic: (i) *unnecessary reasoning burden*, where phrases like "*from the oldest to the youngest*" require additional inference instead of directly specifying "*ordered by age descending*"; (ii) *ambiguity*, when expressions such as "*nation*" or "*average*" mismatch schema terms; (iii) *redundancy*, where extra words add length without new meaning; and (iv) *schema unawareness*, as in "*female*" versus the column value `sex` $\in$ `{F, M}`. These examples reveal that language-level inconsistency is an overlooked but primary source of semantic misalignment in NL2SQL.

A.3 Method

A.3.1 Data Construction

We construct the DBT rewriting corpus from the training splits of Spider and BIRD. We employ the same LLM as both the teacher $\mathcal{M}_T$ and the NL2SQL backbone within the *Execution Consistency Validation (ECV)* loop, while using a much lighter model as the student $\mathcal{M}_S$. Formally, given a natural question q, its corresponding schema S, and the reference *Gold SQL* s^*, the *DBT* data construction process can be described as:

$$\mathcal{D} = \left\{ (q, S, r^*) \;\middle|\; r^* \sim \mathcal{F}_{\text{ECV}}\big(\mathcal{M}_T(q, S, s^*)\big) \right\}, \tag{8}$$

where $\mathcal{M}_T(q, S, s^*) = \{r_i\}_{i=1}^{k}$ denotes k candidate rewrites generated by the teacher model, and

Table 11: Performance breakdown by the number of NLQ issues on the paired 500-query BIRD-dev subset.

Issue Count	Samples (n)	w/o NLQ rewriter	QBridge	Improv.
0	294	77.2	**83.3**	+6.1
1	89	59.6	**66.3**	+6.7
2	50	52.0	**66.0**	+14.0
3	31	61.3	**74.2**	+12.9
4	36	61.1	**69.4**	+8.3

$\mathcal{F}_{\text{ECV}}$ is the filtering operator that enforces:

$$\mathcal{F}_{\text{ECV}}(r_i) = \begin{cases} r_i, & \text{if ExecCheck}(s_i) \wedge \text{ECV}(s_i, s^*), \\ \varnothing, & \text{otherwise}, \end{cases} \tag{9}$$

where $s_i = \mathcal{G}_{\text{SQL}}(r_i, S)$ is the SQL generated from each rewrite under a simple C3-style prompt (Dong et al., 2023). Among all valid candidates passing both checks, one is randomly sampled and retained as the *Gold Query* r^*, forming a clean and executable corpus $\mathcal{D} = \{(q, S, r^*)\}$. This procedure not only ensures correctness but also internalizes the execution behavior of the ECV-validated backbone into the dataset, aligning the rewriter's output distribution with the SQL decoder.

Data Augmentation. To further enhance quality and robustness, we apply three augmentation strategies during training data construction:

(i) **Schema Noise Injection:** For each instance, extract $\mathcal{T}_{\text{used}}, \mathcal{C}_{\text{used}}$ from s^* and augment with random noise elements $\mathcal{T}_{\text{noise}}, \mathcal{C}_{\text{noise}}$, forming a schema as:

$$\mathcal{T}_{\text{input}} = \mathcal{T}_{\text{used}} \cup \mathcal{T}_{\text{noise}}, \mathcal{C}_{\text{input}} = \mathcal{C}_{\text{used}} \cup \mathcal{C}_{\text{noise}}, \tag{10}$$

to simulate imperfect schema linking.

(ii) **Difficulty- and diversity-aware sampling:** We balance the DBT construction corpus across diverse query complexities and benchmark sources to avoid over-concentration on simple Spider-style instances and to improve robustness under more realistic database settings.

(iii) ***M-Schema* representation:** Linearize each database using the *M-schema* format (Gao et al., 2024b), providing concise and semantically complete relational context with explicit foreign keys.

A.3.2 Distillation

The student rewriter is fine-tuned on $\mathcal{D}$ to generate Gold Queries directly from (q, S) without SQL supervision. Formally, given a dataset $\mathcal{D} = \{(q_i, S_i, r_i^*)\}_{i=1}^{N}$ constructed via Step 8, the objective is to minimize the negative log-likelihood of

Algorithm 1: End-to-end QBridge Inference

Input: Natural language query q; schema $S = (T, C, K)$; optional schema linking flag UseSL; zero-shot SQL generator $\mathcal{G}_{\text{SQL}}$; optional self-consistency budget K; max refine steps $t_{\max}$.
Output: Final SQL $\hat{s}$.

Stage I: SL-Independent NLQ Rewriting
if UseSL **then**
  $[T_{\text{link}}, C_{\text{link}}] \leftarrow \textsc{SchemaLinking}(q, S)$
  $S_{\text{opt}} \leftarrow \textsc{AssembleSchema}(S, T_{\text{link}}, C_{\text{link}})$
else
  $S_{\text{opt}} \leftarrow S$
$\tilde{r} \leftarrow R_{\text{rewrite}}(q, S_{\text{opt}})$
Stage II: Feedback-Driven Query Refinement
$[f_{\text{review}},\ \Delta^r] \leftarrow \mathcal{R}_{\text{Review}}(\tilde{r}, q, S)$
$\hat{r} \leftarrow \mathcal{R}_{\text{Refine}}(\tilde{r}, \Delta^r, q, S)$
SQL Generation
if $K \leq 1$ **then**
  $\tilde{s}_0 \leftarrow \mathcal{G}_{\text{SQL}}(\hat{r}, S)$
else
  $\{\tilde{s}^{(k)}\}_{k=1}^{K} \leftarrow \mathcal{G}_{\text{SQL}}(\hat{r}, S)$
  $\tilde{s}_0 \leftarrow \textsc{SelfConsistency}(\{\tilde{s}^{(k)}\}_{k=1}^{K})$
$t \leftarrow 0$
Stage III: Feedback-Driven SQL Refinement
repeat
  $\phi_{\text{exec}} \leftarrow f_{\text{exec}}(\tilde{s}_t, S)$
  $(q_r, \Delta_t^s) \leftarrow \mathcal{D}_{\leftarrow}(\tilde{s}_t; \hat{r}, S)$
  $\Delta_t^e \leftarrow \texttt{fixes}_{\text{exec}}(\phi_{\text{exec}})$
  if $\textsc{Valid}(\phi_{exec}) \wedge \|\Delta_t^s\| = 0$ **then**
    return $\hat{s} \leftarrow \tilde{s}_t$
  $\tilde{s}_{t+1} \leftarrow \mathcal{R}_{\text{SQL}}(\tilde{s}_t, \Delta_t^e, \Delta_t^s, S)$
  $t \leftarrow t + 1$
until $t \geq t_{\max}$
return $\hat{s} \leftarrow \tilde{s}_t$

generating r_i^* given (q_i, S_i): $\mathcal{L}_{\text{SFT}}$. The Gold SQL is only used in Step 8 for data construction and is removed entirely during training. Through distillation, the rewriter learns to infer high-quality Gold Queries from the interaction between only NLQs and schemas, which is a capability distilled from the teacher's use of Gold SQL. During inference, when only (q, S) is available, as in real-world scenarios, the rewriter can naturally approximate the ground truth *Gold Query* without explicit SQL input. This ensures that it functions under realistic deployment conditions while retaining the precision and consistency learned from SQL-grounded supervision. This process can be viewed as a form of *teacher-forced semantic distillation*, where the student rewriter learns under strong supervision from verified teacher outputs, distilling only execution-consistent rewrites that preserve semantic fidelity.

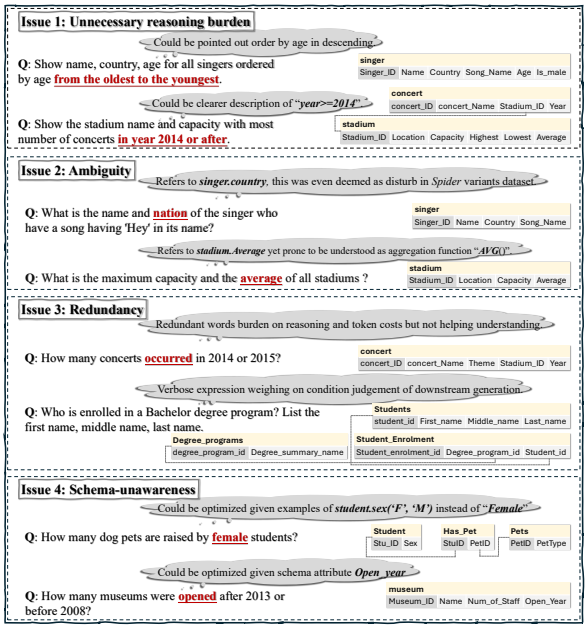

Figure 5: Common NLQ representation issues—unnecessary reasoning burden, ambiguity, redundancy, and schema unawareness—that make grounding and decoding brittle in NL2SQL.

Table 12: Five defining principles of the *Gold Query*.

Principle	Description
Semantic Equivalence	Preserves the original intent and meaning of the NLQ.
Structured Norm	Conforms to a canonical, schema-grounded structured norm.
Schema Awareness	Explicitly identifies all relevant tables, columns, and relationships.
Logical Clarity	States all comparisons and conditions explicitly for unambiguity.
Conciseness	Removes redundancy while keeping the query clear and readable.

A.3.3 SQL Refinement details

Details of SQL Refinement with its generation in QBridge are illustrated in Figure 6.

A.4 Prompts and Checklists

Data Construction for Gold Query Rewriting We provide our explicit Gold Query rewriting prompt as follow. Details are shown in Figure 7.

Rewrite Reviewer Details of rubber-duck-style rewrite reviewer prompt are shown as Figure 9. Details of dimensions in reviewing process are shown as Table 13.

Reverse Deduction Details of reverse deduction prompt of generating reverse query are shown in Figure 10.

SQL Refiner Details of SQL Refiner prompt are shown as Figure 11.

A.5 Experiments Details

Benchmarks We evaluate *QBridge* using five NL2SQL benchmarks/settings: Spider (Yu et al., 2018), BIRD (Li et al., 2023b), Spider-DK (Gan

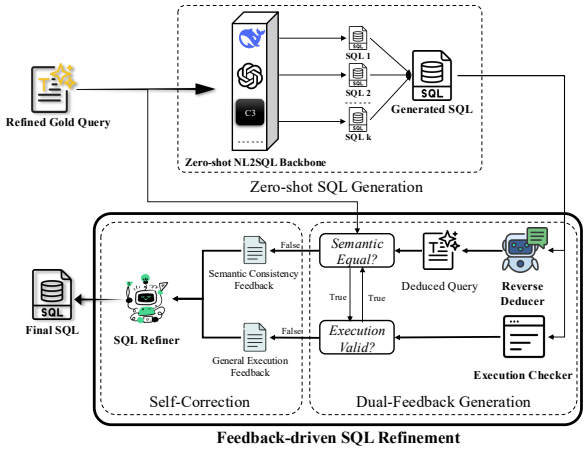

Figure 6: Feedback-driven SQL refinement with a front-end zero-shot SQL generator.

Table 13: Eight standardized dimensions guiding the reviewer's evaluation of the rewrite $\tilde{r}$.

Dimension	Description
Semantic Equivalence	Preserves the intent and meaning of the original NLQ.
Structured Norm	Follows the Gold Query Norm with consistent slot ordering.
Ordering Correctness	Aligns sorting direction and attribute order with user intent.
Grouping Logic	Uses grouping and aggregation only when required.
Join Validity	Ensures joins correspond to valid schema relationships.
Condition Soundness	Specifies complete and coherent filtering and comparison logic.
Schema Coverage	References only valid tables and columns without redundancy.
Conciseness	Removes redundancy while preserving clarity and intent.

et al., 2021b), Spider-SYN (Gan et al., 2021a), and Spider-Realistic (Deng et al., 2021). Benchmark details are summarized in Table 14.

Metrics of Cost Evaluation

- *Average Tokens per Query (Avg. Tokens / Q)*. We measure the realized token usage per query over the full inference pipeline, including all input and output tokens consumed by rewriting, review/refinement, SQL generation, repair, and optional self-consistency sampling.
- *Average Time per Query (Avg. Time / Q)*. We measure the end-to-end wall-clock time from receiving the input question and schema to producing the final SQL output, under a unified deployment setting.
- *Average Cost per Query (Avg. Cost / Q)*. We estimate the actual monetary cost per query from the realized token usage based on the provider pricing used in our experiments.
- *EX / Avg. Cost*. To summarize cost-efficiency, we further report execution accuracy normalized by per-query monetary cost.

ECV-Backbone Coupling As shown in Table 1, the GPT-4o pairing offered slightly lower score not because of model capacity, but the ECV–backbone coupling introduced by our DBT pipeline: during data construction, ECV is performed with a specific backbone $\mathcal{G}_{\text{SQL}}$, and only rewrites that make $\mathcal{G}_{\text{SQL}}$ execute the correct SQL are kept as Gold Queries, implicitly encoding its decoding style into the distilled data. This process not only filters for correctness but also internalizes the SQL generation style and structural preferences of $\mathcal{G}_{\text{SQL}}$ into the distilled dataset, enabling the student rewriter to produce queries naturally aligned with that backbone's decoding behavior. Consequently, when inference-time generation uses the same model as ECV (e.g., DeepSeek-V3), the rewriter operates in-distribution and yields optimal EX, EM, and TS, whereas switching to a different backbone (e.g., GPT-4o) introduces mild stylistic mismatch.

A.6 Qualitative Analysis: Value-Case Noise and Evaluation Artifacts

We follow the official Spider evaluation protocol, and the following test cases illustrate situations where despite appearing to have errors, our approach actually generates more accurate and reasonable SQL queries. These issues are often due to data inconsistencies, misinterpretations in the evaluation process, or inherent limitations of the gold standard.

Value Case Mismatch. In cases where there is a mismatch between the case formatting of literals in the query and the database, our method's ability to handle case normalization leads to more accurate SQL generation compared to the gold standard. For instance, in Case 1 (Status casing) in Figue 12, the gold SQL fails to match because it does not normalize the case of "Stuck", whereas our method correctly normalizes case differences, aligning the query with the database. Similarly, in Case 5 (String literal casing mismatch) in Figure 16, the discrepancy between "Leo Wong" and the database's lowercase value causes the gold SQL to fail, but our system correctly normalizes and matches the value.

Underspecified Queries and Value Mismatches. When a query is underspecified or contains ambiguities, the gold standard SQL may not fully account for the user's intent. In Case 2 (Value casing + target-attribute ambiguity) in Figure 13, the ambiguity of whether to retrieve Document_Object_ID or Related_Document_Object_ID is resolved in our approach by grounding the query to the explicitly

System Prompt: You are now an professional question rewriting specialist. I will provide a natural language question, the corresponding database schema and Gold SQL. Your task is to rewrite the question into a semantically equivalent, structured, yet natural Gold Query format.
Instruction Prompt: CRITICAL RULES:
1. SEMANTIC EQUIVALENCE: The rewritten question must preserve the exact meaning and attribute order of the original question
2. STRUCTURED FORMAT: Follow this exact structure: - "List [aggregation functions] [attributes] from [table(s)] [with join conditions] [with conditions] [grouped by] [ordered by] [limited to]"
3. CONCISENESS: Remove all redundant words while maintaining clarity
4. UNAMBIGUOUS: Explicitly specify aggregations (count, sum, avg, max, min) and comparisons
5. NATURAL LANGUAGE: Maintain natural English flow while using precise database terminology
6. OUTPUT ONLY: Return only the rewritten question, no explanations
[Schema]
{db_schema}
[Question]
{question}
[Gold SQL]
{gold_sql}

Figure 7: Details of NLQ Rewriting Prompt(Green parts are included when constructing training data)

Table 14: Statistics of NL2SQL benchmarks/settings used in this study.

Dataset	Samples	DBs / Domains	Focus
Spider	8659 train / 1034 dev / 2147 test	200 / 138	Large-scale cross-domain benchmark with complex SQL queries over diverse relational schemas.
BIRD	9428 train / 1534 dev / 1789 test	95 / 37	A large-scale, database-grounded benchmark with real-world and often noisy database content, designed to evaluate content-grounded reasoning and SQL efficiency beyond clean schema matching.
Spider-DK	535 (modified from Spider dev)	10 / –	Robustness to implicit domain knowledge beyond explicit schema mentions.
Spider-SYN	1034 (same SQLs as Spider dev)	20 / –	Robustness under synonym-based paraphrasing of schema terms.
Spider-Realistic	508 (revised from Spider dev)	20 / –	Performance under more realistic, less schema-explicit user phrasing.

mentioned field. This discrepancy in interpretation leads to mismatches in the gold standard evaluation, but our method resolves it more effectively by grounding the query to the correct field.

In Case 3 (Whitespace noise in string literals) in Figure 14, the gold SQL does not account for the presence of whitespace around values in the database, leading to empty denotation. Our method, however, handles such discrepancies by trimming whitespace, resulting in a successful match.

Schema Mismatches and Missing Tables. In situations where schema linking fails to recall the correct table or schema element, our method still produces valid SQL queries by grounding the rewrite to the correct logic. In Case 4 (Schema linking false negative + hard pruning) in Figure 15, the failure of schema linking to recall the paintings table leads to an incorrect column (height), but our method's rewrite still captures the user intent, and the issue arises from schema pruning.

Schema Misalignment and SQL Misinterpretation. When the schema misalignment between the gold SQL and our generated SQL occurs, we demonstrate that our method actually aligns more closely with the user's intent. Case 8 (SQL misalignment between cinemas and movies) in Figure 19 shows that while the gold SQL focuses on movie titles, our SQL correctly focuses on cinemas. This misalignment between the gold standard and user intent causes discrepancies in denotation. Adjusting the gold SQL to align with our generated SQL would resolve this issue.

Data Distribution Issues. Sometimes, SQL results are affected by underlying data distribution, and our approach is more adaptable to such issues. In Case 10 (Data distribution issues: Card credit

System
You are now an excellent SQL writer, first I'll give you some tips and examples, and I need you to remember the tips, and do not make same mistakes.

User
Tips 1:
Question: Which A has most number of B?
Gold SQL: select A from B group by A order by count (*) desc limit 1;
Notice that the Gold SQL doesn't select COUNT(*) because the question only wants to know the A and the number should be only used in ORDER BY clause, there are many questions asks in this way, and I need you to remember this in the the following questions.

Assistant
Thank you for the tip! I'll keep in mind that when the question only asks for a certain field, I should not include the COUNT(*) in the SELECT statement, but instead use it in the ORDER BY clause to sort the results based on the count of that field.

User
Tips 2:
Don't use "IN", "OR", "LEFT JOIN" as it might cause extra results, use "INTERSECT" or "EXCEPT" instead, and remember to use "DISTINCT" or "LIMIT" when necessary.
For example,
Question: Who are the A who have been nominated for both B award and C award?
Gold SQL should be: select A from X where award = 'B' intersect select A from X where award = 'C';

Assistant
Thank you for the tip! I'll remember to use "INTERSECT" or "EXCEPT" instead of "IN", "OR", or "LEFT JOIN" when I want to find records that match or don't match across two tables. Additionally, I'll make sure to use "DISTINCT" or "LIMIT" when necessary to avoid repetitive results or limit the number of results returned.

User
You are now a sqlite data analyst, and you are given a database schema as follows:
[Schema]
{db_schema}
[Question]
{question}
Please read and understand the database schema carefully, and generate an executable SQL based on the user's question with no explanations. The generated SQL is protected by "'sql and "'.

Figure 8: Details of Zero-shot SQL Generation Prompt

ranges) in Figure 21, the gold SQL returns only one country, but the data distribution makes our approach return multiple countries, which is logically correct based on the query intent. The gold SQL fails to account for the data distribution correctly, while our method provides more accurate results.

String Matching and Aggregation Errors. Case 9 (String matching and aggregation error) in Figure 20 presents an issue where case sensitivity prevents the query from matching correctly. The gold SQL works under the assumption of lowercase city names, while our method's ability to normalize case resolves the mismatch. This shows our method's flexibility in generating more accurate SQL queries compared to the gold standard.

System Prompt: You are a professional question rewriting quality reviewing specialist. Evaluate a single rewritten question across following dimensions using rubber duck checking.

Instruction Prompt: Dimensions: 1. STRUCTURED FORMAT - Does it follow the structured format for Gold Query?
- Base pattern: "List [aggregation functions] [attributes] from [table(s)] [with join conditions] [with conditions] [grouped by] [ordered by] [limited to]"
2. ORDERING CORRECTNESS - Does the rewritten question specify the correct ordering (e.g., asc/desc) that matches the original question's intent, especially for superlative queries (e.g., "fewest", "most", "highest", "lowest")? Check for misinterpretation of order direction.
3. GROUP BY USAGE - Only use GROUP BY when it is truly necessary; avoid excessive grouping.
4. SET OPERATION LOGIC - Check if the question contains keywords such as "either...or" (should map to UNION), "both...and" (should map to INTERSECT), and "except"/"not in" (should map to EXCEPT or NOT IN depending on context). Ensure the rewritten question correctly reflects the intended set operation or aggregation logic, and matches the original meaning.
5. JOIN CONDITIONS - By default, do not specify explicit join-on columns (including foreign keys); only specify join-on columns when omitting them would cause ambiguity or confusion. Only use join instead of inner join or left join.
6. UNAMBIGUITY - Are table names, column names, and functions clearly specified?
7. DB UTILIZATION - Does the rewritten question utilize the schema effectively but not excessively? Also, pay attention to case sensitivity issues between the schema and the original question.
8. NATURAL LANGUAGE - Does the rewritten question maintain natural English flow while using precise database terminology?

Use rubber duck checking: briefly explain your reasoning per dimension, then provide brief fix suggestions if needed.

Schema: {db_schema}
Original Question: {original_question}
Rewritten Question: {rewritten_question}

Figure 9: Details of Rewrite Reviewer Prompt

System Prompt: You are a SQL analysis expert. Given an NLQ, a SQL query, and the database schema, reverse-deduce a concise Gold Query style that semantically matches the SQL. Assess its consistency with the NLQ, briefly explain any differences, and provide actionable SQL fixes if needed.

Instruction Prompt: **Deduction Rules:** 1. Reverse-deduct a natural language question Gold Query that describes the exact semantics of the SQL, and evaluate whether the reverse-deduced Gold Query is semantically equivalent to the given NLQ.
2. Structured Format: Follow the Gold Query structure:
- "List [aggregation functions] [attributes] from [table(s)] [with join conditions] [with conditions] [grouped by] [ordered by] [limited to]".
3. Unambiguity: mention table/column names and operations explicitly; keep attribute order consistent with the SELECT list.
4. Natural Language: maintain natural English flow while using precise database terminology.

NLQ:{question}
Database Schema:{db_schema}
SQL:{sql}

Return STRICT JSON only, no extra text, with this schema:
```json
{
"reverse_query": "one line natural language Gold Query",
"feedback": {
"consistency_with_given_question": "true or false",
"reasoning": "brief reasoning",
"fixes": ["short, actionable step 1", "step 2", "step 3"]
}
```

Figure 10: Details of Reverse Deduction Prompt

System Prompt: You are a SQL Refiner. Your job is to refine and optimize SQL queries based on execution feedback and semantic consistency feedback.

Instruction Prompt: **Original Question: {original_question}
- Current SQL Query: {current_sql}
- Database Schema: {db_schema}
- Semantic Consistency Feedback (JSON): {semantic_feedback}
- Execution Feedback (JSON): {execution_feedback}

Refine Goals:
1. Fix all issues mentioned in the feedback while preserving the original intent.
2. Optimize SQL correctness based on execution results.
3. Ensure the refined SQL is syntactically correct and executable.
4. Maintain semantic equivalence with the original question.

Output:
- Return ONLY the refined SQL with no explanations.
- The output SQL should be a clean, executable SQL statement and protected by “‘sql and “‘.

Figure 11: Details of SQL Refiner Prompt

Case 1 (Status casing). NLQ: Find all the details of the customers who have been involved in an interaction with status “Stuck” and service and channel detail “bad”.
DB evidence: `SELECT DISTINCT Status_Code FROM Customer_Interactions;` returns `'Stuck'` (not `'stuck'`).
Gold: `... WHERE t2.status_code="stuck" AND ...="bad"` ⇒ `gold_denotation = [ ]`.
Pred: `... WHERE ci.Status_Code='Stuck' AND ...='bad'` ⇒ non-empty denotation.
Diagnosis: With M-Schema value examples and our case canonicalization, QBridge tends to ground literals to the database form, which may disagree with casing-noisy gold values.

Figure 12: Representative test cases where value-case noise in gold SQL yields empty gold denotations.

Case 2 (Value casing + target-attribute ambiguity). NLQ: What are the document object ids of the related to the document owned by Ransom?
Gold: `... SELECT t1.document_object_id ... WHERE t2.owner='ransom'` ⇒ `gold_denotation = [ ]`.
Pred: `... SELECT dsm.Related_Document_Object_ID ... WHERE do.Owner='Ransom'` ⇒ non-empty denotation (possibly with duplicates).
Diagnosis: (i) the gold literal `'ransom'` may mismatch the database casing (e.g., `'Ransom'`), yielding an empty gold denotation; (ii) the NLQ is underspecified on whether to return `Document_Object_ID` or `Related_Document_Object_ID`, while our pipeline tends to ground the intent to the explicitly mentioned “related” field.

Figure 13: Test-set examples where value-case noise and/or underspecified target attributes can lead to empty gold denotations, potentially under-estimating database-grounded pipelines.

Case 3 (Whitespace noise in string literals). NLQ: What are the voice sound quality scores received for the song named `' The Balkan Girls '` in English language?
Gold: `... WHERE T2.name=' The Balkan Girls ' AND T2.language='English'` ⇒ non-empty denotation.
Pred: `... WHERE s.name='The Balkan Girls' AND s.language='English'` ⇒ empty denotation.
Diagnosis: the database appears to store the song name with leading/trailing spaces. Our pipeline normalizes user mentions by trimming surrounding whitespace, which is often closer to user intent but can miss matches when the underlying DB contains whitespace-corrupted values.

Figure 14: Test-set example where whitespace-corrupted string values cause execution-based evaluation to disagree under different literal normalization behaviors.

Case 4 (Schema linking false negative + hard pruning). NLQ: Find the ids of paintings whose height is bigger than 500 and less than 2000?
Schema linker output: only `artists(. . . )` is recalled; `paintings` is missing.
Rewrite ($\vec{r}$): `List the painting IDs with condition of height is between 500 and 2000.`
Pred SQL: `SELECT paintingID FROM paintings WHERE height>500 AND height<2000;` ⇒ `no such column: height`.
Gold SQL: `... WHERE height_mm>500 AND height_mm<2000`.
Diagnosis: the rewrite correctly captures the clause-level intent, but hard-pruned schema prevents the SQL generator/refiner from observing the true column (`height_mm`), leading to a plausible but invalid guess (`height`).

Figure 15: Example where schema linking false negatives and hard schema pruning cause execution failure, despite a faithful structured rewrite.

Case 5 (String literal casing mismatch). NLQ: List all package numbers received by Leo Wong?
Gold: `... WHERE t2.name='leo wong'` ⇒ `gold_denotation = [ ]`.
Pred: `... WHERE Client.Name='Leo Wong'` ⇒ `pred_denotation = [ ]`.
Diagnosis: The database stores the name in lowercase, which causes mismatch when the case-sensitive value is queried. The Gold SQL correctly matches the lowercase value, while Pred SQL uses an uppercase string that does not match.

Figure 16: Test-set example where string literal case mismatch leads to empty denotation.

Case 6 (String matching difference: LIKE "John" vs LIKE "%John%"). NLQ: Find the details of the customers named John.
Gold: `... WHERE t2.name LIKE "\%John\%"` ⇒ non-empty denotation.
Pred: `... WHERE Client.Name LIKE "John"` ⇒ empty denotation.
Diagnosis: The Pred SQL uses `LIKE "John"` which only matches the exact value `'John'` (case-sensitive), whereas the Gold SQL uses `LIKE "%John%"` to capture all cases where John appears as part of the name. This mismatch leads to an empty result in Pred SQL.

Figure 17: Test-set example where a difference in 'LIKE' patterns (exact match vs partial match) leads to mismatch between gold and pred denotations.

Case 7 (Gold SQL misinterpretation of "unrated movies"). NLQ: What is all the information about the unrated movies?
Gold: `... WHERE rating = 'null'` ⇒ gold_denotation = [ ].
Pred: `... WHERE Rating IS NULL OR Rating = ''` ⇒ non-empty denotation [(4, 'The Quiet Man', None), (5, 'North by Northwest', None), (8, 'A Night at the Opera', None)].
Diagnosis: Gold SQL uses `rating = 'null'` to query for unrated movies, which is incorrect as `NULL` should be used to represent unrated movies. The Pred SQL correctly uses `Rating IS NULL` to capture unrated movies, but also includes `Rating = ''` which could return empty string entries. Nevertheless, the Pred SQL's logic better reflects SQL norms and user intent.

Figure 18: Test-set example where the Pred SQL's logic aligns better with SQL standards, whereas the Gold SQL's use of ''null'' leads to errors in matching unrated movies.

Case 8 (SQL misalignment between cinemas and movies). NLQ: Find the name of the cinemas that are playing movies with either rating 'G' or rating 'PG'.
Gold: `SELECT title FROM movies WHERE rating = 'G' OR rating = 'PG'` ⇒ non-empty denotation.
Pred: `SELECT DISTINCT MovieTheaters.Name FROM MovieTheaters JOIN Movies ON MovieTheaters.Movie = Movies.Code WHERE Movies.Rating IN ('G', 'PG');` ⇒ empty denotation.
Diagnosis: The Pred SQL correctly joins the tables `MovieTheaters` and `Movies`, but mistakenly focuses on the `MovieTheaters.Name` rather than the movie title. The Gold SQL queries the movie titles without considering the theater information, which misaligns with the user's intent of finding cinemas. The Pred SQL's focus on theaters is semantically aligned with the NLQ, but its target and table selection differ from the Gold SQL's aim. The correct query should involve finding cinemas that are playing 'G' or 'PG' rated movies, and should return theater names. Adjusting the Gold SQL to focus on cinemas would resolve this misalignment.

Figure 19: Test-set example where misalignment between Pred SQL's focus on theaters and Gold SQL's focus on movies leads to empty denotation.

Case 9 (String matching and aggregation error). NLQ: What are the total enrollment of institutions in city "Vancouver" or "Calgary"?
Gold: `select sum(enrollment) from institution where city = "vancouver" or city = "calgary"` ⇒ non-empty denotation.
Pred: `SELECT SUM(Enrollment) FROM institution WHERE City = 'Vancouver' OR City = 'Calgary';` ⇒ empty denotation.
Diagnosis: The Pred SQL fails to match the cities because the query is case-sensitive, while Gold SQL assumes that cities are stored in lowercase in the database, resulting in a successful match. Additionally, Pred SQL is trying to calculate the total enrollment, but the case mismatch causes failure.

Figure 20: Test-set example where case sensitivity leads to SQL failure in Pred SQL, while Gold SQL works due to the assumption of lowercase city names.

Case 10 (Data distribution issues: Card credit ranges). NLQ: Which nations have both customers with card credit above 50 and customers with card credit below 75?
Gold: `SELECT Nationality FROM customer WHERE Card_Credit < 50 INTERSECT SELECT Nationality FROM customer WHERE Card_Credit > 75` ⇒ non-empty denotation {Australia}.
Pred: `SELECT DISTINCT c1.Nationality FROM customer c1 WHERE c1.Card_Credit > 50 INTERSECT SELECT DISTINCT c2.Nationality FROM customer c2 WHERE c2.Card_Credit < 75;` ⇒ non-empty denotation {Australia, England}.
Diagnosis: The Pred SQL queries customers whose card credit is between 50 and 75. Given the data distribution, this query is logically correct. However, since the data does not match the Gold SQL condition of having card credits both below 50 and above 75, the Pred SQL returned more than one nation.

Figure 21: Test-set example where data distribution issues affect SQL results, causing Pred SQL to return multiple countries while Gold SQL returns only one.

Case 11 (Self-Connection Distance Issue: Excluding City-to-Self Distances). NLQ: What is the total distance between all cities?
Gold: `SELECT T2.city_name , avg(distance) FROM Direct_distance AS T1 JOIN City AS T2 ON T1.city1_code = T2.city_code GROUP BY T1.city1_code` ⇒ non-empty denotation.
Pred: `SELECT c.city_name, AVG(d.distance) AS average_distance FROM City c JOIN Direct_distance d ON c.city_code = d.city1_code WHERE d.city1_code != d.city2_code GROUP BY d.city1_code;` ⇒ incorrect denotation.
Diagnosis: The Pred SQL attempts to calculate average distances for cities while excluding self-references using 'd.city1_code != d.city2_code'. However, this fails to accurately account for the correct relationships and data filtering in the schema. The Gold SQL lacks the 'd.city1_code != d.city2_code' clause, but it would be beneficial to explicitly exclude self-distances in both cases.

Figure 22: Test-set example where self-connection distances affect the computation of total distance between cities, leading to incorrect denotation.