

VCoT-SQL: Enhancing Text-to-SQL Generation through Dual Chain-of-Thought Validation Training

Rui Zhang¹, Yujia Su¹, Ruixuan Li¹, Yuhua Li¹, Yixin Su^{1*}

¹ School of Computer Science and Technology,
Huazhong University of Science and Technology, Wuhan, China
rayteam@yeah.net (www.ruizhang.info), suyixin@hust.edu.cn

Abstract

Open-source LLMs provide a flexible alternative to closed-source models for Text-to-SQL but still struggle with complex queries. Fine-tuning with chain-of-thought (CoT) data improves reasoning, yet standard datasets lack CoT annotations, requiring synthetic generation. Existing synthesis methods only verify the final answer, allowing flawed reasoning steps to corrupt the training data. We propose VCoT-SQL, a framework that improves Text-to-SQL generation through intermediate-step-validated CoT supervision and a dedicated SQL validator, which enables small open-source models to outperform frontier closed-source models. VCoT-SQL trains two complementary models on curated reasoning traces: an SQL generator using clause-level CoT templates where reasoning steps are verified via execution and schema alignment checks, and an SQL validator using diagnostic CoT templates to detect syntactic, semantic, and type-level errors. We further design a hierarchical schema filtering pipeline that integrates and refines complementary retrieval strategies to distill relevant database elements. VCoT-SQL achieves 66.82% execution accuracy on challenging dataset BIRD dev and 86.8% on Spider test, improving the base 7B model by 7% on BIRD. Notably, our CoT synthesis relies on 7B models yet outperforms methods that use 72B-scale or even proprietary closed-source teachers for data generation, demonstrating that rigorous step-level validation can be more effective than scaling up teacher model size. Our dataset and code are available at <https://github.com/INTREBID/VCoT-SQL>.

1 Introduction

Text-to-SQL is the task of converting natural language questions into executable SQL queries, enabling non-technical users to access and manipulate relational databases. As a fundamental problem at

the intersection of natural language processing and database systems, Text-to-SQL attracts significant attention from both the academia and industry.

LLM-based approaches dominate Text-to-SQL benchmarks, falling into two paradigms: prompt-based and fine-tuned. Prompt-based approaches primarily leverage closed-source LLMs such as GPT-4 (OpenAI, 2023), which remain general-purpose models that are hard to fine-tune for Text-to-SQL, and suffer from high API costs and data privacy risks (Li et al., 2024; Pourreza and Rafiei, 2024; Qin et al., 2025). Fine-tuning approaches train open-source LLMs on task-specific Text-to-SQL datasets, and recent work incorporates Chain-of-Thought (CoT) (Wei et al., 2022) reasoning into the fine-tuning objective to strengthen reasoning capabilities (Pourreza et al., 2025; Zhai et al., 2025; Li et al., 2025). However, existing benchmarks typically provide only question–SQL pairs, and the scarcity of high-quality CoT annotations limits the effectiveness of such methods, particularly on complex queries.

Meanwhile, pipeline-based strategies are proposed to improve Text-to-SQL accuracy by augmenting the generator with schema linking (Li et al., 2024; Pourreza and Rafiei, 2024; Qin et al., 2025; Liu et al., 2026) or Self-Correction (Pourreza and Rafiei, 2023; Liu et al., 2026; Talaei et al., 2024). However, existing schema linking methods either require additional model training or rely on superficial semantic matching that struggles with complex schema correspondences. Moreover, self-correction approaches remain largely ineffective to semantic errors, as they focus on fixing syntax errors via execution feedback.

To address the lack of high-quality reasoning data and further improve the accuracy of generated SQL, we introduce **VCoT-SQL**, a Text-to-SQL framework built on the principle of dual validation—validating CoT quality during training and validating SQL correctness during inference.

*Corresponding author.

On the training side, we design dedicated CoT templates for SQL generation and SQL validation respectively, and then apply multi-stage verification (execution consistency, schema grounding, and reasoning faithfulness) to curate reliable training data for a CoT-enhanced Generator and Validator. During inference, a cascaded schema filtering pipeline first isolates relevant database elements to provide focused context; the Generator then produces candidate SQL, which the Validator diagnoses and refines by correcting syntactic errors, semantic inconsistencies, and data-type mismatches. **This design enables smaller models to achieve strong reasoning and self-correction capabilities with less data, without relying on large teacher models.**

Our contributions are as follows:

1) We propose VCoT-SQL, a dual-validation Text-to-SQL framework with a CoT-enhanced Generator and Validator. To facilitate effective reasoning, we design dedicated CoT templates: clause-driven templates for generation and multi-type correction templates for validation, and then curate high-quality training data from Spider/BIRD via multi-stage filtering on execution consistency, schema grounding, and reasoning faithfulness.

2) We also introduce a training-free, three-stage schema retrieval framework (Pre-SQL, Column, and Value retrievers), which progressively reduces schema noise, achieving performance superior to that of trained specialized schema filters.

3) Extensive experiments on six benchmarks validate that our method significantly improves the performance on three different base models. Our best 7B model increases execution accuracy on the BIRD dev set from 59.78% to 66.82%. Furthermore, it surpasses the strong OmniSQL baseline (63.9%) while using only **1/250** of its training data.

2 Related Work

2.1 Text-to-SQL

Early Text-to-SQL systems rely on rule-based methods (Woods et al., 1972; Li and Jagadish, 2014), which struggle with complex and cross-domain queries. Large-scale benchmarks such as WikiSQL (Zhong et al., 2017) and Spider (Yu et al., 2018b) have spurred neural approaches (Zhong et al., 2017; Yu et al., 2018a; Guo et al., 2019), yet these methods still struggle to generalize to unseen schemas and complex query intents.

Pretrained language models such as BERT (Devlin et al., 2019) and T5 (Raffel et al., 2020) bring

contextual representations to Text-to-SQL (Hwang et al., 2019; Qi et al., 2022; Li et al., 2023a,b). With the rise of large language models (LLMs), current approaches broadly fall into two paradigms. Prompt-based methods leverage powerful closed-source LLMs via in-context learning (Pourreza and Rafiei, 2023; Gao et al., 2024; Pourreza et al., 2024; Talaei et al., 2024; Lee et al., 2025), but incur high inference costs, while also precluding task-specific fine-tuning. Conversely, fine-tuning strategies tailor LLMs to the task, aiming to bolster structural robustness and execution fidelity (Pourreza and Rafiei, 2024; Yang et al., 2024; Li et al., 2024). However, constrained by their limited model capacity, these approaches often exhibit insufficient reasoning ability on complex queries.

Schema linking in Text-to-SQL is a type of retrieval-augmented generation, as it retrieves relevant schema elements to guide SQL generation. For a comprehensive survey on multimodal retrieval-augmented generation including the modality of tables and schemas, please refer to Zhang et al. (2025). However, existing schema linking methods typically rely on a single strategy, such as model training (Pourreza and Rafiei, 2024; Li et al., 2024; Qin et al., 2025), embedding matching (Liu et al., 2026), or LLM-based extraction (Lee et al., 2025). Meanwhile, existing self-correction approaches rely either on prompt-based methods (Pourreza and Rafiei, 2023) or on execution-error-driven correctors without CoT-enhanced training (Liu et al., 2026; Qin et al., 2025).

2.2 Chain-of-Thought Fine-tuning

Chain-of-Thought (CoT) reasoning has proven effective for complex tasks. Recent models such as OpenAI o1 (OpenAI, 2024d) and DeepSeek-R1 (DeepSeek-AI, 2025) internalize it through specialized training objectives and reinforcement learning. A key challenge in CoT fine-tuning is acquiring high-quality reasoning traces. Since manual annotation is costly to scale (Chung et al., 2024), subsequent work synthesizes CoT data via LLMs with filtering heuristics (Zelikman et al., 2022; Kim et al., 2023). For Text-to-SQL, OmniSQL (Li et al., 2025) collects a large-scale CoT-enriched dataset via free-form prompting. However, such approaches (Pourreza et al., 2025; Zhai et al., 2025; Li et al., 2025) depend on the capabilities of ultra-large models and rely solely on execution correctness as the filter, without validating the reliability of intermediate reasoning steps.

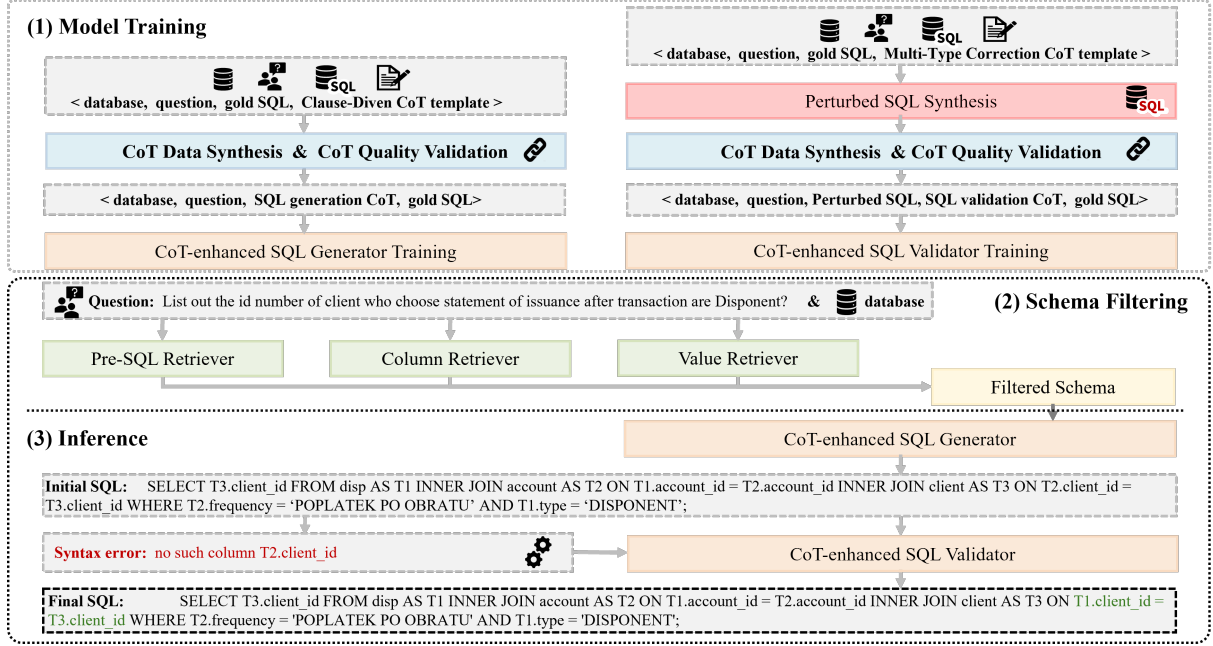


Figure 1: Overview of the proposed VCoT-SQL framework. (1) **Model Training**: two pipelines collect CoT supervision data to train the CoT-enhanced SQL Generator and Validator. (2) **Schema Filtering**: the question and full database are processed through Pre-SQL, Column, and Value Retrieval to obtain a filtered schema (see Figure 2). (3) **Inference**: the Generator produces an initial SQL from the filtered schema; the SQL is executed, and any syntax errors along with the initial SQL are passed to the Validator to yield the final SQL.

3 Methodology

As illustrated in Figure 1, our VCoT-SQL can be divided into three stages. In the model training stage, we synthesize and validate CoT data based on the designed clause-driven CoT template to train the CoT-enhanced SQL Generator (§3.1); similarly, we train a CoT-enhanced SQL Validator based on the multi-perspective correction CoT (§3.2). In the schema filtering stage, complementary Pre-SQL Retriever, Column Retriever, and Value Retriever are applied to progressively simplify the schema. In the inference stage, the SQL Generator produces a candidate SQL based on the filtered schema, which is then fed into the SQL Validator for multi-faceted correction to yield the final SQL (§3.3).

3.1 CoT-enhanced SQL Generator Training

Clause-Driven CoT To guide structured reasoning for SQL generation, we design a clause-driven CoT template that mirrors the inherent structure of SQL, decomposing the task into clause-level reasoning steps, including the SELECT, WHERE, FROM, and other clauses. Each clause is further divided into fine-grained steps that specify how to identify relevant columns, infer filtering conditions, determine join relationships, and construct aggregations. Interestingly, although SQL queries

typically follow the SELECT-FROM-WHERE order, we find that reasoning about the FROM clause only after analyzing the others produces higher-quality CoT and more accurate SQL. This reordering allows the model to select tables only after identifying the required columns, conditions, and aggregations, thereby reducing spurious joins and unnecessary tables. Overall, this structured CoT format aligns with the hierarchical logic of SQL while enabling reliable programmatic verification of reasoning quality.

CoT Data Synthesis & CoT Quality Validation Based on the clause-structured template, we construct CoT data using few-shot strategy. We provide three high-quality CoT exemplars, manually crafted from samples in Spider and BIRD, as references for the LLM. For each training instance, we also provide its gold SQL to improve generation accuracy. However, since a small portion of the original dataset contains mismatches (Pourreza et al., 2025) between questions and gold SQL, we instruct the LLM to first verify whether the provided SQL is correct and then generate structured CoT reasoning steps followed by the final SQL output. The complete CoT template and the prompt for CoT generation are presented in Appendix E.1.

Although providing the gold SQL upfront during

CoT synthesis can alleviate part of the generation burden, we observe that this sometimes leads to cases where the final SQL is correct yet the intermediate reasoning steps are unreliable or unsupported. To address this, we introduce a multi-stage verification mechanism evaluating both execution consistency and schema grounding. We first filter out CoT data that violates the expected clause structure or exceeds a length threshold. Next, we check whether the SQL derived from the CoT executes to the same result as the gold SQL. Finally, and most critically, we validate the CoT reasoning itself. We observe that providing gold SQL during generation can cause answer leakage, where the reasoning references phrases such as 'Gold SQL'; such instances are removed. We then extract schema elements from the gold SQL and verify via heuristic rule-based matching that they are appropriately referenced in the CoT, ensuring the reasoning correctly grounds on the relevant schema.

Generator Training We fine-tune the SQL generator via supervised learning. The loss function is defined as follows:

$$\mathcal{L}_{gen} = -\mathbb{E}_{(x,y) \sim D_{gen}} \left[\sum_t \log P_{\theta_{gen}}(y_t \mid x, y_{<t}) \right] \quad (1)$$

where D_{gen} represents the training set for the generator. The input sequence x consists of the user question and the database schema. The output sequence y comprises the validated clause-driven reasoning followed by the generated SQL. The generator is optimized to predict the CoT solution and the corresponding SQL based on the provided inputs.

3.2 CoT-enhanced SQL Validator Training

Multi-Type Correction CoT We adopt a structured CoT-driven correction framework that guides the LLM to diagnose and repair common SQL error types (Shen et al., 2025; Xu et al., 2025). For syntactic correction, execution-time errors from the database engine are provided as diagnostic evidence. For semantic correction, the model evaluates table-column usage, filtering conditions, and correctness of GROUP BY/ORDER BY/LIMIT clauses. We further incorporate data-type verification, guiding the model to check for issues such as proper numeric casting to preserve precision and correct date formatting. Through this multi-type reasoning process, the LLM incrementally refines its prediction into a query that is both executable and semantically faithful to the question intent.

Perturbed SQL Synthesis To construct training data for SQL validation, we need imperfect SQL queries, as original training sets contain only gold SQL. We generate errors by prompting an LLM to systematically perturb the gold SQL, using exclusively those passing CoT validation during the SQL generation stage to largely ensure semantic alignment with the question. To align synthesized errors with the types handled by our validation template, we organize common SQL mistakes into several categories: column-related, table-related, condition-related, aggregation-related and data-type misuses such as improper numerical casting or incorrect date handling. For each category, we design a concise instruction that prompts the LLM to rewrite the SQL with the targeted defect.

CoT Data Synthesis & CoT Quality Validation After obtaining erroneous-gold SQL pairs, we synthesize validation-oriented CoT following the same procedure as in the SQL generation setting. A small set of handcrafted, high-quality CoT exemplars from real SQL errors serves as few-shot demonstrations. More detailed exemplar design and the complete CoT template for SQL validation are presented in Appendix C.1 and Appendix E.2.

For the correction CoT, we first filter out samples that violate the expected format or exceed a length threshold, and remove those explicitly mentioning "Gold SQL". We then check execution consistency between the final corrected SQL in the CoT and the gold SQL, discarding mismatched instances. Finally, given the complex yet well-structured nature of the correction reasoning, we validate its faithfulness by extracting only the analysis and correction steps from the CoT, feeding them to the LLM, and verifying that the regenerated corrected SQL is consistent with the gold SQL. This ensures the reasoning process itself is reliable, not merely producing correct results by coincidence.

Validator Training Following the similar paradigm as Generator, we leverage high-quality SQL-validation CoT data to train the validator. The loss function is defined as follows:

$$\mathcal{L}_{val} = -\mathbb{E}_{(x,y) \sim D_{val}} \left[\sum_t \log P_{\theta_{val}}(y_t \mid x, y_{<t}) \right] \quad (2)$$

where D_{val} represents the training set. The input sequence x consists of the question, database schema, generated SQL and syntax error (null is correct). The output sequence y comprises the validated correction reasoning followed by the validated SQL.

3.3 Schema Filtering and Inference

Large-scale databases often contain redundant schema elements that introduce noise. Prior methods such as CodeS (Li et al., 2024) address this by training dedicated schema filter models. We propose a training-free schema retrieval framework (Pre-SQL Retriever, Column Retriever, and Value Retriever) that jointly leverages LLM’s code-generation capability and lightweight semantic encoding to progressively prune schema.

Pre-SQL Retriever The first retrieval stage leverages an LLM to generate a coarse-grained Pre-SQL based on the question and complete schema containing only table names, column names, and types. We parse the Pre-SQL to extract tables, columns and literal values, which also provide valuable cues for schema filtering.

Column Retriever To mitigate omissions from the Pre-SQL, we further apply an embedding-based column retrieval step. We encode both the full question and LLM-extracted fine-grained keywords into semantic embeddings, and match them against all database column representations to retrieve the top-K candidates. The full-question embedding captures global context, while keyword embeddings provide complementary fine-grained signals.

Value Retriever Some user queries implicitly reference database values without mentioning corresponding columns. Inspired by CHESS (Talaie et al., 2024), the Value Retriever employs a three-stage filtering pipeline—LSH indexing, semantic similarity, and edit distance—to identify hidden dependencies. We query the value index using literals from the Pre-SQL and LLM-extracted keywords.

Inference Finally, we take the union of all columns retrieved from the three paths and serialize the filtered schema into a compact representation enriched with full table names, nullable indicators, and matched values. During inference, the Generator first produces an initial SQL query, which is executed to collect any syntax errors (empty if correct). The Validator then takes the query and error feedback to perform syntactic, semantic, and data-type diagnostics, producing the refined output. Further details are in Appendix A.

4 Experiments

4.1 Experiments Setting

Evaluation Dataset We evaluate VCoT-SQL on two widely used benchmarks, Spider (Yu et al., 2018b) and BIRD (Li et al., 2023c), along with

three additional datasets, Spider-Syn (Gan et al., 2021a), Spider-Realistic (Deng et al., 2021) and Spider-DK (Gan et al., 2021b). We use Execution Accuracy (EX) (Yu et al., 2018b) as our evaluation metric with the official scripts to ensure fair comparisons with prior work.

Baselines Given that VCoT-SQL is a fine-tuned open-source model, we primarily compare it with other open-source models (Qwen2.5 (Qwen Team, 2024), Qwen2.5-Coder (Hui et al., 2024), DeepSeek-Coder (Guo et al., 2024), XiYan-SQL (Liu et al., 2026), Llama3 (Touvron et al., 2023)), particularly those fine-tuned for Text-to-SQL like CodeS (Li et al., 2024), ROUTE (Qin et al., 2025), OmniSQL (Li et al., 2025). Additionally, to position our work within the broader landscape, we compare against top-performing closed-source models (GPT-4o-mini (OpenAI, 2024b), GPT-4-Turbo (OpenAI, 2024a), GPT-4o (OpenAI, 2024c)). Unless otherwise specified, our Schema Filter is adopted as the default input preprocessing step for all prompting-based and our methods to reduce input length and ensure consistent evaluation. More details can be found in Appendix B.

4.2 Main Results

BIRD Results BIRD is challenging due to its complex schemas and SQL structures. As shown in Table 1, VCoT-SQL achieves state-of-the-art performance among models of comparable size, reaching 66.82% EX. With our dual-module architecture, VCoT-SQL generates high-quality reasoning and corrects potential errors, surpassing even GPT-4-turbo (62.06%) and GPT-4o (61.93%).

Notably, both VCoT-SQL and OmniSQL (Li et al., 2025) excel beyond direct-tuning methods e.g. ROUTE (Qin et al., 2025), CodeS (Li et al., 2024), showing the importance of explicit reasoning for complex SQL generation. Compared with OmniSQL, VCoT-SQL achieves a 2.92% improvement using only 10K validated CoT samples, and even our single-module version (65.38%) surpasses OmniSQL, demonstrating the effectiveness of our multi-stage CoT validation.

Spider and Its Variants Results We further evaluate VCoT-SQL on Spider and its robustness variants (Spider-DK, Spider-Syn, Spider-Realistic). As shown in Table 1, VCoT-SQL consistently improves over its XiYanSQL-QwenCoder-7B backbone across all benchmarks: 88.0% vs. 86.9% on Spider Dev, 86.8% vs. 84.7% on Test, and similar gains on robustness variants. OmniSQL achieves

Method	Model	Training Data Size	Spider Dev	Spider Test	BIRD Dev	Spider-DK	Spider-Syn	Spider-Realistic
Prompting with Closed-Source LLMs								
GPT-4o (OpenAI, 2024c)	-	-	81.9	83.0	61.93	70.8	74.2	82.9
GPT-4o-mini (OpenAI, 2024b)	-	-	82.9	82.8	60.43	73.6	74.2	81.7
GPT-4-turbo (OpenAI, 2024c)	-	-	83.8	85.2	62.06	71.8	77.5	83.2
Prompting with Open-Source LLMs								
Llama3 (Touvron et al., 2023)	Llama3-8B	-	69.3	69.1	39.05	57.2	62.6	72.8
DeepSeek-Coder (Guo et al., 2024)	DeepSeek-Coder-6.7B-Instruct	-	72.4	72.3	49.54	62.4	61.8	68.3
Qwen2.5 (Qwen Team, 2024)	Qwen2.5-7B-Instruct	-	77.9	78.2	53.98	68.0	67.7	76.0
Qwen2.5-Coder (Hui et al., 2024)	Qwen2.5-Coder-7B-Instruct	-	82.3	81.0	58.87	70.9	71.3	80.9
XiYanSQL (Liu et al., 2026)	XiYanSQL-QwenCoder-7B	-	86.9	84.7	59.78	73.1	75.7	77.6
Fine-Tuning with Open-Source LLMs								
CodeS (Li et al., 2024)	StarCoder 7B	18K	85.4	83.3	57.17	66.9	73.8	81.5
ROUTE (Qin et al., 2025)	Llama3-8B	18K	86.0	83.9	57.30	74.6	77.4	80.9
OmniSQL (Li et al., 2025)	Qwen2.5-Coder-7B-Instruct	2.5M	86.2	87.9	63.90	76.1	77.3	81.9
Ours: SQL Generator	XiYanSQL-QwenCoder-7B	9K	<u>87.2</u>	86.4	<u>65.38</u>	<u>76.6</u>	<u>80.0</u>	<u>84.3</u>
Ours: VCoT-SQL	XiYanSQL-QwenCoder-7B	10K	88.0	<u>86.8</u>	66.82	76.8	80.4	85.4

Table 1: Performance of Text-to-SQL Models and Methods. To ensure fairness in comparison, we used the results from the greedy decoding released officially by OmniSQL. SQL Generator denotes single-pass generation; VCoT-SQL further applies our SQL Validator. For Training Data Size, prompting-based methods require no Text-to-SQL data (marked as "-"). Among fine-tuning methods, 18K refers to the combined Spider and BIRD training sets, 2.5M to OmniSQL’s SynSQL-2.5M, and our 9K/10K to CoT data constructed from the Spider and BIRD training splits.

higher EX on Spider Test (87.9%), likely benefiting from its larger 2.5M-sample SynSQL training corpus. However, VCoT-SQL demonstrates superior data efficiency, achieving competitive performance with only 10K validated samples.

We further evaluate robustness on all 17 perturbation test sets of Dr.Spider (Chang et al., 2023). The complete results and category-level analysis are reported in Appendix D.3.

4.3 Ablations on SQL Validation

To assess our SQL Validator and the necessity of task-specific fine-tuning, we conduct an ablation study across three base models. As shown in Table 2, we compare standalone generation against pipelines using either a base or our fine-tuned model for validation. Interestingly, simply appending a base model to verify generated SQL yields highly unstable and often detrimental results. For instance, using base DeepSeek-Coder leads

to a 4.57% absolute decline on BIRD Dev, with similar drops observed on other datasets. Further analysis reveals clear model-dependent behavior: compared with other backbones, DeepSeek-Coder tends to regenerate SQL from scratch rather than preserving and correcting existing queries, introducing more false-positive edits and degrading performance. This indicates that without task-specific alignment, raw LLMs lack the constrained SQL debugging capabilities.

In contrast, our fine-tuned validator consistently improves execution accuracy across all backbones and benchmarks. Trained on validated CoT data, the model acquires effective error detection and correction capabilities. On BIRD Dev, the SFT validator boosts Qwen2.5-Coder by 1.63%, with similar gains on other baselines. Among initially incorrect SQLs, the Validator successfully corrects 1.83% of cases, while among already correct queries, only 0.20% are incorrectly altered. These results demon-

Base Model	Spider Dev		Spider Test		BIRD Dev		Spider-DK		Spider-Syn		Spider-Realistic	
	EX	Δ	EX	Δ	EX	Δ	EX	Δ	EX	Δ	EX	Δ
Only SQL Generator												
DeepSeek-Coder-6.7B-Instruct	84.9	-	86.9	-	60.63	-	74.6	-	77.5	-	80.1	-
Qwen2.5-Coder-7B-Instruct	85.3	-	86.7	-	64.86	-	76.8	-	77.6	-	82.7	-
XiYanSQL-QwenCoder-7B	87.2	-	86.4	-	65.38	-	76.6	-	80.0	-	84.3	-
w/ Base Model Validation												
DeepSeek-Coder-6.7B-Instruct	80.1	-4.8	80.3	-6.6	56.06	-4.57	69.2	-5.4	69.5	-8.0	75.6	-4.5
Qwen2.5-Coder-7B-Instruct	85.5	+0.2	86.9	+0.2	65.91	+1.05	75.9	-0.9	77.8	+0.2	82.9	+0.8
XiYanSQL-QwenCoder-7B	87.7	+0.5	86.7	+0.3	65.71	+0.33	76.6	-	80.5	+0.5	84.6	+0.3
w/ SQL Validator												
DeepSeek-Coder-6.7B-Instruct	85.1	+0.2	87.1	+0.2	61.41	+0.78	75.1	+0.5	77.6	+0.1	80.3	+0.2
Qwen2.5-Coder-7B-Instruct	85.7	+0.4	86.9	+0.2	66.49	+1.63	77.2	+0.4	78.5	+0.9	83.3	+0.6
XiYanSQL-QwenCoder-7B	88.0	+0.8	86.8	+0.4	66.82	+1.44	76.8	+0.2	80.4	+0.4	85.4	+1.1

Table 2: Ablation study on SQL validation strategies across three backbones. EX denotes execution accuracy (%) and Δ indicates absolute improvement over the generation-only baseline.

Method	Spider Dev	Spider Test	BIRD Dev	Spider-DK	Spider-Syn	Spider-Realistic
Base Model (Qwen2.5-Coder-7B-Instruct)	82.3	81.0	58.87	70.9	71.3	80.9
FT w/ original Spider and BIRD	82.6	83.1	60.04	73.6	74.5	79.5
FT w/ unvalidated CoT-enhanced (Syntax-Order) Spider and BIRD	83.8	84.1	60.69	73.3	74.9	80.0
FT w/ unvalidated CoT-enhanced (Logic-Order) Spider and BIRD	84.4	84.9	61.60	73.5	75.0	79.9
FT w/ validated CoT-enhanced (Syntax-Order) Spider and BIRD	85.0	86.8	62.58	76.3	77.2	82.3
FT w/ validated CoT-enhanced (Logic-Order) Spider and BIRD	85.3	86.7	64.86	76.8	77.6	82.7

Table 3: Ablation Results for Validated SQL Generation on different data construction settings (EX).

strate our validator’s reliability and effectiveness. Further analyses demonstrating the stability of the validator are presented in Appendix C.

We also compare our fine-tuned 7B Validator with GPT-5.4 under the same generator outputs and evaluation protocol. We treat this frontier model only as an empirical upper-bound reference; results are provided in Appendix D.2.

4.4 Ablations on Validated SQL Generation

To evaluate the effectiveness of our CoT quality validation strategy for SQL generation, we conduct ablation experiments using the same backbone under five training data settings: original Spider and BIRD; unvalidated and validated CoT data generated with the syntax-order template (§3.1); unvalidated and validated CoT data generated with our logic-order template (used in the final model).

As shown in Table 3, training with synthesized CoT data already improves performance over the original training data, increasing accuracy from 58.87% to 61.60% on BIRD Dev and from 81.0% to 84.9% on Spider Test, demonstrating the general

benefit of CoT for SQL generation. Applying our CoT quality validation boosts performance by filtering flawed reasoning. Among all settings, validated logic-order CoT achieves the best results, reaching 64.86% on BIRD Dev. This represents a consistent improvement of 2–6% over the base model fine-tuned directly on original datasets, highlighting the broad effectiveness of our structured reasoning supervision. Compared with syntax-order, the logic-order performs more consistently, suggesting that organizing reasoning according to query planning is more effective than SQL syntax order. To further disentangle the effects of template ordering and CoT validation, we conduct a 2×2 factorial analysis over the four corresponding settings. Detailed results are provided in Appendix D.1.

4.5 Ablations on Schema Filter

We evaluate our three-path schema filter against the full schema and the trained schema filter from CodeS. As shown in Table 4, our method improves retrieval quality across all datasets, increasing Precision while maintaining near-complete Recall. For

Method	Spider Dev			Spider Test			BIRD Dev		
	Precision	Recall	EX	Precision	Recall	EX	Precision	Recall	EX
Full schema	-	-	83.6	-	-	86.6	-	-	59.71
CodeS schema filter	18.02	99.83	85.4	20.22	99.87	85.1	16.44	97.76	62.63
VCoT-SQL schema filter	30.85	99.79	85.3	34.51	99.89	86.7	31.36	98.09	64.86
w/o Pre-SQL retriever	30.77	98.94	83.8	33.92	97.74	84.4	31.39	94.91	61.15
w/o column retriever	67.35	97.02	83.6	70.02	96.26	84.5	65.48	92.30	61.08
w/o value retriever	30.47	99.77	85.0	34.14	99.82	86.6	31.37	97.84	63.23

Table 4: Comparison and ablation results of the three-path schema retrieval framework. Precision and Recall assess the discrepancy between the retrieved schema and the gold schema (only the tables and columns appearing in the gold SQL). EX evaluates the downstream generation accuracy of our Generator (Qwen2.5-Coder-7B-Instruct).

Qwen2.5-Coder-7B-Instruct	Full Schema	Schema Filter	Δ Schema Filter
Base Model	53.92	58.87	+4.95
Dual-Validation	61.34	66.49	+5.15
Δ Dual-Validation	+7.42	+7.62	—

Table 5: 2x2 ablation of Dual-Validation and Schema Filter on BIRD Dev (EX).

example, Precision improves from 16.44% (CodeS) to 31.36% on BIRD Dev, while Recall remains above 98%. This indicates that our method effectively removes irrelevant schema elements without sacrificing critical schema coverage, which further translates into stronger and more stable downstream EX performance. To investigate the contribution of each retrieval component, we further ablate each retrieval path individually. Removing either the Pre-SQL retriever or the Value retriever degrades both retrieval quality and EX, confirming their roles in structural grounding and value-aware schema matching. Removing the Column Retriever increases Precision but reduces Recall and EX, indicating that although broader retrieval introduces redundancy, it helps ensure coverage of necessary schema elements for SQL generation. This suggests that missing critical schema elements is more harmful than introducing moderate redundancy.

We further conduct a 2x2 ablation on the more challenging BIRD Dev dataset using Qwen2.5-Coder-7B-Instruct as the backbone to analyze the contributions of Dual-Validation (including CoT data quality validation and SQL generation validation) and Schema Filter. As shown in Table 5, Schema Filter consistently improves performance under both settings (+4.95 / +5.15), while Dual-Validation yields larger gains (+7.42 / +7.62). This indicates that Dual-Validation is the primary source

Method	Data Synthesis Model	Spider Dev	Spider Test	BIRD Dev
EXCoT	GPT-4o	-	81.2	59.65
OmniSQL	Qwen2.5-Coder-32B-Instruct Qwen2.5-72B-Instruct	80.3	86.6	59.60
VCoT-SQL	Qwen2.5-Coder-7B-Instruct Qwen2.5-7B-Instruct	85.3	86.7	64.86
VCoT-SQL	Qwen2.5-Coder-32B-Instruct Qwen2.5-72B-Instruct	86.2	87.3	65.06

Table 6: Data synthesis setting and downstream performance (EX). ExCoT and OmniSQL report results trained on their synthesized CoT data from the Spider and BIRD training sets, VCoT-SQL follows the same setting. For the last row, the downstream Generator is fixed to Qwen2.5-Coder-7B-Instruct; only the data synthesis models are scaled.

of improvement, while Schema Filter also provides a non-negligible complementary contribution.

5 Analysis at Efficiency of Data Synthesis

Beyond final execution accuracy, a key advantage of VCoT-SQL is the efficiency of its data synthesis pipeline. Existing methods rely on heavy external resources. For example, OmniSQL uses a brute-force scaling strategy requiring ensembles of large models (32B and 72B) for knowledge distillation. Similarly, EXCoT (Zhai et al., 2025) adopts a divide-and-conquer CoT template (Pourreza et al., 2024) and relies on GPT-4o for synthetic SFT data. In contrast, the entire VCoT-SQL pipeline is self-driven by 7B-parameter models.

Under the premise that both methods use only the Spider and BIRD training sets as founda-

tional data, Table 6 shows that despite relying on smaller teacher models, VCoT-SQL achieves superior downstream performance. Compared to OmniSQL, our method delivers an absolute improvement of 5.26% on BIRD Dev and 5.0% on Spider Dev, while maintaining competitive performance on Spider Test. It also surpasses EXCoT on BIRD Dev despite not relying on GPT-4o-generated data. Scaling the data synthesis models to 32B/72B improves VCoT-SQL to 86.2% on Spider Dev, 87.3% on Spider Test, and 65.06% on BIRD Dev, while the downstream Generator remains fixed at 7B. These results suggest that scaling model size is not the only path to stronger Text-to-SQL reasoning. With a structured CoT template and strict validation, high-quality reasoning trajectories can be bootstrapped by 7B-class models. This "quality over quantity" paradigm reduces dependence on expensive large models and massive datasets.

6 Conclusion

This paper introduces VCoT-SQL, a generalizable Text-to-SQL framework that enhances SQL generation through dual validation by training on validated chain-of-thought data and validating generated SQL. We design tailored CoT templates and CoT quality validation mechanisms, enabling the collection of high-quality training data even with smaller models. Through supervised training on two specialized datasets constructed with this approach, we obtain the CoT-enhanced generator and validator. Extensive experiments on Spider, BIRD, and related benchmarks demonstrate that VCoT-SQL achieves competitive execution accuracy and robustness, effectively handling complex queries.

Limitations

Our VCoT-SQL framework relies on supervised fine-tuning (SFT) to train both the SQL generator and SQL validator. While SFT provides a solid foundation, it has a key drawback: it teaches the model to mimic the training examples but does not directly optimize for the true goal of producing executable and accurate SQL queries. Recent research has begun to enhance SFT by incorporating reinforcement learning algorithms such as GRPO, which can improve the model using feedback signals. We plan to address this in future work by incorporating reinforcement learning methods, which can use execution results and designed reward signals to directly optimize the model for

better performance.

Acknowledgements

This work is supported by the National Key Research and Development Program of China under grant 2025YFC3309504 and the National Natural Science Foundation of China under grants 62436003.

References

- Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. 2023. [Dr.Spider: A diagnostic evaluation benchmark towards text-to-SQL robustness](#). In *The Eleventh International Conference on Learning Representations*.
- Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, and 16 others. 2024. [Scaling instruction-finetuned language models](#). *Journal of Machine Learning Research*, 25(70):1–53.
- DeepSeek-AI. 2025. [DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning](#). *arXiv preprint arXiv:2501.12948*.
- 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. Association for Computational Linguistics.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. [BERT: Pre-training of deep bidirectional transformers for language understanding](#). In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.
- 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](#). *arXiv preprint arXiv:2106.01065*.
- Yujian Gan, Xinyun Chen, and Matthew Purver. 2021b. [Exploring underexplored limitations of cross-domain text-to-SQL generalization](#). *arXiv preprint arXiv:2109.05157*.

- Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. [Text-to-SQL empowered by large language models: A benchmark evaluation](#). *Proceedings of the VLDB Endowment*, 17(5):1132–1145.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. [DeepSeek-Coder: When the large language model meets programming—the rise of code intelligence](#). *arXiv preprint arXiv:2401.14196*.
- Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jianguang Lou, Ting Liu, and Dongmei Zhang. 2019. [Towards complex text-to-SQL in cross-domain database with intermediate representation](#). *arXiv preprint arXiv:1905.08205*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. [Qwen2.5-Coder technical report](#). *arXiv preprint arXiv:2409.12186*.
- Wonseok Hwang, Jinyeong Yim, Seunghyun Park, and Minjoon Seo. 2019. [A comprehensive exploration on WikiSQL with table-aware word contextualization](#). *arXiv preprint arXiv:1902.01069*.
- Seungone Kim, Se Joo, Doyoung Kim, Joel Jang, Seonghyeon Ye, Jamin Shin, and Minjoon Seo. 2023. [The CoT collection: Improving zero-shot and few-shot learning of language models via chain-of-thought fine-tuning](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 12685–12708, Singapore. Association for Computational Linguistics.
- Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. [MCS-SQL: Leveraging multiple prompts and multiple-choice selection for text-to-SQL generation](#). In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 337–353, Abu Dhabi, UAE. Association for Computational Linguistics.
- Fei Li and H. V. Jagadish. 2014. [NaLIR: An interactive natural language interface for querying relational databases](#). In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 709–712. Association for Computing Machinery.
- Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tiejing Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. [OmniSQL: Synthesizing high-quality text-to-SQL data at scale](#). *arXiv preprint arXiv:2503.02240*.
- 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, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023b. [Graphix-T5: Mixing pre-trained transformers with graph-aware layers for text-to-SQL parsing](#). In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 13076–13084.
- Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023c. [Can LLM already serve as a database interface? a BIG bench for large-scale database grounded text-to-SQLs](#). In *Advances in Neural Information Processing Systems*, volume 36, pages 42330–42357.
- Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2026. [XiYan-SQL: A novel multi-generator framework for text-to-SQL](#). *IEEE Transactions on Knowledge and Data Engineering*, 38(4):2474–2487.
- OpenAI. 2023. [GPT-4 technical report](#). *arXiv preprint arXiv:2303.08774*.
- OpenAI. 2024a. [GPT-4 model](#).
- OpenAI. 2024b. [GPT-4o mini: Advancing cost-efficient intelligence](#).
- OpenAI. 2024c. [Hello GPT-4o](#).
- OpenAI. 2024d. [OpenAI o1 system card](#). *arXiv preprint arXiv:2412.16720*.
- Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Serkan 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.
- 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, and Sercan O. Arik. 2025. [Reasoning-SQL: Reinforcement learning with SQL tailored partial rewards for reasoning-enhanced text-to-SQL](#). *arXiv preprint arXiv:2503.23157*.
- Jiexing Qi, Jingyao Tang, Ziwei He, Xiangpeng Wan, Yu Cheng, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. 2022. [RASAT: Integrating relational structures into pretrained Seq2Seq model for text-to-SQL](#). *arXiv preprint arXiv:2205.06983*.
- Yang Qin, Chao Chen, Zhihang Fu, Ze Chen, Dezhong Peng, Peng Hu, and Jieping Ye. 2025. [ROUTE: Robust multitask tuning and collaboration for text-to-SQL](#). In *The Thirteenth International Conference on Learning Representations*.
- Qwen Team. 2024. [Qwen2.5 technical report](#). *arXiv preprint arXiv:2412.15115*.
- Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. [Exploring the limits of transfer learning with a unified text-to-text transformer](#). *Journal of Machine Learning Research*, 21(140):1–67.
- Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. [ZeRO: Memory optimizations toward training trillion parameter models](#). In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.
- Jiawei Shen, Chengcheng Wan, Ruoyi Qiao, Jiazhen Zou, Hang Xu, Yuchen Shao, Yueling Zhang, Weikai Miao, and Geguang Pu. 2025. [Understanding, detecting, and repairing real-world in-context-learning-based text-to-SQL errors](#). *arXiv preprint arXiv:2501.09310*.
- Lei Sheng and Shuai-Shuai Xu. 2025a. [CSC-SQL: Corrective self-consistency in text-to-SQL via reinforcement learning](#). *arXiv preprint arXiv:2505.13271*.
- Lei Sheng and Shuai-Shuai Xu. 2025b. [SLM-SQL: An exploration of small language models for text-to-SQL](#). In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 1497–1512, Mumbai, India. The Asian Federation of Natural Language Processing and The Association for Computational Linguistics.
- 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*.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothee Lacroix, Baptiste Roziere, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. [LLaMA: Open and efficient foundation language models](#). *arXiv preprint arXiv:2302.13971*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. [Chain-of-thought prompting elicits reasoning in large language models](#). In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.
- William A. Woods, Ronald M. Kaplan, and Bonnie L. Nash-Webber. 1972. The lunar sciences natural language information system: Final report. Technical Report BBN Report 2378, Bolt Beranek and Newman Inc., Cambridge, MA.
- Bo Xu, Shufei Li, Hongyu Jing, Ming Du, Hui Song, Hongya Wang, and Yanghua Xiao. 2025. [Boosting text-to-SQL through multi-grained error identification](#). In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 4282–4292, Abu Dhabi, UAE. Association for Computational Linguistics.
- Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. 2024. [Synthesizing text-to-SQL data from weak and strong LLMs](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7864–7875, Bangkok, Thailand. Association for Computational Linguistics.
- Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018a. [SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task](#). *arXiv preprint arXiv:1810.05237*.
- 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. 2018b. [Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task](#). *arXiv preprint arXiv:1809.08887*.
- Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. [STaR: Bootstrapping reasoning with reasoning](#). In *Advances in Neural Information Processing Systems*, volume 35, pages 15476–15488.
- Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. 2025. [ExCoT: Optimizing reasoning for text-to-SQL with execution feedback](#). *arXiv preprint arXiv:2503.19988*.
- Rui Zhang, Chen Liu, Yixin Su, Ruixuan Li, Xuanjing Huang, Xuelong Li, and Philip S. Yu. 2025. [A comprehensive survey on multimodal RAG: All combinations of modalities as input and output](#). *TechRxiv*, DOI: 10.36227/techrxiv.176341513.38473003/v2.
- Victor Zhong, Caiming Xiong, and Richard Socher. 2017. [Seq2SQL: Generating structured queries from natural language using reinforcement learning](#). *arXiv preprint arXiv:1709.00103*.

A Schema Filtering Details and Serialization Format

Regarding the implementation details of the LLM-based components in our schema filter, both the coarse-grained Pre-SQL generation and the keyword extraction processes are performed using the Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) model. This lightweight yet powerful code-centric LLM provides efficient and reliable initial cues for the subsequent retrieval stages.

Furthermore, some datasets provide additional auxiliary information beyond basic table and column names, such as full names of tables and columns, textual descriptions, and external evidence. We leverage this auxiliary information to enable more accurate schema filtering.

When external evidence is available, we combine it with the user’s question and extracted keywords as query signals to retrieve relevant columns and values. This allows the retrieval process to capture implicit semantic cues that may not be explicitly mentioned in the question. For the column retriever, when a column is associated with a full name or a textual description, we encode each of these representations separately. The final similarity score between a column and a query (i.e., the question, extracted keywords, or external evidence) is computed as the maximum similarity among the column name, its full name, and its description. This strategy enables more robust matching, especially in cases where abbreviated schema names are insufficient to convey semantic meaning. And in our experiments, we set the top-K of the column retriever to 3, meaning each embedding retrieves at most 3 most relevant columns.

After obtaining the filtered schema elements from the three retrieval paths, we design a structured serialization format to present the schema to the LLM for downstream reasoning and SQL generation. Each table in the serialized schema includes both its table name and full name (if available). The filtered columns are organized in the following order: primary keys, foreign keys, and other columns. For each column, we include the following attributes: the column name, full name, column description, data type, whether it is a primary key, whether it is a foreign key, whether it contains null values, and two representative example values. In addition, all detected foreign key relations and matched values retrieved during the retrieval process are explicitly listed to further guide the model

during reasoning and generation. Figure 2 illustrates a concrete example of the serialized schema representation used in our framework.

B Experimental Details

For a more detailed discussion of baseline selection, we compare our method with representative Text-to-SQL systems under comparable inference settings. Some recent works (Sheng and Xu, 2025a,b) are not included in the main comparison because they adopt a multi-candidate generation and selection strategy with substantially larger inference budgets (from 4 to 64 candidates), whereas our method performs only one generation and one correction step. Under lower-budget settings (4 and 8 candidates), our method achieves better performance, indicating favorable efficiency-performance trade-off under constrained inference budgets.

We describe the construction process of our training datasets for both modules as follows. For the SQL Generator, we use the Spider training set (8,659) and BIRD training set (9,428) to generate CoT data. After applying our CoT quality validation mechanism, we obtain 5,477 and 3,669 high-quality CoT-augmented samples for Spider and BIRD, respectively. For the SQL Validator, we use the validated BIRD training set as a more challenging source, to generate validation CoT data, yielding 947 filtered samples for validator training.

Due to GPU memory limitations, VCoT-SQL was fine-tuned using bfloat16 mixed precision and model sharding, implemented via the integration of the Accelerate framework and DeepSpeed ZeRO-3 (Rajbhandari et al., 2020). Training was carried out on a server equipped with eight NVIDIA L40 GPUs, and the complete fine-tuning of both models was completed within a few hours. For each base model, we all set the number of epochs, context length, weight decay, and gradient clipping to 4, 2048, 0.1, and 1.0, respectively. These hyperparameter settings follow OmniSQL (Li et al., 2025) and are commonly adopted in fine-tuning practices.

C Validator Data Quality and Robustness Analysis

To better understand the effectiveness of the proposed validator, we further analyze two complementary aspects: the quality of the chain-of-thought (CoT) training data and the reliability of validator behavior during inference. Specifically, we first examine the construction and verification

The database schema section in the Prompt

【SQLite Database Schema】

```
account [  
  account_id (the id of the account) [INTEGER | PRIMARY_KEY | Examples: 1, 2];  
  frequency (frequency of the account) [TEXT | Examples: POPLATEK PO OBRATU]  
];  
client [  
  client_id (the unique number) [INTEGER | PRIMARY_KEY | Examples: 1, 2];  
  gender [TEXT | Examples: F, M]  
];  
disp (disposition) [  
  disp_id (unique number of identifying this row of record) [INTEGER | PRIMARY_KEY | Examples: 1, 2];  
  client_id (id number of client) [INTEGER | FOREIGN_KEY | Examples: 1, 2];  
  account_id (id number of account) [INTEGER | FOREIGN_KEY | Examples: 1, 2];  
  type (type of disposition) [TEXT | Examples: DISPONENT]  
];  
trans (transaction) [  
  account_id [INTEGER | FOREIGN_KEY | Examples: 1, 2];  
  k_symbol [TEXT | HAS_NONE | Examples: SIPO, SLUZBY];  
  account [INTEGER | HAS_NONE | Examples: 41403269, 87144583]  
];  
loan [  
  loan_id (the id number identifying the loan data) [INTEGER | PRIMARY_KEY | Examples: 4959, 4961];  
  account_id (the id number identifying the account) [INTEGER | FOREIGN_KEY | Examples: 2, 19]  
];  
`order` [  
  order_id (identifying the unique order) [INTEGER | PRIMARY_KEY | Examples: 29401, 29402];  
  account_id (id number of account) [INTEGER | FOREIGN_KEY | Examples: 1, 2];  
  bank_to (bank of the recipient) [TEXT | Examples: YZ, ST]  
];  
  
Foreign keys:  
  loan.account_id = account.account_id;  
  trans.account_id = account.account_id;  
  disp.account_id = account.account_id;  
  `order`.account_id = account.account_id;  
  disp.client_id = client.client_id;  
  
Matched values:  
  account.frequency : POPLATEK PO OBRATU  
  disp.type : DISPONENT
```

Figure 2: An example of the serialized schema representation used in our framework

Method	Spider Dev	Spider Test	BIRD Dev	Spider-DK	Spider-Syn	Spider-Realistic
only SQL Generator	85.3	86.7	64.86	76.8	77.6	82.7
Validator FT w/o multi-stage verification	85.6	86.7	65.78	77.0	77.9	83.1
Validator FT w/ multi-stage verification	85.7	86.9	66.49	77.2	78.5	83.3

Table 7: Effect of multi-stage verification on CoT training data quality.

		After Validation		
		Correct	Incorrect	Total
Before	Correct	64.67	0.20	64.86
	Incorrect	1.83	33.31	35.14
Validation	Total	66.49	33.51	100

Table 8: Transition matrix of execution correctness (%) before and after applying the validator on BIRD Dev.

process of CoT supervision, and then evaluate whether the validator introduces reliable corrections while avoiding harmful modifications.

C.1 Quality Assurance of CoT Examples

To ensure the quality of the validator training data, we employ a hybrid human–LLM data construction pipeline with multi-stage verification mechanisms.

1) Error-oriented CoT design. The validator’s CoT reasoning process is explicitly structured around six representative SQL error categories that are commonly adopted in prior Text-to-SQL research (Shen et al., 2025; Xu et al., 2025): syntactic error, column misuse, table misuse, condition errors, aggregation errors, and data type misuse. By constraining the reasoning process to these targeted failure modes, the design improves both the interpretability and consistency of validator decisions while reducing unnecessary reasoning variance.

2) Human-curated seed examples. We construct a small set (fewer than 10) of error-correction CoT demonstrations from real generation errors observed on Spider and BIRD datasets. These examples are manually curated by Text-to-SQL experts to ensure correctness and alignment with realistic error distributions.

3) Scalable CoT expansion with verification. Starting from the curated seed examples, we use LLMs to generate additional CoT training samples. To maintain data quality, all generated samples are filtered using the multi-stage verification pipeline introduced in Section 3.2.

To quantitatively assess the contribution of the verification process, we additionally train valida-

	Unval.	Val.	Val. Gain
Syntax-order	76.13	78.36	+2.23
Logic-order	76.55	78.99	+2.44
Logic-order Gain	+0.42	+0.63	+0.21

Table 9: 2×2 factorial analysis of template ordering and CoT validation.

tors using CoT data generated both with and without verification filtering. As shown in Table 7, verification consistently improves execution accuracy, demonstrating that the proposed filtering pipeline effectively enhances the quality of CoT supervision and leads to more reliable validator behavior.

C.2 Assessment of Validator Reliability

To assess validator reliability, we analyze how often the validator modifies generator outputs correctly or incorrectly. Using Qwen2.5-Coder-7B-Instruct on BIRD Dev, we compare Generator-only and Generator+Validator settings. As shown in Table 8, Generator-only outputs are correct on 64.86% of all examples and incorrect on 35.14%. After validation, 0.20% of all examples move from correct to incorrect, whereas 1.83% move from incorrect to correct. These percentages are measured over the full dataset. Overall, the validator yields a net execution accuracy gain of 1.63% (1.83% – 0.20%), which is consistent with the improvements reported in Table 2. These results indicate that the validator rarely introduces detrimental modifications while providing measurable corrective benefits, demonstrating both the robustness and effectiveness of the proposed validation framework.

D Additional Ablation Analysis

This section presents additional analyses of validation effectiveness and method robustness.

D.1 Factorized Analysis of CoT Validation

To disentangle the effects of template ordering and CoT validation, we perform a 2×2 factorial analysis over the four corresponding settings in Table 3.

Method	Spider Dev	Spider Test	BIRD Dev	Spider-DK	Spider-Syn	Spider-Realistic
Generator only	85.3	86.7	64.86	76.8	77.6	82.7
w/ fine-tuned 7B Validator	85.7	86.9	66.49	77.2	78.5	83.3
w/ GPT-5.4 Validator	86.3	87.4	67.99	80.9	79.3	82.7

Table 10: Upper-bound comparison of validators using outputs from the Qwen2.5-Coder-7B-Instruct Generator.

Backbone	DB perturbation	NLQ perturbation	SQL perturbation	Average
DeepSeek-Coder-6.7B-Instruct	86.2–69.6	85.8–74.2	84.3–82.3	85.4–75.8
Qwen2.5-Coder-7B-Instruct	85.0–72.5	84.8–74.9	85.7–81.7	85.1–76.5
XiYanSQL-QwenCoder-7B	87.0–73.5	87.3–76.6	86.6–84.2	87.1–78.3

Table 11: VCoT-SQL robustness on Dr.Spider (pre-/post-perturbation EX).

Table 9 shows that CoT validation consistently improves EX under both template orders, with gains of 2.23% for syntax-order and 2.44% for logic-order. Logic-order improves over syntax-order by 0.42% without validation and 0.63% with validation. The resulting interaction effect is only 0.21%, suggesting that the benefit of CoT validation is largely independent of template ordering.

D.2 Frontier Validator

To estimate SQL-validation headroom, we replace the Validator with GPT-5.4 while fixing the Qwen2.5-Coder-7B-Instruct Generator outputs. This isolates validation. As shown in Table 10, GPT-5.4 improves over the generation-only baseline on five of the six benchmarks and achieves the largest gain on Spider-DK. It also outperforms our fine-tuned 7B Validator on most datasets. The macro-averaged EX is 79.68% for the 7B Validator and 80.77% for GPT-5.4, with an absolute gap of 1.08%.

D.3 Dr. Spider Robustness

Following the official Dr.Spider (Chang et al., 2023) protocol, we evaluate VCoT-SQL on all 17 perturbation test sets: three database (DB), nine natural-language-question (NLQ), and five SQL perturbations. Each test set contains aligned pre- and post-perturbation examples. In Table 11, $x-y$ denotes macro-averaged EX on the corresponding pre- and post-perturbation examples. Across all three backbones, SQL perturbations cause the smallest drops, while DB perturbations cause the largest drops, with NLQ perturbations in between. XiYanSQL-QwenCoder achieves the best overall performance (87.1/78.3). DB perturbations directly alter schema representations and can therefore introduce schema-linking errors that propagate to

SQL generation, whereas SQL perturbations generally preserve the question–schema correspondence.

E CoT Data Synthesis Prompt Templates

In this section, we present the prompts used for generating our CoT dataset.

E.1 Prompt for SQL Generation

In our prompt, we use instructions to decompose the generation process into clause-level steps: SELECT, WHERE, GROUP BY/ORDER BY/LIMIT/HAVING, and FROM, and provide 3 examples as few-shot demonstrations to guide Qwen2.5-Coder-7B-Instruct in generating intermediate reasoning steps. The prompt used for synthesizing the generation CoT is illustrated in Fig 3, and the three few-shot examples are presented in Fig 4, Fig 5 and Fig 6.

E.2 Prompt for SQL Correction

In our prompt, we use instructions to decompose the validation process into multi-type checking steps: Check Syntax Errors, Check Columns, Tables and Joins, Check Conditions, Check GROUP BY/ORDER BY/LIMIT, Check Division Operations and Date Formats, and provide 3 examples as few-shot demonstrations to guide Qwen2.5-Coder-7B-Instruct in generating intermediate validation reasoning steps. The prompt used for synthesizing the validation CoT is illustrated in Fig. 7, and the three few-shot examples are presented in Fig. 8, Fig. 9 and Fig. 10.

F Inference Prompt Templates

For the inference stage, we employ the concise prompts for SQL generation and validation, as shown in Fig 11 and Fig 12, respectively.

Prompt for CoT Data Synthesis in SQL Generation

Your task is to generate a step-by-step, structured rationale to derive only one correct, executable SQL query based on the SQLite Database Schema, Question and Reference SQL which may potentially be incorrect.

IMPORTANT RULES:

You will receive the Reference SQL, which may or may not be correct. First, analyze whether the Reference SQL correctly satisfies the semantic requirements of the question. If correct, generate the rationale that aligns with its structure and style. If incorrect, create a rationale that maintains the same SQL style while correcting the semantics.

Rationale Structure:

1. Analyze which columns need to be used in the SELECT clause.
2. Determine whether a WHERE clause is required to meet certain conditions.
3. Assess whether clauses such as GROUP BY, ORDER BY, LIMIT, and HAVING are needed, and if so, which columns they require.
4. Based on the above analysis and the remaining semantic analysis of the question, identify all the tables that need to be used.

MUST follow the format and logical method shown in the examples below when writing your rationale.

Example 1

...

Example 2

...

Example 3

...

【SQLite Database Schema】

{TABLES}

【Question】

{QUESTION}

({EVIDENCE})

【Reference SQL】

{GOLD_SQL}

【Rationale】

Figure 3: The prompt template for synthesizing clause-level CoT rationales in SQL generation.

Example1 for SQL Generation

【SQLite Database Schema】

...

【Question】

Please show the lowest five eligible free rates for k-12 students with their corresponding SAT taker count in continuation schools that are directly funded.

(Eligible free rate for K-12 = `Free Meal Count (K-12)` / `Enrollment (K-12)`)

【Reference SQL】

```
SELECT T1.`Free Meal Count (K-12)` / T1.`Enrollment (K-12)` FROM frpm AS T1 INNER JOIN
satscores AS T2 ON T1.CDSCode = T2.cds WHERE T1.`Educational Option Type` = 'Continuation School'
AND T1.`Free Meal Count (K-12)` / T1.`Enrollment (K-12)` IS NOT NULL ORDER BY T1.`Free Meal
Count (K-12)` / T1.`Enrollment (K-12)` ASC LIMIT 5;
```

【Rationale】

1. ****Main Question****: Please show the lowest five eligible free rates for k-12 students with their corresponding SAT taker count in continuation schools that are directly funded.
2. ****Find SELECT columns****:
 - return eligible free rates for k-12 students ("Eligible free rate for K-12 = `Free Meal Count (K-12)` / `Enrollment (K-12)`"):
We need to use `Free Meal Count (K-12)` and `Enrollment (K-12)` column from `frpm` table.
Since both columns are of type REAL, no type casting is needed when performing division.
 - return corresponding SAT taker count:
We need to use `NumTstTskr` column from `satscores` table.
3. ****Find WHERE conditions****:
 - filter rows where the rows in continuation schools:
We need to use 'Continuation School' value in `Educational Option Type` column from `frpm` table to minimize redundant joins.
 - filter rows where the rows are directly funded:
We need to use 'Directly funded' value in `Charter Funding Type` column from `frpm` table to minimize redundant joins.
4. ****Find ORDER BY conditions****:
 - filter rows where the rows are lowest five:
We need to order the results by the `Free Meal Count (K-12)` / `Enrollment (K-12)` from `frpm` table ASC and `LIMIT 5`. Since ascending order is used here, null values in `Free Meal Count (K-12)` / `Enrollment (K-12)` must be excluded.
5. ****Find FROM tables****:
 - `frpm` table join `satscores` table:
we need to join the `frpm` table and `satscores` table on `frpm.CDSCode = satscores.cds`, based on the above analysis and the semantics of the question.
6. ****The final SQL****:

```
``sql
SELECT T1.`Free Meal Count (K-12)` / T1.`Enrollment (K-12)`, T2.NumTstTskr
FROM frpm AS T1 INNER JOIN satscores AS T2 ON T1.CDSCode = T2.cds
WHERE T1.`Educational Option Type` = 'Continuation School'
AND T1.`Charter Funding Type` = 'Directly funded'
AND T1.`Free Meal Count (K-12)` / T1.`Enrollment (K-12)` IS NOT NULL
ORDER BY T1.`Free Meal Count (K-12)` / T1.`Enrollment (K-12)` ASC LIMIT 5;
``
```

Figure 4: The example 1 for synthesizing clause-level CoT rationales in SQL generation.

Example2 for SQL Generation

【SQLite Database Schema】

...

【Question】

What is the percentage of loan amount that has been fully paid with no issue restricted to accounts?

(Loan paid with no issue means contract finished, no problems; status = 'A' means contract finished, no problems; Percentage of accounts by condition = $[(\text{total}(\text{amount}) \ \& \ \text{condition}) / (\text{total amount})] * 100\%$)

【Reference SQL】

```
SELECT (CAST(SUM(CASE WHEN status = 'A' THEN amount ELSE 0 END) AS REAL) * 100) /  
SUM(amount) FROM loan;
```

【Rationale】

1. ****Main Question****: What is the percentage of loan amount that has been fully paid with no issue restricted to accounts?
2. ****Find SELECT columns****:
 - return the percentage of loan amount that has been fully paid with no issue restricted to accounts("Loan paid with no issue means contract finished, no problems; status = 'A' means contract finished, no problems; Percentage of accounts by condition = $[(\text{total}(\text{amount}) \ \& \ \text{condition}) / (\text{total amount})] * 100\%$ ");

We need to divide the SUM of `amount` column from the `loan` table where `status` column from the `loan` table = 'A' by the total SUM of `amount` column from the `loan` table and then multiplying by 100.

Since the calculated sum is of INTEGER type, it needs to be converted to a REAL type to ensure precision in division operations and perform floating-point division.

3. ****Find FROM tables****:
 - from `loan` table:
- We only need the `loan` table, based on the above analysis and the semantics of the question.

4. ****The final SQL****:

```
``sql  
SELECT (CAST(SUM(CASE WHEN status = 'A' THEN amount ELSE 0 END) AS REAL) * 100) /  
SUM(amount)  
FROM loan;  
``
```

Figure 5: The example 2 for synthesizing clause-level CoT rationales in SQL generation.

Example3 for SQL Generation

【SQLite Database Schema】

...

【Question】

What is the status code of the items being ordered and shipped and its corresponding shipment tracking number?

(None)

【Reference SQL】

```
SELECT T1.order_item_status_code , T3.shipment_tracking_number FROM Order_items AS T1 JOIN
Shipment_Items AS T2 ON T1.order_item_id = T2.order_item_id JOIN Shipments AS T3 ON
T2.shipment_id = T3.shipment_id WHERE T1.order_id = 5;
```

【Rationale】

1. ****Main Question****: What is the status code of the items being ordered and shipped and its corresponding shipment tracking number?
2. ****Find SELECT columns****:
 - return the status code of the items being ordered:
We need to use 'order_item_status_code' column from 'Order_Items' table.
 - return the shipment tracking number:
We need to use 'shipment_tracking_number' column from 'Shipments' table.
3. ****Find FROM tables****:
 - 'order_items' table join 'Shipment_Items' table:
We need to join the 'order_items' table and 'Shipment_Items' table on 'Shipment_Items.order_item_id = Order_Items.order_item_id' to filter the items being ordered and shipped.
 - 'Shipment_Items' table join 'Shipments' table:
We need to join the 'Shipment_Items' table and 'Shipments' table on 'Shipment_Items.shipment_id = Shipments.shipment_id' to get the corresponding shipment tracking number.
4. ****The final SQL****:

```
```sql
SELECT T1.order_item_status_code , T3.shipment_tracking_number
FROM Order_items AS T1
JOIN Shipment_Items AS T2 ON T1.order_item_id = T2.order_item_id
JOIN Shipments AS T3 ON T2.shipment_id = T3.shipment_id;
```
```

Figure 6: The example 3 for synthesizing clause-level CoT rationales in SQL generation.

Prompt for CoT Data Synthesis in SQL Validation

Your task is to generate a step-by-step, structured, SQL validation rationale to check and correct the generated SQL(syntax errors will be provided directly) before based on the given database schema, natural language question and the final Modified SQL.

Validation Process Structure:

1. ****Check Syntax Errors****: Any errors in the generated SQL will be provided directly; please identify all syntax errors in the generated SQL based on the provided syntax errors and correct them.
2. ****Check Columns, Tables and Joins****: Only check whether the columns in the SELECT clause of generated SQL have issues of extra columns, missing columns, or incorrect columns. And whether the tables in the generated SQL have issues of extra tables, missing tables, or incorrect tables, and how to correct them. If there are foreign keys, check whether the foreign keys are used correctly.
3. ****Check Conditions****: Only check whether the condition(ignore data types) in the generated SQL meets the requirements of the question, including issues such as excessive conditions, insufficient conditions, or incorrect conditions.
4. ****Check GROUP BY, ORDER BY and LIMIT****: Only check whether ORDER BY/GROUP BY/LIMIT clauses are needed in the generated SQL, and if they are needed, whether they are applied correctly.
5. ****Check Division Operations and Date Formats****: Only check whether the generated SQL uses division operations. When division is used, the dividend must be converted to a REAL type to ensure precision, or check whether the generated SQL involves date-related columns and whether their usage matches the storage format in the database.
6. ****The final Modified SQL****: Finally, based on the above five steps of checking, return the final corrected SQL must be inside ```sql and ```.

Follow the format and logical method shown in the examples below when writing validation process.

Example 1

...

Example 2

...

Example 3

...

【SQLite Database Schema】

{TABLES}

【Question】

{QUESTION}

({EVIDENCE})

【Generated SQL】

{PERTURBED_SQL}

【Syntax error】

{SYNTAX_ERROR}

【the final Modified SQL】

{GOLD_SQL}

【Validation Process】

Figure 7: The prompt template for synthesizing multi-type CoT rationales in SQL validation.

Example1 for SQL Validation

【SQLite Database Schema】

...

【Question】

List the name, born state and age of the heads of departments ordered by age.

【Generated SQL】

SELECT head.name, head.born_state, head.age FROM head INNER JOIN management ON head.head_ID = management.head_ID ORDER BY head.age;

【Syntax error】

None

【the final Modified SQL】

SELECT name, born_state, age FROM head ORDER BY age;

【Validation Process】

1. ****Check Syntax Errors****:
 - Analysis: There are no syntax errors in the generated SQL.
 - Correction: No change.
2. ****Check Columns, Tables and Joins****:
 - Analysis: The generated SQL correctly selects 'head.name' column for name, 'head.born_state' column for born state, and 'head.age' column for age in the question.
 - Analysis: The generated SQL incorrectly joins with unnecessary 'management' table, only the 'head' table is required for the fields ('head.name', 'head.born_state', 'head.age').
 - Correction: Remove 'INNER JOIN management ON head.head_ID = management.head_ID'.
3. ****Check Conditions****:
 - Analysis: There are no explicitly filtering conditions in the question and generated SQL.
 - Correction: No change.
4. ****Check GROUP BY, ORDER BY and LIMIT****:
 - Analysis: The generated SQL correctly use 'ORDER BY head.age' for ordering by age.
 - Correction: No change.
5. ****Check Division Operations and Date Formats****:
 - Analysis: There are no division operations or date-related columns in the question and generated SQL.
 - Correction: No change.
6. ****The final Modified SQL****:

```
```sql
SELECT name, born_state, age FROM head ORDER BY age;
```
```

Figure 8: The example 1 for synthesizing multi-type CoT rationales in SQL validation.

Example2 for SQL Validation

【SQLite Database Schema】

...

【Question】

List all the distinct product names, price and details which are bought by female customers.
(details means product's description)

【Generated SQL】

```
SELECT DISTINCT T2.product_name, T2.product_price, T2.product_description FROM Products AS T1
JOIN Order_items AS T2 ON T1.product_id = T2.product_id JOIN Orders AS T3 ON T2.order_id =
T3.order_id JOIN Customers AS T4 ON T3.customer_id = T4.customer_id WHERE T4.gender_code = 'F';
```

【Syntax error】

no such columns T2.product_name;

【the final Modified SQL】

```
SELECT DISTINCT T1.product_name , T1.product_price , T1.product_description FROM Products AS
T1 JOIN Order_items AS T2 ON T1.product_id = T2.product_id JOIN Orders AS T3 ON T2.order_id =
T3.order_id JOIN Customers AS T4 ON T3.customer_id = T4.customer_id WHERE T4.gender_code =
'Female';
```

【Validation Process】

1. ****Check Syntax Errors****:
 - Analysis: There has a syntax error referencing "no such columns T2.product_name".
 - Analysis: The generated SQL incorrectly selects 'T2.product_name' column for product names, 'T2.product_price' column for product price, and 'T2.product_description' column for product details(means product's description), 'T2' refers to 'Order_Items' table, not 'Products' table.
 - Correction: Change 'T2.product_name, T2.product_price, T2.product_description' to 'T1.product_name, T1.product_price, T1.product_description' to refer to the 'Products' table.
2. ****Check Columns, Tables and Joins****:
 - Analysis: Issues in the SELECT clause of the generated SQL have been identified and corrected above.
 - Analysis: The generated SQL correctly joins 'Products' and 'Order_items' on 'Products.product_id = Order_items.product_id', joins 'Order_items' and 'Orders' on 'Order_Items.order_id = Orders.order_id', joins 'Orders' and 'Customers' on 'Orders.customer_id = Customers.customer_id', which all align with the database schema.
 - Correction: No change.
3. ****Check Conditions****:
 - Analysis: The generated SQL incorrectly uses 'T4.gender_code = 'F'' to get the products bought by female customers, matched values in the database schema is 'Customers.gender_code : Female'.
 - Correction: Change 'T4.gender_code = 'F'' to 'T4.gender_code = 'Female''.
4. ****Check GROUP BY, ORDER BY and LIMIT****:
 - Analysis: There are no such requirement in the question and generated SQL.
 - Correction: No change.
5. ****Check Division Operations and Date Formats****:
 - Analysis: There are no division operations or date-related columns in the question and generated SQL.
 - Correction: No change.
6. ****The final Modified SQL****:

```
```sql
SELECT DISTINCT T1.product_name , T1.product_price , T1.product_description FROM Products AS
T1 JOIN Order_items AS T2 ON T1.product_id = T2.product_id JOIN Orders AS T3 ON T2.order_id =
T3.order_id JOIN Customers AS T4 ON T3.customer_id = T4.customer_id WHERE T4.gender_code =
'Female';
```
```

Figure 9: The example 2 for synthesizing multi-type CoT rationales in SQL validation.

Example3 for SQL Validation

【SQLite Database Schema】

...

【Question】

What is the list ID that was first created by user 85981819?
(first created list refers to oldest list_creation_date_utc;)

【Generated SQL】

```
SELECT list_id FROM lists_users WHERE user_id = 85981819 GROUP BY list_id ORDER BY  
list_creation_date_utc ASC LIMIT 1
```

【Syntax error】

None

【the final Modified SQL】

```
SELECT list_id FROM lists_users WHERE user_id = 85981819 ORDER BY list_creation_date_utc ASC  
LIMIT 1
```

【Validation Process】

1. ****Check Syntax Errors****:
 - Analysis: There are no syntax errors in the generated SQL.
 - Correction: No change.
2. ****Check Columns, Tables and Joins****:
 - Analysis: The generated SQL correctly selects 'lists_users.list_id' column for the list ID.
 - Analysis: The generated SQL correctly uses the 'lists_users' table for the fields ('lists_users.list_id', 'lists_users.user_id', 'lists_users.list_creation_date_utc').
 - Correction: No change.
3. ****Check Conditions****:
 - Analysis: The generated SQL correctly uses 'user_id = 85981819' to get the user 85981819.
 - Correction: No change.
4. ****Check GROUP BY, ORDER BY and LIMIT****:
 - Analysis: The generated SQL correctly uses 'ORDER BY list_creation_date_utc ASC LIMIT 1' to get first created list(refers to oldest list_creation_date_utc).
 - Analysis: The generated SQL incorrectly uses 'GROUP BY list_id', because 'user_id' is primary key and each user has only one 'list_id' in the 'lists_users' table.
 - Correction: Remove the 'GROUP BY list_id'.
5. ****Check Division Operations and Date Formats****:
 - Analysis: There are no division operations in the question and generated SQL.
 - Analysis: The generated SQL use the date-related column 'list_creation_date_utc', but there is no specific matching date value.
 - Correction: No change.
6. ****The final Modified SQL****:

```
```sql  
SELECT list_id FROM lists_users WHERE user_id = 85981819 ORDER BY list_creation_date_utc ASC
LIMIT 1
```
```

Figure 10: The example 3 for synthesizing multi-type CoT rationales in SQL validation.

Prompts for inferring in SQL Generation

Your task is to convert the natural language question into a single, executable SQLite SQL query based on the given database schema through step-by-step analysis.

【SQLite Database Schema】

{TABLES}

【Question】

{QUESTION}

({EVIDENCE})

【Answer】

Figure 11: Inference prompt template for the SQL generation stage.

Prompts for inferring in SQL Validation

Your task is to examine the SQL generated during the text-to-SQL process for errors and perform corrections through step-by-step validation.

【SQLite Database Schema】

{TABLES}

【Question】

{QUESTION}

({EVIDENCE})

【Generated SQL】

{PRED_SQL}

【Syntax error】

{SYNTAX_ERROR}

【Answer】

Figure 12: Inference prompt template for the SQL validation stage.