Reward-SQL: Boosting Text-to-SQL via Stepwise Execution-Aware Reasoning and Process-Supervised Rewards

YUXIN ZHANG, Renmin University of China, China
MEIHAO FAN, Renmin University of China, China
JU FAN*, Renmin University of China, China
MINGYANG YI, Renmin University of China, China
YUYU LUO, HKUST (GZ), China
GUOLIANG LI, Tsinghua University, China
BIN WU, Alibaba Cloud Computing, China
WENCHAO ZHOU, Alibaba Cloud Computing, China

Recent advances in large language models (LLMs) trained with reinforcement learning (RL) have improved Text-to-SQL performance. However, RL-based approaches still struggle with complex queries due to two key limitations: insufficient stepwise execution-aware reasoning grounded in database feedback, and the lack of process-level rewards for guiding reasoning optimization. To address these issues, we propose CoCTE, a divide-and-conquer and execution-aware reasoning framework that progressively composes SQL queries through intermediate view validation and structured Common Table Expressions (CTEs), improving both accuracy and interpretability. To realize a CoCTE reasoning process, we develop Reward-SQL, a unified approach with three stages: (1) model initialization, which equips LLMs with structured CoCTE reasoning capabilities; (2) process reward design, which delivers fine-grained, execution-aware supervision; and (3) process-supervised RL and inference, which integrates process rewards into training and guides the inference stage by process rewards. This paper addresses the core challenges in Reward-SQL and makes the following contributions. We introduce a process reward model (PRM) that combines execution-aware trajectory scoring with entropy-based step weighting, providing dense and interpretable supervision across reasoning steps. We integrate PRM into both RL training and inference stages, stabilizing optimization and improving trajectory exploration with process-level signals. Experiments show that Reward-SQL significantly outperforms baselines with comparable model sizes, and exhibits strong cross-domain generalization.

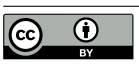
CCS Concepts: • **Computing methodologies** → **Natural language processing**; *Knowledge representation and reasoning*.

Additional Key Words and Phrases: Text-to-SQL, Process Reward Models, Large Language Models

ACM Reference Format:
Yuxin Zhang, Meihao Fan, Ju Fan, Mingyang Yi, Yuyu Luo, Guoliang Li, Bin Wu, and Wenchao Zhou. 2026. Reward-SQL: Boosting Text-to-SQL via Stepwise Execution-Aware Reasoning and Process-Supervised Rewards. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 228 (June 2026), 27 pages. https://doi.org/10.1145/3802105

*Ju Fan is the corresponding author.

Authors' Contact Information: Yuxin Zhang, yuxin.zhang@ruc.edu.cn, Renmin University of China, Beijing, China; Meihao Fan, fmh1art@ruc.edu.cn, Renmin University of China, Beijing, China; Ju Fan, fanj@ruc.edu.cn, Renmin University of China, Beijing, China; Mingyang Yi, yimingyang@ruc.edu.cn, Renmin University of China, Beijing, China; Yuyu Luo, yuyuluo@hkust-gz.edu.cn, HKUST (GZ), Guangzhou, China; Guoliang Li, liguoliang@tsinghua.edu.cn, Tsinghua University, Beijing, China; Bin Wu, wubeen@gmail.com, Alibaba Cloud Computing, Hangzhou, China; Wenchao Zhou, zwc231487@alibaba-inc.com, Alibaba Cloud Computing, Hangzhou, China.



ACM 2836-6573/2026/6-ART228
https://doi.org/10.1145/3802105

1 Introduction

Text-to-SQL translates natural language (NL) questions into executable SQL queries, allowing non-technical users to interact with relational databases [8, 11, 15, 28, 32, 54, 58]. Despite recent advances in large language models (LLMs), generating complex SQL queries based on LLMs (see a recent survey [31]) still remains a significant challenge, particularly for queries involving multi-table joins and nested structures over complex databases [20, 55].

RL-based Approaches and Their Limitations. Recent work has explored using reinforcement learning (RL) as a post-training strategy to enhance LLMs' reasoning and SQL generation capabilities [27, 35, 41, 51, 56]. The core idea is to enhance LLMs' reasoning for Text-to-SQL by decomposing complex query generation into *intermediate reasoning steps* and leveraging *feedback-driven optimization* to better align model outputs with accurate SQL queries. To achieve this, these methods typically employ policy optimization algorithms, such as Grouped Reinforcement Policy Optimization (GRPO) in Reasoning-SQL [41] and Direct Preference Optimization (DPO) in Ex-CoT variants [53], to guide models through structured and step-by-step reasoning. Despite these advances, current RL-based Text-to-SQL approaches face two key limitations:

(1) Limited Stepwise Execution-Aware Reasoning. Most RL-based Text-to-SQL approaches [30, 35, 41, 51] enhance model reasoning by introducing intermediate reasoning steps before generating the final SQL query. However, these steps are not *execution-aware*: the model does not interact with the database during generation and only observes inconsistencies after the final SQL is executed. The lack of *stepwise execution signals* limits the model's performance on complex SQL generation.

(2) Lack of Process-Supervised Rewards. Existing approaches [30, 41, 51] primarily rely on outcome-supervised rewards, where feedback is provided after the final SQL query is generated. While such rewards encourage execution correctness, the lack of fine-grained, *process-level supervision* hinders effective adjustment of the reasoning trajectory and leads to error accumulation in complex queries.

Empirical Evidence from Existing RL Approaches. To empirically validate these limitations, we conduct a preliminary study by training a Qwen3-8B model using GRPO with outcome-only rewards on the BIRD training set. We found this baseline achieved only 63.0% execution accuracy on BIRD Dev. set. Figure 1 shows the error distribution of the GRPO baseline. The most frequent errors involve incorrect filter conditions (161 cases, 32.8%), schema linking errors including table selection (115 cases, 23.4%) and column selection (81 cases, 16.5%). These failures stem from a common root cause: the model generates the entire SQL query in one pass without observing intermediate database states. Without stepwise execution feedback, the model cannot verify whether its table joins are correct, or whether filtering conditions align with actual data distributions. This limited ability to ground reasoning in intermediate execution results leads to cascading errors that only become apparent after the final query fails.

Example 1.1. Figure 2(a) illustrates a real failure case from our Qwen3-8B GRPO baseline. The NL question asks for the top 5 players with the highest average runs per match in season 5. The model generates logically sound reasoning steps: (1) aggregate runs by player, (2) count matches per player, (3) calculate average, (4) sort and select top 5. However, the generated SQL contains a critical error: it unnecessarily joins the `player` table, causing **duplication of run counts** when aggregating. Specifically, the formula `SUM(runs) / COUNT(DISTINCT match_id)` fails because the erroneous JOIN inflates the numerator while the denominator cannot correct this over-counting. This mistake stems from the model's inability to observe intermediate results during generation. Without executing an intermediate step to verify the join result, the model cannot detect that its JOIN strategy produces duplicates. By the time the final SQL executes and returns incorrect results, it is too late to identify which step failed—despite the reasoning being logically sound. □

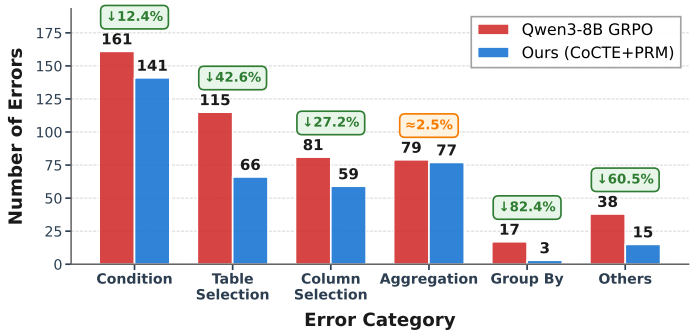
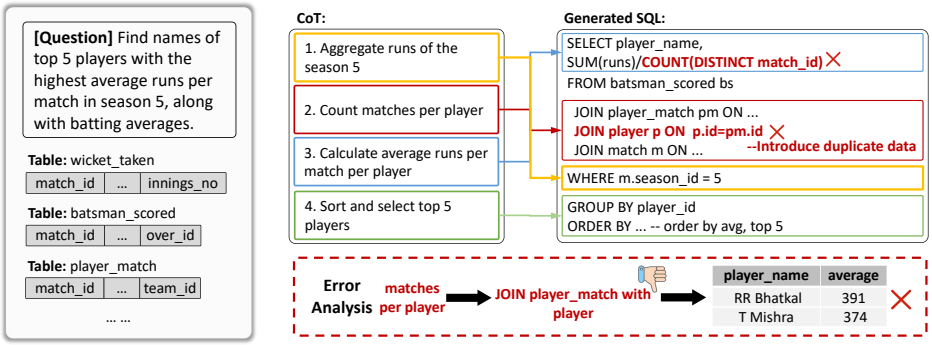
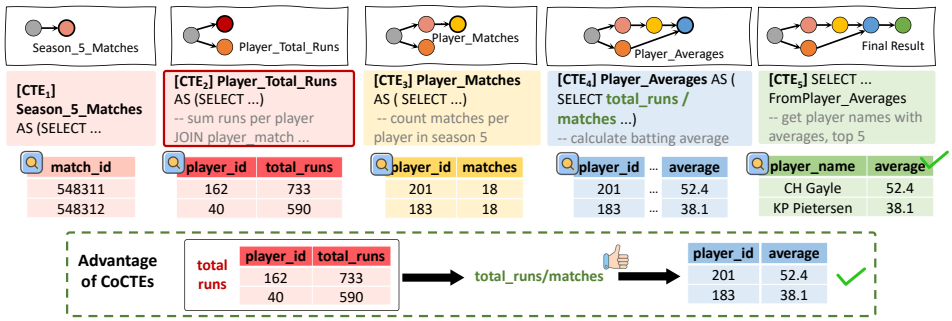


Fig. 1. Error distribution comparison between Qwen3-8B GRPO baseline (red) and our approach (blue).



(a) Traditional Reasoning Framework: introducing intermediate reasoning steps before generating the final SQL query.



(b) Our Proposed CoCTE: breaking a structurally complex SQL query into a sequence of short and executable subqueries, *i.e.,* CTEs.

Fig. 2. A real example from the Spider 2.0 benchmark (id:local026-4) [20] comparing traditional Chain-of-Thought (CoT) reasoning by Qwen3-8B GRPO baseline with our proposed framework CoCTE.

CoCTE: A Divide-and-Conquer and Execution-Aware Reasoning Framework. We propose CoCTE, a divide-and-conquer and execution-aware reasoning framework for complex SQL generation. The key idea is inspired by how experienced database engineers write SQL queries: instead of producing a single and complex SQL query at once, they build SQL progressively, i.e., **creating intermediate views to validate intermediate results and structuring logic with Common Table Expressions (CTEs) [1, 3]**. Each CTE serves as a building block that can be validated and

reused, enabling the final query to be built by systematically composing verified components rather than generating a complex SQL query at once.

Following this idea, our CoCTE decomposes a complex query into a sequence of executable CTEs. Specifically, each step is executed as it is generated, providing *stepwise execution-aware feedback* that grounds reasoning in database. This feedback enables *process-level supervision*, allowing the model to detect and correct errors during generation, thus significantly improving the accuracy and interpretability of complex SQL generation.

Example 1.2. Figure 2(b) illustrates the step-by-step reasoning of CoCTE for example in Figure 2(a). CoCTE divides the complex SQL generation process into four executable steps: (1) $\texttt{CTE}_1$ filters season 5 matches from base tables. (2) $\texttt{CTE}_2$ computes total runs per player from base tables. (3) $\texttt{CTE}_3$ counts matches per player based on $\texttt{CTE}_1$. (4) $\texttt{CTE}_4$ calculates averages per player from $\texttt{CTE}_2$ and $\texttt{CTE}_3$. Finally, the model generates an SQL query to obtain player names with the highest average runs from the results of $\texttt{CTE}_4$.

Through stepwise execution-aware feedback, CoCTE observes that each player participates in multiple matches and computes the average as `total_runs/matches`, a subtle but crucial step missed by existing methods. By decomposing a structured complex SQL query into a sequence of subqueries, CoCTE simplifies generation while improving accuracy and interpretability. □

Our Proposed Reward-SQL Approach. To effectively realize the proposed CoCTE reasoning framework, we present Reward-SQL, a unified approach that integrates stepwise execution-aware reasoning with process-supervised learning. Specifically, Reward-SQL is designed with three stages for complex SQL generation.

Stage 1 - Model Initialization. Unlike standard instruction-following, off-the-shelf LLMs lack the ability to follow CoCTE-style reasoning. To fix this, Reward-SQL synthesizes CoCTE-formatted training data and fine-tunes the LLM to learn structured reasoning trajectories, providing a desired initialization for downstream RL process.

Stage 2 - Process Reward Design. We design reliable process-level rewards that deliver fine-grained, execution-aware supervision at each reasoning step. Instead of providing feedback after the SQL is completely generated, these rewards are tied to intermediate execution signals, enabling the model to align reasoning with database.

Stage 3 - Process-Supervised RL and Inference. We integrate the designed process rewards into the RL stage, enabling execution-aware supervision throughout training while maintaining alignment with final outcomes. During inference, these rewards guide both efficient selection and structured exploration of reasoning trajectories, enhancing the model's ability to generate accurate queries.

Challenges and Solutions. Effectively realizing the proposed Reward-SQL solution raises two key technical challenges.

The first challenge is designing a process rewards tailored to CoCTE reasoning. As discussed above, process rewards aim to overcome the limitations of sparse outcome-based signals in existing RL-based Text-to-SQL approaches [35, 41, 51]. To achieve this goal, we define a process reward model (PRM) that evaluates intermediate trajectories generated within CoCTE. This design introduces two challenges: (1) estimating rewards for each intermediate trajectory based on execution-aware feedback from the corresponding CTE, and (2) weighting rewards across trajectories according to their relative contributions to the final query. To address the challenges, we design a composite process reward consisting of: (1) a trajectory scoring model that estimates the correctness of intermediate trajectories, enabling step-level reward estimation; and (2) inverse entropy weighting that emphasizes informative steps and allocates rewards across trajectories accordingly.

The second challenge is integrating the process reward into RL training, which requires a careful balance between the dense reward signal and training stability. An improper combination results

in divergent output or reward hacking [44], where the model over-optimizes high process scores instead of producing the correct final answers. To address this, during RL, we introduce a unified objective that properly integrates the process and outcome reward signals, leading to a correct and stable training process. Moreover, in inference stage, we use process rewards to enable Best-of-N sampling to select high-quality trajectories.

Effectiveness of Our Approach. As in Figure 1, our Reward-SQL approach substantially reduces errors across all categories through two complementary mechanisms. *Stepwise execution-aware reasoning* validates schema choices at each CTE, reducing table selection errors by 42.6% and column selection errors by 27.2%. *Process-supervised rewards* provide fine-grained feedback during generation, proving particularly effective for complex operations like GROUP BY (82.4% reduction). Together, these mechanisms transform sparse outcome feedback into dense step-level supervision.

Contributions. We summarize our contributions as follows.

(1) We introduce CoCTE, a *divide-and-conquer, execution-aware reasoning framework* that decomposes complex queries into a sequence of executable CTEs (Section 4).

(2) We propose Reward-SQL to effectively realize CoCTE by addressing the key technical challenges of process reward design and process-supervised RL and inference (Sections 5 and 6).

(3) We conduct extensive experiments on standard Text-to-SQL benchmarks. Results show that Reward-SQL achieves superior performance on BIRD, reaching 70.3% execution accuracy with an 8B model and outperforming baselines with comparable parameter sizes. Moreover, Reward-SQL demonstrates strong cross-domain generalization, maintaining high performance on five out-of-distribution benchmarks without retraining (Section 7). [1]

2 Preliminaries

2.1 Text-to-SQL

The Text-to-SQL task maps a natural language (NL) question $q \in \mathcal{Q}$ and a database schema $\mathcal{S}$ to an executable SQL query $y \in \mathcal{Y}$. We denote the schema as $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{R})$, where $\mathcal{T}$, $\mathcal{C}$, and $\mathcal{R}$ are the sets of tables, columns, and inter-table relations, respectively. Given $(q, \mathcal{S})$, a Text-to-SQL model generates y, which is then executed on a database instance $\mathcal{D}$. For example, as shown in Figure 2, given the question "*Find names of the top 5 players with the highest average runs per match in season 5, along with batting averages*", a Text-to-SQL model reasons over multiple tables and constructs a complex SQL query involving filtering, joining, and aggregation, which can be challenging due to multi-table joins and nested structures.

2.2 Common Table Expressions (CTEs)

A **Common Table Expression (CTE)** defines a temporary, named result set within a single SQL query, enabling modular and structured construction of complex queries. Conceptually, a CTE functions similarly to a lightweight, inline *view* as it is defined once and can be referenced multiple times within the same query, without requiring persistent schema changes. A minimal form is:

```
WITH cte_name AS ( SELECT ... )
SELECT ... FROM cte_name;
```

By encapsulating intermediate results as reusable logical views, CTEs allow a complex query to be decomposed into smaller, verifiable subqueries. This modular structure aligns naturally with the stepwise reasoning representation adopted in our CoCTE framework, making it easier to verify intermediate steps and compose them into the final query.

[1] Code is available at https://github.com/ruc-datalab/RewardSQL

2.3 Design of CoCTE

Recently, Chain-of-Thought (CoT) [49] has been applied to Text-to-SQL to encourage a step by step reasoning. However, existing CoT-based approaches [9, 21, 41, 51] decompose reasoning solely in the natural language, without grounding intermediate steps in database feedback. As in Figure 2, this lack of execution awareness prevents the model from verifying whether each step reflects the actual database state, causing reasoning errors to accumulate and ultimately leading to incorrect SQL generation. Overcoming this limitation requires a reasoning representation that bridges linguistic abstraction and relational composition in SQL, enabling semantic subgoals to be immediately realized as a verifiable SQL fragments.

Formalization of CoCTE. To this end, we introduce Chain of Common Table Expressions (**CoCTE**). As mentioned previously, a CTE defines a named, temporary result set within a query, which serves as a modular and interpretable computation unit. Because each CTE can be independently executed and verified, it naturally aligns with how human engineers construct queries, by decomposing problems into smaller, verifiable components.

Formally, a CoCTE trajectory interleaves natural language reasoning with executable SQL fragments:

$$\mathcal{T} = \{(r_1, q_1), (r_2, q_2), \ldots, (r_k, q_k), (r_f, q_f)\}$$

where r_i is the natural language rationale for step i and q_i is the corresponding CTE. The final query q_f composes all intermediate results to produce the answer.

In Figure 2, CoCTE decomposes the query generation process into modular, stepwise reasoning steps: CTE_1 retrieves season 5 matches, CTE_2 computes total runs per player, CTE_3 counts matches per player, and CTE_4 calculates averages before the final selection. By executing and composing these intermediate subqueries, the model maintains a correct logical flow, and mitigates structural errors common in single-pass generation.

2.4 Rewards in Reinforcement Learning

In reinforcement learning (RL), the objective is to optimize a policy through reward signals that measure task performance. The following two primary forms of reward are commonly used.

Outcome Reward (OR). The model receives a single reward after completing the entire task, reflecting only the final success or failure. Formally, R_{out} provides binary or scalar feedback based on the final outcome. While this reward aligns well with the end objective, its sparse and delayed nature makes credit assignment difficult, as it offers no signal about which intermediate decisions contributed to success or error.

Process Reward (PR). Unlike outcome rewards, process rewards provide step-level feedback that evaluates intermediate actions or reasoning quality. Each step a_i in a trajectory receives a reward $r_i = f_\phi(a_i, a_{<i}) \in [0, 1]$, enabling the model to assign credit to specific parts of the reasoning process. This dense supervision improves learning efficiency and stabilizes optimization, but also requires reliable step evaluation to avoid introducing reward bias.

3 Overview of Reward-SQL

Figure 3 presents our proposed Reward-SQL approach, which unifies SQL-level decomposition with process rewards to enhance both training and inference for Text-to-SQL. Given an NL question q and a database schema $\mathcal{S}$, Reward-SQL generates an executable SQL query by constructing a structured CoCTE trajectory $\mathcal{T}$. To this end, Reward-SQL consists of the following three stages.

Stage 1: Model Initialization. We first introduce Chain-of-CTEs (namely CoCTE), a structured reasoning format that incrementally constructs complex SQL queries through a sequence of intermediate CTEs. Unlike conventional CoT, which remains purely linguistic, CoCTE grounds

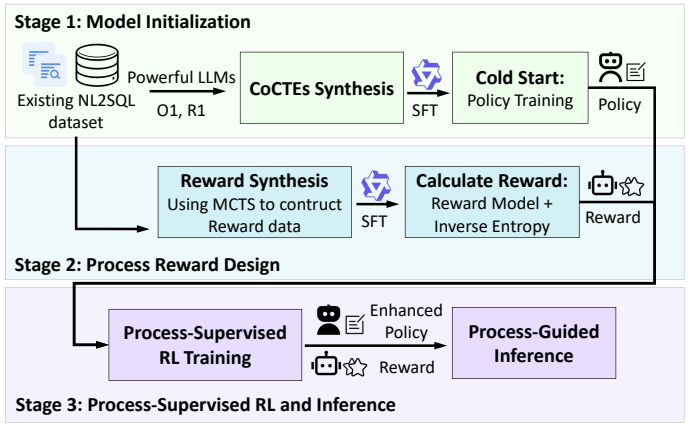


Fig. 3. Overview of the REWARD-SQL framework. It consists of three stages: (1) Model Initialization, which enables the model to generate structured CoCTE-style reasoning; (2) Process Reward Design, which provides stepwise rewards for intermediate trajectories; and (3) Process-Supervised RL and Inference, which integrates process rewards into both training and inference.

each reasoning step in an executable subquery, enabling stepwise verification and modular query construction. Since standard instruction-following LLMs do not inherently produce such structured trajectories, we construct a dedicated training corpus through a semi-automatic pipeline: a strong generator (DEEPSEEK-V3.1) converts existing NL2SQL queries from BIRD [25] and SPIDER [52] into CoCTE trajectories, guided by a small set of manually annotated seed examples. The policy model π_θ is then initialized via supervised fine-tuning on this corpus, equipping it with the ability to generate coherent, stepwise CoCTE trajectories rather than a single SQL query. This structured generation provides a natural foundation for applying step-level supervision in the subsequent RL training stages (see Section 4 for details).

Stage 2: Process Reward Design. Building on CoCTE, we construct a *process reward* R_{proc} that delivers process supervision by aggregating intermediate correctness with uncertainty-aware weighting, which consists of two complementary components:

- **Trajectory Score Model (R_ϕ)** is trained using labels generated via Monte Carlo Tree Search (MCTS) to estimate the correctness of each intermediate trajectory $\mathcal{T}_{\leq i}$. Specifically, for each NL question, MCTS explores multiple full CoCTE trajectories and propagates the terminal execution outcomes back through the search tree to compute Monte Carlo estimates of step-level score for each intermediate node.
- **Inverse Entropy Weight (IH)** measures each intermediate trajectory's contribution to reducing reasoning uncertainty. Trajectories with lower entropy (i.e., higher certainty) receive larger weights, emphasizing informative steps over redundant yet valid ones.

By combining R_ϕ and IH, R_{proc} converts sparse outcome-only feedback into process-supervised signals, enabling fine-grained supervision throughout the reasoning process (see Section 5).

Stage 3: Process-Supervised RL Training and Inference. We integrate the process reward into both the RL training and inference. During training, R_{proc} is combined with the outcome reward R_{out} under the GRPO framework, providing stepwise supervision for policy optimization. During inference, R_{proc} acts as a dynamic guidance signal: in Best-of-N sampling, it selects the trajectory with the highest process reward among N candidates. The above integration of process rewards allows the model to benefit from process-level supervision throughout both training and inference. More details are provided in Section 6.

4 Cold Start for CoCTE Policy Model

As discussed in Section 2.3, CoCTE provides a better reasoning paradigm. However, LLMs will not output this pattern as expected. Oppositely, the existing Text-to-SQL model outputs either a single SQL statement or a SQL generated after CoT that consists of natural language. Thus, to adapt LLM to the CoCTE format, we apply a *cold-start supervised training process* to the policy model (a LLM) π_θ on structured CoCTE trajectories.

Semi-Automatic Corpus Construction. We construct the training corpus through a pipeline as below. Given the natural-language question $q \in \mathcal{Q}$, database schema $\mathcal{S}$, and gold SQL query $y^\star \in \mathcal{Y}$ from the training set as inputs, we first create some handcrafted CoCTE as examples to prompt a strong generator (DeepSeek-V3.1) [4] to transform SQL queries from the Bird [25] and Spider [52] training sets into CoCTE trajectories. To improve coverage and diversity, each input question is sampled three times at temperature 1.0, producing multiple candidate trajectories that explore different reasoning paths. Each generated output is then parsed into an abstract syntax tree (AST) and filtered to ensure structural consistency—removing cases with malformed CTE blocks, undefined references, or naming conflicts. The resulting dataset provides syntactically and semantically valid CoCTE trajectories.

Table 1 presents statistics of the constructed corpus. From 17,462 initial SQL queries, the pipeline produces 51,996 valid CoCTE trajectories with a 90.7% acceptance rate after filtering. The average trajectory contains 3.3 CTE steps, with rationales averaging 28.1 tokens and CTEs averaging 11.6 tokens, demonstrating the granular decomposition achieved by the CoCTE format.

Table 1. Statistics of the CoCTE corpus constructed for cold-start training. The dataset combines Bird and Spider queries transformed via a semi-automatic pipeline.

Category	Metric	Value
Data Scale	# Manually annotated seeds	5
	# Initial SQL queries	17,462
	# Valid CoCTEs (post-filtering)	51,996
	Acceptance rate	90.7%
Structure Statistics	Avg. CTE steps	3.3
	Max. CTE steps	9
	Avg. rationale length	28.1 tokens
	Avg. CTE length	11.6 tokens

Supervised Fine-Tuning We fine-tune the policy model π_θ on this constructed corpus to imitate the reasoning pattern of CoCTE by minimizing a next-token prediction loss [18]:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_t \log \pi_\theta((r_t^\star, q_t^\star) \mid (r_{<t}^\star, q_{<t}^\star), q, \mathcal{S}),$$

where each $(r_t^\star, q_t^\star)$ denotes a paired intermediate step of NL rationale and executable CTE.

By doing so, the policy model generates coherent, executable CoCTE trajectories instead of isolated SQL or textual explanations. The model learns to alternate between semantic planning (rationales) and structural realization (CTEs), producing outputs that are both interpretable and executable. More importantly, such structured CoCTE provides a natural basis for step-level supervision: each intermediate CTE can be independently executed and evaluated, enabling the process reward design described in the next Section.

5 Process Reward Design

5.1 Motivation and Formulation

The existing RL approaches for Text-to-SQL mostly rely on the *Outcome Reward* (OR), a binary variable of execution correctness:

$$R_{\text{out}}(y) = \begin{cases} 1, & \text{Exe}(y, \mathcal{D}) = \text{Exe}(y^\star, \mathcal{D}), \\ 0, & \text{otherwise.} \end{cases}$$

Although conceptually straightforward, this reward is extremely sparse and non-decomposable. Even if most intermediate CTEs in a CoCTE trajectory are correct, a single error in the final clause nullifies the entire signal, providing no clue as to which reasoning step failed. This "all-or-nothing" feedback amplifies the credit assignment problem and leads to high-variance policy gradients, often scaling super-linearly with the reasoning horizon [42, 50].

To overcome these limitations, we redefine the reward formulation from a *terminal* outcome to a *process-oriented* metric. Concretely, we introduce the **Process Reward**, a dense reward signal that distributes credit across each executable step of a CoCTE trajectory. For a reasoning trajectory $\mathcal{T} = \{(r_i, q_i)\}_{i=1}^{k}$, we define the process reward over each intermediate reasoning trajectory $\mathcal{T}_{\leq d} = \{q_1, \cdots, q_d\}$ $(d \leq k)$ as:

$$R_{\text{proc}}(\mathcal{T}_{\leq d}) = \sum_{i=1}^{d} \left(\frac{\sum_{j \leq i} IH(\mathcal{T}_{\leq j}) \cdot R_\phi(\mathcal{T}_{\leq j})}{\sum_{j \leq i} IH(\mathcal{T}_{\leq j})} \right). \tag{1}$$

Intuitively, this formulation is an **uncertainty-reweighted sum** of intermediate trajectory correctness, where:

- $R_\phi(\mathcal{T}_{\leq j})$: **Trajectory Score Model** that estimates the correctness of intermediate trajectory $\mathcal{T}_{\leq j}$
- $IH(\mathcal{T}_{\leq j})$: **Inverse Entropy Weight** that quantifies the contribution of trajectory $\mathcal{T}_{\leq j}$ to reducing uncertainty

This design ensures that our process reward captures both *semantic correctness* (via R_ϕ) and *reasoning efficiency* (via IH). Steps that are both correct and informative receive higher credit, while redundant or uninformative steps are automatically down-weighted. Next, we describe how each component is constructed.

5.2 Trajectory Score Model

To estimate the quality of CTE steps, we train a Trajectory Score Model R_ϕ that maps the reasoning trajectory to a confidence score:

$$s_i = R_\phi(\mathcal{T}_{\leq i} \mid q, \mathcal{S}) \in (0, 1).$$

A higher s_i indicates stronger confidence that the current CTE q_i is locally valid and globally consistent with the target query. The core challenge lies in obtaining accurate step-level supervision without human annotation. Inspired by the use of Monte Carlo estimation in mathematical reasoning [33], we adopt a Monte Carlo Tree Search (MCTS) strategy to generate step-level labels automatically.

Concretely, for each question in the Bird training set, we sample n trajectories from the cold-started policy model π_{ref} and perform MCTS to explore the space of CoCTE completions.

Each node in the search tree corresponds to an intermediate trajectory $\mathcal{T}_{\leq i}$, and the leaf nodes represent complete SQL candidates whose execution yields binary rewards $R_{\text{out}} \in \{0, 1\}$. To ensure sufficient diversity, we use $n = 8$ rollouts with temperature $T = 1.0$, yielding an average of 38 distinct root-to-leaf trajectories per question, effectively preventing collapse to repeated paths. By

backing up these terminal outcomes through the tree, we compute a Monte Carlo estimate of each intermediate step's expected utility:

$$\sigma_i = \mathbb{E}_{\pi_{\text{ref}}}[R_{\text{out}} \mid \mathcal{T}_{\le i}] = \sum_{\mathcal{T}_{\le i} \subseteq \tau} p(\tau \mid \mathcal{T}_{\le i})\, R_{\text{out}}(\tau).$$

Intuitively, $\sigma_i \in [0, 1]$ can be viewed as the success rate from intermediate trajectory $\mathcal{T}_{\le i}$ to the end. To mitigate noise in σ_i, we apply syntax-level filters to remove malformed CTEs and resolve name conflicts, ensuring structural validity.

In our CoCTE structure, each CTE step is paired with its execution output using `SELECT * FROM` queries, exposing both syntactic structure and runtime semantics to the model. The model input thus interleaves each CTE fragment, its execution result, and a `<step>` marker token indicating the prediction checkpoint. Given these step-labeled samples, the Trajectory Score Model $R_\phi(\mathcal{T}_{\le i} \mid q, \mathcal{S})$ is optimized via binary cross-entropy:

$$\mathcal{L}_{\text{PRM}} = -\sum_{i=1}^{k} \big[\sigma_i \log s_i + (1 - \sigma_i) \log(1 - s_i)\big].$$

By doing so, the trajectory score model provides correctness estimation on reasoning trajectories. We implement the Trajectory Score Model R_ϕ using Qwen2.5-7B-Coder-Instruct as the base architecture, training it independently without parameter sharing with the policy model. However, correctness is insufficient to capture a step's *usefulness*. For example, a redundant step can be correct in syntactic sense, while it may fail to advance the overall reasoning (e.g., redundant joins or irrelevant filters).

5.3 Inverse Entropy Weight

During the construction of the training set, we observe that the Trajectory Score Model indeed sometimes rewards *spurious but valid* steps—CTEs that are syntactically correct and executable yet fail to advance the overall reasoning. To distinguish genuinely contributive steps from merely valid ones, we incorporate an additional process-level signal derived from the entropy of policy model.

At each generation step, the entropy of the policy model over next tokens $\pi_\theta(\cdot \mid \mathcal{T}_{\le i})$ is

$$H(a \mid \mathcal{T}_{\le i}) = -\sum_{a} \pi_\theta(a \mid \mathcal{T}_{\le i}) \log \pi_\theta(a \mid \mathcal{T}_{\le i}),$$

where a denotes a single token from the vocabulary, which measures the model's uncertainty over the next token: a lower $H(a \mid \mathcal{T}_{\le i})$ corresponds to less uncertainty. Empirically, we find that entropy tends to decrease monotonically along successful reasoning steps as in Figure 4.

Based on this observation, we propose to assign higher weights to informative (low entropy) intermediate steps. We define the entropy of step $q_{i+1} = (a_{i+1}^1, \cdots, a_{i+1}^{|q_{i+1}|})$ as:

$$H(q_{i+1} \mid \mathcal{T}_{\le i}) = \frac{1}{|q_{i+1}|} \sum_{j=1}^{|q_{i+1}|} H\left(a_{i+1}^j \mid \mathcal{T}_{\le i}, a_{i+1}^{<j}\right),$$

where $|q_{i+1}|$ is the sequence length. Then, the entropy of intermediate trajectory $\mathcal{T}_{\le i} = \{q_1, \cdots, q_i\}$ is defined as $H(\mathcal{T}_{\le i}) = \sum_{j=1}^{i} H(q_j \mid \mathcal{T}_{<j})$.

We then normalize the trajectory entropy as $\tilde{H}(\mathcal{T}_{\le i}) = \frac{H(\mathcal{T}_{\le i})}{\max_i H(\mathcal{T}_{\le i})}$, and define the *Inverse Entropy*:

$$IH(\mathcal{T}_{\le i}) = 1 - \tilde{H}(\mathcal{T}_{\le i}). \tag{2}$$

A larger inverse entropy $IH(\mathcal{T}_{\le i})$ indicates a more certain intermediate trajectory, meaning the prior steps have effectively constrained the reasoning space and provided strong guidance for

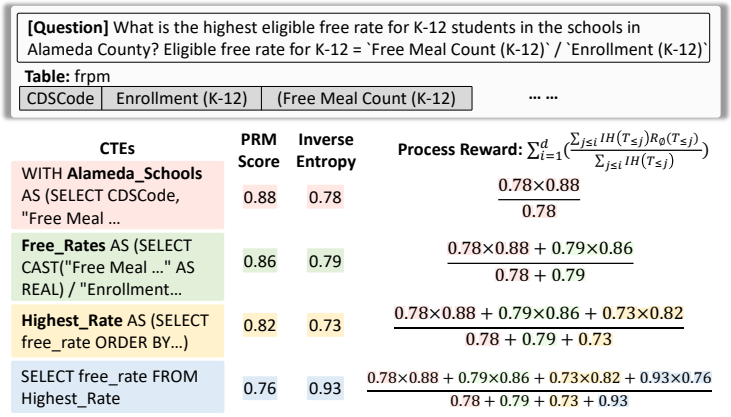


Fig. 4. The computation of process reward, which is an inverse entropy weighted sum of trajectory scores.

subsequent generation. This inverse entropy serves as the weight in Equation 1, ensuring that informative steps contribute more to the overall process reward.

5.4 Summary and Example

By combining the Trajectory Score Model R_ϕ with the Inverse Entropy weight IH, the process reward design achieves two complementary goals: (1) verifiable correctness through the trajectory scoring model R_ϕ, and (2) reward allocation that reflects the exploratory contribution of each intermediate step via the IH weights.

Example 5.1. Figure 4 illustrates the process reward computation for the Alameda County query. The trajectory consists of four CTE steps with their corresponding Trajectory Scores and entropy values and following Equation 1, we can compute the process reward at each step. Notably, the first CTE (`Alameda_Schools`) has both high correctness (0.88) and low inverse entropy (0.78), indicating it is both valid and informative, thus receiving strong positive signal. In contrast, `Highest_Rate` has slightly lower correctness and higher entropy, resulting in reduced contribution to the overall reward. The final SELECT step exhibits very high inverse entropy (0.93), reflecting high certainty after all intermediate reasoning steps.

5.5 Decomposition Quality Analysis

While CoCTE trajectories provide verifiable intermediate results, they may also introduce structural redundancy or non-optimal decompositions. To understand this phenomenon and validate our inverse-entropy weighting mechanism, we manually analyzed 500 randomly sampled trajectories from the cold-start corpus.

Redundancy Statistics. We categorize trajectories into two groups: those containing redundant CTEs (e.g., duplicate joins, unnecessary filtering steps) and those with minimal redundancy. As shown in Table 2, approximately 37% (184/500) of trajectories exhibit some form of structural redundancy. These include cases where intermediate CTEs perform semantically equivalent operations or fail to progressively narrow the result set.

Entropy as Quality Indicator. Crucially, we observe that redundant trajectories exhibit higher average entropy (0.3142 vs. 0.2966) and correspondingly lower inverse entropy (0.47 vs. 0.50). This validates our design intuition: when the policy model generates redundant steps, it exhibits greater uncertainty in token-level predictions, reflected in elevated entropy values. By incorporating

Table 2. Entropy statistics for 500 sampled trajectories with and without redundancy. Redundant trajectories exhibit higher entropy, which our inverse-entropy weighting mechanism down-weights during training.

Metric	With Redundancy	Without Redundancy
Sample Count	184	316
Avg. Entropy	0.3142	0.2966
Avg. Inverse Entropy	0.47	0.50

inverse-entropy weighting in Equation 1, our PRM automatically assigns lower credit to such uncertain steps, mitigating their negative impact during RL training.

6 Process-Supervised RL and Inference

6.1 Process-Supervised RL Training

As demonstrated in recent work [21, 22, 35, 36, 41], reinforcement learning post-training significantly improves Text-to-SQL capabilities. However, as discussed in Section 1, existing RL approaches rely on sparse Outcome Reward (OR) signals that provide only binary feedback on final execution correctness. This sparsity leads to unstable training dynamics and poor credit assignment, particularly for multi-step CoCTE trajectories.

To address this limitation, we integrate our process reward (Equation 1) into the RL optimization objective, providing dense, step-level supervision throughout the reasoning process.

Unified Process Reward Objective. Formally, the RL optimization objective is maximizing the expected reward:

$$J(\theta) = \mathbb{E}_{(\mathcal{T},y)\sim\pi_\theta(\cdot|q,\mathcal{S})}\left[R_{\text{proc}}(\mathcal{T}) + R_{\text{out}}(y) - \lambda D_{KL}(\pi_\theta \parallel \pi_{\text{ref}})\right], \tag{3}$$

Here $D_{KL}(\pi_\theta \parallel \pi_{\text{ref}})$ penalizes the policy model away from the initialized pre-trained model π_{ref}, and $\lambda > 0$ is a hyperparameter. This formulation transforms reinforcement learning from sparse outcome-level supervision into dense, interpretable, step-level guidance. Each reasoning step in an CoCTE trajectory contributes its own verifiable credit, allowing gradients to flow through intermediate reasoning processes rather than being determined solely by final execution success.

Integration into GRPO. We conduct the RL process under the GRPO framework [44], which has been proven to be an effective and stable RL training framework.

Given query $(q, \mathcal{S})$, the GRPO samples G trajectories $\{(\mathcal{T}_g, y_g)\}_{g=1}^{G}$ from the current policy model $\pi_\theta(,y \mid q, \mathcal{S})$. Then for every token $a_{g,i}^j$ in step $q_{g,i}$, we define the reward

$$A_{g,i} = R_{out}(y_g) + \sum_{k=i}^{d}\left(\frac{\sum_{j\le k} IH(\mathcal{T}_{g,\le j})R_\phi(\mathcal{T}_{g,\le j})}{\sum_{j\le k} IH(\mathcal{T}_{g,\le j})}\right),$$

and standardize it across the group as $\hat{A}_i = \frac{A_i - \mu_R}{\sigma_R}$, where μ_R and σ_R are the mean and standard deviation of all A_i in groups. The GRPO maximizes the following loss:

$$\begin{aligned}\mathcal{L}_{\text{GRPO}} = &\\ &\frac{1}{G}\sum_{g=1}^{G}\frac{1}{\sum|a_{g,i}^j| + |y_g|}\sum_{q_{g,i}\in\mathcal{T}_g}\sum_j \min\left(\rho_j\hat{A}_j,\ \text{clip}(\rho_j, 1\pm\epsilon)\hat{A}_j\right)\\ &- \lambda D_{KL}(\pi_\theta \parallel \pi_{\text{ref}}),\end{aligned} \tag{4}$$

where $\rho_j = \pi_\theta(a_{g,i}^j \mid q, \mathcal{S}, \mathcal{T}_{<g}, a_{g,i}^{<j})/\pi_{\text{ref}}(a_{g,i}^j \mid q, \mathcal{S}, \mathcal{T}_{<g}, a_{g,i}^{<j})$, the KL divergence is estimated as in [44], and $\sum|a_{g,i}^j|$ represents the number of tokens in trajectory $\mathcal{T}_g$. During training, the PRM

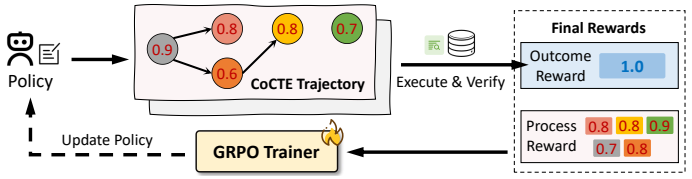


Fig. 5. The GRPO with process reward signals integrated.

parameters ϕ remain frozen, while only policy π_θ is updated. This converts process-level feedback into a differentiable supervision signal, enabling stable credit assignment with reduced variance.

6.2 Process-guided Inference

Inference-Time Process Guidance. Beyond training, the process reward also serves as a principled mechanism for inference-time guidance. During inference, reasoning trajectories are generated auto-regressively, and the process reward provides step-level quality estimates that can guide trajectory selection and exploration. Unlike existing Text-to-SQL systems that rely on heuristic methods such as majority voting or self-consistency [7], our approach leverages the trained process reward to evaluate and rank candidate reasoning paths based on their semantic correctness and reasoning efficiency.

We integrate the process reward into a widely used inference strategies: Best-of-N sampling.

Best-of-N Decoding. The policy model π_θ generates N CoCTE trajectories for a given query. Each candidate trajectory $\mathcal{T}$ is assigned a process reward $R_{\text{proc}}(\mathcal{T})$ by (1). Then, the trajectory with the highest process reward is selected as the final output:

$$\mathcal{T}^* = \arg \max_{\mathcal{T} \in \{\mathcal{T}_1, \ldots, \mathcal{T}_N\}} R_{\text{proc}}(\mathcal{T}).$$

This selection strategy replaces the heuristic voting mechanisms with a principled, learned evaluation metric that accounts for both correctness and reasoning quality.

In summary, the process reward serves as a dynamic inference-time controller that enables principled trajectory selection. By replacing static heuristics with learned, step-level evaluations, our approach completes a closed loop where the model learns from process supervision during training and reasons with process evaluation during inference.

7 Experiments

As summarized in Figure 3, our Reward-SQL framework consists of three stages: (1) *Cold start*, which guides the policy model to generate reasoning in the CoCTE format; (2) *Process reward-assisted RL post-training*, which improves the generation probability on trajectories with higher process rewards; and (3) *Process reward-assisted inference*, which leverages process rewards to select high-quality trajectories during inference. In this section, we conduct a comprehensive empirical evaluation of the proposed Reward-SQL.

7.1 Experimental Setup

Evaluation Datasets. We evaluate Reward-SQL on a suite of benchmarks covering both cross-domain and domain-specific SQL reasoning. For standard evaluation, we adopt two representative large-scale cross-domain datasets: BIRD [25] and Spider [52]. BIRD features industrial-scale databases with rich numerical and textual attributes (9,428 train, 1,534 dev, 1,789 test), where database schemas are strictly disjoint across splits. Spider follows the same cross-domain split protocol (7,000 train, 1,034 dev, 2,147 test) and remains the de-facto benchmark for complex multi-table

SQL generation. Unless otherwise specified, all reported results are measured on the filtered BIRD development and Spider test sets.

Generalization Benchmarks. To assess model generalization beyond the training distribution, we evaluate on five out-of-distribution (OOD) benchmarks spanning two levels of domain shift: *Robustness-level generalization* is tested on three Spider-based variants that preserve schema familiarity while introducing linguistic variations: **Spider-DK** [13] (535 samples) tests inference of implicit domain knowledge; **Spider-Syn** [12] (1,034 samples) replaces schema mentions with synonyms to examine lexical robustness; and **Spider-Realistic** [6] (508 samples) uses real-user questions with practical ambiguities. *Cross-domain generalization* is evaluated on two domain-specific benchmarks with entirely different schemas and specialized terminology: **ScienceBenchmark** [57] (299 samples) covers research policy, astrophysics, and cancer studies with scientific databases, while **EHRSQL** [19] (1,008 samples) focuses on clinical databases requiring medical domain expertise. None of these five benchmarks are used during training, representing strict zero-shot evaluation.

Model and Training Configuration. We adopt Qwen3-8B as the base model for all experiments. Training proceeds in three consecutive stages: (1) *Supervised fine-tuning* (SFT) on the CoCTE trajectories introduced in Section 4, enabling the model to generate stepwise reasoning and compositional SQL structures; (2) *Online reinforcement learning* using the unified reward $R_{\text{proc}} + R_{\text{out}}$ under the GRPO framework (Section 6); and (3) *Process-guided inference* using the Process Rewards (5) as the selection criterion during decoding. The SFT stage employs a learning rate of 1×10^{-5}, batch size 64, maximum context length 4,096, and 3 epochs of training. For GRPO, the group size G is set to 8, clipping ratio 0.2, KL coefficient 0.02, and sampling temperature 1.0. During inference, the model generates N trajectories and ranks them with process scores; $N = 8$ is used by default.

Metrics and Decoding Strategies. Model quality is evaluated by *Execution Accuracy (EX)*, which compares the outputs of predicted and reference SQL queries executed on the target database. Two decoding protocols are used: (1) greedy decoding with temperature zero, and (2) PRM-guided Vote@N, where multiple candidates are generated and scored by their process reward.

Implementation and Environment. All models are implemented in PyTorch 2.6.0 with DeepSpeed ZeRO-3 optimization. Experiments are conducted on a Rocky Linux 8.10 server equipped with four NVIDIA H20-3e GPUs (143 GB each), dual Intel Xeon Platinum 8468V processors (96 cores in total), and 2 TB system memory. Evaluation is performed on the same hardware to ensure consistent latency measurements.

7.2 Main Results: Comparison with Text-to-SQL Baselines

We first compare Reward-SQL with existing specialized Text-to-SQL approaches on standard benchmarks. Table 3 presents results on BIRD development set and Spider test set, which are the two most widely used benchmarks for evaluating Text-to-SQL systems.

Baselines. We compare with three main categories of specialized Text-to-SQL approaches: (1) *Structure-aware methods* that encode database schema and foreign key relations into graph neural networks, including RAT-SQL [46] and RESDSQL [23]; (2) *Prompting-based methods* that leverage large language models with carefully designed prompts for schema linking and query construction, including DIN-SQL [39], DAIL-SQL [14], MAC-SQL [45], Chase-SQL [38], and Alpha-SQL [21]; (3) *Post-training methods* includes supervised fine-tuning approaches such as CodeS [24], DTS-SQL [40], SQL-o1 [35] and Omni-SQL [22], as well as RL-based methods such as SQL-R1 [37], Reasoning-SQL [41], and Arctic-SQL [51].

Results and Analysis. The results in Table 3 demonstrate several key advantages of our approach:

(1) Improved Performance over Structure-aware Models. Our Reward-SQL improves the results obtained under traditional structure-aware methods like RAT-SQL and RESDSQL, validating the effectiveness of LLM and RL-post training.

Table 3. Comparison with Text-to-SQL methods on benchmarks. The methods evaluated under Vote@N (self-consist. or selection agent or MCTS) are marked with ‡. For open-source methods, we reproduce the results ourselves; for closed-source methods, we report results from original papers. Best results are **bolded**.

Model / Method	BIRD (dev)	Spider (test)
Structure-aware Methods		
RAT-SQL + GraPPa	-	73.3
RESDSQL + T5-3B	-	79.9
Prompting-based Methods		
DIN-SQL + GPT-4	50.7	85.3
DAIL-SQL + GPT-4	54.8	**86.2**
MAC-SQL + GPT-4	59.4	-
Alpha-SQL + Qwen2.5-7B‡	66.8	84.0
GPT-4o	61.9	-
openPangu 7B	41.9	-
Supervised Fine-tuning Methods		
DTS-SQL + DeepSeek-7B	56.0	-
CodeS + StarCoder-15B	58.5	79.4
SQL-o1 + Qwen2.5-7B‡	66.7	85.1
Omni-SQL + Qwen2.5-7B‡	63.8	79.4
RL-based Methods		
Reasoning-SQL + Qwen2.5-7B	64.0	78.7
+ Vote@N‡	68.1	-
SQL-R1 + Qwen2.5-7B‡	65.1	81.4
Arctic-SQL + Qwen2.5-7B‡	66.8	82.1
Our Approach (Qwen3-8B)		
Reward-SQL + Greedy	65.9	81.2
Reward-SQL + PRM@8 (Best-of-N)†	68.5	83.1
Reward-SQL + PRM@32 (Best-of-N)†	**70.3**	85.4

(2) Improved Performance over other LLM-based Methods. Firstly, for all post-training methods with comparable model sizes (7-8B parameters), Reward-SQL achieves the best performances on BIRD and Spider. These improvements are consistent across both greedy decoding and test-time scaling. Besides, even compared to strong (100× larger) closed-source LLMs (e.g., GPT-4 based methods), our Reward-SQL has comparable or improved results. These results together show that the RL-post training combined with our CoCTEs reasoning structures and PRM can make a small LLM have strong performance on Text-to-SQL.

7.3 Generalization Study

We evaluate generalization on five OOD benchmarks, explicitly categorizing them into two levels:

- **Robustness-level OOD**: Spider-DK, Spider-Syn, and Spider-Realistic evaluate robustness to linguistic variations (domain knowledge, synonym substitution, and realistic ambiguity) while retaining schema similarity to the training data.
- **Cross-domain-level OOD**: ScienceBenchmark and EHR-SQL evaluate transfer to entirely unseen domains with different schemas and specialized terminology.

Setup. We compare against three categories of models to ensure fair comparisons: (1) *Specialized 7-8B models* that are post-trained on Text-to-SQL data, including reproducible RL-based methods (SQL-R1, Arctic-SQL) and SFT methods (Omni-SQL); (2) *General-purpose models* of similar size

Table 4. Out-of-distribution generalization across five challenging benchmarks. We distinguish two levels of OOD evaluation: **robustness-level** and **cross-domain-level**. Results show execution accuracy (%) for greedy decoding and sampling-based selection. Best results in each category are in bold.

Model	Robustness-Level OOD								Cross-Domain-Level OOD						Overall	
	Spider-DK		Spider-Syn		Spider-Real		Avg.		ScienceBench		EHR-Bench		Avg.			
	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj
Closed-Source Models																
GPT-4-Turbo	72.3	72.1	62.9	63.5	67.5	68.3	67.6	68.0	**59.2**	**59.5**	43.1	44.8	**51.2**	**52.2**	61.0	61.6
GPT-4o	72.9	73.5	59.6	62.3	66.5	66.7	66.3	67.5	55.5	56.2	**44.9**	**45.5**	50.2	50.9	59.9	60.8
Large Open-Source Models																
Qwen2.5-72B-Instruct	**76.4**	**77.6**	64.1	64.3	70.1	68.5	70.2	70.1	52.8	58.2	35.0	41.2	43.9	49.7	59.7	62.0
DeepSeek-V3 (671B MoE)	72.9	73.8	64.4	65.1	67.9	66.9	68.4	68.6	56.2	57.9	43.2	43.5	49.7	50.7	60.9	61.4
Similar-Sized Models (∼7-8B)																
Qwen2.5-Coder-7B-Instruct	67.5	73.6	63.1	66.9	66.7	70.5	65.8	70.3	45.2	51.2	24.3	36.9	34.8	44.1	53.4	59.8
Meta-Llama-3.1-8B-Instruct	62.6	69.9	53.1	59.3	57.5	61.0	57.7	63.4	36.8	43.1	24.6	33.7	30.7	38.4	46.9	53.4
Post-Trained Models (∼7-8B)																
Arctic-SQL + Qwen2.5-7B	69.7	71.8	75.2	76.0	80.3	81.7	75.1	76.5	52.5	53.8	36.3	38.9	44.4	46.4	**62.8**	64.4
OmniSQL + Qwen2.5-7B	64.9	68.2	70.7	73.9	75.2	80.7	70.3	74.3	50.5	51.8	34.3	39.5	42.4	45.7	59.1	62.8
SQL-R1 + Qwen2.5-7B	67.9	70.5	72.6	75.4	78.3	80.5	72.9	75.5	53.8	51.8	36.0	38.8	44.9	45.3	61.7	63.4
Our Model																
Reward-SQL + Qwen3-8B	69.4	73.1	**75.7**	**79.0**	**84.1**	**83.3**	**76.4**	**78.5**	50.8	54.6	29.6	34.4	40.2	44.5	61.9	**64.9**

(7-8B): Qwen2.5-Coder-7B and Meta-Llama-3.1-8B; (3) *Large-scale models*: Qwen2.5-72B, DeepSeek-V3, GPT-4-Turbo, and GPT-4o. For fair comparison, Vote@8 results for all open-source models use self-consistency voting.

Results and Analysis. Table 4 presents comprehensive results across all five OOD benchmarks. Our analysis reveals that:

(1) Robustness-Level OOD: Superior Performance Against RL Baselines. On Spider-DK, Spider-Syn, and Spider-Realistic, where schemas remain similar but linguistic variations increase, our approach achieves SOTA performance among all 7-8B models. Notably, we surpass even 100× larger models including GPT-4-Turbo and DeepSeek-V3. This demonstrates that process-aware training produces superior robustness to linguistic variations compared to both RL approaches and massive general-purpose models.

(2) Cross-Domain-Level OOD: Competitive with RL but Dropped Performance due to Schema Transfer. On the ScienceBenchmark and EHR-SQL, where domains and schemas are entirely unseen, the performance drops for all specialized 7-8B models. Though Reward-SQL achieves comparable results with other post-trained baselines, all these RL-post trained models significantly underperform large general models on these benchmarks. This is because the post-training process pushes the model to overfit to domain-specific schemas and terminology, which deteriorates the performance of the model on the other domains. Similar phenomena have been observed in [17]. Handling such OOD generalization is left as a challenge in future work.

(3) Overall Trade-offs. When averaged across all OOD benchmarks, Reward-SQL achieves 64.9%, marginally exceeding Arctic-SQL (64.4%)—while substantially outperforming SQL-R1 (63.4%) and OmniSQL (62.8%). This demonstrates that Reward-SQL achieves superior OOD generalization compared to RL methods, while maintaining strong performance across both cross-domain and linguistic variation settings.

(4) Effectiveness of Test-Time Scaling Across OOD Types. PRM-guided selection provides consistent gains in both OOD scenarios, though the magnitude varies. On robustness-level tasks, scaling from greedy to Maj@8 improves performance by +2.1% on average (76.4% → 78.5%), validating that process rewards effectively capture reasoning patterns resilient to linguistic variations.

On cross-domain tasks, gains are comparable (+4.3%, 40.2% → 44.5%), suggesting that process-level supervision remains beneficial even when schema knowledge is incomplete, though it cannot fully compensate for domain-specific knowledge gaps.

The consistent improvements across all benchmarks (averaging +9.2% over the best closed-source model) indicate that combining CoCTE structured reasoning with process reward supervision yields robust and generalizable SQL generation rather than dataset-specific overfitting.

7.4 Training Stability Analysis

We examine how process-level supervision influences reinforcement learning. Traditional RL post-training for Text-to-SQL relies solely on the *Outcome Reward* R_{out}, a binary signal indicating execution correctness. While simple, this signal is highly sparse: most trajectories receive zero reward, and its gradient variance grows with trajectory length, leading to unstable optimization. REWARD-SQL augments the outcome reward with the dense *Process Reward* R_{proc} defined in Section 6, forming a composite objective $R_{\text{proc}} + R_{\text{out}}$ under the GRPO framework. This design provides credit signals at intermediate steps of a CoCTE trajectory rather than only at the final SQL.

Training Dynamics. Figure 6 compares the dev-set execution accuracy during GRPO training using only R_{out} versus the unified R_{proc}+R_{out} objective. Both models start from the same SFT checkpoint and share identical hyperparameters. In the first 150 training steps, both variants exhibit similar upward trends, with $GRPO(R_{\text{proc}}+R_{\text{out}})$ achieving a higher peak performance of 62.1%. After 150 steps, the training dynamics diverge significantly: the process-guided model maintains near-stable performance with variations not exceeding 1%, whereas the outcome-only $GRPO(R_{\text{out}})$ variant experiences substantial instability, showing noticeable performance degradation and variance of approximately 1.5%.

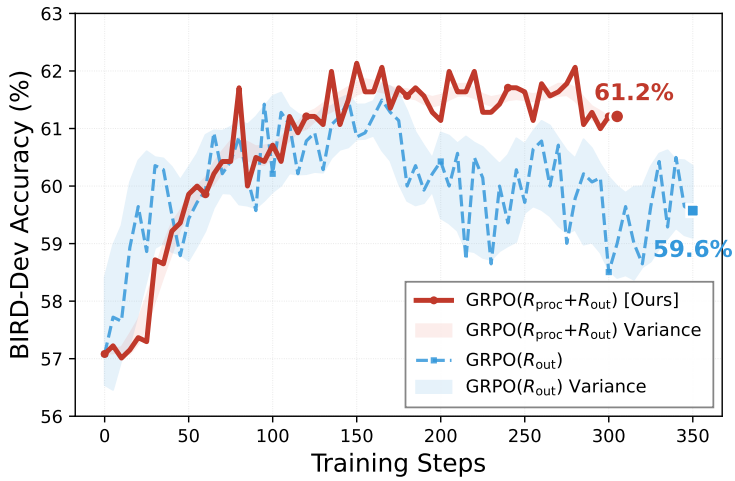


Fig. 6. Training dynamics of GRPO with and without process reward. Process supervision accelerates convergence and reduces variance in dev-set execution accuracy.

7.5 Ablation Studies

Quantitative Comparison. Table 5 summarizes the final performance and training stability. Under identical configurations, GRPO(R_{proc}+R_{out}) achieves 61.8% execution accuracy on the BIRD development set, outperforming the outcome-only baseline (59.6%) while exhibiting a substantially lower variance after the initial 50 steps (±0.58 vs ±0.76). The number of steps to convergence

Table 5. Effect of process reward on GRPO training stability. Results show final execution accuracy, mean performance, and standard deviation after initial 50 steps.

Method	Dev EX (%)	Mean (±)	Std (±)
GRPO (R_{out})	59.6	60.2	0.76
GRPO (R_{proc}+R_{out})	**61.8**	**61.5**	**0.58**

Table 6. Ablation study on key components of the proposed framework. Results compare different reasoning formats, training methods, and selection strategies on BIRD Dev.

Configuration	Greedy EX (%)	PRM@8 EX (%)
Full Reward-SQL (SFT + GRPO + PRM@8)	66.0	68.7
Reasoning Format Variants		
Direct SQL (no reasoning)	63.6	-
CoCTE (prompting Deepseek-V3.1)	60.1	-
NL-CoT (generic reasoning)	64.2	-
CoCTE (ours)	**66.0**	**68.7**
Training Method Variants		
SFT only	63.2	67.4
GRPO(R_{out} only)	62.6	67.0
GRPO(R_{proc} + R_{out})	**66.0**	**68.7**
Selection Method Variants		
Self-Consistency	-	66.6
Outcome Reward (Model)	-	67.1
Process Reward (ours)	-	**68.7**

decreases from 350 to 250, demonstrating that the dense process reward provides a more informative learning signal and facilitates early stabilization of policy gradients.

In this section, we conduct ablation studies on the main components of our Reward-SQL. The main results are summarized in Table 6.

(1) Reasoning Format. CoCTE outperforms both direct SQL generation (66.0% vs 63.6% in greedy, +2.4%) and generic natural language chain-of-thought (66.0% vs 64.2%, +1.8%). This validates our design of executable intermediate reasoning steps that can be verified against the database, providing more reliable supervision signals than unverifiable natural language reasoning or format-only prompting. To further validate that gains come from our training/reward design rather than merely using CTE formatting, we compare against 5-shot CoCTE prompting with DeepSeek-V3.1, which achieves only 60.1%. The 5.9% gap (66.0% vs 60.1%) demonstrates that prompting alone cannot reliably induce proper SQL decomposition—our process-supervised training is essential for learning correct substep generation.

(2) Training Method. GRPO training with process rewards improves over SFT baseline by 2.7% in greedy decoding (66.0% vs 63.2%). More importantly, comparing GRPO(R_{proc} + R_{out}) with GRPO(R_{out}) shows that adding process rewards during training improves both greedy (66.0% vs 62.6%, +3.4% absolute improvement) and PRM@8 performance (68.7% vs 67.0%, +1.7% gain). This demonstrates that process-level supervision during training enhances model's ability to generate high-quality intermediate reasoning steps.

Table 7. Sensitivity to trajectory source: training on different single-LLM corpora versus mixed-source corpus.

Training Source	Greedy	PRM@8
DeepSeek-only	58.15	68.4
GPT-4-only	58.87	67.6
Claude-only	62.97	68.9
Mixed (DeepSeek+GPT+Claude)	**64.86**	**70.0**

(3) Selection Method. PRM selection substantially outperforms alternatives, achieving 68.7% compared to self-consistency (66.6%, +2.1% improvement) and ORM selection (67.1%, +1.6% gain). This confirms that process-level evaluation provides more fine-grained discrimination among candidate trajectories than outcome-only or voting-based selection. Overall, these ablations confirm that all three components—CoCTE format, process-aware GRPO training, and PRM-guided selection—contribute to the final performance.

Sensitivity to Cold-Start Corpus Source. To evaluate whether our policy model overfits to a specific LLM's decomposition style during cold-start training, we conduct experiments using trajectories generated exclusively by DeepSeek, GPT-4, or Claude as the initial training corpus (Section 4). As shown in Table 7, policy models trained on single-source corpora achieve comparable performance : DeepSeek-only (68.4% at PRM@8), GPT-only (67.6%), and Claude-only (68.9%). The small variance (±0.6%) demonstrates that our trained policy model generalizes across different reasoning styles and does not overfit to any particular LLM's CTE decomposition patterns. Moreover, training on a mixed corpus combining trajectories from all three LLMs yields further improvements. This suggests that exposure to diverse decomposition styles enhances the policy model's ability to generate varied reasoning paths.

7.6 Computational Cost Analysis

While test-time scaling improves accuracy, it incurs additional computational costs. We analyze the trade-offs between performance gains and resource requirements, distinguishing between one-time training costs and per-query inference overhead.

Training-Time vs. Inference-Time Costs. We first clarify that the expensive MCTS search is performed *only once* during offline training-data construction (Section 5.2) and incurs no overhead during inference. Specifically, MCTS sampling processed **16,248** training instances, consuming approximately **18 hours** on a single server. The subsequent PRM training utilized **4×A800 GPUs for 7 hours** (28 GPU-hours total). These one-time offline costs amortize across all future queries.

For online inference, the computational overhead comprises three stages: (1) candidate generation via VLLM, (2) SQL execution to obtain intermediate results, and (3) PRM scoring for candidate selection. Below we analyze the latency of each component.

Latency Analysis. Figure 7 presents a breakdown of the latency components for different values of n in our PRM@n approach. The inference pipeline consists of three main stages:

- **Chat Completion:** Leveraging VLLM's optimized batch inference, resulting in sublinear latency growth (from 0.8s at n=1 to 1.65s at n=32)
- **SQL Processing:** Collecting intermediate results through execution, scaling linearly with n. In this part, our Reward-SQL has similar latencies compared to standard inference methods without our intermediate CoCTEs, due to database cache.
- **PRM Scoring:** Evaluating candidates with the Process Reward Model, whereas the extra computational costs of our Reward-SQL arise from it. Rising linearly from 0.21s-3.61s (depending on n). This part takes up 5%-20% cost of total latency.

Table 8. Error category analysis on BIRD Dev. Comparison between Qwen3-8B and Reward-SQL with different PRM@N.

Category	Error Type	Qwen3-8B Errors (%)	Reward-SQL PRM@8 (%)	Reward-SQL PRM@32 (%)
Schema Linking	Table Selection	100 (6.52)	65 (4.24)	62 (4.04)
	Column Selection	88 (5.74)	51 (3.32)	48 (3.13)
	JOIN Keys	49 (3.19)	14 (0.91)	13 (0.85)
	Hallucination	96 (6.26)	6 (0.39)	4 (0.26)
	Condition Error	102 (6.65)	65 (4.24)	63 (4.11)
	NULL Handling	22 (1.43)	52 (3.39)	52 (3.39)
Value Retrieval	Format Error	41 (2.67)	24 (1.56)	25 (1.63)
	Manipulation	8 (0.52)	12 (0.78)	9 (0.59)
Operation	Math Formula	43 (2.80)	46 (3.00)	34 (2.22)
	Aggregation	42 (2.74)	37 (2.41)	36 (2.35)
	Complex Operation	7 (0.46)	5 (0.33)	2 (0.13)
Information	Filter Condition	29 (1.89)	47 (3.06)	53 (3.45)
	ORDER BY/LIMIT	26 (1.69)	31 (2.02)	30 (1.96)
	Return Format	12 (0.78)	13 (0.85)	11 (0.72)
Other	Domain Knowledge	6 (0.39)	9 (0.59)	9 (0.59)
	Invalid SQL	20 (1.30)	4 (0.26)	2 (0.13)
Total		**699 (45.57)**	**483 (31.49)**	**455 (29.66)**

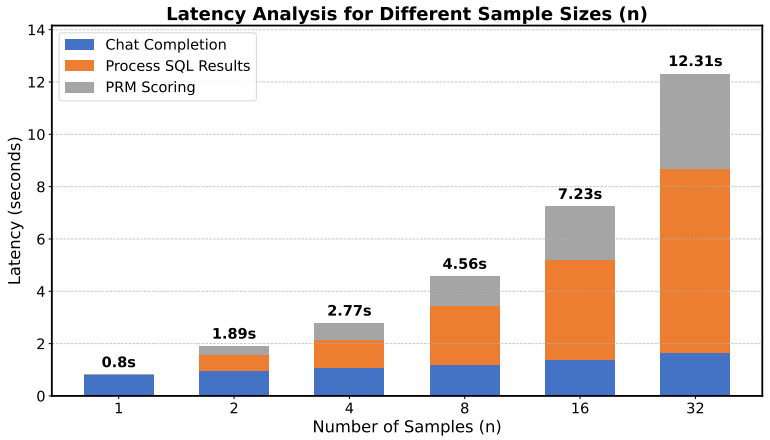


Fig. 7. Latency analysis for different values of n in PRM@n. The chart shows the time contribution of each component.

The total latency increases from 0.8 seconds at n=1 to 12.31 seconds at n=32, with SQL processing becoming the dominant factor at higher values of n. This analysis reveals significant optimization opportunities, particularly in parallelizing the SQL execution and scoring components.

Accuracy-Latency Trade-off. Figure 8 presents the Pareto frontier comparing three inference strategies: (1) Direct SQL generation (**Greedy**), (2) **Self-Consistency** (majority voting over N samples), and (3) **Reward-SQL** (PRM-guided selection, PRM@N). While greedy decoding offers

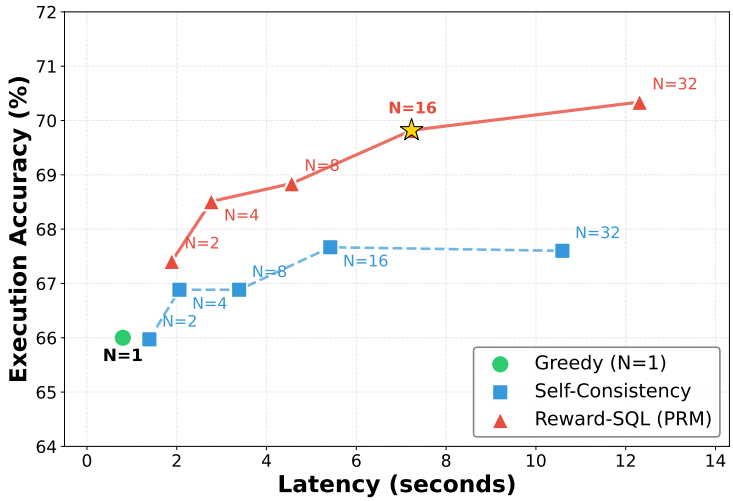


Fig. 8. Accuracy-Latency Pareto frontier on BIRD Dev., comparing greedy decoding, self-consistency (majority voting), and PRM-guided selection (ours).

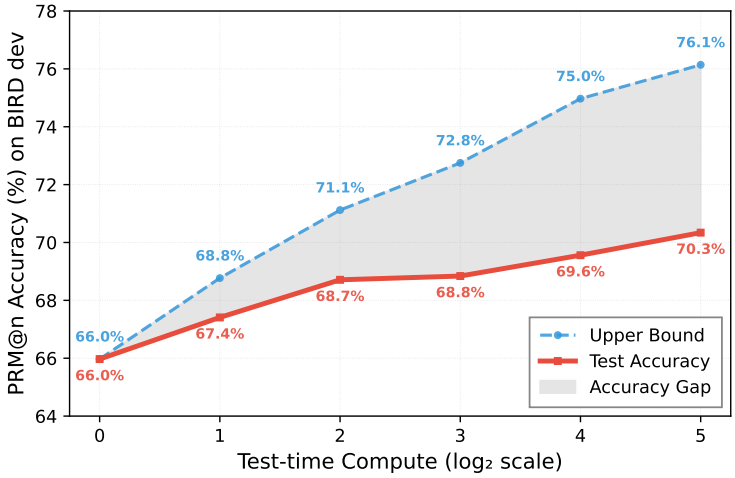


Fig. 9. Test-time compute vs accuracy on BIRD Dev. set. The upper bound (Pass@N) represents the theoretical maximum performance achievable with perfect selection, while PRM@N shows our model's actual performance.

minimal latency (0.8s) with 66.0% accuracy, scaling to PRM@16 achieves **69.6% accuracy at 6.3s latency**, representing a substantial improvement of +3.6 percentage points. Self-consistency provides a middle ground but saturates quickly, even experiencing a slight decline of 0.1% from N=16 to N=32. In contrast, PRM maintains steady improvements through effective candidate selection. Beyond N=16, however, PRM's marginal gains also diminish (only +0.7% from N=16 to N=32) while latency doubles, establishing PRM@16 as the practical sweet-spot that balances accuracy gains against computational cost.

Scaling Behavior. As shown in Figure 9, performance scales logarithmically with the number of samples N. The upper bound (Pass@N) shows a nearly linear growth as test-time compute increases, climbing from 65.9% at the lowest compute level to 76.1% at the highest compute level. Our PRM@n accuracy follows a similar upward trajectory, starting at 65.9% with only one sample and reaching 70.3% with 32 samples. The gap between the theoretical upper bound and actual PRM@n performance gradually widens as compute increases, indicating untapped potential for further improvement through enhanced selection mechanisms.

Query Execution Efficiency under CoCTEs Format. We evaluate the runtime overhead of CTE-based queries compared to flattened SQL on BIRD Dev. (SQLite). For fair comparison, we selected queries where Reward-SQL produced correct results and compared their latency against the corresponding gold SQL (optimized, flattened expert queries). Each query was executed 5 times with cache cleared between runs. On average, CoCTE queries execute in **230ms** versus **210ms** for gold SQL, representing a **<10% overhead** (9.5%). While this overhead is acceptable on BIRD-scale datasets, it may impact latency-sensitive enterprise applications with complex schemas. In future work we will explore (1) rule-based CTE transpilers to generate flattened SQL for production deployment, and (2) cost-aware reward modeling to optimize execution efficiency during training.

7.7 Error Analysis

To understand how our approach improves SQL generation quality, we conduct a detailed error analysis comparing the baseline Qwen3-8B model with our full Reward-SQL framework at different test-time scaling budgets (PRM@8 and PRM@32). We manually annotate error examples and categorize them according to a comprehensive error taxonomy.

Table 8 presents a comprehensive breakdown of error categories. Our analysis reveals several key insights:

Major Improvements from Baseline to PRM@8:

- **Hallucination Reduction (-6.00%):** The most significant improvement is in reducing schema hallucinations, dropping from 6.26% to 0.26%. Process supervision effectively prevents the model from generating invalid schema elements by providing feedback when intermediate CTEs reference undefined database objects.
- **JOIN Key Accuracy (-2.34%):** Structured reasoning through CoCTE helps correctly identify foreign key relationships.
- **Schema Linking:** Significant reductions in Table Selection (-2.48%) and Column Selection (-2.61%) errors demonstrate improved schema comprehension.
- **Invalid SQL (-1.17%):** The CoCTE format naturally encourages syntactically valid SQL generation by decomposing complex queries into verifiable components.

Gains from Scaling PRM@8 to PRM@32: Further increasing the sample budget from 8 to 32 yields additional error reductions, primarily in schema understanding:

- **Math Formula:** Notable 26% relative improvement (46 to 34 instances), suggesting better handling of complex computations with increased sampling.
- **Table Selection:** Errors reduced by an additional 4.6% (from 65 to 62 instances), indicating better table retrieval in complex multi-table scenarios.
- **Column Selection:** Errors reduced by an additional 5.9% (from 51 to 48 instances), showing improved column mapping for nuanced queries.

Overall, Reward-SQL reduces total errors from 45.57% (baseline) to 31.49% (PRM@8) and further to 29.66% (PRM@32), representing a total relative improvement of 34.9%. The remaining challenges primarily involve semantic understanding (Filter Conditions, Domain Knowledge) where additional

Table 9. CTE-level error breakdown on 200 failed cases. The majority of errors are unrelated to CTE decomposition.

Error Category	Count	Percentage
Not CTE-related	189	94.5%
Data Loss in CTE Pipeline	8	4.0%
Redundant CTEs	2	1.0%
Wrong CTE Boundaries	1	0.5%
Total	**200**	**100%**

sampling provides limited benefit, suggesting these require stronger prior knowledge rather than increased test-time compute.

CTE-Level Error Analysis. To further validate that our CoCTE decomposition does not introduce systematic errors, we manually analyzed 200 failed cases from the BIRD Dev. set, examining whether errors stem from CTE-level issues (e.g., wrong decomposition boundaries, redundant steps) or from underlying SQL generation problems.Table 9 shows that 94.5% of errors are not CTE-related, indicating that failures primarily originate from SQL generation challenges (e.g., incorrect filters, schema mismatches) rather than decomposition artifacts. Among CTE-specific issues, the most common is Data Loss in CTE Pipeline (4.0%), where intermediate CTEs inadvertently filter out necessary rows through overly restrictive conditions. Cases of Redundant CTEs (1.0%) and Wrong CTE Boundaries (0.5%) are rare, demonstrating that our framework effectively learns appropriate decomposition granularity.

8 Related Work

SQL Decomposition and CoT. Natural language decomposition and finer-grained reasoning in Text-to-SQL can be broadly categorized into three directions. (i) Natural language decomposition. Representative works such as Reasoning-SQL [41] and Omni-SQL [22] first generate natural language reasoning traces—breaking the question into sub-goals and aligning them with SQL clauses—before producing the final query. Following this idea, Arctic-SQL [51] and ExCoT[53] integrate the reasoning process into the model using different RL algorithms. (ii) SQL structure decomposition. SQL-o1 [35] emphasizes structural planning: it decomposes a query into skeleton-level steps (e.g., deciding the overall layout and clause order) before filling in details. (iii) Operator-level decomposition. Alpha-SQL [21] further refines this idea by decomposing queries into sequences of fine-grained operators (joins, filters, aggregations), treating SQL generation as a stepwise composition of primitive actions.

Despite these advances, most existing approaches decompose only at the natural language or abstract planning level, and the final SQL must still be assembled monolithically, which remains error-prone. A more robust form of SQL-level decomposition would require intermediate states that are executable and verifiable. In practice, Common Table Expressions (CTEs) serve exactly this role, allowing developers to structure complex queries into modular subqueries [1, 3]. Recent works, such as ReFoRCE [5] have begun to incorporate CTEs for partial refinement, but this remains ad-hoc and limited. Overall, the CTE-based SQL decomposition proposed in this paper has not been systematically explored in Text-to-SQL, leaving a gap between natural language reasoning and executable query construction.

RL-based Text-to-SQL Methods. Recent research has increasingly adopted RL to optimize Text-to-SQL models. On the training side, online RL approaches differ mainly in their reward design: SQL-R1 and Arctic-SQL employ execution accuracy (a lightweight but sparse supervision signal)

as the sole reward in RL to improve the model's reasoning capability in Text-to-SQL [34, 36]. To mitigate the sparse reward signals, Reasoning-SQL combines GRPO with partial execution-based rewards [41]. In parallel, offline preference optimization has gained prominence, where DPO-based [43] approaches align models via preference learning, and frameworks such as ExCoT integrate both off-policy and on-policy optimization [30, 53]. On the inference side, the core idea is "test-time scaling", i.e., aggregating multiple candidates and then selecting the best one from them. The variants arise in the selection criteria, e.g., the self-consistency in C3-SQL [7] the Best-of-N in STaR-SQL [16], the Monte Carlo Tree Search (MCTS) in Alpha-SQL [21], the self-rewarding in SQL-o1 [35].

Reward design of the existing approaches is still dominated by outcome-based signals, i.e., whether the final SQL executes correctly. Such sparse feedback provides limited guidance during training and forces inference-time strategies to rely largely on frequency-based selection or heuristic search. To address this sparsity, several methods have explored process-level rewards. Execution-Guided Decoding (EGD) executes partial SQL during generation to prune invalid candidates [47]. Reasoning-SQL introduces partial execution rewards to reduce sparsity during RL training [41]. Search-based methods such as Alpha-SQL (MCTS exploration) and SQL-o1 (self-rewarding search) also leverage process information to improve inference [21, 35]. ReEx-SQL [2] integrates intermediate execution feedback but applies it to full-query revisions rather than stepwise decomposition into simpler executable sub-queries.

Process Reward Models (PRMs) have demonstrated strong potential in other reasoning domains: in mathematics, they evaluate intermediate reasoning to address credit assignment issues [29, 48]; in program synthesis, they assess partial code states to catch errors early [10, 26]. However, existing Text-to-SQL methods remain largely heuristic or task-specific, lacking a learned and generalizable model to evaluate intermediate reasoning quality. Unlike mathematical or code reasoning, Text-to-SQL still lacks an explicitly trained PRM to systematically guide both learning and inference. This gap motivates our work: we introduce a PRM-based framework that provides fine-grained, execution-aware supervision and principled search guidance for stepwise SQL generation.

9 Conclusions

In this paper, we have proposed Reward-SQL, a novel framework that addresses the key limitations of existing RL-based Text-to-SQL approaches through stepwise execution-aware reasoning and process-supervised rewards. We introduced CoCTE, a divide-and-conquer reasoning format that decomposes complex SQL generation into a sequence of executable Common Table Expressions. To provide fine-grained supervision, we designed a process reward model that combines trajectory scoring with inverse entropy weighting, and integrated it into both GRPO training and Best-of-N inference, enabling stable optimization and principled trajectory selection. Extensive experiments on multiple benchmark datasets demonstrate the effectiveness of our method: Reward-SQL achieves 70.3% execution accuracy on BIRD with an 8B model, outperforming baselines of comparable size and showing strong cross-domain generalization across five out-of-distribution benchmarks.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (Grant Nos. 62436010, 62441230, 62506365, and 62402409) and the Scientific Research Innovation Capability Support Project for Young Faculty (Grant No. SRICSPYF-ZY2025001). Guoliang Li was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (62525202, 62232009), Shenzhen Project (CJGJZD20230724093403007), China Railway Science Research Institute Group Co., Ltd, Zhongguancun Lab. This work was done while Yuxin Zhang was an intern at Alibaba Cloud Computing, and was supported by Alibaba Research Intern Program.

References

[1] Airbyte. 2024. Common Table Expressions (CTEs): Syntax, Types, & Benefits. https://airbyte.com/data-engineering-resources/common-table-expression. Accessed: 2025-09-22.

[2] Yaxun Dai, Wenxuan Xie, Xialie Zhuang, Tianyu Yang, Yiying Yang, Haiqin Yang, Yuhang Zhao, Pingfu Chao, and Wenhao Jiang. 2025. ReEx-SQL: Reasoning with Execution-Aware Reinforcement Learning for Text-to-SQL. *CoRR* abs/2505.12768 (2025). https://doi.org/10.48550/arXiv.2505.12768

[3] DataForgeLabs. 2023. Advanced SQL Concepts for Data Engineers. https://www.dataforgelabs.com/advanced-sql-concepts. Accessed: 2025-09-22.

[4] 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, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, and S. S. Li. 2025. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.* Preprint arXiv:2501.12948.

[5] Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. ReFoRCE: A Text-to-SQL Agent with Self-Refinement, Consensus Enforcement, and Column Exploration. *arXiv preprint arXiv:2502.00675* (2025).

[6] Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2020. Structure-grounded pretraining for text-to-sql. *arXiv preprint arXiv:2010.12773* (2020).

[7] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, et al. 2023. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306* (2023).

[8] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining Small Language Models and Large Language Models for Zero-Shot NL2SQL. *Proc. VLDB Endow.* 17, 11 (2024).

[9] Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. 2024. Autoprep: Natural language question-aware data preparation with a multi-agent framework. *arXiv preprint arXiv:2412.10422* (2024).

[10] Meihao Fan, Ju Fan, Yuxin Zhang, Shaolei Zhang, Xiaoyong Du, Jie Song, Peng Li, Fuxin Jiang, Tieying Zhang, and Jianjun Chen. 2026. DeepPrep: An LLM-Powered Agentic System for Autonomous Data Preparation. *arXiv preprint arXiv:2602.07371* (2026).

[11] Meihao Fan, Xiaoyue Han, Ju Fan, Chengliang Chai, Nan Tang, Guoliang Li, and Xiaoyong Du. 2024. Cost-effective in-context learning for entity resolution: A design space exploration. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3696–3709.

[12] Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R Woodward, Jinxia Xie, and Pengsheng Huang. 2021. Towards robustness of text-to-SQL models against synonym substitution. *arXiv preprint arXiv:2106.01065* (2021).

[13] Yujian Gan, Xinyun Chen, and Matthew Purver. 2021. Exploring underexplored limitations of cross-domain text-to-sql generalization. *arXiv preprint arXiv:2109.05157* (2021).

[14] 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. *Proc. VLDB Endow.* 17, 5 (2024).

[15] Zihui Gu, Ju Fan, Nan Tang, Lei Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. 2023. Few-shot text-to-sql translation using structure and content prompt learning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–28.

[16] Mingqian He, Yongliang Shen, Wenqi Zhang, Qiuying Peng, Jun Wang, and Weiming Lu. 2025. STaR-SQL: Self-Taught Reasoner for Text-to-SQL. arXiv:2502.13550 (2025).

[17] Maggie Huan, Yuetai Li, Tuney Zheng, Xiaoyu Xu, Seungone Kim, Minxin Du, Radha Poovendran, Graham Neubig, and Xiang Yue. 2025. Does Math Reasoning Improve General LLM Capabilities? Understanding Transferability of LLM Reasoning. *arXiv preprint arXiv:2507.00432* (2025).

[18] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720* (2024).

[19] Gyubok Lee, Hyeonji Hwang, Seongsu Bae, Yeonsu Kwon, Woncheol Shin, Seongjun Yang, Minjoon Seo, Jong-Yeup Kim, and Edward Choi. 2022. Ehrsql: A practical text-to-sql benchmark for electronic health records. *Advances in Neural Information Processing Systems* 35 (2022), 15589–15601.

[20] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763* (2024).

[21] Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025. Alpha-SQL: Zero-Shot Text-to-SQL using Monte Carlo Tree Search. In *Forty-Second International Conference on Machine Learning, ICML 2025, Vancouver, Canada, July 13-19, 2025*. OpenReview.net.

[22] Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. *OmniSQL: Synthesizing High-quality Text-to-SQL Data at Scale*. Preprint arXiv:2503.02240.

[23] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. In *Thirty-Seventh AAAI Conference on Artificial Intelligence*, Brian Williams, Yiling Chen, and Jennifer Neville (Eds.).

[24] 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. *Proc. ACM Manag. Data* 2, 3 (2024).

[25] 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 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*.

[26] Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025. CodePRM: Execution Feedback-enhanced Process Reward Model for Code Generation. In *Findings of the Association for Computational Linguistics: ACL 2025*. 8169–8182.

[27] Yu Li, Mingyang Yi, Xiuyu Li, Ju Fan, Fuxin Jiang, Binbin Chen, Peng Li, Jie Song, and Tieying Zhang. 2026. Reasoning and Tool-use Compete in Agentic RL: From Quantifying Interference to Disentangled Tuning. *arXiv preprint arXiv:2602.00994* (2026).

[28] Ziming Li, Youhuan Li, Yuyu Luo, Guoliang Li, and Chuxu Zhang. 2025. *Graph Neural Networks for Databases: A Survey*. Preprint arXiv:2502.12908.

[29] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*.

[30] Hanbing Liu, Haoyang Li, Xiaokang Zhang, Ruotong Chen, Haiyong Xu, Tian Tian, Qi Qi, and Jing Zhang. 2025. *Uncovering the Impact of Chain-of-Thought Reasoning for Direct Preference Optimization: Lessons from Text-to-SQL*. Preprint.

[31] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025. A Survey of Text-to-SQL in the Era of LLMs: Where Are We, and Where Are We Going? *IEEE Transactions on Knowledge and Data Engineering* 37, 10 (2025), 5735–5754. doi:10.1109/TKDE.2025.3592032

[32] Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. 2025. *NL2SQL-BUGs: A Benchmark for Detecting Semantic Errors in NL2SQL Translation*. Preprint arXiv:2503.11984.

[33] Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Meiqi Guo, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, et al. 2024. *Improve mathematical reasoning in language models by automated process supervision*. Preprint arXiv:2406.06592.

[34] Yuyu Luo, Guoliang Li, Ju Fan, Chengliang Chai, and Nan Tang. 2025. Natural Language to SQL: State of the Art and Open Problems. *Proc. VLDB Endow.* 18, 12 (2025), 5466–5471.

[35] 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*. Preprint arXiv:2502.11741.

[36] Peixian Ma, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. 2025. *SQL-R1: Training Natural Language to SQL Reasoning Model By Reinforcement Learning*. Preprint arXiv:2504.08600.

[37] Peixian Ma, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. 2025. Sql-r1: Training natural language to sql reasoning model by reinforcement learning. *arXiv preprint arXiv:2504.08600* (2025).

[38] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. 2024. *CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL*. Preprint arXiv:2410.01943.

[39] 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*.

[40] Mohammadreza Pourreza and Davood Rafiei. 2024. DTS-SQL: Decomposed Text-to-SQL with Small Large Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*. Association for Computational Linguistics, 8212–8220.

[41] Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, and Sercan. 2025. *Reasoning-SQL: Reinforcement Learning with SQL Tailored Partial Rewards for Reasoning-Enhanced Text-to-SQL.* Preprint arXiv:2503.23157.

[42] James Preiss, Tom Erez, and Yuval Tassa. 2019. Analyzing the Variance of Policy Gradient Estimators for the REINFORCE Algorithm in Linear Quadratic Regulator Problems. In *Proceedings of the Workshop on Optimization Foundations for Reinforcement Learning (OptRL).*

[43] 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 *Advances in Neural Information Processing Systems 36.*

[44] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. 2024. *Deepseekmath: Pushing the limits of mathematical reasoning in open language models.* Preprint arXiv:2402.03300.

[45] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In *International Conference on Computational Linguistics.*

[46] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2019. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942* (2019).

[47] Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. 2018. Robust text-to-sql generation with execution-guided decoding. *arXiv preprint arXiv:1807.03100* (2018).

[48] Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2023. *Math-shepherd: Verify and reinforce llms step-by-step without human annotations.* Preprint arXiv:2312.08935.

[49] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022).

[50] Yuhuai Wu, George Tucker, and Ofir Nachum. 2018. Variance Reduction for Policy Gradient with Action-Dependent Factorized Baselines. *arXiv preprint arXiv:1803.07246* (2018).

[51] 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. *arXiv preprint arXiv:2505.20315* (2025).

[52] 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.* 3911–3921.

[53] 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* (2025).

[54] Chao Zhang, Shaolei Zhang, Quehuan Liu, Sibei Chen, Tong Li, and Ju Fan. 2025. TAIJI: MCP-based Multi-Modal Data Analytics on Data Lakes. *arXiv preprint arXiv:2505.11270* (2025).

[55] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.

[56] Shaolei Zhang, Ju Fan, Meihao Fan, Guoliang Li, and Xiaoyong Du. 2025. Deepanalyze: Agentic large language models for autonomous data science. *arXiv preprint arXiv:2510.16872* (2025).

[57] Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. 2023. Sciencebenchmark: A complex real-world benchmark for evaluating natural language to sql systems. *arXiv preprint arXiv:2306.04743* (2023).

[58] Yizhang Zhu, Runzhi Jiang, Boyan Li, Nan Tang, and Yuyu Luo. 2025. *EllieSQL: Cost-Efficient Text-to-SQL with Complexity-Aware Routing.* Preprint arXiv:2503.22402.

Received October 2025; revised January 2026; accepted February 2026