RSTAR-SQL: SELF-EVOLVING PROCESS PREFERENCE LEARNING FOR COMPLEX-REASONING TEXT-TO-SQL

Anonymous authors
Paper under double-blind review

ABSTRACT

Complex reasoning Text-to-SQL requires models to jointly resolve schema grounding, implicit constraints, domain knowledge, and SQL construction. While recent MCTS-based methods improve test-time exploration, their search feedback is typically used only for terminal candidate selection, providing limited reusable supervision for subsequent reasoning. We introduce **RStar-SQL**, a self-evolving process preference framework that converts SQL search outcomes into persistent training signals. **RStar-SQL** performs schema-grounded MCTS to generate diverse reasoning trajectories and evaluates them using execution consistency together with SQL-specific semantic signals, including schema grounding, execution behavior, join validity, and type operator compatibility. Verified high-quality trajectories refine the policy model, while preferred and dispreferred trajectory pairs train a SQL Process Preference Model. The learned SQL-PPM then guides action selection in subsequent MCTS rounds before terminal feedback is available, forming an iterative loop of search, verification, and policy preference refinement. Experiments on LogicCat and Archer show that RStar-SQL achieves execution accuracies of 35.41% and 47.82%, respectively, demonstrating strong effectiveness on complex reasoning tasks.

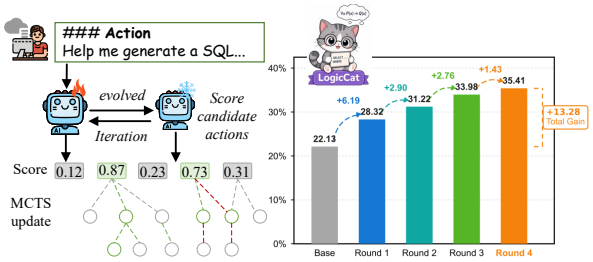


Figure 1: **Effectiveness of RStar-SQL self-evolution.** SQL-PPM-guided MCTS iteratively improves the policy and preference model, raising LogicCat execution accuracy from 22.13% to 35.41% over four rounds (**+13.28 points**).

1 INTRODUCTION

Text-to-SQL aims to translate natural-language questions into SQL queries over relational databases. Recent LLM-based systems have achieved strong performance on conventional benchmarks such as Spider (Yu et al., 2018) and BIRD (Li et al., 2023), through schema linking, candidate selection (Pourreza & Rafiei, 2023; Sun et al., 2024; Talaei et al., 2024; Pourreza et al., 2025a), and reinforcement scaling (Wang et al., 2025; Yang et al., 2026a). These advances suggest that modern systems can handle many routine database queries whose main difficulty lies in identifying relevant schema elements and composing standard SQL structures. However, strong performance (Yao et al., 2026; Li et al., 2026a; Talaei et al., 2024) on such schema-centric settings does not imply reliable

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

reasoning over complex mathematical and logical constraints. This gap is exposed by reasoning-intensive benchmarks such as Archer (Zheng et al., 2024) and LogicCat (Liu et al., 2026b), which require mathematical, physical, commonsense, and hypothetical reasoning beyond conventional schema matching. Such reasoning must be accurately translated into SQL operations (Song et al., 2025), where a single intermediate error can invalidate the final query. Complex-reasoning Text-to-SQL therefore requires jointly resolving semantic reasoning, computation, and SQL construction.

Recent studies have explored search-based methods to address complex-reasoning Text-to-SQL (Li et al., 2025; Liu et al., 2026a; Yuan et al., 2025). We argue that the value of SQL search should extend beyond selecting a final candidate (Lyu et al., 2025; Xie et al., 2024; Liu et al., 2026a). The diverse trajectories produced during MCTS (Yuan et al., 2026; Tang et al., 2026) reveal relative evidence about which schema selections, value mappings, operators, and SQL revisions are more likely to lead to semantically grounded solutions. Although estimating MCTS values requires substantial search resources, individual trajectory values can still be noisy due to limited search coverage and stochastic rollouts (Zhang et al., 2024; Guan et al., 2025), and may not provide reliable guidance. In contrast, relative preferences between competing trajectories offer a more robust form of process supervision (Singh et al., 2024). The central question is therefore,

How can transient SQL search outcomes and reasoning trajectories be transformed into persistent process knowledge that improves subsequent reasoning and search?

Existing approaches provide complementary foundations for this direction. rStar-Math (Guan et al., 2025) demonstrates iterative policy preference evolution (Park et al., 2025) for reasoning, while Alpha-SQL (Li et al., 2025) introduces structured MCTS for SQL generation. However, complex SQL trajectories additionally involve database-specific failure modes in schema grounding, value alignment, joins, execution behavior, and operator compatibility, making reliable preference construction itself a central challenge. This raises a further question:

How can SQL-specific verification convert structured search trajectories into reliable process preferences for subsequent self-evolution?

To address these questions, we propose **RStar-SQL**, a self-evolving process preference framework shown as Figure 1 for complex-reasoning Text-to-SQL. It employs schema-grounded (Li et al., 2026a) MCTS to generate diverse reasoning trajectories and evaluates them using execution consistency and SQL-specific semantic signals. High-quality trajectories refine the policy model, while trajectory preference pairs train a SQL Process Preference Model (SQL-PPM). The learned SQL-PPM subsequently guides intermediate action selection in MCTS, forming a self-evolving loop between search, SQL-specific verification, policy refinement, and process preference learning.

We evaluate **RStar-SQL** on LogicCat and Archer, which cover mathematical, physical, commonsense, and hypothetical reasoning. **RStar-SQL** achieves execution accuracies of 35.41% and 47.82% on LogicCat and Archer, respectively. Ablation studies further demonstrate the complementary effects of SQL-specific semantic verification, process preference guidance, and iterative self-evolution.

Our main contributions are summarized as follows:

❶ We develop SQL-specific trajectory verification that combines execution consistency with semantic evidence over schema grounding, execution behavior, join validity, and type–operator compatibility, enabling reliable process preference construction from MCTS trajectories.

❷ We introduce SQL-PPM to convert verified trajectory preferences into reusable process guidance for subsequent MCTS, integrating structured SQL search with iterative policy–preference self-evolution.

❸ RStar-SQL achieves execution accuracies of 35.41% and 47.82% on LogicCat and Archer, respectively, outperforming closed-source models on LogicCat and open-source coder models on both benchmarks with only a 7B backbone.

2 PRELIMINARIES

Problem formulation. Let a relational database (Liu et al., 2024) be represented as $\mathcal{D} = (\mathcal{T}, \mathcal{C}, \mathcal{R}, \mathcal{V})$, where $\mathcal{T}$ is the set of tables, $\mathcal{C}$ denotes their columns, $\mathcal{R}$ contains primary–foreign

key relations, and $\mathcal{V}$ denotes database values (Hong et al., 2025). Given a natural-language question q, optional evidence e, and database $\mathcal{D}$, the goal is to generate an SQL query y whose execution result correctly answers q:

$$y^* = \arg\max_{y \in \mathcal{Y}(\mathcal{D})} P(y \mid q, e, \mathcal{D}), \tag{1}$$

where $\mathcal{Y}(\mathcal{D})$ is the set of valid SQL queries over $\mathcal{D}$. Complex-reasoning Text-to-SQL (Liu et al., 2026b) differs from conventional schema matching because the question may involve arithmetic computation, physical or domain knowledge, commonsense inference, hypothetical assumptions, and implicit constraints. A model must therefore reason not only about which tables and columns are relevant, but also about how the inferred logic should be instantiated as SQL predicates, functions, joins, aggregations, orderings, and nested queries.

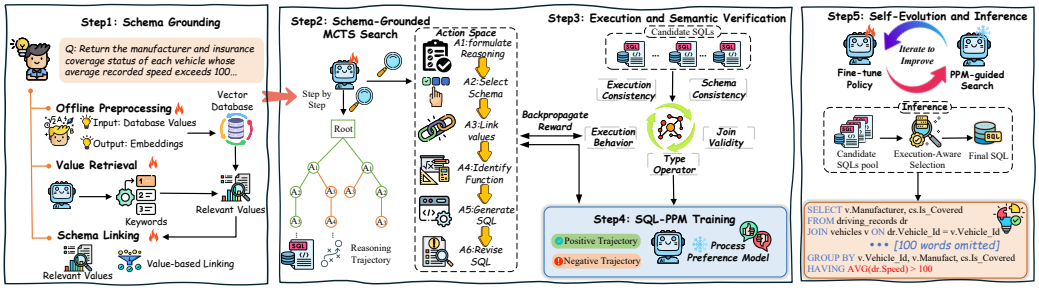


Figure 2: Overview of the RStar-SQL pipeline. (1) schema grounding and retrieval; (2) schema-grounded MCTS for step-by-step SQL reasoning; (3) execution and semantic verification with reward backpropagation; (4) SQL-PPM training from trajectory preferences; and (5) iterative self-evolution and execution-aware SQL selection.

Monte Carlo Tree Search. Monte Carlo Tree Search (Zhang et al., 2024) incrementally constructs a search tree $\Psi = (\mathcal{S}, \mathcal{E})$, where each node $s \in \mathcal{S}$ represents a partial reasoning state and each edge $(s, a) \in \mathcal{E}$ corresponds to an action a. Starting from the root state s_0, MCTS repeatedly performs four operations:

- **Selection.** Starting from s_0, MCTS recursively selects an action that balances exploitation and exploration using the Upper Confidence Bound for Trees:

$$a^* = \arg\max_{a \in \mathcal{A}(s)} \left[\frac{Q(s,a)}{N(s,a)} + c\sqrt{\frac{\log N(s)}{N(s,a)}} \right], \tag{2}$$

  where $Q(s,a)$ is the accumulated return, $N(s,a)$ and $N(s)$ are the visit counts of the action and state, respectively, and c controls exploration.
- **Expansion.** At a selected non-terminal state s_i, the policy model samples one or more actions as $a_{i,j} \sim \pi_\phi(\cdot \mid x, h_i)$ and generates the corresponding child states $s_{i+1,j} = f(s_i, a_{i,j})$, where x is the task input, h_i is the preceding reasoning history, and f is the state transition function.
- **Simulation.** From an expanded state, MCTS continues selecting or sampling actions until reaching a terminal state s_d. The resulting root-to-leaf path forms a trajectory $\tau = s_0 \oplus a_1 \oplus s_1 \oplus \cdots \oplus a_d \oplus s_d$, which is evaluated by a task-specific terminal reward $R(\tau)$.
- **Backpropagation.** The terminal reward $R(\tau)$ is accumulated into $Q(s_i, a_i)$ for every state–action pair visited by trajectory τ, while the corresponding visit count $N(s_i, a_i)$ is increased by one. The empirical action value is then computed as

$$\overline{Q}(s_i, a_i) = \frac{Q(s_i, a_i)}{N(s_i, a_i)}. \tag{3}$$

For complex-reasoning Text-to-SQL, MCTS searches over intermediate schema-grounding and SQL-construction decisions until producing a complete query. Its terminal rewards and retrospective action values are subsequently used for trajectory filtering and SQL-PPM training.

3 METHODOLOGY

Given a user question $x = (q, e, \mathcal{D})$, RStar-SQL follows a self-evolving search-and-learning pipeline, as illustrated in Figure 2. It first performs schema-grounded MCTS to generate and verify diverse SQL reasoning trajectories. The verified trajectories are then used to refine the policy model and train SQL-PPM, which guides subsequent search rounds. At inference time, the latest policy and SQL-PPM jointly produce and select the final SQL query.

3.1 SCHEMA-GROUNDED SQL TRAJECTORY SEARCH

Schema-grounded initialization. Before search, we retrieve question-relevant database values and obtain an initial schema-linking result $\mathcal{L}_0 = (\mathcal{T}_0, \mathcal{C}_0, \mathcal{V}_0, \mathcal{G}_0)$, where $\mathcal{T}_0$ and $\mathcal{C}_0$ denote candidate tables and columns, $\mathcal{V}_0$ contains retrieved value evidence, and $\mathcal{G}_0$ describes their relational connections. Rather than treating $\mathcal{L}_0$ as an irrevocable prediction, we use it only to initialize the root state. Subsequent schema-selection actions can therefore revise the grounding result when it becomes inconsistent with later reasoning or execution feedback.

Search state and structured actions. At step i, the search state is represented as

$$s_i = (\widetilde{q}_i, \mathcal{S}_i, \mathcal{V}_i, \mathcal{F}_i, y_i, h_i)\,, \tag{4}$$

where $\widetilde{q}_i$ denotes the current interpretation of the question, $\mathcal{S}_i$ the selected schema context, $\mathcal{V}_i$ the column–value mappings, $\mathcal{F}_i$ the required SQL functions, y_i the current SQL candidate, and h_i the accumulated reasoning and execution history. To expose complex SQL construction as a sequence of revisable decisions, we define six action types (Liu et al., 2026a): *reasoning reformulation*, *schema selection*, *column–value identification*, *column–function identification*, *SQL generation*, and *SQL revision*. The first four actions progressively ground the question into explicit schema, value, and operator constraints, while the latter two construct and refine the executable SQL. Given state s_i and action type a_i, the policy samples the next state as $s_{i+1} \sim \pi_\phi(\cdot \mid x, h_i, \mathrm{P}(a_i))$. For each valid action, we sample multiple child states and remove duplicates with identical schema selections or SQL queries, preserving diverse reasoning paths while avoiding redundant exploration.

MCTS trajectory generation. Starting from the schema-grounded root state, MCTS follows the standard selection, expansion, simulation, and backpropagation procedure introduced in Section Preliminaries. Our modifications lie in the SQL-specific state representation, the structured action space, and the verification-based terminal reward described in Section Verification. A root-to-leaf path ending at terminal state s_d forms a complete SQL reasoning trajectory

$$\tau = x \oplus a_1 \oplus s_1 \oplus \cdots \oplus a_d \oplus s_d, \tag{5}$$

where $\oplus$ denotes trajectory concatenation and s_d contains a terminal SQL query y_τ. Before SQL-PPM is available, search is guided only by standard MCTS statistics. In later self-evolution rounds, SQL-PPM provides an additional process prior for node selection and simulation.

3.2 SQL-SPECIFIC VERIFICATION AND TRAJECTORY SUPERVISION

Execution consistency. For an SQL-generation or SQL-revision action, the policy produces a candidate set $\mathcal{Y} = \{y_1, \ldots, y_K\}$. We execute the candidates against the target database and measure the execution consistency of candidate y as

$$R_{\mathrm{sc}}(y) = \frac{1}{K} \sum_{j=1}^{K} \mathbb{I}\left[\mathrm{EX}(y, \mathcal{D}) = \mathrm{EX}(y_j, \mathcal{D})\right], \tag{6}$$

where $\mathrm{EX}(\cdot, \mathcal{D})$ denotes SQL execution and $\mathbb{I}[\cdot]$ is the indicator function. This signal rewards candidates supported by independently generated queries while assigning low scores to invalid or unstable generations. It is used as a terminal search signal; final candidate aggregation is described separately in later Section.

SQL semantic verification. Execution agreement alone is unreliable because distinct incorrect queries may produce identical results. We therefore augment it with four weak semantic signals: schema grounding G_{sch} verifies referenced tables, columns, and literals; execution behavior G_{exe} assesses executability, degenerate outputs, and answer-type compatibility; join validity G_{join} detects missing predicates and Cartesian products; and type–operator compatibility G_{op} checks whether operators match column types. The terminal reward is defined as

$$R(\tau) = w_{\text{sc}} R_{\text{sc}}(y_\tau) + w_{\text{sch}} G_{\text{sch}}(y_\tau) + w_{\text{exe}} G_{\text{exe}}(y_\tau) + w_{\text{join}} G_{\text{join}}(y_\tau) + w_{\text{op}} G_{\text{op}}(y_\tau), \quad (7)$$

where all components are normalized to $[0, 1]$ and used as weak ranking evidence rather than oracle correctness labels.

3.3 SQL Process Preference Learning

Directly regressing the numerical MCTS values is unreliable because their scale depends on search coverage, rollout stochasticity, and downstream policy quality. We instead convert verified trajectories into relative preferences, requiring SQL-PPM (He et al., 2025; Zhang et al., 2026) only to distinguish which of two reasoning paths is more promising.

Trajectory preference construction. For input x, let $\mathcal{T}(x)$ denote the set of trajectories generated by MCTS. We construct positive trajectories from executable candidates that pass terminal verification, and negative trajectories from unsuccessful or semantically implausible candidates. The process quality of a trajectory τ is measured by its average and minimum retrospective action values:

$$m(\tau) = \frac{1}{|\tau|} \sum_{(s_i, a_i) \in \tau} \overline{Q}(s_i, a_i),$$
$$b(\tau) = \min_{(s_i, a_i) \in \tau} \overline{Q}(s_i, a_i). \quad (8)$$

Here, $m(\tau)$ measures the overall trajectory quality, while $b(\tau)$ penalizes trajectories containing unreliable intermediate decisions. We select positive trajectories with high $(m(\tau), b(\tau))$ values and use high-valued unsuccessful trajectories as hard negatives. Each preference example is represented as (x, τ^+, τ^-), where $\tau^+ \succ \tau^-$. Although preferences are constructed over complete trajectories, SQL-PPM can also score a partial trajectory paired with a candidate next action during search. Details of the MCTS cold-start stage are provided in Appendix A.2.

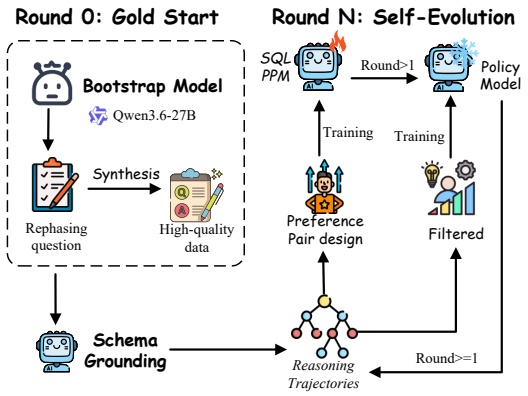


Figure 3: RStar-SQL self-evolution pipeline.

Preference model and objective. SQL-PPM is initialized from a pretrained code-oriented language model whose language-model head is replaced by a scalar value head. For encoded sequence u, its score is $r_\theta(u) = \mathbf{w}^\top \mathbf{h}_\theta^{\text{last}}(u) + b$. Given a preferred–dispreferred pair, we optimize a Bradley–Terry ranking objective:

$$\mathcal{L}_{\text{PPM}}(\theta) = -\mathbb{E}_{(x, \tau^+, \tau^-) \sim \mathcal{D}_{\text{pref}}} \left[\omega_x \log \sigma\big(r_\theta(x \oplus \tau^+) - r_\theta(x \oplus \tau^-)\big) \right]. \quad (9)$$

where σ is the sigmoid function and $\omega_x = 1/\max(1, |\mathcal{T}^+(x)|\,|\mathcal{T}^-(x)|)$ balances questions producing different numbers of trajectory pairs. The objective encourages a higher score for the preferred reasoning process without requiring calibrated absolute rewards.

SQL-PPM-guided MCTS. For parent state s and candidate action a, SQL-PPM scores the complete preceding context together with the candidate action as $p_\theta(s, a) = r_\theta(\text{Encode}(s) \oplus \text{Encode}(a))$. The scores are min–max normalized among sibling nodes to obtain $\widetilde{p}_\theta(s, a) \in [0, 1]$. We then augment the standard UCT score with the learned process prior:

$$U_{\text{SQL}}(s, a) = \overline{Q}(s, a) + c\sqrt{\frac{\log N(s)}{N(s, a)}} + \lambda \widetilde{p}_\theta(s, a), \quad (10)$$

where λ controls the strength of preference guidance. SQL-PPM provides prospective guidance for action selection, while terminal execution and semantic rewards provide retrospective supervision.

Algorithm 1 RStar-SQL Policy–Preference Evolution

Require: Training set $\mathcal{D}$, bootstrap policy π^{boot}, base policy π^{base}, maximum rounds R
Ensure: Evolved policy π_ϕ and SQL-PPM r_θ

$\mathcal{T}^{(0)} \leftarrow \text{MCTS}(\mathcal{D}, \pi^{\text{boot}}, \varnothing)$
$\mathcal{D}_{\text{SFT}} \leftarrow \textsc{FilterSFT}(\mathcal{T}^{(0)})$
$\pi_\phi^{(1)} \leftarrow \text{SFT}(\pi^{\text{base}}, \mathcal{D}_{\text{SFT}})$
$\mathcal{D}_{\text{pref}} \leftarrow \textsc{Pair}(\mathcal{T}^{(0)})$
$r_\theta^{(0)} \leftarrow \textsc{PPMTrain}(\mathcal{D}_{\text{pref}})$
for $r = 1, \ldots, R$ **do**
  $\mathcal{T}^{(r)} \leftarrow \text{MCTS}(\mathcal{D}, \pi_\phi^{(r)}, r_\theta^{(r-1)})$
  $\mathcal{D}_{\text{SFT}} \leftarrow \mathcal{D}_{\text{SFT}} \cup \textsc{FilterSFT}(\mathcal{T}^{(r)})$
  $\mathcal{D}_{\text{pref}} \leftarrow \mathcal{D}_{\text{pref}} \cup \textsc{Pair}(\mathcal{T}^{(r)})$
  $\pi_\phi^{(r+1)} \leftarrow \text{SFT}(\pi_\phi^{(r)}, \mathcal{D}_{\text{SFT}})$
  $r_\theta^{(r)} \leftarrow \textsc{PPMTrain}(\mathcal{D}_{\text{pref}})$
end for
return $\pi_\phi^{(R+1)}, r_\theta^{(R)}$

3.4 Iterative Self-Evolution and Inference

As illustrated in Figure 3, bootstrap MCTS without preference guidance produces trajectories for filtered SFT of the smaller base policy and initial SQL-PPM training. Each subsequent round uses the previous SQL-PPM to guide MCTS: $\textsc{FilterSFT}(\cdot)$ supplies policy supervision, while $\textsc{Pair}(\cdot)$ supplies preference pairs. This loop jointly improves the policy and search guidance. SQL-PPM is trained separately and only scores candidate actions.

At inference time, the latest policy and SQL-PPM run schema-grounded MCTS. Verified terminal queries are grouped by execution equivalence, and the lowest-latency query in the largest valid cluster $\mathcal{C}^*$ is selected:

$$y_{\text{final}} = \arg\min_{y \in \mathcal{C}^*} \mathrm{T}(y, \mathcal{D}). \tag{11}$$

Here, $\mathrm{T}(y, \mathcal{D})$ is the benchmark executor's wall-clock latency under identical hardware and timeout settings. Ties favor fewer SQL tokens, then earlier generation. The procedure requires no gold SQL, gold execution results, or annotated reasoning trajectories.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate RStar-SQL on LogicCat (Liu et al., 2026b), Archer (Zheng et al., 2024), BIRD (Li et al., 2023), Spider (Yu et al., 2018), and Spider-DK (Gan et al., 2021). LogicCat and Archer are the primary reasoning-intensive benchmarks, while the remaining datasets evaluate conventional Text-to-SQL performance and cross-domain robustness.

Models. We use Qwen3-Embedding-0.6B (Zhang et al., 2025) for schema retrieval and adopt Qwen2.5-Coder-7B-Instruct (Zhang et al., 2025) and Qwen3-8B (Yang et al., 2025) as policy backbones, and we use Qwen3.6-27B (Yang et al., 2025) as the bootstrap model for the initial cold-start round, with details provided in Appendix A.2. SQL-PPM is initialized from a backbone of comparable scale and trained on preference pairs derived from verified MCTS trajectories.

Evaluation Metric. Following prior work (Pourreza et al., 2025a), we report Execution Accuracy (EX) using the official evaluator of each benchmark.

Implementation Details. RStar-SQL is implemented in PyTorch, and model training is conducted on 24 NVIDIA H100 GPUs. In Round 0, we construct approximately 100K training instances. The training and inference hyperparameters follow the settings of prior work (Li et al., 2025; Guan et al., 2025), with a default MCTS rollout budget of 8; full implementation details are provided in Appendix B.1.

Table 1: Main results on reasoning-intensive and general Text-to-SQL benchmarks. LogicCat and Archer emphasize complex reasoning, while BIRD, Spider, and Spider-DK evaluate general Text-to-SQL performance. All results are execution accuracy (%).

Model	LogicCat	Archer	BIRD	Spider	Spider-DK	Method	LogicCat	Archer	BIRD	Spider	Spider-DK
Large-Scale Systems						*Open-Source Specialized Methods: Qwen2.5-Coder-7B*					
o3-mini (Lai et al., 2025)	14.96	39.28	59.29	85.18	68.28	DIN-SQL (Pourreza & Rafiei, 2023)	11.56	34.41	52.45	79.30	63.29
GPT-5.4 (Singh et al., 2026)	18.24	52.21	69.55	87.13	74.89	CHESS (Talaei et al., 2024)	19.21	35.54	68.31	86.30	76.17
Claude-Ops-4.6 (Joshi, 2026)	20.84	**54.89**	68.77	88.10	**80.97**	CHASE-SQL (Pourreza et al., 2025a)	22.13	35.78	69.68	85.12	78.82
DeepSeek-V4-Pro (DeepSeek-AI et al., 2026)	20.67	50.11	68.28	88.05	79.45	DeepEye-SQL (Li et al., 2026a)	21.22	21.15	**71.27**	86.89	80.34
Gemini-3.1 (Cazzaniga, 2026)	18.31	46.12	66.24	87.85	78.21	Alpha-SQL (Li et al., 2025)	19.78	30.12	67.92	84.93	76.21
GLM-4.7 (Team et al., 2025)	19.11	47.94	64.82	87.67	79.01	SQL-O1 (Lyu et al., 2025)	16.42	26.56	66.70	85.10	75.84
Kimi-2.5 (Team et al., 2026)	18.97	47.87	64.46	86.60	77.86	IESR (Liu et al., 2026a)	24.28	37.28	62.22	83.36	67.19
Generalized Base Model: Qwen3-8B						*Specialized Base Model: Qwen2.5-Coder-7B*					
SFT Policy	30.46	41.26	61.27	85.23	72.98	SFT Policy	28.32	44.67	63.78	86.12	75.23
SFT + SQL-PPM	31.17	44.74	63.12	86.01	74.12	SFT + SQL-PPM	31.22	46.43	64.64	87.46	76.85
RStar-SQL	**33.26**	**45.16**	**66.23**	**88.19**	**77.21**	**RStar-SQL**	**35.41**	**47.82**	**68.54**	**88.63**	**78.64**

Table 2: Ablation of SQL-specific semantic reward signals with Qwen2.5-Coder.

Semantic Signal	LogicCat EX ↑	Archer EX ↑	Valid SQL ↑	Schema Acc. ↑	Join Err. Rate ↓
Exec. consistency	26.34	43.26	74.29	76.38	26.96
+ All signals	**28.32**	**44.67**	**78.19**	**79.77**	**24.12**
w/o schema	27.17	43.28	76.37	78.97	24.98
w/o execution behavior	28.18	44.12	76.85	79.21	24.46
w/o join validity	27.34	43.54	77.89	78.11	27.38
w/o type–operator	27.67	41.84	73.95	79.43	24.78

Table 3: Performance across self-evolution rounds with SQL-PPM on Qwen2.5-Coder.

Round	SQL-PPM	LogicCat	Archer	BIRD	Spider	Spider-DK
Round 1	✗	28.32	44.67	63.78	86.12	75.23
	✓	31.22	46.43	64.64	87.46	76.85
Round 2	✗	29.34	45.62	64.98	86.76	75.92
	✓	33.67	46.99	65.78	87.92	76.73
Round 3	✗	30.28	45.96	64.89	86.81	76.06
	✓	34.49	47.21	66.20	88.46	77.06
Round 4	✗	30.36	46.07	64.64	86.78	75.98
	✓	**35.41**	**47.82**	**68.54**	**88.63**	**78.64**

4.2 Main Results

As shown in Table 1, RStar-SQL achieves its strongest gains on the reasoning-intensive LogicCat and Archer benchmarks, reaching 35.41% and 47.82% EX with Qwen2.5-Coder-7B. The progression from SFT Policy to SFT with SQL-PPM and then to full RStar-SQL indicates that process preference guidance and iterative policy–preference refinement provide complementary improvements beyond one-pass training. RStar-SQL also maintains competitive performance on BIRD, Spider, and Spider-DK, suggesting that the learned process preferences transfer to general SQL construction while benefiting most when reasoning and SQL decisions are tightly coupled. More error trajectories and details provided in Appendix A.3 and A.6.

4.3 Ablation Study

Analysis by Core Components Figure 4 shows that each component contributes to RStar-SQL. Schema-grounded MCTS provides the largest initial gain, while semantic verification and verified SFT improve trajectory quality and transfer successful search experience into the policy. SQL-PPM further guides intermediate action selection before terminal feedback is available. The stronger performance of the full model suggests that iterative policy–preference co-evolution provides additional gains beyond a one-pass combination of these components.

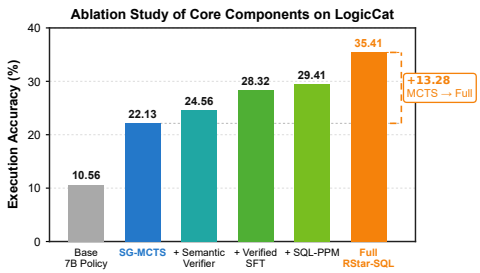


Figure 4: Ablation study of the core components of RStar-SQL on LogicCat. **Base 7B Policy** denotes Few-shot, **SG-MCTS** denotes Schema-Grounded MCTS, **SFT** denotes supervised fine-tuning on verified trajectories, and **SQL-PPM** denotes the SQL Process Preference Model.

Effect of SQL-Specific Semantic Rewards. As shown in Table 2, execution consistency alone provides a weak supervision signal, since incorrect SQL queries may still produce identical execution outcomes. Incorporating schema, execution-behavior, join, and type–operator signals consistently improves both execution accuracy and the correspond-

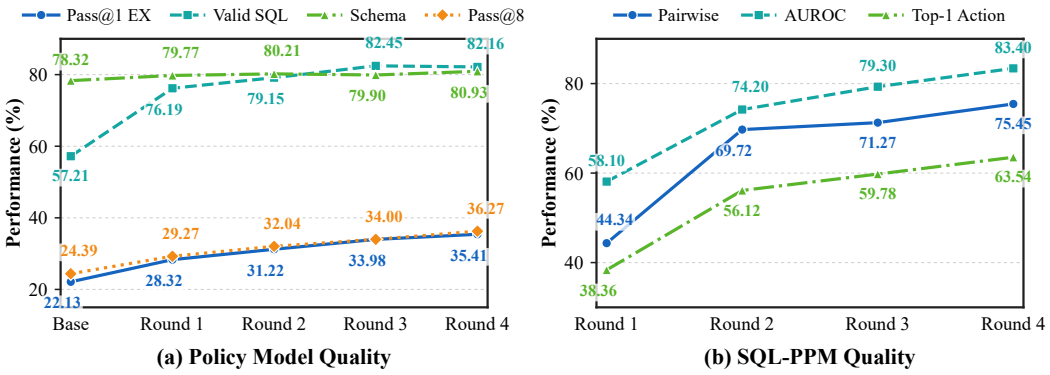


Figure 5: Evolution of policy and preference model quality across self-evolution rounds. **Left** for Policy model: Pass@1 EX, valid-SQL rate, schema accuracy, and Pass@8. **Right** for SQL-PPM: pairwise accuracy, AUROC, and Top-1 action-ranking accuracy.

ing diagnostic metrics. These complementary signals provide a more reliable estimate of trajectory quality, yielding stronger supervision for MCTS from the first self-evolution round.

Effect of Iterative Self-Evolution. As shown in Table 3, iterative self-evolution offers a key advantage: each round improves the policy and provides better supervision for the next round. As the policy becomes stronger, MCTS explores more executable and semantically plausible trajectories, yielding cleaner positive examples and more informative hard negatives for both policy refinement and SQL-PPM training. In turn, the improved SQL-PPM better prioritizes promising schema-grounding and SQL-construction decisions, reducing search effort on unproductive branches and further improving the trajectory distribution. This mutually reinforcing policy–preference loop allows RStar-SQL to progressively convert transient search outcomes into reusable decision knowledge, producing gains that cannot be achieved by a single round of search or fine-tuning alone.

Evolution of Policy and Preference Quality. Figure 5 illustrates the joint evolution of the policy and SQL-PPM. In Figure 5(a), the policy shows a rapid improvement in valid-SQL rate during the early rounds, while schema accuracy changes more moderately. Meanwhile, *Pass@1* and *Pass@8* continue to increase across rounds, suggesting that later gains increasingly come from improved reasoning over constraints, operators, joins, and query composition rather than basic executability or schema grounding alone. Figure 5(b) shows a parallel improvement in SQL-PPM, with stronger *pairwise discrimination*, *AUROC*, and *top-1* action ranking over successive rounds. This indicates that higher-quality trajectories produced by the evolving policy provide cleaner positives and harder negatives for preference learning. The improved SQL-PPM then guides subsequent MCTS toward more promising intermediate decisions, forming a mutually reinforcing policy–preference evolution loop.

4.4 Case Study

Analysis by Error Case. At Round 2, we sample 10,000 erroneous trajectories to analyze the failure patterns shown in Figure 6; additional cases are provided in Appendix A.6. Filtering and value-grounding errors dominate the distribution, indicating that the main bottleneck is no longer SQL executability but incorrect semantic constraints and database-value alignment. The case on the right illustrates this issue: although the SQL is executable, an incorrectly grounded literal produces a result inconsistent with the gold answer.

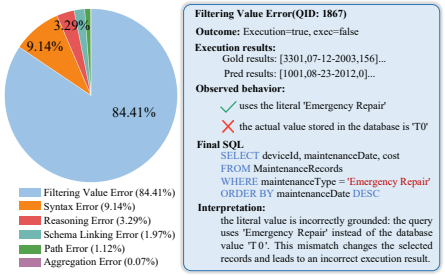


Figure 6: Error distribution and example trajectories at Round 2.

Analysis by Rollout Efficiency. As shown in Table 4, increasing the MCTS rollout budget substantially raises inference cost, highlighting the limitation of scaling search through broader exploration alone. Rather than simply increasing the number of rollouts, RStar-SQL uses SQL-PPM to prioritize more promising intermediate decisions and concentrate computation on higher-quality regions of the search space. This improves search efficiency by favoring exploration quality over quantity. Further analysis is provided in Appendix A.4.

Table 4: Effect of the MCTS rollout budget on inference efficiency in the first round on LogicCat on H100.

Rollouts	LLM Calls	Avg. Tokens	Latency	EX
4	8.9	10.4k	1.5 min	26.17
8	16.6	18.3k	3.9 min	28.32
16	30.2	28.9k	7.7 min	28.45
32	42.1	46.8k	14.3 min	27.99
64	61.4	73.6k	27.2 min	28.63

5 Related Work

5.1 LLM-Based Text-to-SQL

LLM-based Text-to-SQL has evolved from one-shot (Rajkumar et al., 2022; Nan et al., 2023; Chang & Fosler-Lussier, 2023; Sun et al., 2023) generation toward modular systems that coordinate schema grounding, retrieval, generation, execution, verification, and revision. Early methods improve robustness through task decomposition, diverse candidate generation, and iterative correction (Pourreza & Rafiei, 2023; Gao et al., 2024; Sun et al., 2024; Talaei et al., 2024; Pourreza et al., 2025a; Li et al., 2026a). More recent approaches formulate SQL generation as a long-horizon interaction process, allowing models to repeatedly execute and refine queries using database feedback (Xu et al., 2025; Dai et al., 2026b; Hua et al., 2026). While these methods establish interaction and refinement as key components, their feedback is primarily consumed within the current trajectory or training objective. RStar-SQL instead transforms verified search experience into reusable guidance for subsequent exploration.

5.2 Search-Based Text-to-SQL

Test-time search improves Text-to-SQL through candidate exploration, execution-guided revision, and result-based selection. Existing methods use self-consistency, execution agreement, or specialized workflows to identify reliable queries (Sun et al., 2024; Zhai et al., 2025; Talaei et al., 2024; Pourreza et al., 2025a; Li et al., 2026a; Tang et al., 2026). However, most follow predefined search procedures rather than explicitly modeling reasoning as a state–action tree. Alpha-SQL (Li et al., 2025) introduces MCTS over partial SQL states, but focuses primarily on SQL construction and terminal candidates. RStar-SQL instead treats schema grounding and SQL construction as jointly revisable decisions and learns process preferences from verified trajectories to guide future search.

5.3 Process Preference Learning and Self-Evolution

Outcome supervision evaluates complete trajectories but provides limited credit to intermediate decisions. Process supervision addresses this limitation by deriving step-level signals or trajectory preferences from search outcomes (Xie et al., 2024; Park et al., 2025; Yang et al., 2026b). Iterative frameworks such as rStar-Math (Guan et al., 2025) further co-evolve policies and process preference models, while recent Text-to-SQL reinforcement learning methods reduce reward sparsity through partial rewards, trajectory aggregation, and step-level credit assignment (Pourreza et al., 2025b; Li et al., 2026b; Dai et al., 2026a). Since SQL reasoning must satisfy schema, value, type, join, and execution constraints, generic process signals remain insufficient. RStar-SQL therefore learns SQL-specific trajectory preferences and uses SQL-PPM to guide successive MCTS rounds.

6 Conclusion

We introduced RStar-SQL, a self-evolving process preference framework for complex-reasoning Text-to-SQL. By converting verified MCTS trajectories into reusable supervision, RStar-SQL jointly improves the policy and SQL-PPM across self-evolution rounds. Experiments confirm the benefits of schema-grounded search, SQL-specific verification, and process preference learning.

AI USE STATEMENT

In this work, we used generative AI tools to assist with the interpretation and presentation of experimental results, preparation of visualization code, literature search, and language editing of the manuscript. Generative AI was not used to generate experimental results or replace empirical validation. We take full responsibility for the final content of this work, including all text, claims, experiments, and artifacts.

ETHICS STATEMENT

This work uses publicly available datasets and models and does not involve human subjects or sensitive personal data. The proposed method is intended for research on Text-to-SQL reasoning; potential risks mainly concern inappropriate use of automated SQL generation in privacy- or security-sensitive applications. Appropriate safeguards and human oversight are recommended for real-world deployment.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide detailed descriptions of the RStar-SQL algorithm, training objectives, search procedure, and experimental settings in the main paper and Appendices A and B. The appendices further report implementation details (Appendix A), hyperparameters (Appendix B.1), action definitions (Appendix A.5), prompts (Appendix B.4), and additional ablation studies (Appendices A.3 and A.4). All evaluations are conducted on publicly available Text-to-SQL benchmarks using their official execution-based metrics. We will also release the code and configuration files upon publication.

REFERENCES

Luca Cazzaniga. Schema for gemini 3 pro image: A structured methodology for controlled ai image generation on google's native multimodal model., 2026. URL https://zenodo.org/doi/10.5281/zenodo.18721380.

Shuaichen Chang and Eric Fosler-Lussier. How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings. *ArXiv*, abs/2305.11853, 2023. URL https://api.semanticscholar.org/CorpusID:258823166.

Yaxun Dai, Baolin Sun, Junying Wang, Pengfei Wang, Yingqi Gao, Xuemei Dong, Mengdie Chu, Xiang Qi, and Pingfu Chao. Every step counts: Step-level credit assignment for tool-integrated text-to-SQL. *arXiv preprint arXiv:2605.04719*, 2026a. doi: 10.48550/arXiv.2605.04719. URL https://arxiv.org/abs/2605.04719.

Yaxun Dai, Wenxuan Xie, Xialie Zhuang, Tianyu Yang, Ziyi Liu, Haiqin Yang, Yiying Yang, Yuhang Zhao, Pingfu Chao, and Wenhao Jiang. ReEx-SQL: Reasoning with execution-aware reinforcement learning for text-to-SQL. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 824–847, San Diego, California, United States, July 2026b. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.35. URL https://aclanthology.org/2026.acl-long.35/.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, and Bochao Wu. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Yujian Gan, Xinyun Chen, and Matthew Purver. Exploring underexplored limitations of cross-domain text-to-sql generalization, 2021. URL https://arxiv.org/abs/2109.05157.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145, January 2024. ISSN 2150-8097. doi: 10.14778/3641204.3641221. URL https://doi.org/10.14778/3641204.3641221.

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

Mingqian He, Yongliang Shen, Wenqi Zhang, Qiuying Peng, Jun Wang, and Weiming Lu. STaR-SQL: Self-taught reasoner for text-to-SQL. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 24365–24375, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1187. URL https://aclanthology.org/2025.acl-long.1187/.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. Next-generation database interfaces: A survey of llm-based text-to-sql. *IEEE Transactions on Knowledge and Data Engineering*, 37(12):7328–7345, 2025. doi: 10.1109/TKDE.2025.3609486.

Harper Hua, Zhen Han, Zhengyuan Shen, Meng-Chieh Lee, Sheng Guan, Qi Zhu, Sullam Jeoung, Yueyan Chen, Yunfei Bai, Shuai Wang, Vassilis N. Ioannidis, and Huzefa Rangwala. SQL-trail: Multi-turn reinforcement learning with interleaved feedback for text-to-SQL. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 36224–36246, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1677. URL https://aclanthology.org/2026.acl-long.1677/.

Satyadhar Joshi. Architectural advances and performance benchmarks of large language models in light of anthropic's claude opus 4.6. *Preprints*, February 2026. doi: 10.20944/preprints202602.0537.v1. URL https://doi.org/10.20944/preprints202602.0537.v1.

Xin Lai, Junyi Li, Wei Li, Tao Liu, Tianjian Li, and Hengshuang Zhao. Mini-o3: Scaling up reasoning patterns and interaction turns for visual search, 2025. URL https://arxiv.org/abs/2509.07969.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *International Conference on Learning Representations*, volume 2025, pp. 28691–28735, 2025.

Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. Alpha-SQL: Zero-shot text-to-SQL using Monte Carlo tree search. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 36810–36830. PMLR, 2025. URL https://proceedings.mlr.press/v267/li25dt.html.

Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. DeepEye-SQL: A software-engineering-inspired text-to-SQL framework. *Proceedings of the ACM on Management of Data*, 4(3), 2026a. doi: 10.1145/3802035. URL https://arxiv.org/abs/2510.17586.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM already serve as a database interface? a BIg bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems*, volume 36, pp. 42330–42357, 2023. URL https://arxiv.org/abs/2305.03111.

Long Li, Zhijian Zhou, Jiangxuan Long, Peiyang Liu, Weidi Xu, Zhe Wang, Shirui Pan, and Chao Qu. SQL-ASTRA: Alleviating sparse feedback in agentic SQL via column-set matching and trajectory aggregation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 18474–18491, San Diego, California, United States, July 2026b. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.920. URL https://aclanthology.org/2026.findings-acl.920/.

Tao Liu, Jiafan Lu, Bohan Yu, Pengcheng Wu, Liu Haixin, Guoyu Xu, Li Xiangheng, Lixiao Li, Jiaming Hou, Zhao Shijun, Xinglin Lyu, Kunli Zhang, Yuxiang Jia, and Hongyin Zan. Iesr:efficient mcts-based modular reasoning for text-to-sql with large language models, 2026a. URL https://arxiv.org/abs/2602.05385.

Tao Liu, Xutao Mao, Hongying Zan, Dixuan Zhang, Yifan Li, Haixin Liu, Lulu Kong, Jiaming Hou, Rui Li, Yunlong Li, Aoze Zheng, Zhiqiang Zhang, Zhewei Luo, Kunli Zhang, and Min Peng. LogicCat: A chain-of-thought text-to-SQL benchmark for complex reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29958–29966, 2026b. doi: 10.1609/aaai.v40i36.40243. URL https://ojs.aaai.org/index.php/AAAI/article/view/40243.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yu xin Zhang, Ju Fan, Guo-Liang Li, Nan Tang, and Yuyu Luo. 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*, 37:5735–5754, 2024. URL https://api.semanticscholar.org/CorpusID:271843296.

Shuai Lyu, Haoran Luo, Ripeng Li, Zhonghong Ou, Jiangfeng Sun, Yang Qin, Xiaoran Shang, Meina Song, and Yifan Zhu. Sql-o1: A self-reward heuristic dynamic search method for text-to-sql, 2025. URL https://arxiv.org/abs/2502.11741.

Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. Enhancing text-to-SQL capabilities of large language models: A study on prompt design strategies. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 14935–14956, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.996. URL https://aclanthology.org/2023.findings-emnlp.996/.

Sungjin Park, Xiao Liu, Yeyun Gong, and Edward Choi. Ensembling large language models with process reward-guided tree search for better complex reasoning. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 10256–10277, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.515. URL https://aclanthology.org/2025.naacl-long.515/.

Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. *arXiv preprint arXiv:2304.11015*, 2023. URL https://arxiv.org/abs/2304.11015.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *International Conference on Learning Representations*, 2025a. URL https://arxiv.org/abs/2410.01943.

Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, and Sercan O. Arik. Reasoning-SQL: Reinforcement learning with SQL-tailored partial rewards for reasoning-enhanced text-to-SQL. *arXiv preprint arXiv:2503.23157*, 2025b. doi: 10.48550/arXiv.2503.23157. URL https://arxiv.org/abs/2503.23157.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. Evaluating the text-to-sql capabilities of large language models. *ArXiv*, abs/2204.00498, 2022. URL https://api.semanticscholar.org/CorpusID:247922681.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, and Adi Ganesh. Openai gpt-5 system card, 2026. URL https://arxiv.org/abs/2601.03267.

Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron Parisi, Abhishek Kumar, Alex Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Gamaleldin Elsayed, Hanie Sedghi, Igor Mordatch, Isabelle Simpson, Izzeddin Gur, Jasper Snoek, Jeffrey Pennington, Jiri Hron, Kathleen Kenealy, Kevin Swersky, Kshiteej Mahajan, Laura Culp, Lechao Xiao, Maxwell L.

Bileschi, Noah Constant, Roman Novak, Rosanne Liu, Tris Warkentin, Yundi Qian, Yamini Bansal, Ethan Dyer, Behnam Neyshabur, Jascha Sohl-Dickstein, and Noah Fiedel. Beyond human data: Scaling self-training for problem-solving with language models, 2024. URL https://arxiv.org/abs/2312.06585.

Jinwang Song, Hongying Zan, Kunli Zhang, Lingling Mu, Yingjie Han, Haobo Hua, and Min Peng. JOLT-SQL: Joint loss tuning of text-to-SQL with confusion-aware noisy schema sampling. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 6040–6053, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.308. URL https://aclanthology.org/2025.emnlp-main.308/.

Ruoxi Sun, Sercan Arik, Rajarishi Sinha, Hootan Nakhost, Hanjun Dai, Pengcheng Yin, and Tomas Pfister. SQLPrompt: In-context text-to-SQL with minimal labeled data. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 542–550, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.39. URL https://aclanthology.org/2023.findings-emnlp.39/.

Ruoxi Sun, Sercan O. Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and Tomas Pfister. SQL-PaLM: Improved large language model adaptation for text-to-SQL. *Transactions on Machine Learning Research*, 2024. URL https://arxiv.org/abs/2306.00739.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. CHESS: Contextual harnessing for efficient SQL synthesis. *arXiv preprint arXiv:2405.16755*, 2024. URL https://arxiv.org/abs/2405.16755.

Min Tang, Lixin Zou, Shujie Cui, Weiqing Wang, Zhe Jin, Chengliang Li, and Shiuan-Ni Liang. Q-value guided text-to-sql generation: Structured reasoning meets efficient inference exploration. *Inf. Process. Manage.*, 63(4), June 2026. ISSN 0306-4573. doi: 10.1016/j.ipm.2025.104607. URL https://doi.org/10.1016/j.ipm.2025.104607.

5 Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, and Bin Chen. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models, 2025. URL https://arxiv.org/abs/2508.06471.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, and Ziwei Chai. Kimi k2.5: Visual agentic intelligence, 2026. URL https://arxiv.org/abs/2602.02276.

Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. Agentar-scale-sql: Advancing text-to-sql through orchestrated test-time scaling, 2025. URL https://arxiv.org/abs/2509.24403.

Yuxi Xie, Anirudh Goyal, Wenyue Zheng, Min-Yen Kan, Timothy P. Lillicrap, Kenji Kawaguchi, and Michael Shieh. Monte Carlo tree search boosts reasoning via iterative preference learning. *arXiv preprint arXiv:2405.00451*, 2024. URL https://arxiv.org/abs/2405.00451.

Zekun Xu, Siyu Xia, Chuhuai Yue, Jiajun Chai, Mingxue Tian, Xiaohan Wang, Wei Lin, Haoxuan Li, and Guojun Yin. Mtir-sql: Multi-turn tool-integrated reasoning reinforcement learning for text-to-sql, 2025. URL https://arxiv.org/abs/2510.25510.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, and Binyuan Hui. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Haolin Yang, Jipeng Zhang, Zhitao He, Alexander Zhou, and Yi R. Fung. Mars-sql: A multi-agent reinforcement learning framework for text-to-sql, 2026a. URL https://arxiv.org/abs/2511.01008.

Zerui Yang, Weichuan Wang, Yanwei Xu, Linqi Song, Yudai Matsuda, Wei Han, and Bo Bai. Memo-sql: Structured decomposition and experience-driven self-correction for training-free nl2sql, 2026b. URL https://arxiv.org/abs/2601.10011.

Zhewei Yao, Guoheng Sun, Łukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. Arctic-Text2SQL-r1: Simple rewards, strong reasoning in text-to-SQL. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 26966–26995, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1345. URL https://aclanthology.org/2026.findings-acl.1345/.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL https://aclanthology.org/D18-1425/.

Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter S Lasecki, and Dragomir Radev. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases, 2019. URL https://arxiv.org/abs/1909.05378.

Shuozhi Yuan, Limin Chen, Miaomiao Yuan, and Zhao Jin. Mcts-sql: Light-weight llms can master the text-to-sql through monte carlo tree search, 2025. URL https://arxiv.org/abs/2501.16607.

Shuozhi Yuan, Liming Chen, Miaomiao Yuan, and Zhao Jin. Mcts-sql: light-weight llms can master the text-to-sql through monte carlo tree search. In *Proceedings of the Fortieth AAAI Conference on Artificial Intelligence and Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence and Sixteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'26/IAAI'26/EAAI'26. AAAI Press, 2026. ISBN 978-1-57735-906-7. doi: 10.1609/aaai.v40i41.40751. URL https://doi.org/10.1609/aaai.v40i41.40751.

Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. Excot: Optimizing reasoning for text-to-sql with execution feedback, 2025. URL https://arxiv.org/abs/2503.19988.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. Rest-mcts*: Llm self-training via process reward guided tree search. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models, 2025. URL https://arxiv.org/abs/2506.05176.

Yuxin Zhang, Meihao Fan, Ju Fan, Mingyang Yi, Yuyu Luo, Guoliang Li, Bin Wu, and Wenchao Zhou. Reward-sql: Boosting text-to-sql via stepwise execution-aware reasoning and process-supervised rewards. *Proc. ACM Manag. Data*, 4(3), May 2026. doi: 10.1145/3802105. URL https://doi.org/10.1145/3802105.

Danna Zheng, Mirella Lapata, and Jeff Z. Pan. Archer: A human-labeled text-to-SQL dataset with arithmetic, commonsense and hypothetical reasoning. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 94–111, St. Julian's, Malta, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.eacl-long.6. URL https://aclanthology.org/2024.eacl-long.6/.

Table 5: Synthetic-data statistics after semantics-preserving rewriting and execution-grounded filtering. **Retention** is the fraction of rewrites for which at least three of five independently sampled SQL candidates are execution-equivalent to the gold query.

Dataset	Rewritten (K)	Retained (K)	Retention (%)
LogicCat (Liu et al., 2026b)	35.6	15.4	43.3
Archer (Zheng et al., 2024)	5.1	3.0	58.8
BIRD (Li et al., 2023)	95.3	35.2	36.9
Spider (Yu et al., 2018)	92.2	45.1	48.9
Spider 2.0-SQLite (Lei et al., 2025)	1.3	0.4	30.8
CoSQL (Yu et al., 2019)	3.0	1.1	36.7
Total	**232.5**	**100.2**	**43.1**

A Implementation Details

A.1 Synthetic Data Construction

We construct a large synthetic corpus that increases linguistic coverage while retaining an executable semantic anchor. The procedure deliberately separates **diversity generation** from **semantic validation**: rewriting creates varied questions, whereas independently generated SQL candidates test whether each rewrite still denotes the original task.

Seed task normalization. We collect seed instances from the training splits of LogicCat, Archer, BIRD, Spider, Spider 2.0-SQLite, and CoSQL:

$$\mathcal{D}_{\text{seed}} = \{x_i\}_{i=1}^{N}, \qquad x_i = (q_i, e_i, D_i, y_i^*), \tag{12}$$

where q_i is the question, e_i is optional evidence, D_i is the database, and y_i^* is the gold SQL. Retaining this tuple preserves the schema, evidence, and executable supervision associated with every question.

Semantics-preserving diversification. For each q_i, zero-shot rewriting produces ten lexical and syntactic variants,

$$\mathcal{Q}_i' = \{q_{i,j}'\}_{j=1}^{10}. \tag{13}$$

The rewriter is instructed to preserve the intent expressed by (e_i, y_i^*), but its output is not accepted on instruction following alone. This distinction is important: fluent paraphrases may silently alter aggregation, comparison, or temporal constraints while remaining superficially similar.

Execution-grounded consensus filtering. For every rewrite $q_{i,j}'$, Qwen3.6-27B independently samples five SQL candidates $\mathcal{Y}_{i,j} = \{y_{i,j}^{(k)}\}_{k=1}^{5}$. We compute

$$z_{i,j}^{(k)} = \mathbb{I}\Big[\mathrm{EX}(y_{i,j}^{(k)}, D_i) = \mathrm{EX}(y_i^*, D_i)\Big], c_{i,j} = \frac{1}{5}\sum_{k=1}^{5} z_{i,j}^{(k)}, v_{i,j} = \mathbb{I}[c_{i,j} \geq 3/5]. \tag{14}$$

Thus, a rewrite is retained only if a majority of independent generations recover the gold denotation. Unlike a single teacher judgment, this **three-out-of-five consensus** suppresses both ambiguous rewrites and unstable pseudo-labels. Execution equivalence alone can admit coincidental matches, so we additionally deduplicate repeated rewrites before training.

Corpus assembly. Accepted questions inherit the original evidence, database, and gold query:

$$\mathcal{D}_{\text{syn}} = \{(q_{i,j}', e_i, D_i, y_i^*) \mid x_i \in \mathcal{D}_{\text{seed}},\ v_{i,j} = 1\}. \tag{15}$$

The gold SQL is used only as an *offline construction anchor*; the subsequent search stage must still expose and verify intermediate reasoning. After filtering and deduplication, the corpus contains approximately 100K question–SQL instances (Table 5) and serves as the input to the MCTS cold-start stage.

A.2 MCTS COLD START

After constructing the synthetic dataset $\mathcal{D}_{\text{syn}}$, we use Qwen3.6-27B as the bootstrap policy to perform schema-grounded MCTS over all retained synthetic instances. This stage converts question–SQL supervision into structured reasoning trajectories that explicitly expose intermediate schema-grounding, value-linking, function-identification, SQL-generation, and SQL-revision decisions. For each synthetic instance $x \in \mathcal{D}_{\text{syn}}$, the bootstrap search starts from the schema-grounded root state and explores trajectories using the six structured actions introduced in the main paper. Since SQL-PPM is not available before self-evolution begins, the cold-start search is conducted without preference-model guidance:

$$\mathcal{T}^{(0)} = \text{MCTS}\left(\mathcal{D}_{\text{syn}}, \pi_{27\text{B}}, \varnothing\right), \tag{16}$$

where $\pi_{27\text{B}}$ denotes Qwen3.6-27B and $\mathcal{T}^{(0)}$ is the set of bootstrap MCTS trajectories. Terminal SQL candidates are executed against their corresponding databases and evaluated using execution consistency together with the SQL-specific semantic verification signals described in the main paper. Trajectories that produce invalid, execution-inconsistent, or semantically unreliable SQL queries are removed. The remaining high-quality trajectories form the cold-start supervised dataset:

$$\mathcal{D}_{\text{cold}} = \text{FilterSFT}\left(\mathcal{T}^{(0)}\right). \tag{17}$$

We then fine-tune the smaller base policy π_{base} (Qwen2.5-Coder-7B or Qwen3-8B), rather than the 27B bootstrap policy, on $\mathcal{D}_{\text{cold}}$ to obtain the initial policy for the first self-evolution round:

$$\pi_{\phi}^{(1)} = \text{SFT}\left(\pi_{\text{base}}, \mathcal{D}_{\text{cold}}\right). \tag{18}$$

Therefore, Qwen3.6-27B is used for both consensus-verified synthetic data construction and Round-0 bootstrap MCTS. Starting from Round 1, trajectory generation, policy refinement, and subsequent SQL-PPM-guided search are performed using the specialized policy model.

A.3 EFFECT OF BOOTSTRAP MODELS AND TRAJECTORY QUALITY

To examine the sensitivity of RStar-SQL to the bootstrap model, we compare multiple strong code and Text-to-SQL models for Round 0 trajectory generation on LogicCat. Figure 7 reports direct bootstrap MCTS performance, the performance of a fixed Qwen2.5-Coder-7B policy trained on 500 verified trajectories from each source, and the corresponding results with SQL-PPM guidance. Although Round 0 MCTS performance varies substantially across bootstrap models, these differences become much smaller after training the same 7B policy with an equal number of verified trajectories, suggesting that downstream utility depends on the transferable reasoning patterns contained in the trajectories rather than solely on the terminal accuracy of the generating model. SQL-PPM further provides consistent gains of approximately 2.8–3.9 EX points across trajectory sources and reduces the variation between bootstrap choices. Overall, these results indicate that bootstrap quality primarily affects the initial trajectory distribution, while verified policy training and process preference guidance progressively reduce this dependence and yield more robust search behavior.

A.4 INFERENCE EFFICIENCY ACROSS SELF-EVOLUTION ROUNDS

Table 7 reports the per-query search cost of RStar-SQL across self-evolution rounds under different rollout budgets. Under a fixed budget, execution accuracy improves consistently across rounds for both backbones, while the number of LLM calls, token consumption, and latency remain broadly stable, indicating that the gains from self-evolution do not rely on increasing inference-time computation. Moreover, reducing the rollout budget from 16 to 8 substantially lowers search cost while preserving comparable final accuracy. For example, at Round 4, Qwen2.5-Coder achieves 35.41% EX with 8 rollouts versus 35.48% with 16 rollouts, while reducing LLM calls from 32.9 to 17.8 and latency from 7.7 to 4.2 minutes. This suggests that the learned policy and SQL-PPM improve the quality of search, enabling RStar-SQL to maintain strong performance with substantially less exploration.

A.5 ANALYSIS BY MCTS ACTION

Table 6 shows that all six action types contribute consistently across both policy backbones, supporting the effectiveness of the structured action space. Removing SQL generation (A5) causes the

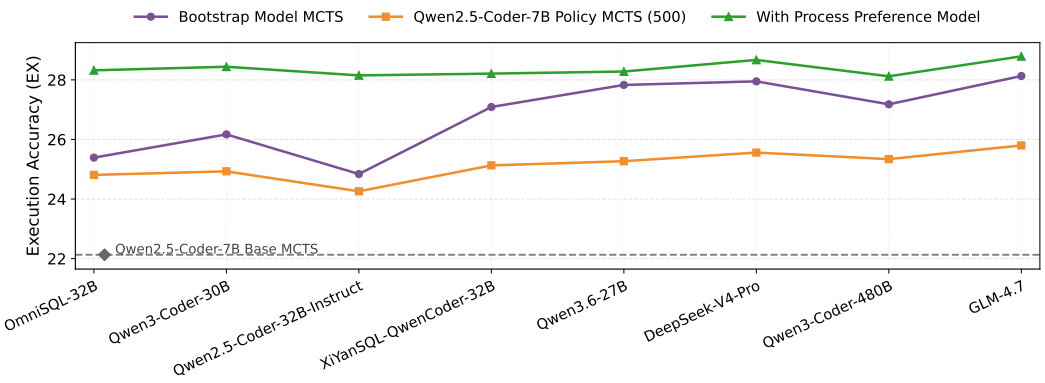


Figure 7: Effect of bootstrap model choice on Round 0 search, downstream policy training, and SQL-PPM-guided search on LogicCat. The three curves report direct bootstrap MCTS performance, the Qwen2.5-Coder-7B policy after training on 500 verified trajectories, and the corresponding performance with SQL-PPM guidance.

Table 6: Ablation of the six action types (A1–A6) in Round 0. Results are execution accuracy; values in parentheses denote absolute performance changes from the complete action space.

Action Space	Qwen2.5-Coder	Δ (%)	Qwen3-8B	Δ (%)
A1–A6	**22.16**	–	**24.28**	–
w/o A1 (formulating reasoning)	21.45	-0.71	24.02	-0.26
w/o A2 (select schema)	20.83	-1.33	23.88	-0.40
w/o A3 (link values)	20.32	-1.84	23.25	-1.03
w/o A4 (identify functions)	20.86	-1.30	23.87	-0.41
w/o A5 (generate SQL)	19.38	**-2.78**	21.47	**-2.81**
w/o A6 (revise SQL)	20.12	-2.04	22.12	-2.16

largest performance drop, followed by SQL revision (A6), indicating that constructing and iteratively correcting candidate queries are the most critical stages of the search process. Value linking (A3) also contributes substantially, highlighting the importance of accurately grounding question mentions to database values. In contrast, reasoning formulation (A1) yields smaller but consistent gains, suggesting that it primarily facilitates subsequent structured decisions. Overall, the similar trends across Qwen2.5-Coder and Qwen3-8B suggest that the proposed action decomposition is robust across different backbone models.

A.6 Analysis of Error Cases

We further examine representative failure cases to understand the remaining limitations of RStar-SQL and the types of intermediate decisions that remain difficult for the policy model.

- **SQL dialect error.** Figure 8 presents a low-level SQL realization failure. The model correctly identifies that the warranty status should be derived from the production date and warranty period, but applies SQLite-style `DATE` syntax and string concatenation in a MySQL environment. Consequently, an otherwise reasonable computation becomes non-executable. This case indicates that correct high-level reasoning does not necessarily guarantee dialect-compatible SQL realization.
- **Join-path error.** Figure 9 illustrates a structurally problematic join decision. Although the two common table expressions independently compute the required statistics, the model combines them through an implicit cross join without an explicit join condition. The query can still be executed, but the join verifier identifies it as structurally unreliable. This highlights the importance of verifying join relationships even when execution alone does not expose an immediate error.
- **Hypothetical reasoning error.** Figure 10 presents a failure to distinguish hypothetical assumptions from database constraints. The model recognizes the variables required for the

Table 7: Per-query search efficiency of RStar-SQL across self-evolution rounds on LogicCat.

Round	LLM Calls	Avg. Tokens	Latency / Query	EX	LLM Calls	Avg. Tokens	Latency / Query	EX
	Qwen2.5-Coder-7B-Instruct (Rollout=16)				*Qwen3-8B (Rollout=16)*			
Round 1	30.2	28.9k	8.3 min	28.45	28.4	27.2k	7.4 min	30.72
Round 2	33.4	30.2k	8.1 min	31.37	30.1	29.8k	7.9 min	31.76
Round 3	32.8	29.7k	7.9 min	34.10	29.7	28.9k	7.5 min	32.89
Round 4	32.9	30.1k	7.7 min	**35.48**	29.4	29.8k	7.5 min	**33.46**
	Qwen2.5-Coder-7B-Instruct (Rollout=8)				*Qwen3-8B (Rollout=8)*			
Round 1	16.6	17.3k	3.9 min	28.32	15.8	16.5k	3.8 min	30.46
Round 2	17.2	18.5k	4.2 min	31.22	16.2	17.5k	4.1 min	31.17
Round 3	14.6	15.9k	3.6 min	33.98	17.3	15.9k	4.1 min	32.75
Round 4	17.8	18.8k	4.2 min	**35.41**	16.0	17.2k	3.9 min	**33.26**

energy calculation, but treats the assumed cooling capacity and efficiency ratio as `WHERE` conditions rather than numerical constants in the expression. Because no database record satisfies these artificial conditions, the query returns an empty result. This case demonstrates that complex-reasoning Text-to-SQL requires not only identifying relevant values, but also determining their semantic roles in the query.

- **Aggregation error.** Figure 11 illustrates an incorrect aggregation decision. The model correctly selects `Water_Resistance` as the grouping key, but directly uses the non-aggregated `Weight_Grams` column in the projected expression. This violates the `ONLY_FULL_GROUP_BY` constraint and causes execution failure. The example shows that selecting the correct schema elements is insufficient unless their aggregation dependencies are also modeled correctly.

- **Schema-linking error.** Figure 12 shows a fine-grained column–table alignment failure. The model identifies the relevant tables and constructs a plausible join, but incorrectly associates `EventType` and `SeverityLevel` with the `WarningTriggerRecord` alias rather than `WarningRule`. This produces an unknown-column error despite otherwise reasonable query structure, demonstrating that precise ownership of schema fields remains essential in multi-table generation.

Overall, these cases reveal that the remaining errors arise at multiple levels of the SQL reasoning process, including dialect realization, join-path construction, value grounding, interpretation of hypothetical conditions, aggregation design, and column–table alignment. Several of the incorrect queries are executable or locally plausible, meaning that execution consistency alone cannot reliably distinguish them from correct solutions. These observations support the use of SQL-specific semantic verification in RStar-SQL, which combines execution behavior with join validity, schema consistency, and type–operator compatibility. They also motivate process preference learning, since identifying and down-weighting unreliable intermediate actions can prevent locally plausible errors from propagating into the final SQL query.

B Experimental Details

B.1 Training and Inference Settings

As summarized in Table 8, RStar-SQL uses a maximum MCTS depth of 6 and a UCT exploration coefficient of 1.4. SQL-generation and SQL-revision actions adopt a higher sampling temperature than intermediate reasoning actions to encourage candidate diversity. The policy model is fine-tuned on verified trajectories, whereas SQL-PPM is optimized separately using trajectory preference pairs. During preference-guided search, normalized SQL-PPM scores are incorporated into UCT with $\lambda = 0.5$. At inference time, executable SQL candidates are grouped by equivalent execution results, and the query with the lowest measured execution latency in the largest valid cluster is selected as the final prediction.

SQL Dialect Error (QID: 1375)

Outcome: Execution=false, status = syntax error

Execution results:

Error: MySQL 1064 near '+' || rc.warranty_period

Pred results: null...

Observed behavior:

✓ Attempts to derive warranty status from date and period.

✗ Uses SQLite DATE syntax and || concatenation in MySQL.

Final SQL

```
CASE WHEN DATE( rc.production_date, '+' || rc.warranty_period || ' month') >= '2023-12-01'
THEN 'Active' ELSE 'Expired'
END
```

Gold SQL

```
CASE WHEN DATE_ADD( rc.production_date, INTERVAL rc.warranty_period MONTH)
>= '2023-12-01'
THEN 'Active' ELSE 'Expired'
END
```

Figure 8: SQL dialect error (QID: 1375). SQLite-style date syntax is used in MySQL, causing execution failure.

Join Path Error (QID: 1625)

Outcome: Execution=true, status = missing join condition

Execution results:

SQL execution: success

Join verifier: missing_join_condition

Observed behavior:

✓ The two CTEs separately compute the required statistics.

✗ Combines CTE outputs using an implicit cross join without ON.

Final SQL

```
WITH avg_cost AS (...), predicted_cost AS (...)
SELECT avg_maintenance_cost,predicted_total_cost
FROM avg_cost, predicted_cost
```

Gold SQL

```
WITH avg_cost AS (...),predicted_cost AS (...)
SELECT *
FROM avg_cost JOIN predicted_cost ON ...
ORDER BY T1.NumTstTakr DESC LIMIT 1
```

Figure 9: Join-path error (QID: 1625). Two CTEs are combined through an implicit cross join without an explicit join condition.

B.2 Open-Source Specialized Text-to-SQL Methods.

We additionally compare against a set of specialized Text-to-SQL systems built upon the same or comparable open-source backbones. These methods represent different directions including decomposition, multi-agent reasoning, execution-guided search, and heterogeneous reasoning optimization.

- **DIN-SQL:** DIN-SQL (Pourreza & Rafiei, 2023) decomposes Text-to-SQL into a sequence of specialized stages, including schema linking, query decomposition, SQL generation, and

Hypothetical Reasoning Error (QID: 2377)

Outcome: Execution=true, status = empty result

Execution results:

Gold results: [extreme_energy_consumption_kwh]...
Pred results: null...

Observed behavior:

✓ Uses cooling capacity and EER in an energy expression.

✗ Treats assumed values as WHERE filters rather than constants.

Final SQL

```
SELECT (aci.cooling_capacity_btu /aci.energy_efficiency_ratio * ...)
FROM air_conditioner_info AS aci
WHERE aci.cooling_capacity_btu = 50000 AND aci.energy_efficiency_ratio = 15
```

Gold SQL

```
SELECT  (50000 * 200 / (15 * 0.3)) * 1055.06 / 1000
AS extreme_energy_consumption_kwh
```

Figure 10: Hypothetical reasoning error (QID: 2377). Assumed values are used as database filters rather than constants in the calculation.

Aggregation Value Error(QID: 2758)

Outcome: Execution=false, status = only_full_group_by

Execution results:

GError: MySQL 1055, expression is not in GROUP BY
Pred results: null...

Observed behavior:

✓ Correctly identifies Water_Resistance as the grouping key.

✗ Uses non-aggregated Weight_Grams in the SELECT expression.

Final SQL

```
SELECT Water_Resistance,
    1000000.0 / Weight_Grams AS unit_price_per_gram
FROM headphones
GROUP BY Water_Resistance
```

Gold SQL

```
SELECT Water_Resistance,
    1000000 / NULLIF(AVG(Weight_Grams), 0) AS Price_Per_Gram
FROM headphones
GROUP BY Water_Resistance
```

Figure 11: Aggregation error (QID: 2758). A non-aggregated column violates the `ONLY_FULL_GROUP_BY` constraint.

self-correction. By explicitly separating complex reasoning steps, it reduces the difficulty of directly generating SQL for challenging cross-domain queries.

- **CHESS:** CHESS (Talaei et al., 2024) adopts a modular agentic pipeline that combines information retrieval, schema selection, candidate generation, and SQL validation. It progressively reduces irrelevant database context and verifies generated queries, enabling more effective reasoning over large and complex database schemas.
- **CHASE-SQL:** CHASE-SQL (Pourreza et al., 2025a) improves Text-to-SQL through multi-path candidate generation and iterative refinement. It explores complementary SQL

Schema Linking Error (QID: 10151)

Outcome: Execution=false, status = unknown column

Execution results:

Error: Unknown column 'wtr.EventType' in 'where clause'
Pred results: null...

Observed behavior:

✓ Correctly joins WarningRule with WarningTriggerRecord.

✗ Links EventType and SeverityLevel to alias wtr instead of wr.

Final SQL

```
SELECT wr.ActionType FROM WarningRule AS wr
INNER JOIN WarningTriggerRecord AS wtr ON …
WHERE wtr.EventType = 'flood'
AND wtr.SeverityLevel = 8
```

Gold SQL

```
SELECT ActionType FROM WarningRule
WHERE EventType = 'flood'
AND SeverityLevel = 8
```

Figure 12: Schema-linking error (QID: 10151). Filtering columns are assigned to the wrong table alias, causing an unknown-column error.

reasoning trajectories and integrates candidate-level verification and selection to improve the robustness of the final prediction.

- **DeepEye-SQL:** DeepEye-SQL (Li et al., 2026a) is a modular Text-to-SQL system that decomposes the overall problem into specialized stages for schema understanding, SQL generation, execution-based refinement, and candidate selection. The modular design enables different components to collaboratively resolve schema, semantic, and execution errors.
- **Alpha-SQL:** Alpha-SQL (Li et al., 2025) formulates SQL generation as a search problem and applies Monte Carlo Tree Search (MCTS) to iteratively construct and refine candidate queries. Dynamic LLM-generated actions and self-supervised feedback guide the search toward promising SQL trajectories without requiring a separately trained external critic.
- **SQL-O1:** SQL-O1 (Lyu et al., 2025) introduces a self-reward-guided dynamic search framework for Text-to-SQL. It generates and iteratively revises candidate SQL queries using execution-aware feedback, allowing the model to explore alternative reasoning paths and select higher-quality solutions at inference time.
- **IESR:** IESR (Liu et al., 2026a) extends search-based Text-to-SQL beyond SQL-only refinement by introducing heterogeneous reasoning actions, including schema grounding, entity and constraint reasoning, symbolic computation, SQL generation, and revision. These actions are jointly explored within a structured search process, enabling the model to handle reasoning-intensive queries that require knowledge or computation beyond direct SQL synthesis.

B.3 Large-Scale Foundation Models.

We additionally compare against strong large-scale general-purpose language models, including o3-mini, GPT-5.4, Claude-Opus-4.6, DeepSeek-V4-Pro, Gemini-3.1, GLM-4.7, and Kimi-2.5. These models represent substantially larger general-purpose reasoning systems with strong capabilities in code generation, multi-step reasoning, and instruction following. Unlike the specialized Text-to-SQL baselines above, they are not explicitly designed around task-specific SQL search or execution-grounded optimization. We include them to provide a reference for how specialized Text-to-SQL systems compare with large-scale foundation models under the same evaluation setting.

Module	Hyperparameter	Setting
MCTS Search	Maximum search depth	6
	Number of rollouts	8 for main experiments; 16+ for efficiency analysis
	UCT exploration coefficient c	1.4
	Random seed	42
	Concurrent workers	64
Reasoning Actions	Sampling temperature	0.2
	Top-p	1.0
	Candidates per action	1
	Maximum output length	1,024 tokens
SQL Generation and Revision	Sampling temperature	0.8
	Candidates per action	5
	Maximum generation attempts	15
	SQL execution timeout	60 seconds
	Candidate aggregation	Largest valid execution-result cluster
Semantic Verification	Execution consistency weight w_{sc}	0.40
	Schema grounding weight w_{sch}	0.25
	Execution behavior weight w_{exe}	0.25
	Join validity weight w_{join}	0.05
	Type–operator weight w_{op}	0.05
Policy SFT	Training epochs	3
	Learning rate	2×10^{-5}
	Optimizer	AdamW
	Maximum sequence length	4,096
	Precision	BF16
	Per-device batch size	1
	Gradient accumulation steps	2
	Effective global batch size	16 on 8 GPUs
SQL-PPM Training	Training objective	Bradley–Terry pairwise ranking loss
	Training epochs	3
	Learning rate	2×10^{-5}
	Maximum sequence length	2,048
	Per-device batch size	8
	Value-head dropout	0.10
	Maximum preference pairs per question	2
Preference-Guided Search	SQL-PPM guidance coefficient λ	0.5
	Score normalization	Sibling-wise min–max normalization
	Simulation policy	Sampling proportional to normalized SQL-PPM scores
Inference Environment	Inference engine	vLLM
	Number of model instances	8, with one instance per GPU
	Maximum context length	8,192 tokens
	GPU memory utilization	0.85–0.90

Table 8: Implementation and training hyperparameters of RStar-SQL. The main experiments use eight MCTS rollouts per question, while the efficiency analysis additionally evaluates a fixed budget of 16 rollouts.

B.4 PROMPT

Figures 13–20 present the prompts used in data construction, trajectory verification, and schema-grounded MCTS. The cold-start question-rephrasing prompt generates semantically equivalent question–SQL pairs, while the SQL verifier checks whether candidate answers are consistent with the observed execution results. During trajectory search, each structured action is implemented with a specialized prompt: A1 extracts the key entities, constraints, and reasoning concepts; A2 selects the necessary tables and columns; A3 grounds question mentions to database-specific values; A4 identifies the required SQL functions and operators; A5 constructs the complete query through structured decomposition; and A6 diagnoses and repairs unsuccessful SQL using execution feedback. This modular prompt design separates heterogeneous reasoning decisions rather than requiring a single generation prompt to simultaneously perform schema grounding, value normalization, operator selection, and SQL construction. It therefore makes intermediate decisions explicit and revisable, facilitates SQL-specific verification at different stages, and provides well-defined state–action transitions for MCTS and process-preference learning.

Question Rephasing

```
You are a MySQL query-rewriting expert.
Your task is to take the provided question + SQL pair and produce **10 rewritten variants**.

Requirements:
1. Each rewritten question must keep the exact meaning of the SQL (i.e., still answerable by the same query).
2. If the original question conflicts with the SQL, treat the SQL as ground truth and adjust the question accordingly.
3. When a rewrite is impossible, reuse the original SQL as-is (do NOT invent new SQL).
4. All output must be in English and contain **no extra commentary**.

Output format:
- Return a JSON array of 10 objects. Each object must have:
  - "question": the rewritten natural-language question.
  - "query": the SQL statement (identical to the provided SQL unless a faithful rewrite requires formatting adjustments only).

Example structure:
[
  {"question": "...", "query": "..."},
  ... (total 10 entries)
]
```

Figure 13: Question-rephrasing prompt used for cold-start data augmentation. Given an original question–SQL pair, the bootstrap model generates multiple semantically equivalent question variants while preserving the SQL query and treating the SQL semantics as the reference when inconsistencies occur.

SQL Verifying

```
You are a precise SQL result verifier.
Input:
- Natural language question
- Proposed SQL query
- Execution result(s)
- Candidate natural-language answers

Task:
1. Check whether each candidate answer is consistent with the SQL execution result.
2. Mark answers as correct/incorrect.
3. Provide a short justification if inconsistent.
4. Output JSON: {"question":..., "correct_answers":[], "incorrect_answers":[], "notes":...}
```

Figure 14: SQL result-verification prompt. Given a natural-language question, a candidate SQL query, its execution result, and candidate answers, the verifier determines which answers are consistent with the execution outcome and returns structured correctness judgments with brief explanations.

Formulate Prompt

````
Objective: Analyze the given question and hint to identify and extract keywords, keyphrases, and named entities. These elements are crucial for understanding the core components of the inquiry and the guidance provided. This process involves recognizing and isolating significant terms and phrases that could be instrumental in formulating searches or queries related to the posed question.

Instructions:

Read the Question Carefully: Understand the primary focus and specific details of the question. Look for any named entities (such as organizations, locations, etc.), technical terms, and other phrases that encapsulate important aspects of the inquiry.

Analyze the Hint: The hint is designed to direct attention toward certain elements relevant to answering the question. Extract any keywords, phrases, or named entities that could provide further clarity or direction in formulating an answer.

List Keyphrases and Entities: Combine your findings from both the question and the hint into a single Python list. This list should contain:

Keywords: Single words that capture essential aspects of the question or hint.
Keyphrases: Short phrases or named entities that represent specific concepts, locations, organizations, or other significant details.
Ensure to maintain the original phrasing or terminology used in the question and hint.

Example 1:
Question: "What is the annual revenue of Acme Corp in the United States for 2022?"
Hint: "Focus on financial reports and U.S. market performance for the fiscal year 2022."

```python
["annual revenue", "Acme Corp", "United States", "2022", "financial reports", "U.S. market performance", "fiscal year"]
```

Example 2:
Question: "In the Winter and Summer Olympics of 1988, which game has the most number of competitors? Find the difference of the number of competitors between the two games."
Hint: "the most number of competitors refer to MAX(COUNT(person_id)); SUBTRACT(COUNT(person_id where games_name = '1988 Summer'), COUNT(person_id where games_name = '1988 Winter'));"

```python
["Winter Olympics", "Summer Olympics", "1988", "1988 Summer", "Summer", "1988 Winter", "Winter", "number of competitors", "difference", "MAX(COUNT(person_id))", "games_name", "person_id"]
```

Example 3:
Question: "How many Men's 200 Metres Freestyle events did Ian James Thorpe compete in?"
Hint: "Men's 200 Metres Freestyle events refer to event_name = 'Swimming Men"s 200 metres Freestyle'; events compete in refers to event_id;"

```python
["Swimming Men's 200 metres Freestyle", "Ian James Thorpe", "Ian", "James", "Thorpe", "compete in", "event_name", "event_id"]
```

Task:
Given the following question and hint, identify and list all relevant keywords, keyphrases, and named entities.

Question: {QUESTION}
Hint: {HINT}

Please provide your findings as a Python list within ```python ``` tags, capturing the essence of both the question and hint through the identified terms and phrases. Only output the Python list, no explanations needed.
````

Figure 15: Reasoning-formulation prompt corresponding to action A1. The model analyzes the question and optional hint to extract task-relevant keywords, keyphrases, named entities, constraints, and SQL-related concepts, which provide an explicit semantic representation for subsequent schema grounding and query construction.

Identify Functions Prompt

```
You are an AI assistant to help me identify the potential column functions (if needed to be used in the SQL query) that are essential for answering the question.

**************************
Example:

Database Schema:
CREATE TABLE businesses
(
    `business_id` INTEGER NOT NULL,
    `name` TEXT NOT NULL, -- Column Description: the name of the eatery
    PRIMARY KEY (`business_id`),
);

CREATE TABLE inspections
(
    `business_id` INTEGER NOT NULL, -- Column Description: the unique id of the business
    `score` INTEGER DEFAULT NULL, -- Column Description: the inspection score
    `date` TEXT NOT NULL, -- Value Examples: `2014-01-24`
    FOREIGN KEY (`business_id`) REFERENCES `businesses` (`business_id`),
);

Question:
What are the names of the businesses that passed with conditions in May 2012?

Hint:
name of business refers to dba_name; passed with conditions refers to results = 'Pass w/ Conditions'; in May 2012 refers to inspection_date like '2012-05%'

Answer:
Since the businesses passed with conditions in May 2012, I should consider a date-related function to filter the `inspections`.`date` column. I find that column is of type TEXT, so I can use the strftime('%Y-%m', `inspections`.`date`) = '2012-05' to filter the date(SQLite) or DATE_FORMAT(`inspections`.`date`, '%Y-%m') = '2012-05'(MYSQL).

**************************

Now, answer the real question, and you need to follow the answer style of the above examples (answer in two sentences).

Database Schema:
{SCHEMA_CONTEXT}

Question:
{QUESTION}

Hint:
{HINT}

Answer:
```

Figure 16: Column–function identification prompt corresponding to action A4. Based on the grounded schema, question, and hint, the model identifies the SQL functions and operators required for relevant columns, such as date transformations, aggregation functions, arithmetic expressions, and filtering operations.

Link Values Prompt

```
You are an AI assistant to help me identify the potential column values (if needed to be used in the SQL query) that are essential for answering the question.

Here is an example:

Database Schema:
CREATE TABLE generalinfo
(
        id_restaurant INTEGER not null primary key,
        food_type TEXT null, -- Value Examples: `thai`| Column Description: the food type
        city TEXT null, -- Column Description: the city where the restaurant is located in
);

CREATE TABLE location
(
        id_restaurant INTEGER not null primary key,
        street_name TEXT null, -- Value Examples: `ave`, `san pablo ave`, `pablo ave`| Column Description: the street name of the restaurant
        city TEXT null, -- Column Description: the city where the restaurant is located in
        foreign key (id_restaurant) references generalinfo (id_restaurant) on update cascade on delete cascade,
);

Question:
How many Thai restaurants can be found in San Pablo Ave, Albany?

Hint:
Thai restaurant refers to food_type = 'thai'; San Pablo Ave Albany refers to street_name = 'san pablo ave' AND T1.city = 'albany'

Answer:
Since the restaurants are located in Albany, based on the schema information and the hint, I need to use `location`.`street_name` = 'san pablo ave' AND
`generalinfo`.`city` = 'albany'.

**************************
Now, answer the real question, and you need to follow the answer style of the above examples (answer in two sentences).

Database Schema:
{SCHEMA_CONTEXT}

Question:
{QUESTION}

Hint:
{HINT}

Answer:
```

Figure 17: Column–value linking prompt corresponding to action A3. The model aligns value mentions and semantic conditions in the question with database-specific columns and normalized literals, producing explicit column–value constraints for subsequent SQL generation.

Schema Selection Prompt

```
You are an expert and very smart data analyst.
Your task is to examine the provided database schema, understand the posed question, and use the hint to pinpoint the specific columns within tables that are essential for crafting a SQL query to answer the question.

The schema offers an in-depth description of the database's architecture, detailing tables, columns, primary keys, foreign keys, and any pertinent information regarding relationships or constraints. Special attention should be given to the examples listed beside each column, as they directly hint at which columns are relevant to our query.

For key phrases mentioned in the question, we have provided the most similar values within the columns denoted by "-- Value Examples" in front of the corresponding column names. This is a critical hint to identify the columns that will be used in the SQL query.

The hint aims to direct your focus towards the specific elements of the database schema that are crucial for answering the question effectively.

Task:
Based on the database schema, question, and hint provided, your task is to identify all and only the columns that are essential for crafting a SQL query to answer the question.
For each of the selected columns, explain why exactly it is necessary for answering the question. Your reasoning should be concise and clear, demonstrating a logical connection between the columns and the question asked.

Please respond with your reasoning process and a JSON object structured as follows:
# Reasoning Process:
(Here, output your reasoning for selecting the columns, be concise and clear)
# Final JSON Result:
```json
{{
  "table_name1": ["column1", "column2", ...],
  "table_name2": ["column1", "column2", ...]
  ...
}}
```

Make sure your response includes the table names as keys, each associated with a list of column names that are necessary for writing a SQL query to answer the question.
For each aspect of the question, provide a clear and concise explanation of your reasoning behind selecting the columns.
Take a deep breath and think logically. If you do the task correctly, I will give you 1 million dollars.

Database Schema Overview:
{SCHEMA_CONTEXT}

Question:
{QUESTION}

Hint:
{HINT}

Please output your reasoning process first after "# Reasoning Process", and output a JSON object within ```json``` tags.
```

Figure 18: Schema-selection prompt corresponding to action A2. Given the database schema, question, retrieved value examples, and optional hint, the model selects only the tables and columns necessary for constructing the target SQL query and explains the relevance of each selected schema element.

SQL Generation Prompt

```
You are an experienced database expert.
Now you need to generate a SQL query given the database information, a question and some additional information.
The database structure is defined by the following table schemas (comments after '--' provide additional column descriptions).
Note that the "Value Examples" are actual values from the column. Some column might contain the values that are directly related to the question. Use it to help you justify which columns to use.

Given the table schema information description and the `Question`. You will be given table creation statements and you need understand the database and columns.

You will be using a way called "recursive divide-and-conquer approach to SQL query generation from natural language".

Here is a high level description of the steps.
1. **Divide (Decompose Sub-question with Pseudo SQL):** The complex natural language question is recursively broken down into simpler sub-questions. Each sub-question targets a specific piece of information or logic required for the final SQL query.
2. **Conquer (Real SQL for sub-questions):** For each sub-question (and the main question initially), a "pseudo-SQL" fragment is formulated. This pseudo-SQL represents the intended SQL logic but might have placeholders for answers to the decomposed sub-questions.
3. **Combine (Reassemble):** Once all sub-questions are resolved and their corresponding SQL fragments are generated, the process reverses. The SQL fragments are recursively combined by replacing the placeholders in the pseudo-SQL with the actual generated SQL from the lower levels.
4. **Final Output:** This bottom-up assembly culminates in the complete and correct SQL query that answers the original complex question.

Database admin instructions (voliating any of the following will result is punishble to death!):
1. **SELECT Clause:**
   - Only select columns mentioned in the user's question.
   - Avoid unnecessary columns or values.
2. **Aggregation (MAX/MIN):**
   - Always perform JOINs before using MAX() or MIN().
3. **ORDER BY with Distinct Values:**
   - Use `GROUP BY <column>` before `ORDER BY <column> ASC|DESC` to ensure distinct values.
4. **Handling NULLs:**
   - If a column may contain NULL values, use `JOIN` or `WHERE <column> IS NOT NULL`.
5. **FROM/JOIN Clauses:**
   - Only include tables essential to answer the question.
6. **Strictly Follow Hints:**
   - Adhere to all provided hints.
7. **Thorough Question Analysis:**
   - Address all conditions mentioned in the question.
8. **DISTINCT Keyword:**
   - Use `SELECT DISTINCT` when the question requires unique values (e.g., IDs, URLs).
9. **Column Selection:**
   - Carefully analyze column descriptions and hints to choose the correct column when similar columns exist across tables.
10. **String Concatenation:**
   - Never use `|| ' ' ||` or any other method to concatenate strings in the `SELECT` clause.
11. **SQLite Or MySQL Functions Only(depending on benchmark):**
   - Use only functions available in SQLite or MYSQL.
12. **Schema Syntax:**
   - Use `table_name`.`column_name` to refer to columns from different tables, when table name or column name contains whitespace.
13. **JOIN Preference:**
   - Prioritize `INNER JOIN` over nested `SELECT` statements. Do not use `CROSS JOIN` or `LEFT/RIGHT JOIN`.

Repeating the question and hint, and generating the SQL with Recursive Divide-and-Conquer, and finally try to simplify the SQL query using `INNER JOIN` over nested `SELECT` statements IF POSSIBLE.

Please respond with xml tags structured as follows:
```xml
<think>
Your detailed reasoning for the SQL query generation, with Recursive Divide-and-Conquer approach.
</think>
<sql>
The final SQL query that answers the question.
</sql>
```
****
【Table creation statements】
{SCHEMA_CONTEXT}
****
【Question】
Question:
{QUESTION}
Hint:
{HINT}
****
Only output xml format (starting with ```xml and ending with ```) as your response.
```

Figure 19: SQL-generation prompt corresponding to action A5. The model applies a recursive divide-and-conquer strategy to decompose the question into simpler subproblems, construct intermediate SQL fragments, and combine them into a complete query while respecting schema, projection, aggregation, join, null-handling, and dialect constraints.

SQL Revision Prompt

```
**Task Description:**
You are an SQL database expert tasked with correcting a SQL query. A previous attempt to run a query did not yield the correct results, either due to errors in execution or because the result returned was empty or unexpected. Your role is to analyze the error based on the provided database schema and the details of the failed execution, and then provide a corrected version of the SQL query.

**Procedure:**
1. Review Database Schema:
        - Examine the table creation statements to understand the database structure.
2. Analyze Query Requirements:
        - Original Question: Consider what information the query is supposed to retrieve.
        - Hint: Use the provided hints to understand the relationships and conditions relevant to the query.
        - Executed SQL Query: Review the SQL query that was previously executed and led to an error or incorrect result.
        - Execution Result: Analyze the outcome of the executed query to identify why it failed (e.g., syntax errors, incorrect column references, logical mistakes), note that the execution result is possibly empty for some queries.
3. Correct the Query:
        - Modify the SQL query to address the identified issues, ensuring it correctly fetches the requested data according to the database schema and query requirements.

Based on the question, table schemas, the previous query, and the execution result, analyze the result following the procedure, and try to fix the query.
You cannot modify the database schema or the question, just output the corrected query.

Please respond with xml tags structured as follows:
```xml
<think>
Your detailed reasoning for the SQL query revision.
</think>
<sql>
The final SQL query that answers the question.
</sql>
```


****
【Table creation statements】
{SCHEMA_CONTEXT}

****
【Question】
Question:
{QUESTION}

Hint:
{HINT}

****

Only output xml format (starting with ```xml and ending with ```) as your response.
```

Figure 20: SQL-revision prompt corresponding to action A6. Given the database schema, original question, previous SQL query, and its execution feedback, the model diagnoses syntax, schema-reference, or semantic errors and produces a corrected SQL query without modifying the question or database schema.