

ReEx-SQL: Reasoning with Execution-Aware Reinforcement Learning for Text-to-SQL

Yaxun Dai^{1*}, Wenxuan Xie³, Xialie Zhuang⁴, Tianyu Yang⁵, Ziyi Liu¹,
Haiqin Yang⁶, Yiying Yang², Yuhang Zhao², Pingfu Chao^{1†}, Wenhao Jiang^{2†}

¹Institute of Computer Science and Technology, Soochow University, China

²Guangdong Laboratory of Artificial Intelligence and Digital Economy (SZ)

³Fudan University, ⁴University of Chinese Academy of Sciences

⁵Meituan, ⁶ Shenzhen Technology University

yaxundai@gmail.com, pfchao@suda.edu.cn, cswjiang@gmail.com

Abstract

Current Text-to-SQL reasoning models often lack integrated execution feedback during generation, and most existing approaches utilize feedback only for post-hoc correction. This separation not only limits real-time error correction, but may also introduce mistakes by altering otherwise correct SQL queries. To address these challenges, we present **ReEx-SQL** (*Reasoning with Execution-Aware Reinforcement Learning*), a Text-to-SQL framework that interacts with the SQL execution engine during decoding and dynamically adjusts reasoning based on execution feedback, thereby enabling context-sensitive query refinement and improved accuracy. ReEx-SQL achieves this through structured prompts with markup tags and a stepwise rollout strategy that incorporates execution feedback at each generation stage. For policy supervision, we design a composite reward function—featuring an exploration reward—to explicitly encourage effective interaction with the database. Furthermore, ReEx-SQL adopts a tree-based decoding strategy to facilitate exploratory reasoning and primarily aims to enhance parallel decoding efficiency. Notably, ReEx-SQL achieves 89.1% accuracy on Spider and 65.3% on BIRD at the 7B scale, surpassing baseline models by 2.7% and 2.6%, respectively. In addition, its tree-based decoding accelerates inference by 51.9% compared to linear decoding during sampling.

1 Introduction

The Text-to-SQL task aims to map natural language questions to executable SQL queries over a given database (Liu et al., 2024a). It plays a critical role in the building intelligent agents, requiring precise reasoning over user intent, database schema, and SQL logic (Zhang et al., 2024; Liu

et al., 2024b,a). Recent advances in large language models (LLMs) have significantly improved Text-to-SQL performance (Li et al., 2025a; Gao et al., 2024b; Li et al., 2024a), especially with the use of Chain-of-Thought (CoT) (Wei et al., 2022), which decomposes the generation process into intermediate steps to better handle complex queries. (Pourreza and Rafiei, 2023; Talaei et al., 2024; Chu et al., 2024; Zhang et al., 2023; Zhai et al., 2025). However, as illustrated in Figure 1, CoT-based standard reasoning still exhibit significant limitations. Specifically, LLMs relies solely on static context and its own reasoning capacity during generation, lacking mechanisms to verify or revise intermediate decisions. As a result, error accumulation from incorrect reasoning steps that the model cannot self-correct (Hsieh et al., 2023; Liu et al., 2025a), which further leads to: (1) syntactic errors due to the absence of execution validation (Zhai et al., 2025; Liu et al., 2025c; Chen, 2025); and (2) semantic misalignment, where the generated SQL fails to faithfully reflect the user’s intent (Shen et al., 2025a; Tyen et al., 2024; Xu et al., 2025).

Recent studies mainly use two strategies to mitigate reasoning errors: (1) improving CoT reasoning, often via reinforcement learning, but this does not fundamentally resolve the issue, as the lack of execution feedback during reasoning leaves models prone to semantic and syntactic errors (Li et al., 2025a; Lyu et al., 2025; Zhu et al., 2025; Liu et al., 2025a; Zhang et al., 2023; Zhai et al., 2025; Wang et al., 2025b); and (2) adding post-decoding verification to fix errors by using execution feedback, which can unintentionally modify correct predictions and introduce new mistakes (Pourreza et al., 2024; Wang et al., 2023; Pourreza and Rafiei, 2023; Qin et al., 2024). Notably, Tyen et al. (2024) found that DIN-SQL introduced 113 additional errors during the SQL correction process on BIRD.

To address these challenges, we propose **ReEx-SQL**, an end-to-end framework that combines CoT-

*Work done during an internship at the Guangdong Laboratory of Artificial Intelligence and Digital Economy (SZ).

†Corresponding authors.

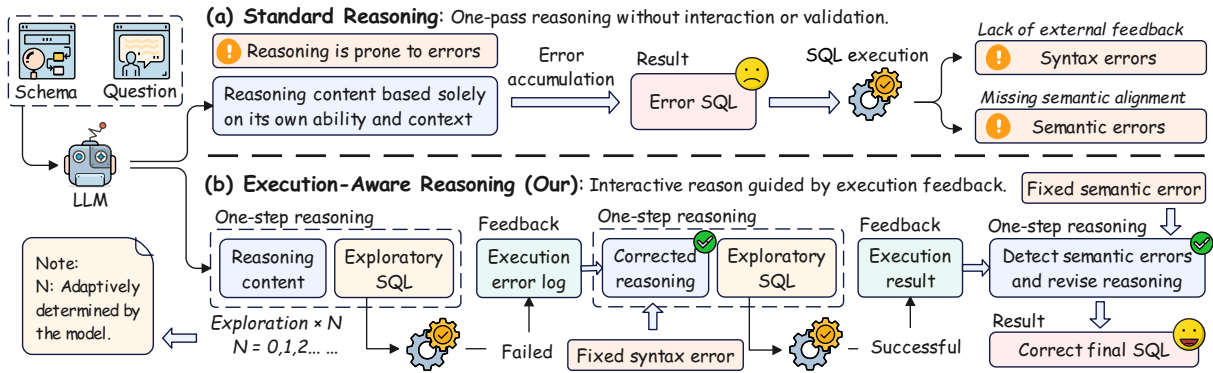


Figure 1: Standard reasoning vs. Execution-aware reasoning. Standard reasoning (e.g., Chain-of-Thought) generates a reasoning path without validation, leading to error accumulation and misalignment. In contrast, execution-aware reasoning uses the Generate-Verify-Refine paradigm to validate each step and fix errors via execution feedback.

based multi-step reasoning with direct execution feedback throughout the decoding process. Unlike previous methods that only enhance reasoning or treat verification as a separate post-processing step, ReEx-SQL enables model to interact with database at each reasoning stage, using execution results to iteratively validate and refine its intermediate SQL queries. This dynamic, feedback-driven process continues until the final SQL query is produced, as shown in Figure 1. To realize this, we extend the Grouped Reinforcement Policy Optimization (GRPO) (Shao et al., 2024) algorithm—enabling multi-round, execution-guided rollouts—and introduce masking of execution feedback to enhance training stability. We further design structured prompts with explicit markup tokens, maximizing interpretability and control. To counteract reward sparsity and potential reward hacking, a composite reward scheme comprising format, execution, exact match, entity match, and exploration is employed for comprehensive guidance. Finally, by adopting a tree decoding strategy, ReEx-SQL adaptively expands promising reasoning paths, boosting accuracy and reducing per-sample inference time by 51.9% over conventional linear decoding.

- We propose ReEx-SQL, an end-to-end framework that incorporates execution feedback to iteratively refine reasoning and adopts tree decoding for broader and faster inference.
- We improve training by extending GRPO for execution-aware rollouts, stabilizing with feedback masking, and jointly optimizing SQL quality and semantics using composite rewards.
- ReEx-SQL-7B achieves strong execution accuracy (EX) on both in-domain (89.1% Spider,

65.3% BIRD) and cross-domain benchmarks (85.5% Spider-Realistic, etc.). Compared to standard reasoning, it boosts EX by 2.6% and reduces syntax errors from 4.5% to 1.9%.

2 Related Work

2.1 LLM Reasoning in Text-to-SQL

With the rapid development of large language models, recent research on text-to-SQL has focused on enhancing model reasoning abilities (Liu et al., 2024b; Talaei et al., 2024; He et al., 2025; Xie et al., 2025). A common approach is supervised fine-tuning, which distills the powerful reasoning capabilities of closed-source models into smaller ones, as seen in methods like ROUTE-SQL (Qin et al., 2024), Omni-SQL (Li et al., 2025b) and PARSQL (Dai et al., 2025). Another line of work, such as ExCoT (Zhai et al., 2025) and EllieSQL (Zhu et al., 2025), leverages execution results to select positive and negative samples for Direct Preference Optimization (DPO) (Rafailov et al., 2023). More recently, methods like reasoning-SQL (Pourreza et al., 2025), SQL-R1 (Ma et al., 2025b), Think2SQL (Papicchio et al., 2025), and Arctic-Text2SQL-R1 (Yao et al., 2025) have adopted GRPO-based reinforcement learning (RL) to further improve reasoning. However, although reinforcement learning enhances reasoning quality, these approaches still lack execution feedback, which limits their ability to generate fully executable SQL queries that are consistent with database content.

2.2 Execution Feedback in Text-to-SQL

Execution feedback is a crucial external signal for enhancing LLM reasoning (Tyen et al., 2024). In

text-to-SQL, it is primarily used for post-hoc self-correction or self-consistency, where execution results revise or filter generated SQL (Shen et al., 2025a; Gao et al., 2024a; Dong et al., 2023). Recent work explores execution-guided selection during inference, such as Query and Conquer (Borchmann and Wydmuch, 2025), which applies partial execution checking via the PipeSQL dialect. By contrast, in other domains, execution feedback is directly integrated into LLM reasoning (Zhang et al., 2025), with RL strengthening model-tool interaction for tasks like code execution (Qian et al., 2025) and information retrieval (Jin et al., 2025; Chen et al., 2025; Song et al., 2025). Although some prior work in text-to-SQL explores tool-based interaction (Yang et al., 2023), these approaches are typically modular, leaving end-to-end, RL-driven interactive methods underexplored. To bridge this gap, we propose ReEx-SQL, which integrates the SQL executor directly into the model’s reasoning process. By leveraging the fact that both tool inputs and model outputs are SQL, ReEx-SQL adopts a generate-verify-correct feedback paradigm. Unlike prior modular self-correction methods (Talaie et al., 2024; Pourreza and Rafiei, 2023), ReEx-SQL incorporates execution feedback during reasoning, reducing the risk of incorrectly modifying correct SQL. This end-to-end feedback mechanism extends the applicability of execution feedback and introduces a new paradigm for structured query.

3 Methodology

End-to-end reinforcement learning with execution feedback for multi-step reasoning poses several key challenges: (1) ensuring stable RL training; (2) preventing reward hacking; and (3) maintaining efficient tool-integrated reasoning. To address these, we improve the GRPO algorithm for multi-turn reasoning with masked tool outputs, introduce custom rollout templates for better control, design composite reward functions, and refine the decoding strategy to enhance efficiency and output quality.

3.1 Execution-Aware Reinforcement Learning

To ensure stable reinforcement learning training with execution feedback, inspired by related work (Jin et al., 2025; Chen et al., 2025; Song et al., 2025), we enhance GRPO algorithm (Shao et al., 2024) by introducing a SQL executor \mathcal{E} into the model reasoning during the rollout process. When the model generates a tool-call token, we interrupt

generation, extract and execute the SQL, and append the feedback to the context. This adaptive process continues automatically until the model outputs a final SQL, without needing a preset number of interactions. Since execution feedback is externally generated, we mask these tokens during loss computation to ensure that policy updates are based only on model outputs.

Formally, given an existing policy $\pi_{\theta_{\text{old}}}$ and a reference policy $\pi_{\theta_{\text{ref}}}$, for each input x sampled from the data distribution \mathcal{D} , Unlike standard methods, we generate G rollouts $\tau = \{y_i\}_{i=1}^G$, where each y_i is drawn from $\pi_{\theta_{\text{old}}}(\cdot|x; \mathcal{E})$. Here, \mathcal{E} denotes the SQL executor, and each rollout y_i can be interpreted as $\pi_{\theta_{\text{old}}}(\cdot|x) \otimes \mathcal{E}$, where \otimes represents the interleaved process, detailed explanations are provided in Section 3.2. Multi-turn execution-aware reinforcement learning aims to update the policy π_{θ} by maximizing the following objective:

$$\mathcal{J}(\theta) = \mathbb{E}_{x \sim \mathcal{D}, \{y_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(\cdot|x; \mathcal{E})} \left[\frac{1}{G} \sum_{i=1}^G \left(\min \left(\frac{\pi_{\theta}(y_i|x; \mathcal{E})}{\pi_{\theta_{\text{old}}}(y_i|x; \mathcal{E})} A_i, \text{clip} \right) \right) - \beta \mathbb{D}_{\text{KL}} \right],$$

$$\text{clip} = \text{clip} \left(\frac{\pi_{\theta}(y_i|x; \mathcal{E})}{\pi_{\theta_{\text{old}}}(y_i|x; \mathcal{E})}, 1 - \epsilon, 1 + \epsilon \right) A_i, \quad (1)$$

where $A_i = (R^i - \text{mean}(\{R^j\}_{j=1}^G)) / \text{std}(\{R^j\}_{j=1}^G)$ denotes advantage of the i -th rollout, with reward R^i , clipping ratio ϵ , and KL loss coefficient β .

3.2 Rollout for Execution-Aware Reasoning

To enable execution-aware reasoning, ReEx-SQL adopts a stepwise rollout mechanism that integrates SQL execution feedback directly into the generation process. As illustrated in Figure 2, ReEx-SQL employs structured prompts with markup tags to distinguish reasoning content, intermediate SQL, execution feedback, and the final SQL. After generating an intermediate SQL, a SQL executor \mathcal{E} executes it and appends the feedback, either results or errors, to the context to guide subsequent reasoning. This iterative process enables progressive refinement until either a final SQL query is produced or a predefined interaction limit N is reached. The complete procedure is presented in Algorithm 1, and the full prompt format is shown in Figure A.1.

3.3 Composite Reward Design

The reward is a crucial component of reinforcement learning, and studies have shown that overly simplistic rewards can lead to reward hacking (Skalse et al., 2022). Reasoning-SQL (Pourreza et al.,

Response Format: `<think>` Reasoning process. `</think>` `<intermediate_sql>` The SQL query for verification is: "Intermediate SQL"`</intermediate_sql>` `<result>` Execution result. `</result>` `<think>` Further reasoning. `</think>` `<final_sql>` The final SQL query is: "Final SQL"`</final_sql>`

Figure 2: Response format for ReEx-SQL.

2025) points out that sparse rewards pose challenges for text-to-SQL. To effectively guide policy optimization, we design a composite reward function encompassing five dimensions: format, execution, exact match, entity match, and exploration. Each component provides complementary supervision, steering the model to generate correct, executable, and semantically consistent SQL queries. Notably, the exploration reward is tailored for multi-turn interactions, discouraging repetitive behavior and inadequate utilization of tools.

Format reward The format reward R_{format} encourages the output y that strictly follow the predefined template by assigning 1 for perfect conformity and 0 otherwise, promoting consistent and well-structured SQL generation.

Exact match reward The exact match reward R_{em} assigns 1 only if the output SQL s_{final} is identical to the reference SQL s_{final} , encouraging alignment with preferred SQL styles.

Execution reward The execution reward R_{exec} consists of two parts: a syntax reward R_{valid} (1 if s_{final} executes without error) and a result reward (1 if $\text{Exec}(s_{\text{final}}) = \text{Exec}(s_{\text{gold}})$, otherwise 0, where $\text{Exec}(s)$ denotes the result of executing SQL s). This encourages semantic alignment with s_{gold} .

Entity match reward The entity match reward R_{entity} encourages correct use of tables and columns by measuring their overlap between s_{final} and s_{gold} . Here, $\text{Entities}(s)$ denotes the set of tables and columns in SQL s , as follows:

$$R_{\text{entity}} = \frac{|\text{Entities}(s_{\text{final}}) \cap \text{Entities}(s_{\text{gold}})|}{|\text{Entities}(s_{\text{gold}})|}. \quad (2)$$

Exploration Reward The exploration reward R_{expl} is specifically designed for multi-turn interaction scenarios to encourage effective and diverse tool usage. It penalizes duplicate intermediate SQLs to discourage redundancy and rewards broader exploration using a sigmoid function $\sigma(n)$

over the number of interaction steps n when the final SQL s_{final} is incorrect. Let $\mathcal{S}_{\text{explore}}$ denote all generated intermediate SQLs, as follows:

$$R_{\text{expl}} = \begin{cases} 0, & \text{if } |\mathcal{S}_{\text{explore}}| \neq |\text{set}(\mathcal{S}_{\text{explore}})| \\ 1, & \text{if } \text{Exec}(s_{\text{final}}) = \text{Exec}(s_{\text{gold}}) \\ \sigma(n), & \text{otherwise} \end{cases}. \quad (3)$$

Composite Reward $R = \sum_{i \in \mathcal{C}} w_i R_i$ is final reward, where $\mathcal{C} = \{\text{format}, \text{em}, \text{exec}, \text{entity}, \text{expl}\}$ and w_i is the weight of R_i . Each term provides distinct supervision to promote syntactic validity, execution correctness, schema alignment.

3.4 Execution-Guided Tree Decoding

Both the interaction with a SQL executor \mathcal{E} during decoding in ReEx-SQL and the use of self-consistency strategies that generate many SQL candidates in text-to-SQL tasks increase computational cost. To address these efficiency challenges, we employ a tree-structured decoding process, drawing inspiration from beam search (Choo et al., 2022) and Tree of Thought (Yao et al., 2023). In contrast to these approaches, each node in our decoding tree captures the reasoning context, a candidate SQL, and its execution result (see Figure 3), thus enabling efficient pruning of error-producing nodes and minimizing redundant exploration. After generating multiple SQL candidates, ReEx-SQL applies an execution-based voting mechanism to select the most reliable output. Overall, compared to the independent linear decoding, our tree decoding eliminates redundant paths, improves efficiency, and, particularly in early stages, reduces the number of candidate nodes and the likelihood of early errors, thereby maintaining high reasoning quality.

4 Experiments

4.1 Experimental Settings

Datasets and Metrics We conduct experiments on two widely-used Text-to-SQL benchmarks: BIRD (Li et al., 2023) and Spider (Yu et al., 2018). BIRD contains over 12,000 queries across 95 databases in diverse domains, designed to better reflect real-world scenarios; we evaluate on its development set (Dev) using execution accuracy (EX) and valid efficiency score (VES), where VES measures how efficiently the predicted SQL runs compared to the ground truth. Spider covers 200 databases in 138 domains, with evaluations

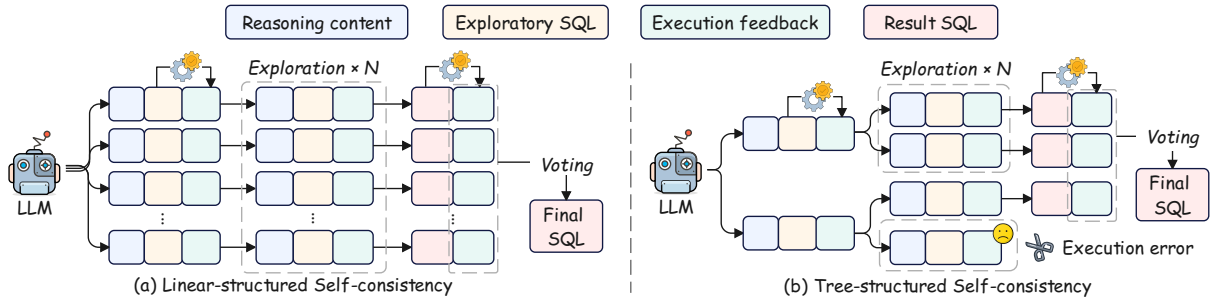


Figure 3: Comparison between linear-structured and tree-structured self-consistency decoding.

on its Dev set using EX and test-suite accuracy (TS) (Zhong et al., 2020), which checks if the predicted SQL yields correct results across multiple database test cases. Additionally, we assess three Spider robustness variants: Spider-Syn (Gan et al., 2021a), Spider-Realistic (Deng et al., 2021), and Spider-DK (Gan et al., 2021b), each introducing specific schema or question perturbations.

Implementation Details We use Qwen2.5-Coder-7B-Instruct model for experiments. Database prompts are adapted from CodeS (Li et al., 2024b), which filter relevant schema components, values, and metadata, and show strong results on the BIRD benchmark. Training is performed with the GRPO (Shao et al., 2024) algorithm within the OpenRLHF (Hu et al., 2024) framework, using a batch size of 64 and a learning rate of $2e-6$. In the rollout phase, we sample 8 outputs per input with a temperature T of 1.0, set the maximum sequence length to 4096, and up to maximum interactions $N=10$. The reward weight vector is defined as $\mathbf{w} = (w_{\text{format}}, w_{\text{em}}, w_{\text{exec}}, w_{\text{entity}}, w_{\text{expl}}) = (2.0, 1.0, 3.0, 1.0, 2.0)$, a configuration referred to as **MaxTune**. For inference, we apply greedy decoding ($T=0.0$), and tree-structured decoding ($T=0.7$), allowing up to 3 children per node and generating up to 16 candidates. SQL execution is handled by SQLite (SQLite, n.d.), which provides feedback including column headers and up to 3 sample rows. All experiments run on a 24-core CPU at 2.10GHz and 8 NVIDIA A800 GPUs.

4.2 Overall Performance

Results on BIRD and Spider Table 1 shows the performance of ReEx-SQL on BIRD and Spider. Using only 9k training samples from BIRD Train, ReEx-SQL achieves 65.3% EX and 73.3% VES on BIRD Dev, narrowing the gap with proprietary solutions. By comparison, OmniSQL-7B (Li et al.,

Methods	BIRD Dev		Spider Dev		Spider Test
	EX	VES	EX	TS	EX
XiYan-SQL (Gao et al., 2024b)	73.3	-	-	-	89.6
CHASE-SQL (Pourreza et al., 2024)	73.0	73.0	-	-	87.6
MCTS-SQL + GPT-4 (Yuan et al., 2025)	69.4	66.2	88.7	-	86.6
PAS-SQL + GPT-4o (Kong et al., 2025)	64.7	65.0	87.9	-	86.8
MCS-SQL + GPT-4 (Lee et al., 2025)	63.4	64.8	89.5	-	89.6
MAC-SQL + GPT-4 (Wang et al., 2023)	59.4	66.2	86.8	82.8	-
SENSE-13B (Yang et al., 2024)	55.5	-	84.1	83.5	86.6
DPO + Qwen2.5-7B* (Liu et al., 2025b)	64.1	-	82.6	80.2	-
SFT CodeS-7B (Li et al., 2024b)	57.2	58.8	85.4	80.3	-
OmniSQL-7B* (Li et al., 2025b)	66.1	-	81.6	-	88.9
DTS-SQL-7B* (Pourreza and Rafiei, 2024)	55.8	60.3	82.7	78.4	-
DB-Explore-7B* (Ma et al., 2025a)	52.1	55.8	84	79.3	-
ROUTE + Qwen2.5-7B (Qin et al., 2024)	55.9	57.4	83.6	77.5	83.7
ROUTE + Qwen2.5-14B (Qin et al., 2024)	60.8	65.2	87.3	80.9	87.1
Reasoning-SQL* (Pourreza et al., 2025)	64.0	-	-	-	78.72
SQI-R1-7B* (Ma et al., 2025b)	63.1	-	84.5	-	86.1
Arcite-Text2SQL-R1* (Yao et al., 2025)	64.8	-	-	-	87.1
SQL-o1 + Llama3-8B (Lyu et al., 2025)	63.4	64.7	87.4	79.6	85.4
ReEx-SQL-7B (Our)	65.3	73.3	89.1	83.9	86.9

* The methods marked use the same base model as ReEx-SQL (Qwen2.5-Coder-Instruction-7B), and this notation is also adopted in Table 2.

Table 1: Performance comparison on BIRD and Spider benchmarks (%).

2025b) uses 2.5M training samples, which is more than 100 times the amount of data, to obtain higher in-domain performance. However, as shown in Table 2, ReEx-SQL demonstrates much stronger generalization on perturbed and realistic benchmarks, highlighting the efficiency and robustness of our execution-aware approach. On Spider Dev, ReEx-SQL reaches 83.9% TS, outperforming several large proprietary models. While recent SOTA methods (Gao et al., 2024b; Pourreza et al., 2024) report higher scores, they often require multiple models or proprietary pipeline. In contrast, ReEx-SQL uses a single 7B open-source model, enabling efficient and accessible deployment.

Robustness Evaluation Table 2 reports robustness results on three Spider variants. (1) ReEx-SQL-7B achieves the highest EX scores, with 78.7% on Spider-Syn and 85.5% on Spider-Realistic, outperforming both open-source and proprietary models. (2) It also leads in TS

Methods	Syn		Realistic		DK
	EX	TS	EX	TS	EX
FastRAT _{ext} +GPT-4 (Shen et al., 2024)	74.4	-	80.9	-	72.3
TA-SQL + GPT-4 (Qu et al., 2024)	-	-	79.5	-	72.9
CYCLESQL + GPT-4 (Fan et al., 2024)	76.0	66.3	70.6	56.9	68.5
SQL-PaLM + PaLM2 (Sun et al., 2024)	74.6	-	77.6	-	66.5
DPO + Qwen2.5-Coder-7B* (Liu et al., 2025b)	76.2	-	79.1	-	72.9
Qwen2.5-Coder-32B (Li et al., 2025b)	70.5	-	74.8	-	78.3
ROUTE + Llama3-8B (Qin et al., 2024)	77.4	70.2	80.9	72.6	74.6
SFT CodeS-7B (Li et al., 2024b)	76.9	70.0	82.9	77.2	72.0
SFT CodeS-15B (Li et al., 2024b)	77.0	69.4	83.1	75.6	70.7
SENSE-7B (Yang et al., 2024)	72.6	64.9	82.7	75.6	77.9
SENSE-13B (Yang et al., 2024)	77.6	70.2	84.1	76.6	80.2
OmniSQL-7B* (Li et al., 2025b)	69.6	-	78.0	-	77.8
OmniSQL-14B (Li et al., 2025b)	72.0	-	78.5	-	74.8
OmniSQL-32B (Li et al., 2025b)	72.1	-	77.2	-	77.6
Reasoning-SQL* (Pourreza et al., 2025)	69.3	-	-	-	73.3
SQL-R1-7B* (Ma et al., 2025b)	76.7	-	83.3	-	78.1
SQL-o1 + Llama3-8B (Lyu et al., 2025)	77.6	69.2	82.7	72.8	78.7
ReEx-SQL-7B (Our)	78.7	72.0	85.5	79.9	80.2

Table 2: Performance comparison on Spider variants robustness benchmarks (%).

scores (72.0% on Spider-Syn and 79.9% on Spider-Realistic), indicating fewer semantic errors and better SQL structure alignment. (3) On Spider-DK, ReEx-SQL-7B achieves 80.2% using only a single 7B model, matching the performance of SENSE-13B (Yang et al., 2024), which relies on a larger model and additional data augmentation. Overall, ReEx-SQL demonstrates strong robustness and generalization across diverse perturbations, while remaining lightweight and cost-effective.

4.3 Comparison with Existing RL Methods

Table 3 reports EX scores on BIRD Dev for existing RL-based methods and our reproduced baseline, all employing the standard reasoning paradigm with vanilla Qwen2.5-coder-7b-Instruct and GRPO optimization, under greedy (G) and sampling-8 voting (V) strategies. Baseline uses the same training setup as ReEx-SQL, differing only in prompting template and the use of exploration reward R_{expl} , which is exclusive to ReEx-SQL (see Appendix A.1). Reasoning-SQL (Pourreza et al., 2025) uses a custom reward design, Arctic-Text2SQL-R1 (Yao et al., 2025) and SQL-R1 (Ma et al., 2025b) benefit from data augmentation, and Baseline is our reproduction without interactive reasoning. Distinctively, ReEx-SQL incorporates execution feedback during inference for iterative interaction. Compared to Arctic-Text2SQL-R1, which relies on extensive data augmentation, ReEx-SQL achieves strong performance using only the original training data. Furthermore, ReEx-SQL demonstrates robust generalization, reaching EX of 78.7% on Spider-Syn and 79.9% on Spider-Realistic.

Method	BIRD Dev		Spider Test		Spider-Syn		Spider-Realistic	
	G	V	G	V	G	V	G	V
Reasoning-SQL*	64.01	-	78.72	-	69.34	-	73.27	-
Arctic-Text2SQL-R1	63.9	64.8	85.0	87.1	-	-	-	-
SQL-R1	-	63.1	-	86.1	-	76.7	-	78.1
Think2SQL	56.1	-	82.4	-	78.0	-	67	-
Baseline (Our)	60.8	63.16	86.2	86.7	78.3	78.5	82.3	83.1
ReEx-SQL (Our)	63.4	65.3	86.4	86.9	78.5	78.7	79.3	79.9

* The decoding strategy used in Reasoning-SQL is not reported.

Table 3: Comparison with existing RL-based methods.

4.4 Ablation Study on Reasoning Paradigms

To highlight the advantage of the execution-aware reasoning paradigm in ReEx-SQL, Table 4 compares its performance with the standard reasoning baseline (see section 4.3 for setup details). Key findings are as follows: (1) After GRPO training, ReEx-SQL consistently outperforms the baseline on all benchmarks, with gains of 2.6% and 2.7% in EX score on BIRD Dev and Spider Dev, respectively. For robustness, ReEx-SQL also improves TS scores on Spider-Syn and Spider-Realistic by 3.2% and 3.1%, and achieves a 4.7% EX score increase on Spider-DK. These results demonstrate that the execution-aware reasoning itself drives the improvement. (2) ReEx-SQL achieves a 14.8% gain after GRPO training (from 48.6% to 63.4%), compared to an 11.3% increase for the baseline. This validates the effectiveness of GRPO and suggests that execution-aware reasoning can unlock greater potential and performance improvements.

ReEx-SQL vs. Baseline To further demonstrate the strengths of ReEx-SQL, Figure 4 provides a comprehensive comparison with the baseline. Key observations are: (1) Training rewards (a): Both models converge rapidly, but ReEx-SQL consistently achieves higher average rewards, indicating a greater performance ceiling, in line with Table 4. (2) Response length (b): Both models produce longer responses over training, echoing DeepSeek (DeepSeek-AI et al., 2025). ReEx-SQL extends reasoning via interaction, while the baseline deepens reasoning after noise reduction. (3) Performance by difficulty (c, e): ReEx-SQL outperforms the baseline across all difficulty levels on BIRD Dev and Spider Dev, and achieves especially notable improvements on challenging and extra hard cases: 6.2% and 11.5% higher on BIRD and Spider Dev, respectively. This highlights ReEx-SQL’s advantage on complex queries. (4) Error analysis (d): On BIRD Dev, ReEx-SQL reduces to-

Reasoning Paradigm	Training Type	BIRD Dev		Spider Dev		Spider Test	Spider-Syn		Spider-Realistic		Spider-DK
		EX (%)	EX (%)	TS (%)	EX (%)	EX (%)	TS (%)	EX (%)	TS (%)	EX (%)	
Direct Output	No Post-Training	48.7	81.4	75.3	79.3	68.3	54.6	77.0	68.4	71.6	
Standard Reasoning	No Post-Training	49.5	81.1	74.8	78.3	69.5	56.9	74.6	63.1	71.4	
Execution-Aware Reasoning	No Post-Training	48.6	80.0	75.9	37.0	57.4	43.1	65.7	53.8	57.6	
Direct Output	SFT	56.3	85.4	81.4	84.5	75.7	68.3	81.9	74.2	74.0	
Standard Reasoning	GRPO	60.8	85.7	83.5	86.2	78.3	68.9	82.3	75.6	74.6	
Execution-Aware Reasoning	GRPO	63.4_{+2.6}	88.4_{+2.7}	83.5_{+0.0}	86.4_{+0.2}	78.5_{+0.2}	72.1_{+3.2}	85.0_{+2.7}	78.7_{+3.1}	79.3_{+4.7}	

* All results are obtained with greedy decoding. "Direct Output" refers to directly predicting SQL without intermediate reasoning.

Table 4: Comparison of reasoning paradigms on multiple Text-to-SQL benchmarks.

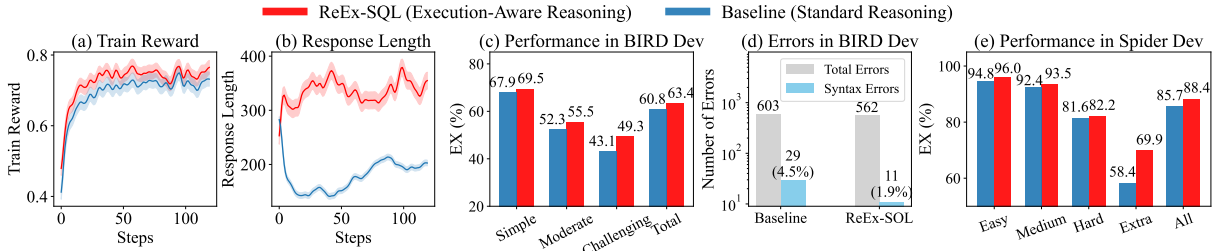


Figure 4: Multi-dimensional comparison between ReEx-SQL and the baseline: (a) Training reward over steps; (b) Response length throughout training; (c) BIRD performance across difficulty levels; (d) Error type distribution on the BIRD Dev; (e) Spider performance across difficulty levels.

Reward Components	BIRD Dev	
	EX (%)	VES (%)
ReEx-SQL (Base Configuration)	60.9	70.8
w/o R_{em}	58.0 (↓ 2.9)	65.4 (↓ 5.4)
w/o R_{exec}	54.4 (↓ 6.5)	62.7 (↓ 8.1)
w/o R_{entity}	57.8 (↓ 3.1)	66.1 (↓ 4.7)
w/o R_{expl}	59.2 (↓ 1.7)	69.3 (↓ 1.5)

Table 5: Ablation of reward components for ReEx-SQL on BIRD Dev.

Configuration Type	Reward Weights					BIRD Dev	
	w_{format}	w_{em}	w_{exec}	w_{entity}	w_{expl}	EX (%)	VES (%)
Base	1.5	0.8	1.5	0.8	0.5	60.9	70.8
HighExploration	1.5	0.8	1.5	0.8	2.0	62.0	69.4
HighFormat-EX	2.0	0.8	3.0	0.8	0.5	62.3	70.1
Uniform	1.0	1.0	1.0	1.0	1.0	62.3	73.8
MaxTune (Best)	2.0	1.0	3.0	1.0	2.0	63.4	72.3

Table 6: Ablation of reward weights for ReEx-SQL on BIRD Dev. Configuration names reflect the characteristic reward weight assignments.

tal errors from 603 to 562 and lowers syntax errors from 4.5% to 1.9%. These results show that ReEx-SQL more effectively mitigates both syntactic and semantic errors, overcoming the shortcomings of the baseline. Representative output comparisons can be found in Appendix A.2.4.

4.5 Ablation Study on Reward Design

Ablation of Reward Components Table 5 shows the effect of removing each reward com-

ponent (configuration "Base" in Table 6). Eliminating any reward decreases ReEx-SQL performance, underscoring the importance of all components. Execution reward R_{exec} has the largest impact, demonstrating the distinctive benefit of execution feedback in Text-to-SQL. While the direct effect of the exploration reward is smaller, as discussed in Appendix A.2.3, it significantly increases exploration diversity, ensuring that each interaction yields informative gains.

Ablation of Reward Weights Table 6 summarizes the influence of different reward weights. Key observations: (1) Increasing exploration reward weight w_{expl} (Base vs. HighExploration) improves EX score by 1.1% on BIRD Dev, indicating better interaction with the database contributes to higher SQL quality. (2) Raising format w_{format} and execution w_{exec} weights (Base vs. HighFormat-EX) increases EX score by 1.4%, showing that format and execution consistency enhance output quality. (3) MaxTune, which increases w_{format} , w_{exec} , and w_{expl} , yields the best results, highlighting the benefits of integrating multiple reward signals.

4.6 Decoding Strategy Comparison

Table 7 summarizes the results of different decoding strategies on BIRD Dev and Spider Dev. With a fixed candidate number, tree-structured and tree pruning decoding consistently outperform linear decoding in both EX score and efficiency. For ex-

Decoding Strategy		Number of Candidates	Bird Dev			Spider Dev		
			EX (%) \uparrow	Latency (s/sample) \downarrow	Cost \downarrow	EX (%) \uparrow	Latency (s/sample) \downarrow	Cost \downarrow
Baseline	Greedy Decoding	1	60.30	0.40	207.8	85.7	0.15	122.6
	Linear Decoding	8	62.70	1.85	1651.9	86.1	0.81	985.1
	Linear Decoding	16	63.16	3.15	3415.5	86.1	2.14	1975.3
ReEx-SQL	Greedy Decoding	1	63.36	0.60	356.4	88.4	0.20	148.2
	Linear Decoding	8	64.08	2.79	2849.9	88.3	1.37	1162.2
	Tree Decoding	8	64.15	2.04 (\downarrow 28.6%)	1784.6	88.4	0.88 (\downarrow 35.7%)	729.4
	Tree Pruning Decoding	8	64.36	2.17 (\downarrow 22.2%)	2031.7	88.5	0.93 (\downarrow 33.1%)	766.8
	Linear Decoding	16	64.54	5.36	5696.4	88.5	2.76	2371.4
	Tree Decoding	16	64.93	2.64 (\downarrow 51.9%)	2356.9	88.8	1.06 (\downarrow 60.7%)	946.7
	Tree Pruning Decoding	16	65.31	2.87 (\downarrow 46.4%)	2563.1	89.1	1.09 (\downarrow 60.5%)	963.9

Table 7: Comparison of different sampling strategies on bird dev and spider dev datasets.

		Simple	Moderate	Challenging	Total
Baseline	EX	67.89	52.26	43.06	60.82
ReEx-SQL	Initial EX	67.03	50.97	43.75	59.97
	Final EX	69.51	55.48	49.31	63.36
	Δ EX	+2.48	+4.51	+5.56	+3.39
	Avg. # of SQLs	1.07	1.21	1.24	1.13

Table 8: Comparison across different difficulty levels.

Method	Success	Failed	All
DIN-SQL+GPT3.5 (Pourreza and Rafiei, 2023)	61	113	-52
MAC-SQL+GPT-4o (Wang et al., 2025a) + MapleRepair (Shen et al., 2025b)	24	1	23
	65	13	52
ReEx-SQL 7B	57	0	57

Table 9: Comparison of SQL Repair Results

ample, with 16 candidates, tree pruning decoding achieves the highest EX scores (65.3% on BIRD Dev and 89.1% on Spider Dev), surpassing linear decoding by up to 1.2%. Additionally, tree-based methods reduce latency and computational cost, where cost is measured as the average number of output tokens generated per data item. Tree decoding with 16 candidates lowers latency by over 50% and cost by more than half compared to linear decoding, and tree pruning further improves EX scores with only minor increases in latency and cost. Overall, tree pruning decoding offers the balance between accuracy and efficiency, mitigating interaction-related latency and enhancing performance. Additional results on Spider variants are provided in Appendix A.2.1.

4.7 Effectiveness of Multi-Turn Interaction

Table 8 compares the execution accuracy of ReEx-SQL and the baseline across different query difficulty levels. Initial EX refers to the performance of ReEx-SQL on the first SQL generated during interaction, while Final EX reflects the accuracy after all rounds of multi-turn correction using execution

Model	Decoding	Candidates = 8		Candidates = 16	
		Voting	Pass@8	Voting	Pass@16
Baseline	Linear	62.70	67.93	63.16	68.71
ReEx-SQL	Linear	64.08	70.80	64.54	71.06
ReEx-SQL	Tree	64.15	70.99	64.93	72.03
ReEx-SQL	Tree pruning	64.36	71.71	65.31	72.75

Table 10: Voting and pass@k results on the BIRD Dev.

feedback. ReEx-SQL significantly improves EX through this iterative process, with the largest gains observed on challenging queries (up to +5.56%), demonstrating its strength in refining and correcting SQL via multi-turn interaction.

Table 9 evaluates the SQL repair capabilities of each method on their initial SQL outputs. Error correction modules in DIN-SQL (Pourreza and Rafiei, 2023) and MAC-SQL (Wang et al., 2025a) can mis-modify correct SQL, leading to additional errors: 113 for DIN-SQL, and 1 and 13 for MAC-SQL and MapleRepair (Shen et al., 2025b), respectively. In contrast, ReEx-SQL refines the initial SQL through multi-turn, end-to-end interaction, successfully repairing all 57 erroneous cases without introducing new errors. This highlights the reliability and precision of ReEx-SQL’s interactive approach, which delivers the greatest overall improvement across both challenging and routine queries.

4.8 Pass@k Metric

We further report voting-based and pass@k execution accuracy (EX, %) on the BIRD dev set under different candidate budgets. As shown in Table 10, ReEx-SQL consistently outperforms the baseline across both voting and pass@k metrics for all candidate settings. In particular, ReEx-SQL achieves substantial improvements in pass@8 and pass@16, indicating that it is more effective at generating high-quality candidate sets that contain correct so-

Feedback Type	Count	Percentage
Correct feedback	1475	86.61%
Error feedback (all)	228	13.39%
- Timeout (only)	25	1.47%
- Null/empty-result related (only)	95	5.58%
- Syntax-related (only)	108	7.32%

Table 11: Feedback type distribution on BIRD Dev.

lutions. Furthermore, comparing different decoding strategies within ReEx-SQL, tree-based decoding slightly improves over linear decoding, while incorporating pruning yields consistent additional gains (Table 10).

Notably, tree pruning improves pass@8 from 70.99% to 71.71% and pass@16 from 72.03% to 72.75%, corresponding to a +0.72% gain over standard tree decoding in both settings. This result demonstrates that pruning enhances candidate selection by filtering out low-quality branches, thereby improving the likelihood of including correct executions among the generated candidates.

4.9 Distribution of Error Types

To contextualize the robustness results discussed above, we analyze the composition of execution feedback on BIRD Dev. As shown in Table 11, correct executions dominate the feedback stream, accounting for the vast majority of cases, while error feedback constitutes only a relatively small fraction. Within the error category, the most frequent types correspond to *syntax-related issues* and *null/empty-result situations*. Specifically, syntax-related errors (7.32%) and null/empty-result cases (5.58%) together make up most of the error feedback, whereas timeout cases are comparatively rare (1.47%), and no locked cases are observed in this evaluation setting (Table 11).

A more fine-grained breakdown of syntax-related errors is presented in Table 12. Among these, missing-column errors are the dominant subtype, accounting for 4.82% of all instances, followed by skeleton/structure errors (1.47%) and function-related errors (0.41%), while missing-table errors occur only marginally (0.12%).

Overall, these statistics suggest that most instances already yield valid execution outputs, with errors being both limited in proportion and concentrated in a few common categories (Table 11). Consequently, improving robustness to missing or partial feedback primarily depends on effectively leveraging informative error messages for correc-

Syntax-related Type	Count	Percentage
Syntax-related	108	7.32%
- Missing column (only)	82	4.82%
- Skeleton/structure error (only)	17	1.47%
- Function error (only)	7	0.41%
- Missing table (only)	2	0.12%

Table 12: Breakdown of syntax-related errors.

tion, rather than relying on fine-grained distinctions within relatively infrequent error types (Table 12).

5 Conclusion

We propose ReEx-SQL, a framework that integrates real-time execution feedback into decoding, enabling progressive refinement of reasoning through structured prompting and iterative database interaction. By extending GRPO with execution-aware rollouts and a composite reward for format, execution, semantics, and exploration, ReEx-SQL achieves strong in-domain and out-of-domain performance, outperforming similarly sized models and some proprietary solutions. The tree-structured decoding strategy further reduces inference time while maintaining or improving accuracy, demonstrating the efficiency of execution-guided reasoning. Overall, ReEx-SQL shows that execution feedback can effectively guide reasoning, improving performance of small open-source LLMs, and narrowing the gap with proprietary solutions.

Limitation

Although ReEx-SQL demonstrates significant improvements by leveraging real-time execution feedback, its dependence on direct database access may restrict its applicability in certain scenarios, such as offline environments or those with privacy constraints. Future research may address this limitation by developing lightweight execution simulators or fallback mechanisms, enabling broader deployment. Moreover, exploring advanced decoding strategies could further enhance reasoning efficiency, especially in settings with limited resources.

References

- Lukasz Borchmann and Marek Wydmuch. 2025. [Query and conquer: Execution-guided SQL generation](#). *CoRR*, abs/2503.24364.
- Jikai Chen. 2025. [Sqlcritic: Correcting text-to-sql generation via clause-wise critic](#). *CoRR*, abs/2503.07996.

- Mingyang Chen, Tianpeng Li, Haoze Sun, Yijie Zhou, Chenzheng Zhu, Haofen Wang, Jeff Z. Pan, Wen Zhang, Huajun Chen, Fan Yang, Zenan Zhou, and Weipeng Chen. 2025. Research: Learning to reason with search for llms via reinforcement learning. *CoRR*, abs/2503.19470.
- Jinho Choo, Yeong-Dae Kwon, Jihoon Kim, Jeongwoo Jae, André Hottung, Kevin Tierney, and Youngjune Gwon. 2022. [Simulation-guided beam search for neural combinatorial optimization](#). In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.
- Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2024. Navigate through enigmatic labyrinth A survey of chain of thought reasoning: Advances, frontiers and future. In *ACL (1)*, pages 1173–1203. Association for Computational Linguistics.
- Yaxun Dai, Haiqin Yang, Hao Mou, and Pingfu Chao. 2025. [PARSQL: enhancing text-to-sql through SQL parsing and reasoning](#). In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 661–681. Association for Computational Linguistics.
- DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 81 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/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, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 1337–1350. Association for Computational Linguistics.
- Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Lu Chen, Jinshu Lin, and Dongfang Lou. 2023. [C3: zero-shot text-to-sql with chatgpt](#). *CoRR*, abs/2307.07306.
- Yuankai Fan, Tonghui Ren, Can Huang, Zhenying He, and X. Sean Wang. 2024. Grounding natural language to SQL translation with data-based self-explanations. *CoRR*, abs/2411.02948.
- 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, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 2505–2515. Association for Computational Linguistics.
- 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, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8926–8931. Association for Computational Linguistics.
- 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.*, 17(5):1132–1145.
- Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. 2024b. [Xiyan-sql: A multi-generator ensemble framework for text-to-sql](#). *CoRR*, abs/2411.08599.
- Mingqian He, Yongliang Shen, Wenqi Zhang, Qiuying Peng, Jun Wang, and Weiming Lu. 2025. [Star-sql: Self-taught reasoner for text-to-sql](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 24365–24375. Association for Computational Linguistics.
- Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. [Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes](#). In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 8003–8017. Association for Computational Linguistics.
- Jian Hu, Xibin Wu, Weixun Wang, Xianyu, Dehao Zhang, and Yu Cao. 2024. [Openrlhf: An easy-to-use, scalable and high-performance RLHF framework](#). *CoRR*, abs/2405.11143.
- Bowen Jin, Hansi Zeng, Zhenrui Yue, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. [Search-r1: Training llms to reason and leverage search engines with reinforcement learning](#). *CoRR*, abs/2503.09516.
- Yonghui Kong, Hongbing Hu, Dan Zhang, Siyuan Chai, Fan Zhang, and Wei Wang. 2025. [Bridging the gap: Transforming natural language questions into SQL queries via abstract query pattern and contextual schema markup](#). *CoRR*, abs/2502.14682.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. [Efficient memory management for large language model](#)

- serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.
- Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. MCS-SQL: leveraging multiple prompts and multiple-choice selection for text-to-sql generation. In *COLING*, pages 337–353. Association for Computational Linguistics.
- Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024a. The dawn of natural language to SQL: are we fully ready? [experiment, analysis \u0026 benchmark]. *Proc. VLDB Endow.*, 17(11):3318–3331.
- Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025a. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. *CoRR*, abs/2502.17248.
- 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. 2025b. Omnisql: Synthesizing high-quality text-to-sql data at scale. *CoRR*, abs/2503.02240.
- Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024b. Codes: Towards building open-source language models for text-to-sql. *Proc. ACM Manag. Data*, 2(3):127.
- Jinyang Li, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. 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 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- Hanbing Liu, Haoyang Li, Xiaokang Zhang, Ruotong Chen, Haiyong Xu, Tian Tian, Qi Qi, and Jing Zhang. 2025a. Uncovering the impact of chain-of-thought reasoning for direct preference optimization: Lessons from text-to-sql. *CoRR*, abs/2502.11656.
- Hanbing Liu, Haoyang Li, Xiaokang Zhang, Ruotong Chen, Haiyong Xu, Tian Tian, Qi Qi, and Jing Zhang. 2025b. Uncovering the impact of chain-of-thought reasoning for direct preference optimization: Lessons from text-to-sql. *CoRR*, abs/2502.11656.
- Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024a. A survey of NL2SQL with large language models: Where are we, and where are we going? *CoRR*, abs/2408.05109.
- Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Yuyu Luo, and Nan Tang. 2024b. A survey of nl2sql with large language models: Where are we, and where are we going? *CoRR*, abs/2408.05109.
- Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. 2025c. NL2sql-bugs: A benchmark for detecting semantic errors in NL2SQL translation. *CoRR*, abs/2503.11984.
- Shuai Lyu, Haoran Luo, Zhonghong Ou, Yifan Zhu, Xiaoran Shang, Yang Qin, and Meina Song. 2025. Sql-o1: A self-reward heuristic dynamic search method for text-to-sql. *CoRR*, abs/2502.11741.
- Haoyuan Ma, Yongliang Shen, Hengwei Liu, Wenqi Zhang, Haolei Xu, Qiuying Peng, Jun Wang, and Weiming Lu. 2025a. Db-explore: Automated database exploration and instruction synthesis for text-to-sql. *CoRR*, abs/2503.04959.
- Peixian Ma, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. 2025b. SQL-R1: training natural language to SQL reasoning model by reinforcement learning. *CoRR*, abs/2504.08600.
- Simone Papicchio, Simone Rossi, Luca Cagliero, and Paolo Papotti. 2025. Think2sql: Reinforce LLM reasoning capabilities for text2sql. *CoRR*, abs/2504.15077.
- Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*.
- Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- Mohammadreza Pourreza and Davood Rafiei. 2024. DTS-SQL: decomposed text-to-sql with small large language models. In *EMNLP (Findings)*, pages 8212–8220. Association for Computational Linguistics.
- Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, and Sercan Ö. Arik. 2025. Reasoning-sql: Reinforcement learning with SQL tailored partial rewards for reasoning-enhanced text-to-sql. *CoRR*, abs/2503.23157.
- Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. Toolrl: Reward is all tool learning needs. *CoRR*, abs/2504.13958.
- Yang Qin, Chao Chen, Zhihang Fu, Ze Chen, Dezhong Peng, Peng Hu, and Jieping Ye. 2024. ROUTE: robust multitask tuning and collaboration for text-to-sql. *CoRR*, abs/2412.10138.

- Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! A novel and effective strategy for mitigating hallucinations in text-to-sql generation. In *ACL (Findings)*, pages 5456–5471. Association for Computational Linguistics.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *NeurIPS*.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300.
- Jiawei Shen, Chengcheng Wan, Ruoyi Qiao, Jiazhen Zou, Hang Xu, Yuchen Shao, Yueling Zhang, Weikai Miao, and Geguang Pu. 2025a. A study of in-context-learning-based text-to-sql errors. *CoRR*, abs/2501.09310.
- Jiawei Shen, Chengcheng Wan, Ruoyi Qiao, Jiazhen Zou, Hang Xu, Yuchen Shao, Yueling Zhang, Weikai Miao, and Geguang Pu. 2025b. A study of in-context-learning-based text-to-sql errors. *CoRR*, abs/2501.09310.
- Zhili Shen, Pavlos Vougiouklis, Chenxin Diao, Kaushtubh Vyas, Yuanyi Ji, and Jeff Z. Pan. 2024. Improving retrieval-augmented text-to-sql with ast-based ranking and schema pruning. In *EMNLP*, pages 7865–7879. Association for Computational Linguistics.
- Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. 2022. Defining and characterizing reward hacking. *CoRR*, abs/2209.13085.
- Huatong Song, Jinhao Jiang, Yingqian Min, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, and Ji-Rong Wen. 2025. R1-searcher: Incentivizing the search capability in llms via reinforcement learning. *CoRR*, abs/2503.05592.
- SQLite. n.d. [Sqlite home page](#). Accessed: 2025-05-10.
- Ruoxi Sun, Sercan Ö. Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2024. Sql-palm: Improved large language model adaptation for text-to-sql. *CoRR*, abs/2306.00739.
- Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: contextual harnessing for efficient SQL synthesis. *CoRR*, abs/2405.16755.
- Gladys Tyen, Hassan Mansoor, Victor Carbune, Peter Chen, and Tony Mak. 2024. Llms cannot find reasoning errors, but can correct them given the error location. In *ACL (Findings)*, pages 13894–13908. Association for Computational Linguistics.
- Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025a. [MAC-SQL: A multi-agent collaborative framework for text-to-sql](#). In *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, pages 540–557. Association for Computational Linguistics.
- Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2023. [MAC-SQL: A multi-agent collaborative framework for text-to-sql](#). *CoRR*, abs/2312.11242.
- Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. 2025b. Agentar-scale-sql: Advancing text-to-sql through orchestrated test-time scaling. *arXiv preprint arXiv:2509.24403*.
- 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 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.
- Wenxuan Xie, Yaxun Dai, and Wenhao Jiang. 2025. Sde-sql: Enhancing text-to-sql generation in large language models via self-driven exploration with sql probes. *arXiv preprint arXiv:2506.07245*.
- 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 *COLING*, pages 4282–4292. 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), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 7864–7875. Association for Computational Linguistics.
- John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. [Intercode: Standardizing and benchmarking interactive coding with execution feedback](#). In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. [Tree of thoughts: Deliberate problem solving with large language models](#). In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2025. [Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql](#). *CoRR*, abs/2505.20315.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. 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, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921. Association for Computational Linguistics.

Shuozhi Yuan, Liming Chen, Miaomiao Yuan, Jin Zhao, Haoran Peng, and Wenming Guo. 2025. [MCTS-SQL: an effective framework for text-to-sql with monte carlo tree search](#). *CoRR*, abs/2501.16607.

Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. 2025. [Excot: Optimizing reasoning for text-to-sql with execution feedback](#). *CoRR*, abs/2503.19988.

Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, and 1 others. 2025. [The landscape of agentic reinforcement learning for llms: A survey](#). *arXiv preprint arXiv:2509.02547*.

Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023. [ACT-SQL: in-context learning for text-to-sql with automatically-generated chain-of-thought](#). In *EMNLP (Findings)*, pages 3501–3532. Association for Computational Linguistics.

Weixu Zhang, Yifei Wang, Yuanfeng Song, Victor Junqiu Wei, Yuxing Tian, Yiyan Qi, Jonathan H. Chan, Raymond Chi-Wing Wong, and Haiqin Yang. 2024. [Natural language interfaces for tabular data querying and visualization: A survey](#). *IEEE Trans. Knowl. Data Eng.*

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. [Semantic evaluation for text-to-sql with distilled test suites](#). In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 396–411. Association for Computational Linguistics.

Yizhang Zhu, Runzhi Jiang, Boyan Li, Nan Tang, and Yuyu Luo. 2025. [Elliesql: Cost-efficient text-to-sql with complexity-aware routing](#). *CoRR*, abs/2503.22402.

A Appendices

Algorithm 1: LLM Response Rollout with Stepwise SQL Execution Feedback

Input: Question x , database DB , policy model π_θ , SQL executor \mathcal{E} , maximum interactions N
Output: Final response y , final SQL s_{final}

- 1 Initialize the model output $y \leftarrow \emptyset$
- 2 Initialize interactions $n \leftarrow 0$
- 3 **while** $n < N$ **do**
- 4 Initialize the current round model output tokens
 $y_{\text{temp}} \leftarrow \emptyset$
- 5 **while** **True** **do**
- 6 Generate the next tokens fragment
 $y_t \leftarrow \pi_\theta(\cdot | x, y + y_{\text{temp}})$
- 7 $y_{\text{temp}} \leftarrow y_{\text{temp}} + y_t$
- 8 **if** y_t in [`</intermediate_sql>`, `<im_end>`]
9 **then**
| **break**
- 10 $y \leftarrow y + y_{\text{temp}}$
- 11 **if** `<intermediate_sql>`...`</intermediate_sql>` in
12 y_{temp} **then**
Intermediate SQL query $s_{\text{explore}} \leftarrow$
Extract(y_{temp} , pattern="<intermediate_sql>...</intermediate_sql>")
Execution Feedback
 $e \leftarrow \mathcal{E}(s_{\text{explore}}, DB)$
 $y \leftarrow y + \text{<result> } e \text{ </result>}$
- 16 **else**
- 17 **break**
- 18 Number of interactions $n \leftarrow n + 1$
- 19 Final SQL $s_{\text{final}} \leftarrow$ Extract(y ,
pattern="<final_sql>...</final_sql>")
- 20 **return** Final response y , Final SQL s_{final}

A.1 Prompt Templates

This section presents the prompts corresponding to the three decoding strategies evaluated in Table 4. The execution-aware reasoning prompt, shown in Figure 8, explicitly incorporates execution feedback to guide the generation process; this prompt is used by ReEx-SQL. The standard reasoning prompt, shown in Figure 9, performs multi-step reasoning without relying on execution signals, and serves as the decoding strategy for the baseline. In contrast, the direct output prompt in Figure 10 generates SQL queries directly, without any intermediate reasoning steps.

A.2 More Details about Experiments

A.2.1 Decoding Strategy Comparison on Spider Variants

Table 13 extends the comparison in Table 7 by evaluating decoding strategies on various Spider robustness benchmarks. Consistent with previous

Decoding Strategy	Number of Candidates	Spider Test	Spider-Syn		Spider-Realistic		Spider-DK
		EX (%)	EX (%)	TS (%)	EX (%)	TS (%)	EX (%)
Greedy decoding	1	86.4	78.5	72.1	85.0	78.7	79.3
Linear decoding	8	86.6	78.3	71.6	84.6	78.5	78.9
Tree decoding	8	86.4	78.3	71.7	85.8	79.5	79.1
Tree Pruning decoding	8	86.7	78.4	71.9	85.6	79.5	79.3
Linear decoding	16	86.5	78.5	71.7	85.0	78.7	79.4
Tree decoding	16	86.6	78.6	72.1	85.2	79.1	79.8
Tree Pruning decoding	16	86.9	78.7	72.0	85.5	79.9	80.2

Table 13: Comparison of decoding strategies on spider variants robustness benchmarks.

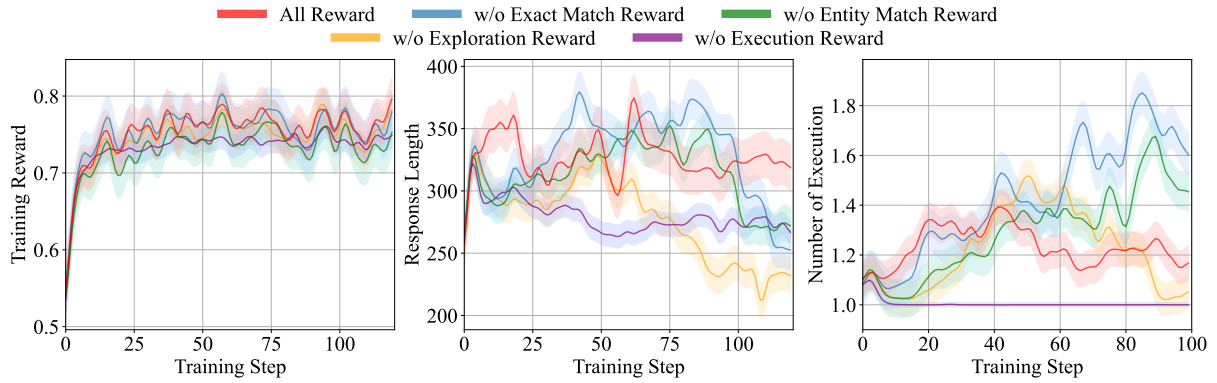


Figure 5: Impact of reward components on training reward (left), response length (middle), and interaction steps (right).

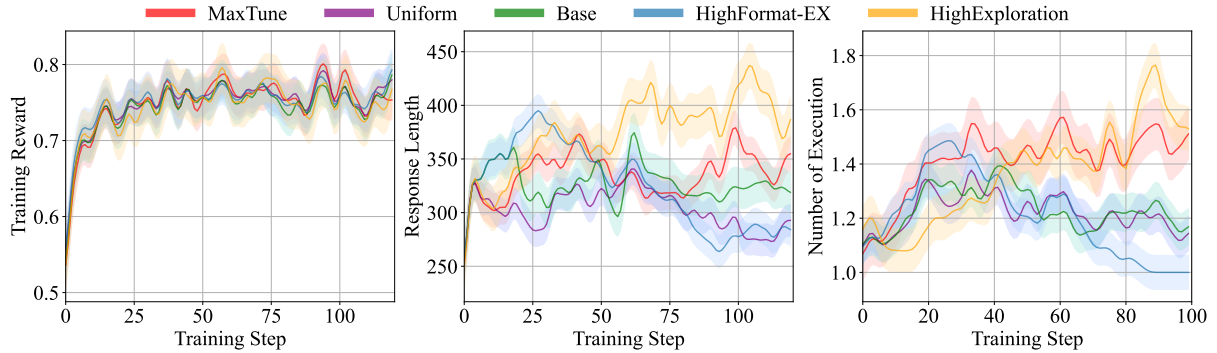


Figure 6: Impact of different reward weight settings on training reward (left), response length (middle), and interaction steps (right).

Database	Nums	BIRD Dev (EX%)				Spider Test (EX%)				Spider-DK (EX%)					
		Simple	Moderate	Challenging	Total	Easy	Medium	Hard	Extra	All	Easy	Medium	Hard	Extra	All
Origin	3	69.5	55.5	49.3	63.4	93.8	89.0	82.1	76.2	86.4	88.2	85.4	62.2	67.6	79.3
Origin	6	69.4	55.7	49.3	63.4	94.0	89.0	82.3	76.5	86.6	88.2	85.8	64.9	66.7	79.6
Origin	8	69.4	55.9	50.0	63.5	93.8	89.0	82.3	76.2	86.5	88.2	85.8	64.9	67.8	79.8
TS-DB	3	-	-	-	-	94.0	89.0	82.7	76.5	86.7	88.2	86.2	64.9	67.6	80.0

* All results are with greedy decoding. TS-DB refers to the augmented databases for TS score evaluation (Zhong et al., 2020), from Spider and its variants.

Table 14: Impact of execution feedback and database complexity on ReEx-SQL.

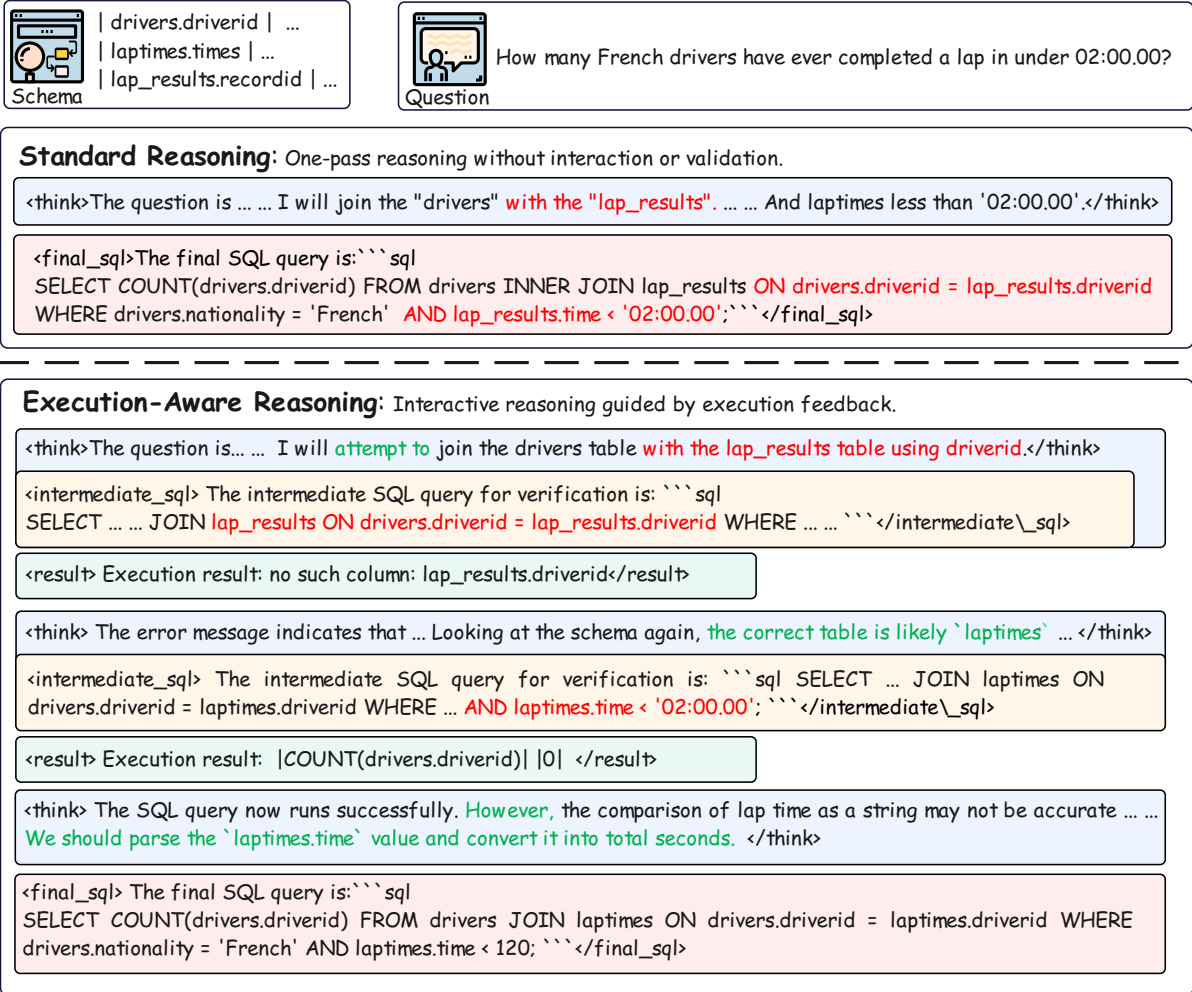


Figure 7: Output comparison: execution-aware (ReEx-SQL) vs. standard reasoning (Baseline).

findings, Tree-structured self-consistency continues to outperform or match other strategies in both effectiveness and robustness. It achieves the highest execution accuracy on Spider-Realistic (85.5%) and Spider-DK (80.2%), which are more challenging due to their paraphrased questions and unseen schema structures. Even when using only 8 candidates, the tree-structured approach performs competitively; and with 16 candidates, it yields further improvements across nearly all metrics. These results reinforce that tree-structured decoding generalizes well under distribution shifts and diverse linguistic perturbations, making it a strong choice for real-world deployment scenarios where robustness is critical.

A.2.2 Impact of Execution Feedback and Database

To investigate how the richness of execution feedback affects performance, we compare different result row limits and database settings, as shown

in Table 14. “Origin” refers to the default evaluation database, while TS-DB contains more diverse cell values. “Nums” indicates the number of rows returned in each execution result. Results show that increasing the number of feedback rows improves performance—for example, raising Nums from 3 to 8 boosts the EX score on BIRD Challenging by 0.7% (49.3% vs. 50.0%). Additionally, using TS-DB leads to further gains, such as a 0.6% improvement on Spider Test Hard (82.1% vs. 82.7%) and a 0.7% gain on Spider-DK (79.3% vs. 80.0%). These findings indicate that richer feedback provides more informative guidance for reasoning, resulting in higher-quality SQL generation, particularly for complex queries.

A.2.3 Impact of Reward Design on Training Dynamics

This section complements Section 4.5 by analyzing the impact of reward design on training dynamics, focusing on two aspects: reward components and

reward weights.

Impact of Reward Components on Training Dynamics Figure 5 shows how the training reward, response length, and interaction count evolve under different reward ablation settings. We highlight the following observations: (1) Training converges rapidly within the first 50 steps and remains stable, indicating that GRPO effectively activates model potential and aligns behavior with the reward signal. (2) Without the execution reward, the average reward is significantly lower, consistent with Table 5, highlighting its essential role in the Text-to-SQL task. (3) Without the exploration reward, response length and interaction count decline in later stages, suggesting the model tends to favor short, low-interaction reasoning paths without explicit exploration incentives. (4) With all reward components present, the model exhibits stable behavior in both response length and interaction count, indicating a synergistic effect that supports more robust training dynamics.

Impact of Reward Weights on Training Dynamics Figure 6 illustrates the impact of different reward weight configurations. Key findings include: (1) Average reward remains similar across configurations, suggesting that the model is robust to moderate changes in reward weights. (2) Increasing the exploration reward weight (HighExploration vs. Base) leads to longer responses and more interactions, showing that stronger incentives promote deeper reasoning. (3) HighFormat-EX increases the format and execution rewards at the expense of exploration, resulting in shorter responses and fewer interactions, indicating that overemphasis on correctness may suppress exploration. (4) MaxTune yields intermediate response length and interaction count, reflecting a balanced trade-off between execution accuracy and exploration.

A.2.4 ReEx-SQL vs. Baseline: Output Analysis

This section compares the reasoning paths of ReEx-SQL and the Baseline to highlight the advantages of Execution-Aware Reasoning. As shown in Figure 7, under the Standard Reasoning paradigm, the model relies solely on its internal knowledge and the given context, often resulting in reasoning errors and incorrect SQL generation. In contrast, Execution-Aware Reasoning allows the model to issue exploratory SQL queries to test its hypotheses and refine its reasoning based on execution

feedback. For instance, in the example shown, the model receives an empty result from an exploratory query and correctly infers that the “time” column is an integer representing seconds rather than a string, leading to a corrected and accurate SQL query.

Beyond this case, ReEx-SQL demonstrates several generalizable behaviors driven by execution feedback. First, it can form and revise hypotheses through exploratory SQL based on observed results (e.g., Figure 11). Second, it is capable of detecting and correcting semantic errors, such as missing conditions, by interpreting empty result sets (e.g., Figures 12 and 13). Third, it can leverage execution error messages to adjust its reasoning, and when uncertain, it may re-run exploratory queries to validate its revisions (e.g., Figures 14 and 15). These examples collectively show that ReEx-SQL moves beyond static context-based reasoning by actively interacting with the environment, resulting in a more robust and adaptive SQL generation process.

A.2.5 Statistical Rigor

To ensure reproducibility and assess result stability, we conduct additional experiments with controlled randomness. All experiments are run using fixed random seeds in vLLM (Kwon et al., 2023), a large-scale LLM inference framework. While most main experiments adopt greedy decoding, for sampling-based settings we report the mean and variance of EX and TS across multiple seeds. Specifically, each configuration is executed four times. The EX score is averaged over the four runs, and results are reported in the format of *mean (minimum, maximum)*. Tables 15 and 16 present the results under candidate sizes of 8 and 16, respectively.

Method	BIRD Dev	Spider Dev
Baseline		
+ Linear decoding	62.66 (-0.08, 0.05)	86.05 (-0.07, 0.12)
ReEx-SQL		
+ Linear decoding	64.08 (-0.06, 0.07)	88.44 (-0.05, 0.05)
+ Tree decoding	64.16 (-0.08, 0.05)	88.42 (-0.02, 0.07)
+ Tree pruning	64.37 (-0.03, 0.03)	88.54 (-0.05, 0.05)

Table 15: Statistical results with 8 candidates.

Overall, the conclusions drawn from the sampling-based experiments are consistent with those obtained under fixed random seeds, indicating that the observed performance gains are stable and not sensitive to stochastic variations.

Method	BIRD Dev	Spider Dev
Baseline		
+ Linear decoding	63.09 (-0.11, 0.08)	86.05 (-0.07, 0.12)
ReEx-SQL		
+ Linear decoding	64.52 (-0.11, 0.15)	88.54 (-0.05, 0.05)
+ Tree decoding	64.99 (-0.06, 0.07)	88.83 (-0.05, 0.05)
+ Tree pruning	65.34 (-0.15, 0.11)	89.14 (-0.07, 0.12)

Table 16: Statistical results with 16 candidates.

A.2.6 Experiments on Spider 2.0

We evaluate ReEx-SQL on the Spider 2.0 benchmark using the schema provided by Omni-SQL for fair comparison, and report results on the Spider-v2.0-SQLite subset. Table 17 summarizes the performance of different models.

Model	Train Dataset	Data Scale	Accuracy
Qwen2.5 Coder-7B	None	None	1.50%
Omni-SQL-7B	Spider + BIRD	17K	3.00%
Baseline	Spider + BIRD	17K	2.96%
ReEx-SQL-7B	Spider + BIRD	17K	3.70%

Table 17: Results on the Spider-v2.0-SQLite subset.

With the same training data, ReEx-SQL consistently outperforms both the standard reasoning baseline and Omni-SQL-7B. We note that the overall accuracy on Spider 2.0 is relatively low due to the increased schema complexity and distribution mismatch between Spider and BIRD. As future work, we plan to expand the training data for ReEx-SQL to further improve its performance on this challenging benchmark.

A.3 Robustness to Execution Feedback

Execution feedback provides an external supervision signal during interactive SQL generation. To study how ReEx-SQL behaves when this signal is partially unavailable or degraded, we run controlled ablations that systematically restrict the feedback content. All results in this section are obtained with greedy decoding on BIRD Dev.

A.3.1 Impact of Feedback Availability

We consider the following feedback settings:

- **Full feedback:** Complete execution feedback, including schema, row values, and error messages.
- **Locked:** All feedback replaced with a locked message, simulating fully restricted DB access.

- **Timeout:** All feedback replaced with a timeout message, simulating unresponsive queries.
- **Error-only:** only error messages are returned; successful executions return a generic success indicator.
- **Schema-only:** Successful executions return schema-related information (e.g., column names) but no cell values; error feedback is preserved.
- **Row-count:** Successful executions return only the number of result rows; error feedback is preserved.

Feedback	Simple	Moderate	Challenging	Total
Full feedback	69.50	55.50	49.30	63.40
Locked	69.19	50.43	48.97	61.60
Timeout	68.86	48.71	46.21	60.63
Error-only	69.19	51.94	48.28	61.99
Schema-only	69.30	53.02	48.97	62.45
Row-count	69.19	52.16	48.97	62.13

Table 18: Greedy decoding results of ReEx-SQL on BIRD Dev under different execution feedback settings.

Overall, ReEx-SQL remains robust when execution feedback is weakened. Performance drops are larger when feedback collapses to non-informative signals (Locked/Timeout), while retaining error messages (Error-only, Schema-only, Row-count) preserves most of the gains. This suggests that informative failure signals play a key role in guiding correction, whereas limiting successful execution feedback (e.g., removing cell values) is less detrimental as long as error feedback is available.

Execution-Aware Reasoning Prompt

You are an experienced database expert. Now you need to generate a SQL query given the database information, a question, and some additional information. Your goal is to generate a single **SQLite** query that can correctly answer the user's question based on the given schema and matched values.

Important Guidelines:

1. The database structure is defined by the following table schemas:
table_name.column_name (data_type | comment : description | values : values)
The “values” are only examples to illustrate the data type and format; They are not directly related to the question.
2. You should:
 - 2.1. **Analyze** the question intent and map it to relevant tables and columns.
 - 2.2. If the question is complex or ambiguous, you may write an **intermediate SQL** for verification.
 - 2.3. **Do not generate an infinite number of intermediate SQLs**. Instead, focus on identifying key areas where the SQL might need improvement and stop when you have sufficient insight.
 - 2.4. When reflecting on the intermediate SQL:
 - First, check whether it **correctly expresses all conditions, columns, and logic** described in the original question.
 - Also, verify whether it follows the Database admin instructions, such as avoiding prohibited patterns or optimizing structure.
 - Then, evaluate whether the **execution result** confirms or contradicts your expectations.
 - If there is any mismatch, misunderstanding, violation of admin rules, or missing condition, **revise your reasoning before generating the final SQL**.
 - In addition, you can generate other possible intermediate SQLs for comparison.

Response Format:

Respond strictly in the following format:

<think> Reasoning process here. **</think>**

<intermediate_sql> The intermediate SQL query for verification is: “sql your intermediate SQL here.” **</intermediate_sql>**

<result> Execution result here. **</result>**

<think> Further reasoning. **</think>**

<final_sql> The final SQL query is: “sql your final SQL here.” **</final_sql>**

[Table creation statements] {database_schema}

[Matched Values] {matched_contents}

[Question] {question} Hint: {evidence}

Figure 8: Execution-aware reasoning prompt for ReEx-SQL.

Standard Reasoning Prompt

You are an experienced database expert. Now you need to generate a SQL query given the database information, a question, and some additional information. Your goal is to generate a single **SQLite** query that can correctly answer the user's question based on the given schema and matched values.

Important Guidelines:

1. The database structure is defined by the following table schemas: **table_name.column_name (data_type | comment : description | values : values)** The "values" are only examples to illustrate the data type and format; They are not directly related to the question.
2. **Analyze** the question intent and map it to relevant tables and columns.
3. First, check whether it **correctly expresses all conditions, columns, and logic** described in the original question.
4. Also, verify whether it follows the Database admin instructions, such as avoiding prohibited patterns or optimizing structure.

Response Format:

Respond strictly in the following format:

<think> Reasoning process here. **</think>**

<final_sql> The final SQL query is: "sql your final SQL here." **</final_sql>**

[Table creation statements] {database_schema}

[Matched Values] {matched_contents}

[Question] {question} Hint: {evidence}

Figure 9: Standard reasoning prompt for baseline.

Direct Output Prompt

You are an experienced database expert. Now you need to generate a SQL query given the database information, a question, and some additional information. Your goal is to generate a single **SQLite** query that can correctly answer the user's question based on the given schema and matched values.

Important Guidelines:

1. The database structure is defined by the following table schemas: **table_name.column_name (data_type | comment : description | values : values)** The "values" are only examples to illustrate the data type and format; They are not directly related to the question.
2. Your task is to generate the correct SQL query **only**. Do not include any explanation, reasoning, or additional text.

Response Format:

Respond strictly in the following format:

<final_sql> The final SQL query is: "sql your final SQL here." **</final_sql>**

[Table creation statements] {database_schema}

[Matched Values] {matched_contents}

[Question] {question} Hint: {evidence}

Figure 10: Direct output prompt.

Question:

The average unemployment ratio of 1995 and 1996, which one has higher percentage?

Hit:

a12 refers to unemployment rate 1995; a13 refers to unemployment rate 1996;

Standard Reasoning:

<think>

The user's question asks for the average unemployment ratio of 1995 and 1996 and which one has a higher percentage. To answer this question, I need to look at the columns a12 (unemployment rate 1995) and a13 (unemployment rate 1996) in the district table. I need to calculate the average of these two columns to compare the percentages. I will start by selecting the columns a12 and a13 from the district table. Then, I will calculate the average of these two columns.

</think>

<final_sql> The final SQL query is: ““sql
SELECT AVG(a12) AS avg_unemployment_1995, AVG(a13) AS avg_unemployment_1996
FROM district;”” </final_sql>

Execution-Aware Reasoning:

<think>

Let's find the average unemployment ratio for both 1995 and 1996 from the 'district' table. We need to calculate the average of 'a12' (unemployment rate 1995) and 'a13' (unemployment rate 1996). Then, we will compare these averages to determine which year has a higher percentage.

</think>

<intermediate_sql>

The intermediate SQL query for verification is: ““sql
SELECT AVG(a12) AS avg_1995, AVG(a13) AS avg_1996 FROM district;””

</intermediate_sql>

<result> avg_1995 | avg_1996 | 3.0723684210526314 | 3.787012987012987 | </result>

<think>

The intermediate SQL query correctly calculates the average unemployment rates for 1995 and 1996. The average unemployment rate for 1996 (3.787) is higher than for 1995 (3.072).

</think>

<final_sql> The final SQL query is: ““sql
SELECT CASE WHEN AVG(a13) > AVG(a12) THEN '1996' ELSE '1995' END AS
higher_unemployment_year FROM district;”” </final_sql>

Figure 11: Reasoning output comparison for example 1.

Question:

Which drivers who were born in 1971 and has the fastest lap time on the race? Give id and code of these drivers.

Standard Reasoning:

<think>

The user's question asks for the drivers who were born in 1971 and had the fastest lap time on a race, along with their id and code. To answer this question, I need to focus on the "dob" column in the "drivers" table, which represents the date of birth of the drivers. I need to find drivers born in 1971, which means I need to look for rows where the year of "dob" is 1971. Additionally, I need to look at the "fastestlaptime" column in the "results" table, which represents the fastest lap time of a driver in a race. I need to join the "drivers" and "results" tables based on the "driverid" column to connect the information about the drivers' birth year and their fastest lap times.

</think>

<final_sql>

The final SQL query is: "sql

```
SELECT drivers.driverid, drivers.code FROM drivers INNER JOIN results ON drivers.driverid = results.driverid WHERE strftime('%Y', drivers.dob) = '1971';"
```

</final_sql>

Execution-Aware Reasoning:

<think>

user is asking for drivers born in 1971 with the fastest lap time in a race. To solve this, we need to join the 'drivers' table with the 'results' table using the 'driverid' column. We then filter for drivers born in 1971 by extracting the year from the 'dob' column. The 'results' table has a 'fastestlaptime' column, which we will use to find the fastest driver. We'll begin with an initial query that selects all such drivers.

</think>

<intermediate_sql>

The intermediate SQL query for verification is: "sql

```
SELECT drivers.driverid, drivers.code, results.fastestlaptime FROM drivers JOIN results ON drivers.driverid = results.driverid WHERE STRFTIME('%Y', drivers.dob) = '1971' ORDER BY results.fastestlaptime ASC LIMIT 1;"
```

</intermediate_sql>

<result>

```
|driverid|code|fastestlaptime|
|14|COU|NULL|
```

</result>

<think>

The intermediate result shows that some 'fastestlaptime' values are NULL. Including NULLs in sorting can lead to incorrect results, as NULL values do not represent valid lap times. Therefore, we must filter out rows where 'fastestlaptime' is NULL before applying ordering and limits. We'll revise the SQL accordingly.

</think>

<final_sql>

The final SQL query is: "sql

```
SELECT drivers.driverid, drivers.code FROM drivers JOIN results ON drivers.driverid = results.driverid WHERE STRFTIME('%Y', drivers.dob) = '1971' AND results.fastestlaptime IS NOT NULL ORDER BY results.fastestlaptime ASC LIMIT 1"
```

</final_sql>

Question:

What was the release date for the set which card "Evacuation" in it?

Standard Reasoning:

<think>

The user's question asks for the release date of the set that contains the card "Evacuation". To answer this question, I need to find the set code for "Evacuation" and then look up its release date. First, I need to locate the card "Evacuation" in the "cards" table by searching for its name. Next, I need to join the "cards" table with the "set_translations" table using the "uuid" column, as the set code is stored in "set_translations". Finally, I need to join the "sets" table with "set_translations" using the "code" column to find the release date of the set.

</think>

<final_sql>

The final SQL query is: "sql
SELECT sets.releasedate FROM sets INNER JOIN set_translations ON sets.code = set_translations.setcode INNER JOIN cards ON set_translations.id = cards.id WHERE cards.name = 'Evacuation';"

</final_sql>

Execution-Aware Reasoning:

<think>

user is asking for the release date of the set that contains the card "Evacuation". To answer this question, we need to join the 'cards' table with the 'sets' table using the 'uuid' column, which is common to both tables. The 'cards' table contains the card name, and the 'sets' table contains the release date of the set. By filtering the 'cards' table for the card named "Evacuation", we can then extract the corresponding release date from the 'sets' table.

</think>

<intermediate_sql>

The intermediate SQL query for verification is: "sql
SELECT sets.releasedate FROM cards JOIN sets ON cards.uuid = sets.mcmid WHERE cards.name = 'Evacuation' "

</intermediate_sql>

<result> lreleaseDate| No data available in the database. </result>

<think>

The previous intermediate SQL query had a mistake in the join condition. The 'sets' table's 'mcmid' column does not match the 'cards' table's 'uuid' column. Instead, we should use the 'sets' table's 'code' column, which is referenced by the 'cards' table's 'setcode' column. Let's correct the join condition and try again.

</think>

<intermediate_sql>

The intermediate SQL query for verification is: "sql
SELECT sets.releasedate FROM cards JOIN sets ON cards.setcode = sets.code WHERE cards.name = 'Evacuation' "

</intermediate_sql>

<result>

lreleaseDate| 2007-07-13| 2003-07-28| 2001-04-11| Omit the sample below.....

</result>

<think>

The intermediate SQL query has been corrected to properly join the 'cards' table with the 'sets' table using the 'setcode' column. The result shows that the release date for the set containing the card 'Evacuation' is '2007-07-13'.

</think>

<final_sql> The final SQL query is: "sql

SELECT sets.releasedate FROM cards JOIN sets ON cards.setcode = sets.code WHERE cards.name = 'Evacuation' "

</final_sql>

Figure 13: Reasoning output comparison for example 3.

Question:

How many posts were created on 21st July, 2010?

Hit:

created on 21st July, 2010 refers to creationdate BETWEEN '2010-07-21 00:00:00' and '2012-07-21 23:59:59';

Standard Reasoning:

<think>

The user's question asks for the count of posts created on 21st July, 2010. To answer this question, I need to look at the creationdate column in the posts table. I need to count the number of rows where the creationdate is between '2010-07-21 00:00:00' and '2010-07-21 23:59:59'. This will give me the number of posts created on that specific date.

</think>

<final_sql>

The final SQL query is: ““sql

```
SELECT COUNT(*) FROM posts WHERE creationdate BETWEEN '2010-07-21 00:00:00' AND '2010-07-21 23:59:59' ”
```

</final_sql>

Execution-Aware Reasoning:

<think>

user wants to find out how many posts were created on July 21, 2010. This requires querying the 'posts' table, specifically the 'creationdate' column, to count the number of posts that fall within the specified date range. The date range is from '2010-07-21 00:00:00' to '2010-07-21 23:59:59'. The SQL query will use the 'COUNT' function to count the number of posts and the 'WHERE' clause to filter the posts based on the 'creationdate'.

</think>

<intermediate_sql>

The intermediate SQL query for verification is: ““sql

```
SELECT COUNT(id) FROM posts WHERE creationdate BETWEEN '2010-07-21 00:00:00' AND '2010-07-21 23:59:59';”
```

</intermediate_sql>

<result>

no such column: creationdate

</result>

<think>

The error indicates that the 'creationdate' column does not exist in the 'posts' table. The 'creationdate' column is actually in the 'posthistory' table, not the 'posts' table. Therefore, the query needs to be adjusted to use the 'posthistory' table instead.

</think>

<final_sql>

The final SQL query is: ““sql

```
SELECT COUNT(id) FROM posthistory WHERE creationdate BETWEEN '2010-07-21 00:00:00' AND '2010-07-21 23:59:59' ”
```

</final_sql>

Figure 14: Reasoning output comparison for example 4.

Question:

Please list the name of the cards in the set Coldsnap with the highest converted mana cost.

Standard Reasoning:

<think>

The user's question asks to list the name of the cards in the set Coldsnap with the highest converted mana cost. To answer this question, I need to find the cards in the set Coldsnap and look at their converted mana costs. I need to join the 'sets' table with the 'cards' table using the 'code' column in 'sets' and the 'uuid' column in 'cards'. I need to find the card with the highest converted mana cost within the Coldsnap set.

</think>

<final_sql>

The final SQL query is: ““sql
SELECT cards.name FROM cards INNER JOIN sets ON cards.uuid = sets.code WHERE sets.name = 'Coldsnap'
ORDER BY cards.convertedmanacost DESC LIMIT 1;”

</final_sql>

Execution-Aware Reasoning:

<think>

user is asking for the names of cards in the set 'Coldsnap' with the highest converted mana cost. To achieve this, we need to join the 'cards' table with the 'sets' table using the 'uuid' column, which is common to both tables. The 'sets' table will help us filter the cards by the set name 'Coldsnap'. We will then sort the cards by their 'convertedmanacost' in descending order to get the card with the highest converted mana cost at the top. Since we are only interested in the name of the card, we will select the 'name' column from the 'cards' table.

</think>

<intermediate_sql>

The intermediate SQL query for verification is: ““sql
SELECT cards.name FROM cards JOIN sets ON cards.uuid = sets.uuid WHERE sets.name = 'Coldsnap' ORDER BY
cards.convertedmanacost DESC LIMIT 1;”

</intermediate_sql>

<result> no such column: sets.uuid </result>

<think>

The error indicates that we cannot directly join the 'cards' table with the 'sets' table using the 'uuid' column because the 'sets' table does not have a 'uuid' column. Instead, we should use the 'code' column from the 'sets' table to match with the 'code' column in the 'cards' table. Let's correct the intermediate SQL query accordingly.

</think>

<intermediate_sql>

The intermediate SQL query for verification is: ““sql
SELECT cards.name FROM cards JOIN sets ON cards.setcode = sets.code WHERE sets.name = 'Coldsnap' ORDER
BY cards.convertedmanacost DESC LIMIT 1;”

</intermediate_sql>

<result> Inamel |Sunscount| </result>

<think>

The intermediate SQL query has been corrected to join the 'cards' table with the 'sets' table using the 'setcode' column, which is common to both tables. The query filters the cards by the set name 'Coldsnap' and sorts them by their 'convertedmanacost' in descending order. The result confirms that the card with the highest converted mana cost in the 'Coldsnap' set is 'Sunscount'.

</think>

<final_sql>

The final SQL query is: ““sql
SELECT cards.name FROM cards JOIN sets ON cards.setcode = sets.code WHERE sets.name = 'Coldsnap' ORDER
BY cards.convertedmanacost DESC LIMIT 1”

</final_sql>

Figure 15: Reasoning output comparison for example 5.