EXPO-SQL: Execution-based Clause-level Policy Optimization for Text-to-SQL

Jaehoon Lee, CheolWon Na, Suyoung Bae,
Jin-Seop Lee, Jihyung Lee, YunSeok Choi*, Jee-Hyong Lee*
College of Computing and Informatics
Sungkyunkwan University, South Korea
{hoon1223, ncw0034, sybae01, wlstjq0602, jjklle, ys.choi, john}@skku.edu

Abstract

Text-to-SQL enables users to query databases using natural language by generating executable SQL queries. Recent methods have increasingly adopted Large Language Models based reinforcement learning (RL) to leverage execution feedback for training. However, existing RL methods assign uniform query-level rewards to all clauses in a SQL query, treating correct and incorrect clauses equally. This coarse-grained reward design leads to insufficient learning signals for correct SQL generation. To address this issue, we propose **EXPO-SQL** (**EX**ecution-based clause-level **P**olicy **O**ptimization for Text-to-**SQL**) which provides fine-grained supervision through clause-level rewards. To assign clause-level rewards, our method identifies erroneous clauses by analyzing execution results, including error messages and clause-wise incremental execution. Experiments on widely-used Text-to-SQL benchmarks demonstrate that EXPO-SQL significantly outperforms existing supervised fine-tuning, prompting, and RL-based methods through fine-grained clause-level learning. Our code is available at `https://github.com/jhn25/EXPO-SQL`.

1 Introduction

Text-to-SQL aims to generate executable SQL queries from given natural language question and database schema. It is a core technology that facilitates data retrieval by enabling non-expert users to query databases directly (Iacob et al., 2020; Affolter et al., 2019). Recent works on Text-to-SQL has increasingly adopted large language models (LLMs), leveraging their strong reasoning ability (Liu et al., 2024; Deng et al., 2022; Maamari et al., 2024; Hong et al., 2024).

Early approaches adopted supervised fine-tuning (SFT) to optimize token-level matching with gold SQL queries (Gao et al., 2024; Pourreza and Rafiei, 2024). Prompt-based methods leverage in-context learning with iterative refinement at inference time (Gao et al., 2024; Pourreza and Rafiei, 2024). However, both approaches still produce erroneous SQL queries due to the lack of execution-aware supervision for Text-to-SQL reasoning.

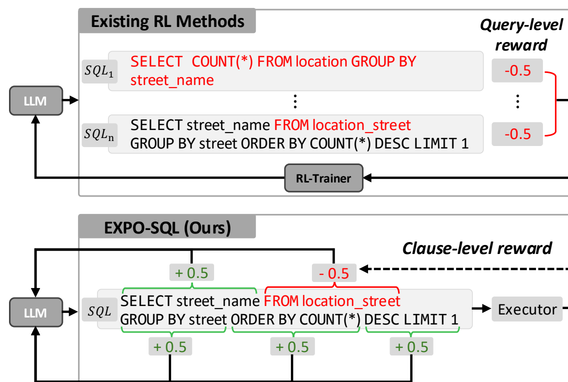

Figure 1: Comparison of reward assignment strategies. Existing RL methods assign query-level rewards uniformly across all clauses, while EXPO-SQL assigns clause-level rewards, only penalizing erroneous ones.

To address this issue, several studies have explored reinforcement learning (RL) approaches for Text-to-SQL generation (Zhong et al., 2017; Liang et al., 2018; Shi et al., 2025; Liu et al., 2025). These methods optimize model policies using execution results as reward signals (Ma et al., 2025; Yao et al., 2025; Zhai et al., 2025; Pourreza et al., 2025b; Weng et al., 2025; Zhang et al., 2025). However, these RL-based approaches have a limitation that they rely on query-level learning signals.

Usually, an SQL query is composed of several clauses, such as `SELECT`, `FROM`, `WHERE`, etc. In practice, execution failures are often caused by errors in a few clauses rather than the entire clauses in the SQL query. However, existing methods assign an identical reward to all clauses in a generated SQL query, as illustrated in Figure 1 where an identical reward is given to all clauses even when the `FROM` clause is only incorrect. This query-

*Corresponding authors

Findings of the Association for Computational Linguistics: ACL 2026, pages 22000–22019
July 2-7, 2026 ©2026 Association for Computational Linguistics

level rewards treat correct and incorrect clauses equally, leading to coarse credit assignment (Pignatelli et al., 2024) and penalizing even correctly generated clauses. As a result, the model receives only coarse learning signals, which are insufficient for correct SQL generation.

To address this coarse credit assignment problem, we propose **EXPO-SQL** (**EX**ecution-based clause-level **P**olicy **O**ptimization for Text-to-**SQL**), which separately evaluates the correctness of each clause within a SQL query and provides fine-grained supervision with clause-level reward. However, evaluating each clause remains highly challenging in online reinforcement learning. First, lexical analysis of SQL query cannot determine which clauses have error. Second, due to the one-to-many nature of SQL queries, whereby various queries can produce the same execution result, direct token matching with the gold SQL is not appropriate. Finally, execution results only provide binary feedback on whether the entire query is correct.

To overcome these limitations, we analyze execution results to identify erroneous clauses, rather than using them directly as binary rewards. We first classify them into three cases and design corresponding strategies: correct results, incorrect results, and execution errors. In the case of *correct results* where the query is executed successfully and produces the correct answer, we assign positive rewards to all clauses. For *incorrect results*, where the query is executable but produces an incorrect answer, we decompose the generated SQL query following the logical execution order and incrementally execute each clause. Then, we analyze how the result changes before and after adding each clause to identify erroneous clauses. Moreover, in the case of *execution errors* where the query fails to be executed, we analyze the error message to identify clauses that caused the failure.

For each case, we analyze how each clause affects the result and design clause-level rewards that provide fine-grained learning signals. Through the execution-based analysis, EXPO-SQL provides more accurate clause-level learning signals, which existing RL methods with query-level rewards cannot achieve.

We evaluate the superiority of EXPO-SQL on widely-used Text-to-SQL benchmarks including Spider (Yu et al., 2018) and BIRD (Li et al., 2023), comparing with a wide range of baselines including SFT, prompting, and recent RL-based methods. Experimental results demonstrate that EXPO-SQL significantly improves execution accuracy through effective clause-level learning signals, achieving state-of-the-art performance. Specifically, EXPO-SQL outperforms the best RL baseline by 1.2%p on Spider-Dev and 2.4%p on BIRD-Dev. The improvements are more pronounced on complex queries, showing 5.6%p gains.

2 Related Work

2.1 Conventional Text-to-SQL Methods

Early Text-to-SQL methods fine-tuned models using large-scale text-SQL pair datasets (Yang et al., 2024; Pourreza and Rafiei, 2024; Li et al., 2024, 2025b), with recent work incorporating chain-of-thought reasoning (Wei et al., 2022) through schema linking and query decomposition (Wang et al., 2025b; Qin et al., 2025). With the emergence of LLMs, prompting-based methods have been extensively studied, including in-context learning with well-designed demonstrations (Gao et al., 2023; Pourreza and Rafiei, 2023; Lee et al., 2025b), SQL-specific chain-of-thought prompting (Dong et al., 2023; Liu and Tan, 2023), and execution consistency based selection strategies with multiple generated candidates (Lee et al., 2025a). However, these methods did not leverage execution feedback as learning signals.

2.2 Reinforcement Learning in Text-to-SQL

Recently, reinforcement learning has been applied to Text-to-SQL leveraging execution feedback (Ma et al., 2025; Pourreza et al., 2025b; Yao et al., 2025). Most methods adopted GRPO (Shao et al., 2024) for policy optimization (Yao et al., 2025; Ma et al., 2025; Zhang et al., 2025; Pourreza et al., 2025b), while some explored DPO (Rafailov et al., 2023) by constructing preference pairs from execution results (Zhai et al., 2025).

Since execution results provide only binary feedback, some methods designed additional rewards for more informative signals. ReasoningSQL (Pourreza et al., 2025b) combined LLM-as-a-judge evaluation with syntactic validity and schema matching rewards. RewardSQL (Zhang et al., 2025) introduced a process reward model that evaluates intermediate reasoning steps. GraphRewardSQL (Weng et al., 2025) proposed the graph-based reward with gold SQL comparison.

However, those methods assigned rewards at the query-level, treating all clauses equally despite only specific clauses being erroneous. They failed to

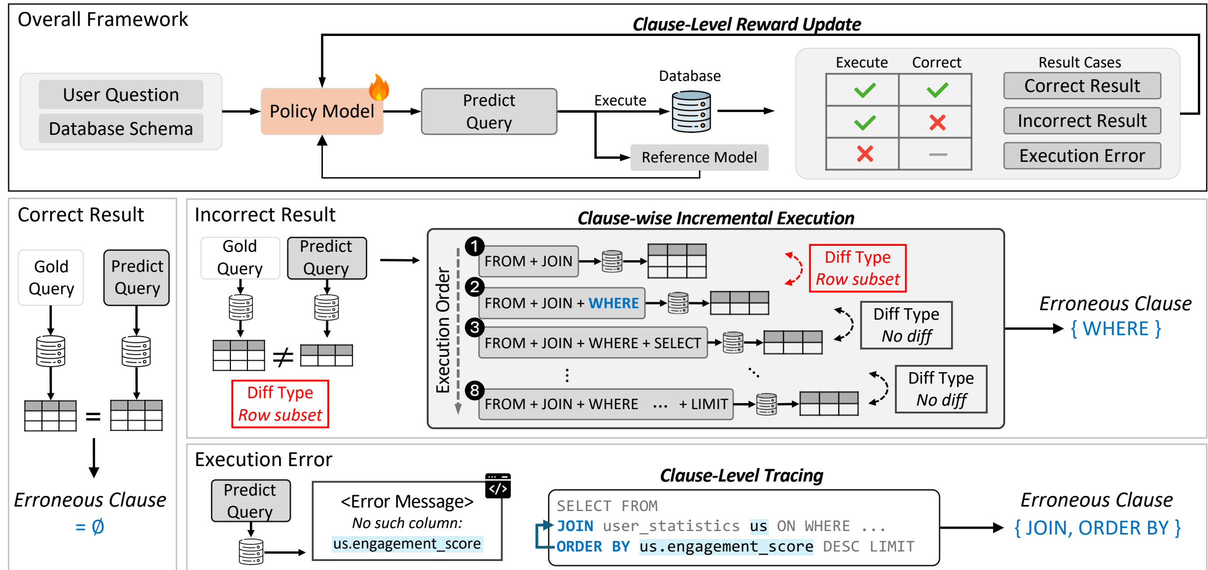

Figure 2: The overall framework of EXPO-SQL. Predicted queries are classified into three cases based on execution results and correctness. For each case, erroneous clauses are detected to assign clause-level rewards: correct results have no erroneous clauses (Section 3.2.1), incorrect results use difference type detection through incremental execution (Section 3.2.2), and execution errors employ error messages and clause-level tracing (Section 3.2.3).

distinguish correct clauses from erroneous ones, resulting in insufficient learning signals.

3 EXPO-SQL

In this section, we introduce EXPO-SQL, a novel clause-level reinforcement learning framework. Figure 2 illustrates the overall framework of EXPO-SQL. Instead of using execution feedback as a query-level reward, we utilize execution results to identify clause-level errors and design clause-level rewards. We first formulate Text-to-SQL generation as clause-level policy optimization in Section 3.1. We then present execution-based clause-level reward design for different execution results in Section 3.2, and describe how these rewards are incorportaed into policy optimization in Section 3.3.

3.1 Problem Formulation

Given a natural language question and a database schema, the policy model π_θ generates a SQL query O. We define the generated SQL query as a sequence of n clauses: $(c_1, c_2, \ldots, c_n)$, where each clause corresponds to a structural component of the SQL query. The generated query is executed against the database to obtain an execution result table R_{pred}, then the reward signal is extracted by comparing R_{pred} with ground-truth R_{gold}.

Our goal is to provide clause-level learning signals, rather than query-level rewards, by assigning differentiated rewards r_c to individual clauses $c \in O$. To this end, we erroneous clauses $C_{\text{err}} \subseteq \{c_1, \ldots, c_n\}$ by leveraging detailed execution feedback, including error messages and differences between execution results. Based on the identified erroneous clauses, we design clause-level rewards r_c that guide the policy model toward improved SQL generation.

3.2 Clause-level Reward Design

In order to adopt fine-grained clause-level rewards, we identify erroneous clauses by analyzing the execution result of the generated SQL query. Since the information available for identifying erroneous clauses varies depending on execution results, it is necessary to apply different approaches accordingly. Therefore, we categorize execution results into the following three cases: (1) Correct Result: the query is executed successfully and $R_{pred} = R_{gold}$. (2) Incorrect Result: the query is executable but $R_{pred} \neq R_{gold}$. (3) Execution Error: the query fails to be executed. For each case, we effectively identify C_{err} by leveraging detailed execution feedback and assign different rewards r_c to each clause $c \in O$ to provide fine-grained learning signals.

3.2.1 Correct Result

When the generated SQL query is successfully executed and produces the correct result, all clauses are considered correct, meaning $C_{err} = \emptyset$. Therefore,

Level	Diff Type	Description
Column	col_count	Number of columns differs
	col_name	Column names differ
Row	row_order	Row sequence differs
	row_dedup	Duplicate counts differ
	row_subset	Some rows are missing
	row_superset	Extra rows are present
	row_emptied	Result is empty, gold is not
	row_created	Result has rows, gold is empty
	row_disjoint	No intersection between results
Value	row_partial	Cell values differ

Table 1: The 10 difference types (Diff Type) based on changes in columns, rows, values.

we assign the same positive reward to all clauses.

$$r_c = +1.5 \quad \forall c \in O \tag{1}$$

3.2.2 Incorrect Result

When the generated SQL query is executable but the execution result is incorrect, there is no direct information about which clause caused the error. In this case, to identify C_{err}, we consider analyzing the difference between R_{pred} and R_{gold}. However, since both results show the tables with the cumulative effect of all clauses, we cannot directly identify which clauses lead to the incorrect result. To address this challenge, we leverage each SQL clause performs a distinct operation on the result. By analyzing these clause-specific effects, we can trace back which clause caused the differences in the final result. We accomplish this by two steps. First, we define *difference types from execution results* to specify which difference each clause can produce. Second, with those types, we perform *clause-wise incremental execution* to detect the actual effect caused by each clause, thereby identifying C_{err}.

Difference Types from Execution Result. We first define 10 difference types, denoted as diff_type from execution result based on the logical operations of SQL clauses. Since SQL results are returned as tables, any difference between two execution results appear as changes in the columns, rows or values. Table 1 shows the 10 diff_type. More detail explanations are provided in Appendix A.1.

Clause-wise Incremental Execution. After defining the diff_type, we detect C_{err} that caused the diff_type observed between R_{pred} and R_{gold}. Identifying C_{err} is straightforward when each diff_type is produced by only one clause. However, in most SQL queries, diff_type can be caused by multiple clauses. Therefore, we need a comprehensive analysis across the entire generated SQL query to determine how each clause affects the execution result. To enable this, we leverage the deterministic execution order of SQL. When executing an SQL query, clauses are operated in a predefined logical order, with each clause accessing tables and transforming the previous clause's outcome (Guagliardo and Libkin, 2017a; Benzaken and Contejean, 2019). The logical execution order is as follows: `FROM/JOIN` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY` → `LIMIT`. We first decompose the generated SQL query based on the order, and incrementally add clauses one by one starting from the `FROM/JOIN` clause in the order. By observing how the execution result R_i changes as each clause c_i is added, we can identify which clause leads to the difference. Algorithm 1 provides the detailed procedure.

Algorithm 1 Clause-wise Incremental Execution

Require: SQL $O = (c_1, ..., c_n)$, R_{gold}
Ensure: Erroneous clauses C_{err}
1: $C_{err} \leftarrow \emptyset$
2: Replace `SELECT` clause with `SELECT *`
3: $(c'_1, ..., c'_n) \leftarrow$ Reorder(O) in logical execution order
4: $R_1 \leftarrow$ Execute(c'_1) ▷ `FROM/JOIN`
5: $d_1 \leftarrow$ diff(R_{gold}, R_1)
6: **for** $i = 2$ to n **do**
7: $R_i \leftarrow$ Execute$(c'_1, ..., c'_i)$
8: $d_i \leftarrow$ diff(R_{i-1}, R_i)
9: **end for**
10: $D^* \leftarrow$ diff(R_{gold}, R_{pred})
11: **for** each clause c'_i **do**
12: **if** $d_i \in D^*$ **then**
13: $C_{err} \leftarrow C_{err} \cup \{c'_i\}$
14: **end if**
15: **end for**
16: **return** C_{err}

By comparing the results between the cumulative executions, we can classify the diff_type $d_i = \text{diff}(R_{i-1}, R_i)$ for each clause c_i. For the first execution where only the `FROM/JOIN` clause is executed, there is no previous result to compare. In this case, we classify the diff_type against R_{gold}, and only consider whether the diff_type is `column_count`.

After obtaining all diff_type through clause-wise incremental execution, we also classify the final difference $D^* = \text{diff}(R_{pred}, R_{gold})$, which represents the cumulative effect of all individual clause changes during execution. A clause c_i is identified as erroneous clause and added to C_{err} if its corresponding diff_type d_i appears in D^*. More details are provided in Appendix A.2. We assign

clause-level rewards as follows:

$$r_c = \begin{cases} -0.5 & \text{if } c \in \mathcal{C}_{\text{err}} \\ +0.5 & \text{otherwise} \end{cases} \quad (2)$$

Since the query is executable with valid syntax, we use relatively weak rewards to fix only C_{err} while preserving the correct clauses. This allows the model to learn which clauses to fix while retaining ones that are already correct. A detailed case study is provided in Appendix C.2.

3.2.3 Execution Error

When the query fails to be executed, we leverage the error message from the executor to provide information about C_{err}. The message typically contains the error location, type, and the names of tables or columns that caused the error. Details of the error messages are provided in Appendix A.3.

However, the clause indicated by the error message is where the execution failed, not necessarily the root cause. As shown in Figure 2, while the error message points to `us.engagement_count` in the `ORDER BY` clause, the actual cause is the preceding JOIN clause that joined a table without this column. To identify such root causes, we trace the reference relationships between clauses.

To trace those relationships, we utilize the defined logical order of SQL execution. In SQL, clauses executed in the order, with later clauses referencing tables and columns defined by earlier clauses. We trace these references backward along the execution order to find the root C_{err}. We first parse the error message to extract the table or column that caused the error, and identify the clause containing it. We then trace backward through the preceding clauses, adding any clause that references or defines the error-causing element. All identified clauses form C_{err}. Based on the identified C_{err}, we assign rewards as follows:

$$r_c = \begin{cases} -1.5 & \text{if } c \in \mathcal{C}_{\text{err}} \\ -0.5 & \text{otherwise} \end{cases} \quad (3)$$

Since the query failed to execute, we assign weak penalties even to clauses outside C_{err} to discourage generating invalid SQL overall. A detailed case study is provided in Appendix C.3.

3.3 Execution-based Clause-level Policy Optimization

We assign the clause-level rewards computed in Section 3.2, into clause-level policy optimization. Since SQL execution provides deterministic feedback on correctness, we directly assign clause-level rewards without relative comparison across multiple samples. Prior methods assign the query-level reward to all tokens in a generated SQL query. In contrast, our method assigns rewards at the clause level: all tokens within the same clause receive the same reward r_c, while tokens in different clauses receive different rewards based on their clause's correctness. We define the policy optimization loss as:

$$\mathcal{L} = -\mathbb{E}\left[\sum_{c \in O}\sum_{t_i \in c}\left(r_c - \beta D_{KL}^{(i)}\right)\log \pi_\theta(t_i|t_{<i})\right] \quad (4)$$

where O is the generated SQL query as a sequence of clauses, c denotes each clause in O, t_i denotes a token belonging to clause c, $\pi_\theta(t_i|t_{<i})$ is the probability of generating token t_i given previous tokens, and $D_{KL}^{(i)}$ is the KL divergence penalty to prevent the policy from deviating too far from the reference model.

Through the clause-level optimization, tokens in erroneous clause ($c \in C_{err}$) receive negative rewards, decreasing their generation probability, while tokens in correct clauses receive positive rewards, reinforcing their generation. This enables the model to selectively correct erroneous clauses while preserving correct ones.

4 Experiment Setups

4.1 Datasets

We adopt the SynSQL-Complex-5k dataset for training following SQL-R1 (Ma et al., 2025), which consists of 5k complex NL-SQL pairs sampled from the widely-used SynSQL-2.5M (Li et al., 2025b).

For evaluation, we use Spider (Yu et al., 2018) and BIRD (Li et al., 2023), two widely-used cross-domain Text-to-SQL benchmarks. We report results on Spider development set, Spider test set, and BIRD development set. We also further evaluate on Spider-DK (Gan et al., 2021b), Spider-Syn (Gan et al., 2021a), and Spider-Realistic (Deng et al., 2021). Dataset statistics and details are provided in Appendix B.1

4.2 Baselines

We comprehensively compare EXPO-SQL with SFT-based, prompting-based, and state-of-the-art RL-based methods. Detailed descriptions of each baseline are provided in Appendix B.2.

Methods	Base Model	Spider (Dev)	Spider (Test)	BIRD (Dev)
SFT-based methods				
OmniSQL-7B (Li et al., 2025b)	Qwen2.5-Coder-7B	85.5	88.9	66.1
ROUTE (Qin et al., 2025)	Qwen2.5-Coder-7B	84.7	85.1	66.7
SENSE-7B (Yang et al., 2024)	CodeLLaMA-7B	83.2	83.5	51.8
DTS-SQL (Pourreza and Rafiei, 2024)	DeepSeek-7B	85.5	84.4	55.8
Prompt-based methods				
DAIL-SQL (Gao et al., 2023)	GPT-4	83.1	86.6	54.8
MCS-SQL (Lee et al., 2025a)	GPT-4	89.5	89.6	63.4
Alpha-SQL (Li et al., 2025a)	Qwen2.5-Coder-7B	84.0	-	66.8
CHASE-SQL (Pourreza et al., 2025a)	Gemini-1.5-Pro	-	87.6	73.0
CHESS-SQL (Talaei et al., 2024)	Deepseek-33B	-	87.2	65.0
SGU-SQL (Zhang et al., 2024)	GPT-4	87.9	-	61.8
RL-based methods				
SQL-R1 (Ma et al., 2025)	Qwen2.5-Coder-7B	87.6	88.7	66.6
SQL-R1 (Ma et al., 2025)	Qwen2.5-Coder-14B	86.7	88.1	67.1
Reward-SQL (Zhang et al., 2025)	Qwen2.5-Coder-7B	81.7	-	68.9
Reasoning-SQL (Pourreza et al., 2025b)	Qwen2.5-Coder-7B	78.7	-	64.0
Reasoning-SQL (Pourreza et al., 2025b)	Qwen2.5-Coder-14B	81.4	-	65.3
Graph-Reward-SQL (Weng et al., 2025)	Qwen2.5-Coder-7B	81.6	-	63.0
Arctic-SQL-R1 (Yao et al., 2025)	Qwen2.5-Coder-7B	87.3*	88.8	68.9
Arctic-SQL-R1 (Yao et al., 2025)	Qwen2.5-Coder-14B	-	89.4	70.1
ExCoT (Zhai et al., 2025)	Qwen2.5-Coder-32B	-	85.1	68.2
EXPO-SQL (ours)	Qwen2.5-Coder-7B	**88.5**	**89.1**	**71.3**
EXPO-SQL (ours)	Qwen2.5-Coder-14B	**89.2**	**89.5**	**73.0**

Table 2: Performance comparison with baselines on Spider and Bird benchmarks. We use execution accuracy (%) as our primary metric, which measures whether the predicted SQL produces the same execution result as the gold SQL. All baseline results are reported from the original papers ($*$: reproduced by us).

4.3 Implementation Details

We use Qwen2.5-Coder 7B and 14B (Hui et al., 2024) as our base model. Following prior work (Talaei et al., 2024; Ma et al., 2025; Li et al., 2025b), we represent database schema using CREATE TABLE statements. For fair comparison, we evaluate using zero-shot inference on datasets independent of training data. Training and inference details are provided in Appendix B.3.

5 Experiment Results

5.1 Overall Results

Table 2 presents the performance comparison of various Text-to-SQL methods. EXPO-SQL consistently achieves the best performance across all benchmarks for both 7B and 14B, highlighting the effectiveness of our clause-level reward approach.

Our method shows substantial improvements over SFT-based methods across all benchmarks. With 7B parameters, it outperforms ROUTE, the best-performing SFT method, by 4.6%p on BIRD-Dev. For prompting-based methods that employ multi-agent pipelines or iterative refinement, EXPO-SQL achieves comparable or superior results with significantly fewer parameters. Among RL-based methods, our method outperforms all baselines with both 7B and 14B models by providing clause-level rewards that enable fine-grained supervision. Specifically, the 7B model outperforms Arctic-SQL-R1 7B by 2.4%p and EXPO-SQL 14B outperforms Arctic-SQL-R1 14B by 2.9%p on BIRD-Dev. The improvement is more pronounced on BIRD, which contains more complex queries requiring multiple clauses. Notably, our 7B and 14B models outperform the 32B ExCoT by 3.1%p and 4.8%p respectively, achieving superior performance with substantially fewer parameters.

Table 3 presents execution accuracy on Spider-series datasets, which evaluate generalization under domain shift (Spider-DK), synonym transformations (Spider-Syn), and realistic user queries (Spider-Realistic). EXPO-SQL outperforms existing methods across all three datasets, demonstrating the generalizability of our approach.

5.2 Ablation Study

We conduct an ablation study on the clause-level reward design on BIRD-Dev. Since correct results assign uniform positive rewards to all clauses, we

Method	Detail	Spider-DK	Spider-Syn	Spider-Realistic
Qwen2.5-Coder-7B	-	67.9	70.2	75.4
OmniSQL-7B	SFT	77.8	69.6	78.0
SENSE-7B	SFT	77.9	72.6	82.7
ACT-SQL	Prompt	68.2	67.9	75.8
MAC-SQL	Prompt	71.4	72.5	79.9
SQL-R1	RL	78.1	76.7	83.3
Reasoning-SQL	RL	73.3	69.3	-
EXPO-SQL (Ours)	RL	**79.9**	**83.1**	**83.4**

Table 3: Execution accuracy (%) on three Spider-series datasets. The existing prompting-based methods, ACT-SQL (Zhang et al., 2023) and MAC-SQL (Wang et al., 2025a), use GPT-4 as the base model, while all other baselines use Qwen2.5-Coder-7B.

Configuration	BIRD (Dev)
EXPO-SQL	**71.3**
w/o Execution Error	69.6 (-1.7)
w/o Incorrect Result	68.9 (-2.4)
w/o Both	68.5 (-2.8)

Table 4: Ablation study on clause-level reward design on BIRD dev. "w/o" denotes replacing clause-level rewards with uniform query-level rewards for that case.

C_{err} Identification Method	BIRD (Dev)
Query-level reward	
None	68.9
Clause-level reward	
+ Diff Types	69.8 (+0.9)
+ Clause-wise Incr. Exec.	70.2 (+1.3)
+ Diff Types, Clause-wise Incr. Exec.	**71.3** (+2.4)

Table 5: Component analysis on incorrect result case. We compare methods for identifying C_{err}: using diff types alone, using clause-wise incremental execution (Incr. Exec.) alone, and combining both.

C_{err} Identification Method	BIRD (Dev)
Query-level reward	
None	69.6
Clause-level reward	
+ Error Message Parsing	70.1 (+0.5)
+ Error Message Parsing, Clause-level Tracing	**71.3** (+1.7)

Table 6: Component analysis on execution error case. We compare methods for identifying C_{err}: using error message parsing alone, and adding clause-level tracing to find root causes.

ablate the clause-level rewards for incorrect result and execution error cases by replacing them with uniform query-level rewards. As shown in Table 4, applying clause-level rewards for both cases show the best performance. Removing clause-level rewards for execution error results in 1.7%p drop, while removing them for incorrect result leads to a larger drop of 2.4%p. The incorrect result case benefits more from clause-level rewards as it requires identifying specific erroneous clauses within a partially correct query. Removing both results in the largest drop of 2.8%p confirming that clause-level rewards provide more fine-grained learning signals than query-level rewards.

6 Further Analysis

For in-depth analysis of our method, we conduct additional experiments. We provide case-wise component analysis (Section 6.1), comparison across various model scales (Section 6.2), analysis by SQL complexity (Section 6.3), effectiveness across backbone models (Section 6.4), training efficiency analysis (Section 6.5), reward sensitivity analysis (Section 6.6), and statistical significance (Section 6.7).

6.1 Case-wise Component Analysis

We analyze the effectiveness of each component in the clause-level reward design for each case described in Section 3.2.

Incorrect Result. We apply diff_type detection and clause-wise incremental execution to identify C_{err}. Each can be used independently: diff_type can identify C_{err} directly when the matching is unique to one clause, while incremental execution can trace each clause's actual change without relying on diff type matchings. As shown in Table 5, using diff types alone achieves 0.9%p higher over query-level reward, while clause-wise incremental execution alone achieves 1.3%p improvement. Combining both yields the best performance of 71.3%, 2.4%p improvement as diff types narrow down candidates while incremental execution pinpoints the exact clause.

Execution Error. We identify C_{err} through error message parsing and clause-level tracing. Error message parsing indicates where execution failed, while clause-level tracing finds root causes in preceding clauses. As shown in Table 6, error message parsing alone achieves 0.5%p improvement over query-level reward. Adding clause-level tracing yields 71.3%, 1.7%p improvement, as it traces back to the actual source of errors.

These results confirm that each component of our clause-level reward design contributes to more accurate C_{err} identification, leading to more effective clause-level rewards.

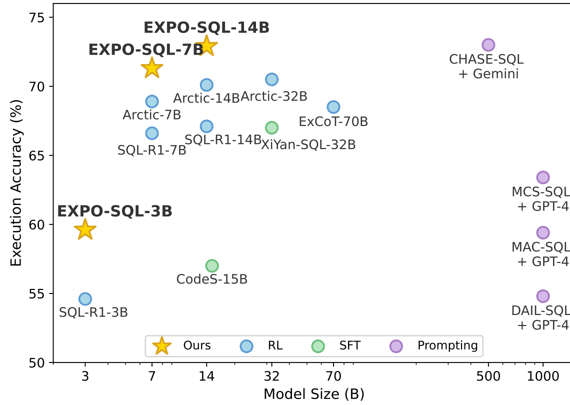

Figure 3: Execution accuracy across model scales on BIRD-Dev. EXPO-SQL achieves superior performance with significantly smaller models compared to existing RL, SFT, and prompting-based methods.

Method	Simple	Moderate	Challenging
PPO	74.3	62.8	58.3
GRPO	74.2	63.1	58.3
EXPO-SQL	**76.2**	**63.9**	**63.9**

Table 7: Execution accuracy (%) of different difficulty levels on BIRD-Dev. All methods use the same reward values (ours); EXPO-SQL assigns rewards at clause-level while PPO and GRPO assign at query-level.

6.2 Comparison across Various Model Scales

Figure 3 analyzes the relationship between model size and performance. EXPO-SQL consistently outperforms both RL-based and SFT-based methods at identical model sizes. Furthermore, EXPO-SQL-7B achieves higher accuracy than prompting-based methods using much larger LLMs (GPT-4). These results demonstrate that clause-level rewards provide more effective learning signals, enabling smaller models to achieve comparable or superior performance with substantially fewer parameters.

6.3 Analysis by SQL complexity

Table 7 compares EXPO-SQL with PPO and GRPO on BIRD-Dev subsets of varying difficulty. EXPO-SQL outperforms the baselines across all levels, with notable 5.6%p improvement on challenging queries. This demonstrates that clause-level rewards are particularly effective for complex SQL, as they enable the model to identify and correct specific erroneous clauses rather than penalizing the entire query. Further details on how complex SQL structures (nested subqueries, CTEs, set operations) are handled are provided in Appendix A.4.

Model	Spider (Dev)	Spider (Test)	BIRD (Dev)
Qwen2.5-Coder-7B	83.5	81.5	51.5
w/ EXPO-SQL	**88.5** (+5.0)	**89.1** (+7.6)	**71.3** (+19.8)
Qwen2.5-Coder-14B	83.8	84.8	56.9
w/ EXPO-SQL	**89.2** (+5.4)	**89.5** (+4.7)	**72.9** (+16.0)
DeepseekCoder-6.7B	78.2	78.5	58.5
w/ EXPO-SQL	**80.4** (+2.2)	**81.2** (+2.7)	**60.7** (+2.2)
Ministral-8B	61.7	62.5	45.9
w/ EXPO-SQL	**69.4** (+7.7)	**69.7** (+7.2)	**48.4** (+2.5)
llama3.1-8B	72.4	73.1	49.7
w/ EXPO-SQL	**76.7** (+4.3)	**77.7** (+4.6)	**58.6** (+8.9)

Table 8: Generalization across different base models. We compare execution accuracy (%) of various base models with EXPO-SQL on three evaluation datasets.

Model	Spider (Dev)	Spider (Test)	BIRD (Dev)
OmniSQL-7B	85.5	**88.9**	66.1
w/ SQL-R1	**87.6** (+2.1)	88.7 (-0.2)	66.6 (+0.5)
w/ EXPO-SQL	87.2 (+1.7)	88.2 (-0.7)	**72.5** (+6.4)
OmniSQL-14B	86.2	88.3	65.9
w/ SQL-R1	86.4 (+0.2)	87.6 (-0.7)	66.6 (+0.7)
w/ EXPO-SQL	**87.9** (+1.7)	**88.7** (+0.4)	**73.2** (+7.3)

Table 9: Performance comparison of RL methods applied to OmniSQL (SFT model). OmniSQL is fine-tuned from Qwen2.5-Coder on synsql-2.5M.

6.4 Effectiveness Across Various Backbone Models

We verify whether EXPO-SQL achieves consistent improvements across various backbone models. Table 8 shows improvement when EXPO-SQL is applied to different pre-trained models (Guo et al., 2024; Jiang et al., 2023; Grattafiori et al., 2024). Our method achieves consistent improvements across all benchmarks regardless of model type and size.

Table 9 shows the results when applying RL methods to OmniSQL, which is fine-tuned from Qwen2.5-Coder on SynSQL-2.5M. On BIRD-Dev, EXPO-SQL achieves 6.4%p (7B) and 7.3%p (14B) improvement, significantly outperforming SQL-R1. This demonstrates that clause-level rewards are effective regardless of the base model.

6.5 Training Efficiency

Table 10 compares the wall-clock time per training step. Despite performing more database calls per sample for clause-wise incremental execution, EXPO-SQL is 22% faster than GRPO and 7% faster than PPO. The speedup comes from sample efficiency: GRPO requires n=8 samples per prompt for group-relative optimization, while EXPO-SQL uses REINFORCE++ (Hu, 2025) with

Metric	GRPO	PPO	EXPO-SQL
Generation time (s)	15.59	11.46	10.16
Reward time (s)	0.69	0.72	0.60
Actor update (s)	6.61	6.61	6.66
Other (ref / critic) (s)	1.15	2.65	1.16
Total step time (s)	**25.67**	**21.45**	**20.02**
DB calls / sample	2.0	2.0	∼3.4
DB time / batch (s)	∼0.38	∼0.38	∼0.52
DB % of step time	1.5%	1.8%	2.6%

Table 10: Wall-clock time breakdown per training step (8× H100, Qwen2.5-Coder-7B, batch size 64).

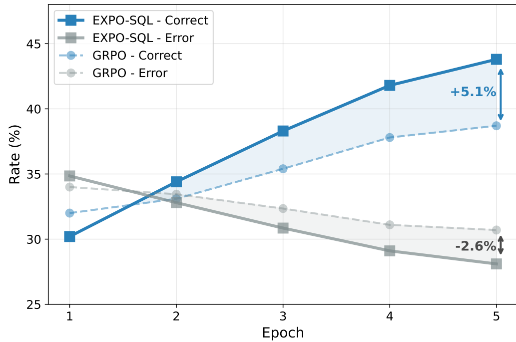

Figure 4: Correct and error rates during training for EXPO-SQL and GRPO. Correct rate represents the ratio of queries producing correct execution results. Error rate denotes the average of incorrect result and execution error rates.

n=1 sample per prompt, saving 5.43s in generation time. Compared to PPO, EXPO-SQL requires no critic model, eliminating ∼1.5s per step for value network updates. The extra DB calls from clause-wise execution add less than 3% to total step time, as each call takes only ∼3ms.

Beyond wall-clock efficiency, Figure 4 shows that EXPO-SQL also achieves better *learning* efficiency. Throughout training, EXPO-SQL consistently maintains a higher correct rate and lower error rate than GRPO, achieving a 5.1%p higher correct rate and 2.6%p lower error rate at the final epoch. This indicates that clause-level rewards provide more informative gradient signals per training step, enabling the model to improve more rapidly than under query-level rewards.

6.6 Reward Sensitivity Analysis

Table 11 examines the sensitivity of performance to reward magnitude. We scale the gap between C_{err} and non-C_{err} clause rewards while preserving the hierarchical structure (Match > Mismatch > Execution Error). All differentiated configurations fall within a 0.5-point range (70.8–71.3), while removing clause-level differentiation entirely drops performance to 68.5. This confirms that the structure of separating erroneous from correct clauses drives performance, not the specific magnitudes.

Configuration	Gap Scale	BIRD (Dev)
No differentiation (uniform)	×0	68.5
Halved	×0.5	70.8
Ours (default)	×1.0	**71.3**
Doubled	×2.0	70.9

Table 11: Reward sensitivity analysis. We scale the clause-level reward gap while preserving the hierarchical structure. Performance is robust across scales (70.8–71.3), while removing differentiation entirely drops to 68.5.

6.7 Statistical Significance

We conducted paired bootstrap resampling (10,000 iterations) comparing EXPO-SQL against the uniform query-level reward baseline. The improvements are statistically significant: BIRD-Dev ($p < 0.001$) and Spider-Dev ($p < 0.05$). Additionally, consistent gains across five independent backbone architectures (Table 8), where all 15 combinations of backbone × benchmark show positive improvement, further support that the effect is not due to random variance.

7 Conclusion

We proposed EXPO-SQL, a novel reinforcement learning framework for Text-to-SQL. This method enabled clause-level policy optimization by leveraging execution results to identify erroneous clauses and assign clause-level rewards. To the best of our knowledge, this is the first work to introduce clause-level rewards in RL-based Text-to-SQL. Extensive experiments showed that clause-level supervision enables more precise and fine-grained learning.

Limitations

We demonstrate that leveraging execution feedback enables effective clause-level reward assignment for Text-to-SQL. However, our experiments are conducted on SQLite-based benchmarks, and generalization to other database dialects (e.g., PostgreSQL, MySQL) requires further investigation. Additionally, error message formats vary across database systems, which require adaptation of our parsing approach for different executors.

Acknowledgments

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (RS-2019-II190421, 13%; IITP-2026-RS-2024-00437633, 12%; RS-2025-25442569, 12%; IITP-2026-RS-2024-00360227, 13%; and No.RS-2023-00228970, 12%) and National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (RS-2024-00352717, 13%), funded by the Korea government(MSIT) (RS-2025-00521391, 13%), and funded by the Ministry of Education (RS-2025-25433088, 12%)

References

Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. 2019. A comparative survey of recent natural language interfaces for databases. *The VLDB Journal*, 28(5):793–819.

Véronique Benzaken and Évelyne Contejean. 2019. A coq mechanised formal semantics for realistic sql queries: formally reconciling sql and bag relational algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 249–261.

Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. 2015. Data generation for testing and grading SQL queries. *The VLDB Journal*, 24(6):731–755.

Adriane Chapman and H. V. Jagadish. 2009. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 523–534. ACM.

Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 510–524. ACM.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent advances in text-to-SQL: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

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

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

Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. 2021a. Towards robustness of text-to-SQL models against synonym substitution. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2505–2515, Online. Association for Computational Linguistics.

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

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *CoRR*, abs/2308.15363.

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

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 31–40. ACM.

Paolo Guagliardo and Leonid Libkin. 2017a. A formal semantics of sql queries, its validation, and applications. *Proc. VLDB Endow.*, 11:27–39.

Paolo Guagliardo and Leonid Libkin. 2017b. A formal semantics of SQL queries, its validation, and applications. *Proceedings of the VLDB Endowment*, 11(1):27–39.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-generation database interfaces: A survey of llm-based text-to-sql. *arXiv preprint arXiv:2406.08426*.

Jian Hu. 2025. Reinforce++: A simple and efficient approach for aligning large language models. *arXiv preprint arXiv:2501.03262*.

Jian Hu, Xibin Wu, Zilin Zhu, Xianyu, Weixun Wang, Dehao Zhang, and Yu Cao. 2024. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*.

Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. 2008. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment*, 1(1):736–747.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Radu Cristian Alexandru Iacob, Florin Brad, Elena-Simona Apostol, Ciprian-Octavian Truică, Ionel Alexandru Hosu, and Traian Rebedea. 2020. Neural approaches for natural language interfaces to databases: A survey. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 381–395, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Dongsheng Jiang, Yuchen Liu, Songlin Liu, Jin'e Zhao, Hao Zhang, Zhen Gao, Xiaopeng Zhang, Jin Li, and Hongkai Xiong. 2023. From clip to dino: Visual encoders shout in multi-modal large language models. *arXiv preprint arXiv:2310.08825*.

Mahmoud Abo Khamis, Phokion G. Kolaitis, Hung Q. Ngo, and Dan Suciu. 2020. Bag query containment and information theory. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, pages 95–112. ACM.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025a. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 337–353.

Jihyung Lee, Jin-Seop Lee, Jaehoon Lee, YunSeok Choi, and Jee-Hyong Lee. 2025b. DCG-SQL: Enhancing in-context learning for text-to-SQL with deep contextual schema link graph. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15397–15412, Vienna, Austria. Association for Computational Linguistics.

Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025a. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. In *Forty-Second International Conference on Machine Learning, ICML 2025, Vancouver, Canada, July 13-19, 2025*. OpenReview.net.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, and 1 others. 2025b. Omnisql: Synthesizing high-quality text-to-sql data at scale. *arXiv preprint arXiv:2503.02240*.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards building open-source language models for text-to-sql. *Proc. ACM Manag. Data*, 2(3).

Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Preprint*, arXiv:2305.03111.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. 2018. Memory augmented policy optimization for program synthesis and semantic parsing. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Hanbing Liu, Haoyang Li, Xiaokang Zhang, Ruotong Chen, Haiyong Xu, Tian Tian, Qi Qi, and Jing Zhang. 2025. Uncovering the impact of chain-of-thought reasoning for direct preference optimization: Lessons from text-to-SQL. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 21223–21261, Vienna, Austria. Association for Computational Linguistics.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024. A survey of NL2SQL with large language models: Where are we, and where are we going? *CoRR*, abs/2408.05109.

Xiping Liu and Zhao Tan. 2023. Divide and prompt: Chain of thought prompting for text-to-sql. *arXiv preprint arXiv:2304.11556*.

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

Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-sql in the age of well-reasoned language models. *arXiv preprint arXiv:2408.07702*.

Jerzy Marcinkowski and Mateusz Orda. 2024. Bag semantics conjunctive query containment: Four small steps towards undecidability. *Proceedings of the ACM on Management of Data (PACMMOD)*, 2(2):1–27.

Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining wrong queries using small examples. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, pages 503–520. ACM.

Eduardo Pignatelli, Johan Ferret, Matthieu Geist, Thomas Mesnard, Hado van Hasselt, and Laura Toni. 2024. A survey of temporal credit assignment in deep reinforcement learning. *Transactions on Machine Learning Research*. Survey Certification.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Arik. 2025a. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. In *International Conference on Representation Learning*, volume 2025, pages 60385–60415.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems*, volume 36, pages 36339–36348. Curran Associates, Inc.

Mohammadreza Pourreza and Davood Rafiei. 2024. DTS-SQL: Decomposed text-to-SQL with small large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8212–8220, Miami, Florida, USA. Association for Computational Linguistics.

Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, Sercan Arik, and 1 others. 2025b. Reasoning-sql: Reinforcement learning with sql tailored partial rewards for reasoning-enhanced text-to-sql. *arXiv preprint arXiv:2503.23157*.

Yang Qin, Chao Chen, Zhihang Fu, Ze Chen, Dezhong Peng, Peng Hu, and Jieping Ye. 2025. Route: Robust multitask tuning and collaboration for text-to-sql. In *International Conference on Representation Learning*, volume 2025, pages 12425–12448.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, volume 36, pages 53728–53741. Curran Associates, Inc.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. 2025. A survey on employing large language models for text-to-sql tasks. *ACM Comput. Surv.*, 58(2).

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*.

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

Yihan Wang, Peiyu Liu, and Xin Yang. 2025b. LinkAlign: Scalable schema linking for real-world large-scale multi-database text-to-SQL. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 977–991, Suzhou, China. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Han Weng, Puzhen Wu, Cui Longjie, Yi Zhan, Boyi Liu, Yuanfeng Song, Dun Zeng, Yingxiang Yang, Qianru Zhang, Dong Huang, Xiaoming Yin, Yang Sun, and Xing Chen. 2025. Graph-reward-SQL: Execution-free reinforcement learning for text-to-SQL via graph matching and stepwise reward. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 12917–12943, Suzhou, China. Association for Computational Linguistics.

Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. 2024. Synthesizing text-to-SQL data from weak and strong LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7864–7875, Bangkok, Thailand. Association for Computational Linguistics.

Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2025. Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql. *arXiv preprint arXiv:2505.20315*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018*

Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium. Association for Computational Linguistics.

Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. 2025. Optimizing reasoning for text-to-SQL with execution feedback. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 19206–19218, Vienna, Austria. Association for Computational Linguistics.

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

Qinggang Zhang, Hao Chen, Junnan Dong, Shengyuan Chen, Feiran Huang, and Xiao Huang. 2024. Structure guided large language model for sql generation. *arXiv preprint arXiv:2402.13284*.

Yuxin Zhang, Meihao Fan, Ju Fan, Mingyang Yi, Yuyu Luo, Jian Tan, and Guoliang Li. 2025. Reward-sql: Boosting text-to-sql via stepwise reasoning and process-supervised rewards. *arXiv preprint arXiv:2505.04671*.

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

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

Appendix

A Additional Details About EXPO-SQL

A.1 Difference Type Classification

We classify the differences between predicted execution result R_{pred} and gold result R_{gold} into 10 diff_type. Since SQL results are returned as tables, any difference appear as changes in columns, rows, or values.

Theoretical Intractability of Unique Identification. Uniquely identifying a single erroneous clause from execution results alone is theoretically intractable, as it reduces to the SQL equivalence problem, which is undecidable in the general case (Chu et al., 2017) and remains open even for conjunctive queries under the bag semantics that SQL uses (Khamis et al., 2020; Marcinkowski and Orda, 2024). Execution-based evaluation on finite database instances is also inherently incomplete for determining semantic equivalence (Zhong et al., 2020). Instead of targeting unique identification, we narrow down the set of candidate erroneous clauses by leveraging the operational semantics of individual SQL clauses, providing substantially finer-grained learning signals than query-level rewards that penalize the entire query uniformly.

Derivation from SQL Semantics. A SQL result is a bag of tuples over a fixed schema, with optional ordering (Guagliardo and Libkin, 2017b; Green et al., 2007). Any difference between two SQL results must therefore manifest in one of four axes: *schema*, *content*, *ordering*, or *value*. Table 13 shows how these axes yield our 10 diff types.

This decomposition aligns with the dichotomy between why and why-not provenance (Chapman and Jagadish, 2009; Huang et al., 2008) and with prior SQL testing and evaluation work (Chandra et al., 2015; Miao et al., 2019; Zhong et al., 2020).

Column-Level Differences. Column-level differences occur when the schema of the two results differs. If the number of columns differs, we classify it as `col_count`. If the column count matches but the column names differ, we classify it as `col_name`. These differences typically originate from the `SELECT` clause.

Row-Level Differences. Row-level differences occur when the schema matches but the rows differ. We further distinguish these based on how the row sets relate to each other. If both results contain the same rows but in different order, we classify it as `row_order`, which typically originates from `ORDER BY`. If the unique rows match but duplicate counts differ, we classify it as `row_dedup`, typically caused by `DISTINCT` or `GROUP BY`. For differences in row existence, we consider five cases: `row_subset` when the predicted result is missing some rows, `row_superset` when extra rows are present, `row_emptied` when the predicted result is empty but gold is not, `row_created` when the predicted result has rows but gold is empty, and `row_disjoint` when the two results have no intersection. These typically originate from `WHERE`, `JOIN`, or `HAVING` clauses.

Value-Level Differences. Value-level differences occur when the row structure aligns but specific cell values differ. We classify this as `row_partial`, which typically indicates errors in aggregate functions or arithmetic expressions in the `SELECT` clause.

Table 12 provides the full mapping between diff types and potential erroneous clauses.

Why Each Type Matters. Each diff type signals the *direction* of the error. `row_subset` (missing rows) indicates an over-restrictive clause, while `row_superset` (extra rows) indicates an under-restrictive one. These map to different candidate clauses: `row_subset` → {`WHERE`, `JOIN`, `HAVING`, `LIMIT`}, `row_superset` → {`JOIN`, `WHERE`}. Merging them loses this direction. This asymmetry is consistent with prior work on why vs. why-not provenance (Chapman and Jagadish, 2009; Huang et al., 2008), SQL mutation testing (Chandra et al., 2015), and Text-to-SQL evaluation (Zhong et al., 2020).

A.2 Clause-wise Incremental Execution Algorithm

Algorithm 1 describes the procedure for identifying C_{err} in the incorrect result case with clause-wise incremental execution. The algorithm consists of three phases: preprocessing, incremental execution, and erroneous clause identification.

Preprocessing. We first replace the `SELECT` clause with `SELECT *` to isolate the effect of each clause during incremental execution. When the query contains `GROUP BY`, `SELECT *` violates SQL constraints, so we process `SELECT` and `GROUP BY` together as a single unit. We then reorder the clauses according to the logical execution order.

Level	Diff Type	Natural Language Description	Potential Erroneous Clauses
Column	`col_count`	The number of columns in the output schema is incorrect.	`SELECT`, `FROM`, `JOIN`
	`col_name`	Column count is correct, but the wrong attributes were projected.	`SELECT`, `FROM`, `JOIN`
Row	`row_order`	Rows are correct, but the sorting sequence is misaligned.	`ORDER BY`
	`row_dedup`	Unique rows match, but the use of `DISTINCT` or grouping is incorrect.	`SELECT (DISTINCT)`, `GROUP BY`
	`row_subset`	The result is missing rows due to overly restrictive conditions.	`WHERE`, `JOIN`, `HAVING`, `LIMIT`
	`row_superset`	Extra rows are generated (e.g., unintended Cartesian products).	`JOIN`, `WHERE` (OR/IN)
	`row_emptied`	The query returns no data, while the gold result is non-empty.	`WHERE`, `JOIN` (No match), `HAVING`
	`row_created`	Rows are returned for a query that should yield an empty result.	`FROM`, `JOIN`
	`row_disjoint`	Predicted and gold rows have zero intersection (completely wrong data).	`FROM`, `JOIN`, `WHERE`
Value	`row_partial`	Structure aligns, but calculated values (aggregates/math) are wrong.	`SELECT` (Aggregates, Expressions)

Table 12: Comprehensive taxonomy of SQL result discrepancies. This classification assumes an alias-free environment and maps each diff type to its structural or logical origins in SQL clauses. Our incremental execution strategy resolves the 1:N ambiguity between diff types and clauses by capturing the exact moment a discrepancy first manifests.

Axis	Diff Types
Schema	`col_count`, `col_name`
Content (bag)	`row_subset`, `row_superset`, `row_disjoint`, `row_emptied`, `row_created`, `row_dedup`
Ordering	`row_order`
Value	`row_partial`

Table 13: Mapping of SQL result axes to diff types. Schema differences follow from the relational model, bag-content differences enumerate all pointwise multiplicity relations (Green et al., 2007), ordering is separate when `ORDER BY` is present (Zhong et al., 2020), and value-level differences cover cell-wise mismatches.

Incremental Execution. We execute clauses incrementally following the logical execution order: `FROM/JOIN` $\rightarrow$ `WHERE` $\rightarrow$ `GROUP BY` $\rightarrow$ `HAVING` $\rightarrow$ `SELECT` $\rightarrow$ `ORDER BY` $\rightarrow$ `LIMIT`. For the first clause (`FROM/JOIN`), there is no previous result to compare, so we compute $d_1 = \text{diff}(R_{gold}, R_1)$ by comparing directly with R_{gold}. For subsequent clauses ($i \geq 2$), we compute $d_i = \text{diff}(R_{i-1}, R_i)$ by comparing the results before and after adding each clause.

Erroneous Clause Identification. After obtaining all d_i through incremental execution, we compute the final difference $D^* = \text{diff}(R_{gold}, R_{pred})$. A clause c_i' is identified as erroneous and added to C_{err} if its corresponding d_i appears in D^*, meaning the change introduced by c_i' contributed to the final difference.

A.3 SQLite Execution Error Taxonomy

When an execution error occurs, our system parses the error string to identify the faulty components. Table 14 categorizes common SQLite error messages and defines how they serve as triggers for the clause-level tracing mechanism described in Section 3.2.3.

The error messages listed in Table 14 act as diagnostic signals for identifying C_{err}. The system employs different strategies based on the error category:

Schema and Constraint Tracing. For schema reference errors, the system extracts the identifier (e.g., `age` from `no such column: age`) and performs a clause-level tracing through the SQL clauses to identify where this object was omitted or incorrectly joined. Similarly, constraint failures prompt an analysis of the `JOIN` or `WHERE` conditions that might have introduced invalid data.

Logical and Syntax Errors. For logical misuse, the specific function in the message serves as an anchor to flag the clause that attempted the illegal operation. For syntax errors, the `"near [token]"` hint isolates the physical location of the error clause.

Category	Common SQLite Error Patterns	Detailed Description

Schema Reference	`no such column: [col_name]`
	`no such table: [table_name]`
	`ambiguous column name: [col_name]`
		Occurs when the query attempts to access an object not present in the database or the current subquery context. These typically indicate a failure in the data retrieval stage (`FROM`/`JOIN`).

Logical Misuse	`misuse of aggregate function [func]()`
	`aggregate functions are not allowed in...`
	`misuse of aliased identifier`
		Triggered when an operation violates the semantic rules of SQL execution order, such as using an aggregate function in a prohibited clause (e.g., `WHERE`).

Data & Constraint	`datatype mismatch`
	`NOT NULL constraint failed: [col]`
	`UNIQUE constraint failed: [col]`
		Indicates that the query structure is correct, but the operation violates data integrity or involves incompatible data types during calculation.

Syntax Error	`near "[token]": syntax error`
	`unrecognized token: "[char]"`
	`incomplete input`
		Occurs when the SQL engine cannot parse the query string due to grammatical failures, preventing the formation of a logical execution plan.

Table 14: Classification of common SQLite execution errors. These messages are utilized to extract key identifiers (e.g., column/table names) as initial triggers for the clause-level tracing process.

Structure	Attribution Strategy
Nested sub-query	Handled as part of its enclosing clause.
Set operations (UNION / INTERSECT / EXCEPT)	Split at set operation boundaries; each sub-query is analyzed independently.
Window function	Treated as part of `SELECT`; errors appear as `row_partial` at the `SELECT` step.
Alias	Resolved to source tables/columns before incremental execution.
CTE	Identified as erroneous when the referencing `FROM`/`JOIN` step produces a diff; internal decomposition is skipped.

Table 15: Attribution strategies for complex SQL structures.

Clause-level Tracing. A `"no such column"` error is interpreted as a failure in the data flow. The tracing logic investigates whether the column was lost during a join operation or if the model failed to account for a table alias, allowing us to pinpoint the root-cause clause even if the error was caught later in the execution sequence (e.g., in `ORDER BY`).

A.4 Handling Complex SQL Structures

For complex SQL structures, we attribute errors at the granularity of the enclosing clause, as summarized in Table 15. Even without decomposing internal structures, this still distinguishes correct and erroneous clauses: the 5.6%p improvement on Challenging queries (Table 7) confirms its effectiveness.

B Additional Details About Experimental Setups

B.1 Datasets

Training. SynSQL-Complex-5K is a subset of SynSQL-2.5M (Li et al., 2025b), containing 5K complex NL-SQL pairs. The queries involve multiple joins, nested subqueries, and aggregations.

Evaluation. Spider (Yu et al., 2018) contains 10,181 question-SQL pairs across 200 databases covering 138 domains (dev: 1,034, test: 2,147). BIRD (Li et al., 2023) contains 12,751 pairs from 95 large-scale databases across 37 specialized domains (dev: 1,534). For robustness evaluation, we use Spider-DK (Gan et al., 2021b) which requires domain knowledge, Spider-Syn (Gan et al., 2021a) which replaces schema words with synonyms, and Spider-Realistic (Deng et al., 2021) which removes explicit schema mentions.

B.2 Baselines

SFT-based methods fine-tune models on text-SQL pairs, including SENSE (Yang et al., 2024), DTS-SQL (Pourreza and Rafiei, 2024) which decomposes the task into subtasks, and OmniSQL (Li et al., 2025b) which synthesizes chain-of-thought reasoning data. Prompting-based methods leverage LLMs without training, including DAIL-SQL (Gao et al., 2023) with in-context learning, MCS-SQL (Lee et al., 2025a) with execution-based selection, and CHASE-SQL (Pourreza et al., 2025a) with multi-agent refinement. RL-based methods optimize with execution feedback, including SQL-R1 (Ma et al., 2025), ReasoningSQL (Pourreza

Clause	Early (Ep 1–2)	Late (Ep 5)	Change
FROM	90.1%	82.1%	−8.0 pp
SELECT	72.7%	64.7%	−8.0 pp
JOIN	58.9%	67.4%	+8.5 pp
GROUP BY	47.9%	39.7%	−8.2 pp
WHERE	32.9%	26.2%	−6.7 pp
ORDER BY	15.9%	14.7%	−1.2 pp
LIMIT	8.3%	6.0%	−2.3 pp

Table 16: Causal error rate per clause type during training on SynSQL-Complex-5K. Values represent the fraction of mismatch cases in which each clause appears in C_{err}.

et al., 2025b) with partial rewards, and Arctic-SQL-R1 (Yao et al., 2025).

B.3 Implementation Details

We implement EXPO-SQL using the Ray + vLLM framework (Hu et al., 2024) for efficient inference, and extend the RL++ framework(Hu, 2025) to support clause-level reward propagation. We train for 5 epochs with batch size 64, learning rate 1e-6, and temperature 0.8 on 8 NVIDIA H100 80GB GPUs. The KL penalty coefficient β is set to 0.01. We use AdamW optimizer with cosine annealing scheduler and warmup ratio 0.1. For inference, we follow SQL-R1 (Ma et al., 2025) and use self-consistency with 8 samples. All experiments are conducted with Python 3.10 and CUDA 12.0+.

C Additional Experimental Results

C.1 Per-Clause Error Rate Dynamics

Since each incorrect-result case produces a set of causal clauses C_{err}, we can track the *causal error rate* per clause type, the fraction of mismatch cases in which a given clause appears in C_{err}, as a diagnostic view of how error distribution shifts during training on the SynSQL-Complex-5K training set. Table 16 compares the rates between the early (Epoch 1–2) and late (Epoch 5) training stages.

Most clauses show consistent reduction in their causal rate, with FROM, SELECT, and GROUP BY each dropping by around 8 pp. JOIN increases (+8.5 pp) not because JOIN performance worsens, but because as the model resolves earlier syntax errors (the unexecutable rate drops from 26.8% to 7.5%), previously failing queries become executable and expose underlying JOIN errors that were previously masked. Such diagnostic signals, unavailable under query-level rewards, reveal that easy errors are resolved first while harder cases persist, providing insight into how execution-based RL training progresses.

C.2 Case Study: Clause-wise Incremental Execution Example

Figure 5 and Figure 6 illustrate how clause-wise incremental execution identifies erroneous clauses when the query produces an incorrect result. By incrementally executing each clause and comparing intermediate results, we can trace which clause introduced the difference observed in the final result.

C.3 Case Study: Clause-level Tracing for Execution Errors

Figure 7 and Figure 8 demonstrate how clause-level tracing identifies erroneous clauses from database error messages. Figure 7 shows an ambiguous column error case, and Figure 8 shows a column not found error case.

Example for incorrect result

<table>
<tr><th>Question</th><th>Gold Result</th><th>Diiff_type</th></tr>
<tr><td>Who is the most recently hired coach for the Chicago Bears?</td><td>coach_came
Jane Smith</td><td rowspan="4">Column Count, Row Disjoint</td></tr>
<tr><th>Predicted SQL Query</th><th>Predicted Result</th></tr>
<tr><td><code>SELECT c.coach_name, c.hire_date FROM coaches c
JOIN teams t ON c.team_id = t.team_id
WHERE t.team_name = ‘Chicago Bears’
ORDER BY c.hire_date DESC
LIMIT 1</code></td><td>coach_name | hire_date
Jane Smith | 2020-02-01</td></tr>
</table>

Clause-wise Incremental Execution

<table>
<tr><th>Execution Order</th><th>SQL Query</th><th>Execution Result</th><th>Diff_type</th></tr>
<tr><td>FROM/JOIN</td><td><code>SELECT * FROM coaches c
JOIN teams t ON c.team_id = t.team_id</code></td><td>coach_name | Team_name | hire_date
John Doe | Packers | 2021-01-01
Jane Smith | Chicago Bears | 2020-02-01</td><td rowspan="2">Row Subset (by WHERE)</td></tr>
<tr><td>+ WHERE</td><td><code>SELECT * FROM coaches c
JOIN teams t ON c.team_id = t.team_id
WHERE t.team_name = ‘Chicago Bears’</code></td><td>coach_name | Team_name | hire_date
John Doe | Packers | 2021-01-01
Jane Smith | Chicago Bears | 2020-02-01</td></tr>
<tr><td>+ SELECT</td><td><code>SELECT c.coach_name, c.hire_date
FROM coaches c
JOIN teams t ON c.team_id = t.team_id
WHERE t.team_name = ‘Chicago Bears’</code></td><td>coach_name | hire_date
Jane Smith | 2020-02-01</td><td>Column Count, Row Disjoint (by SELECT)</td></tr>
<tr><td>+ ORDER BY</td><td><code>SELECT c.coach_name, c.hire_date
FROM coaches c
JOIN teams t ON c.team_id = t.team_id
WHERE t.team_name = ‘Chicago Bears’
ORDER BY c.hire_date DESC</code></td><td>coach_name | hire_date
Jane Smith | 2020-02-01</td><td>No Diff</td></tr>
<tr><td>+ LIMIT</td><td><code>SELECT c.coach_name, c.hire_date FROM coaches c
JOIN teams t ON c.team_id = t.team_id
WHERE t.team_name = ‘Chicago Bears’
ORDER BY c.hire_date DESC
LIMIT 1</code></td><td>coach_name | hire_date
Jane Smith | 2020-02-01</td><td>No Diff</td></tr>
<tr><td>Clause-level Reward</td><td colspan="3">{SELECT} : − 0.5
Others : + 0.5</td></tr>
</table>

Figure 5: Case study of clause-wise incremental execution - (1)

Example for incorrect result

Question	Gold Result	Diiff_type
Could you tell me the name of the geological sequence with the highest total composition where the microbial composition is greater than 60% or the oolitic composition is greater than 35%?	sequence_name Late early Cambrian	Row Emptied
Predicted SQL Query	**Predicted Result**	
SELECT g.sequence_name FROM geological_periods g JOIN biological_composition b ON g.period_id = b.period_id WHERE b.microbial > 60 OR b.oolitic > 35 ORDER BY b.total DESC LIMIT 1	sequence_name (empty)	

Clause-wise Incremental Execution

Execution Order	SQL Query	Execution Result	Diff_type
FROM/JOIN	SELECT * **FROM geological_periods g** **JOIN biological_composition b ON** **g.period_id = b.period_id**	sequence_name, ..., total Late early Cambrian, ..., 0.997531 Mid Cambrian, ..., 1.0	Row Emptied (by WHERE)
+ WHERE	SELECT * FROM geological_periods g JOIN biological_composition b ON g.period_id = b.period_id **WHERE b.microbial > 60 OR b.oolitic > 35**	sequence_name, ..., total (empty)	Column Count (by SELECT)
+ SELECT	**SELECT g.sequence_name** FROM geological_periods g JOIN biological_composition b ON g.period_id = b.period_id WHERE b.microbial > 60 OR b.oolitic > 35	sequence_name (empty)	No Diff
+ ORDER BY	SELECT g.sequence_name FROM geological_periods g JOIN biological_composition b ON g.period_id = b.period_id WHERE b.microbial > 60 OR b.oolitic > 35 **ORDER BY b.total DESC**	sequence_name (empty)	No Diff
+ LIMIT	SELECT g.sequence_name FROM geological_periods g JOIN biological_composition b ON g.period_id = b.period_id WHERE b.microbial > 60 OR b.oolitic > 35 ORDER BY b.total DESC **LIMIT 1**	sequence_name (empty)	
Clause-level Reward	{WHERE} : − 0.5 Others : + 0.5		

Figure 6: Case study of clause-wise incremental execution- (2)

Example for execution error: *ambiguous column name*	
Question	I need to find the top 3 access points with the most log entries.
Predicted SQL Query	SELECT ap_name, COUNT(log_id) AS total_log_entries FROM access_points **JOIN** log_entries ON **access_points.ap_id = log_entries.ap_id** **GROUP BY** **ap_id** ORDER BY total_log_entries DESC LIMIT 3 ;
Execution Error Message	*<Error Msg> Ambiguous column:* ***ap_id***
Clause-level Reward	{ **JOIN , GROUP BY** } : -1.0 Others : -0.5

Figure 7: Case study of clause-level tracing: ambiguous column error

Example for execution error: *no such column*	
Question	Can you please show me the names of top 5 compressed images and the users who upload them, based on the highest compression ratio?
Predicted SQL Query	SELECT i.image_name, i.user_name FROM images i **INNER JOIN** compression_results cr ON i.image_id = **cr.image_id** ORDER BY cr.compression_ratio DESC LIMIT 5 ;
Execution Error Message	*<Error Msg> No such column:* ***cr.image_id***
Clause-level Reward	{ **INNER JOIN** } : -1.0 Others : -0.5

Figure 8: Case study of clause-level tracing: column not found error