CORES: Code-Oriented Reasoning for Complex Text-to-SQL and Generalizable TableQA

Meng Zhang[1], **Ruochun Jin**[1], **Yuanxi Peng**[1], **Wenjing Yang**[1], **Haotian Wang**[1], **Liting Sun**[1], **Kun Hu**[1], **Silin Yang**[2], **Kedi Zhang**[1*]

[1]College of Computer Science and Technology,
National University of Defense Technology, Changsha, China
[2]School of Computer Science, Peking University, Beijing, China
{zhangmeng18, jinrc, zhangkedi10}@nudt.edu.cn

Abstract

Text-to-SQL aims to bridge the gap between human intent and relational databases. While LLMs have shown proficiency in generating simple SQL queries, they struggle with complex analytical tasks. Moreover, models fine-tuned on SQL generation often suffer from catastrophic forgetting, which lose the versatility of procedural reasoning and pertaining to generation constraints. Inspired by the usage of high-resource programming languages as LLM reasoning intermediaries, we propose **CORES** model, which leverages Python as a procedural reasoning pivot to enhance both complex SQL generation and tabular reasoning. It decomposes complex queries into Python reasoning traces before generating the final SQL, which bridges the gap between procedural reasoning and declarative expression. In order to internalize this reasoning capability, we fine-tune LLMs via GRPO with tailored process reward functions that mitigate the sparse feedback problem. We experimentally verify the effectiveness of CORES on six text-to-SQL benchmarks, where ours outperforms baselines by 6.44% on average, while maintains good capability on three tableQA benchmarks.

1 Introduction

Text-to-SQL democratizes data access which enables users to query relational databases via natural language, and large language models (LLMs) have achieved remarkable performance on benchmarks (Qin et al., 2022; Hong et al., 2024). As a financial technology company, our analysts primarily execute SQL queries on databases for daily transactions such as business intelligence and financial analysis (Zhang et al., 2024a; Kumar et al., 2024), while we also need to generate insights from semi-structured tables for monthly audit reports. This workflow demands complex reasoning over

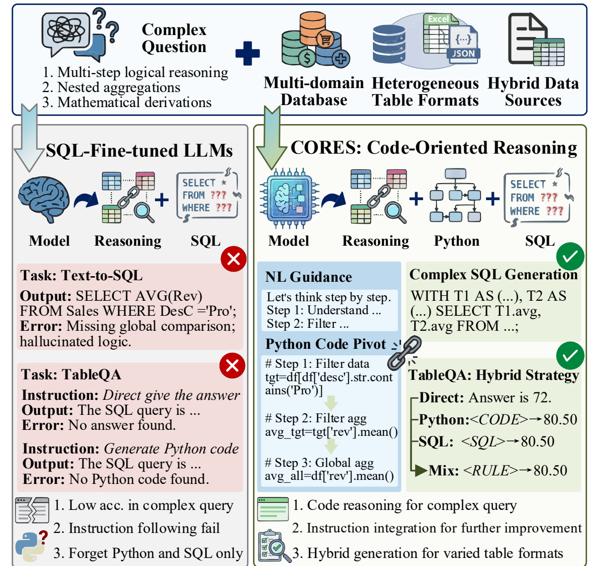

Figure 1: Comparison between current SQL-fine-tuned LLMs, which suffer from catastrophic forgetting, and our CORES model, which excels in complex SQL generation and generalizes to tableQA tasks.

structured data, involving multi-step logical reasoning, nested aggregations, and mathematical derivations (Liu et al., 2025a; Zheng et al., 2024), as well as the generalization of these capabilities to semi-structured tables. Moreover, strict data privacy regulations and on-site auditing requires the local deployment of lightweight models. Unfortunately, existing specialized text-to-SQL models fall short of satisfying these practical demands, presenting challenges in two critical aspects.

First, state-of-the-art (SOTA) models that are extensively trained on SQL corpus suffer from catastrophic forgetting and instruction drift, which significantly limits their practical applicability. Our empirical analysis reveals that these SQL-fine-tuned models such as OmniSQL-7B (Li et al., 2025), SQL-R1-7B (Ma et al., 2025), and Arctic-Text2SQL-R1-7B (Yao et al., 2025) completely lose their ability to generate Python code or pro-

*The corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 4827–4847
July 2-7, 2026 ©2026 Association for Computational Linguistics

vide direct answers, with accuracy dropping to zero even when given explicit instructions. These models unfortunately become “SQL-only” systems that rigidly output thought of constructing SQL queries, which renders them incapable of adapting to table question answering (TableQA) tasks, where the direct answering and Python-based solutions often outperform SQL (Zhang et al., 2025b). *Can we develop a model that achieves expert-level proficiency in text-to-SQL while preserving the generalization capacity to tableQA tasks?*

Second, complex reasoning demands procedural logic, which presents a fundamental mismatch with the declarative nature of SQL. Specifically, the user question and SQL specifies what to retrieve rather than how to compute it, which inherently prevents explicit representation of intermediate reasoning steps (Date, 2011). Although existing methods attempt to bridge this gap through natural language chain-of-thought prompting (Wei et al., 2022; Li et al., 2025; Ma et al., 2025), the non-executable nature and unstructured expressions of these reasoning steps often leads to inconsistency of the final SQL (Madaan et al., 2022; Gao et al., 2023). In contrast, recent studies have explored using programming languages as the intermediate representation to improve LLM reasoning, as executable code provides a precise structure and verifiability that natural language lacks (Chen et al., 2022; Gao et al., 2023; Chi et al., 2025). *Can we integrate code strengths into the text-to-SQL model that enhances the complex reasoning capacity?*

In view of these challenges, we propose the **C**ode-**O**riented **RE**asoning for **S**QL (**CORES**) model, which leverages Python as a reasoning pivot to improve complex SQL generation and tableQA tasks generalization[1]. As shown in Figure 1, first, the Code-Oriented Prompting for SQL (CoPS) strategy decomposes complex queries into Python reasoning steps before generating the final SQL. Then, the LLMs are fine-tuned using Group Relative Policy Optimization (GRPO) with a process reward function, which evaluates both the intermediate Python reasoning and the SQL result. This reward acts as a dense supervision signal that mitigates the sparsity of outcome rewards while preserving the model’s instruction-following abilities. Moreover, ProcQA benchmark with 12k complex queries with aligned Python-SQL pairs is constructed to support training and evaluation. Our framework maintains the general coding skills of the base model while improving its specific SQL capabilities, thus obtaining generalization to tableQA by employing diverse generation strategies such as direct answering, Python-based and SQL-based solution.

Experimental results across nine multi-domain benchmarks show that CORES sets a new SOTA. Our 7B and 3B models achieve average execution accuracies of 59.89% and 49.91% in text-to-SQL benchmarks, which surpasses leading baselines by 6.44%. CORES resolves the overfitting issue in hybrid tableQA tasks where previous models often fail. Moreover, we recover Python generation accuracy from zero to 51.40%, thus the hybrid strategy combining SQL, Python, and direct answering significantly outperforms rigid SQL-only methods.

Our main contributions are as follows.

- We propose CORES model that integrates the procedural reasoning of Python steps into text-to-SQL, which enhances the complex reasoning for the declarative SQL generation.
- We design a process reward function for GRPO training, which provides dense supervision signals for intermediate reasoning steps.
- We curate ProcQA benchmark and conduct experiments across nine datasets, which shows CORES outperforms SOTA in complex text-to-SQL and maintains tableQA capabilities.

2 Related Work

LLM-based Text-to-SQL Text-to-SQL research has shifted towards LLM-based in-context learning with advanced prompting strategies (Pourreza and Rafiei, 2024; Gao et al., 2024; Talaei et al., 2024; Zhang et al., 2025a) and supervised fine-tuning with large-scale text-to-SQL training corpus (Li et al., 2024a, 2025; Papicchio et al., 2025; Ma et al., 2025; Yao et al., 2025). Recently, PI-SQL (Chi et al., 2025) demonstrated that using Python as an intermediate pivot effectively enhance the procedural reasoning. However, it relies on an external inference pipeline with high inference-time costs rather than intrinsic model capabilities. In contrast, CORES addresses this by internalizing code-oriented reasoning via reinforcement learning, thereby enabling strong performance in end-to-end generation while allowing for further enhancement when integrated with the PI-SQL framework.

[1]Our code can be available at: https://github.com/mengzhang18/CORES.

LLM-based TableQA TableQA extends beyond SQL generation and requires models to directly reason over diverse semi-structured tables (Chen et al., 2020; Jin et al., 2025). Numerous models have achieved impressive performance on tableQA benchmarks (Cheng et al., 2022; Wu et al., 2025) by fine-tuning on specialized datasets (Zhang et al., 2024b; Yang et al., 2025) or employing tool-augmented strategies such as Python and SQL (Gao et al., 2023; Chen et al., 2022). However, these models frequently lack strong ability for complex text-to-SQL (Lei et al., 2025). Worse yet, we observe that SOTA SQL-fine-tuned models typically exhibit limited generalization when extended to tableQA tasks. Different from these models, CORES focuses on mastering complex text-to-SQL reasoning while simultaneously demonstrating strong generalization capabilities in tableQA.

LLM Reasoning Enhancement Recent studies have employed programming languages and reinforcement learning to enhance the reasoning capabilities of LLMs (Gao et al., 2023; Sui et al., 2025). Program-aided paradigms leverage code as a rigorous reasoning scaffold to decompose complex problems into procedural steps (Chen et al., 2022; Gao et al., 2023). Concurrently, reinforcement learning algorithms, such as GRPO, have proven effective in enhance model reasoning through supervised rewards (Shao et al., 2024; Wang et al., 2024). These methodologies have been individually explored to address reward sparsity and reasoning gaps in text-to-SQL (Ma et al., 2025; Yao et al., 2025) and tableQA domains(Yang et al., 2025; Lei et al., 2025). In this work, we integrate these two paradigms by integrating the Python code into the GRPO training loop, thereby internalizing code-oriented reasoning capabilities into the model.

3 Preliminaries

3.1 Task Formulation

Text-to-SQL Given a natural language question $Q = (q_1, ..., q_{|Q|})$ and a database schema $S = \langle T, C \rangle$ with tables $T = \{t_1, ..., t_{|T|}\}$ and columns $C = \{c_1, ..., c_{|C|}\}$, the goal of text-to-SQL is to generate a SQL query y that correctly executes.

TableQA Given a natural language question $Q = (q_1, ..., q_{|Q|})$ and a table $T = \langle H, R \rangle$ consisting of column headers $H = \{h_1, ..., h_{|H|}\}$ and data rows $R = \{r_1, ..., r_{|R|}\}$ where each row $r_i = (v_{i,1}, ..., v_{i,|H|})$ contains cell values corresponding to the headers, the goal of tableQA is to give a correct answer a to Q based on the table content.

3.2 Reinforcement Learning Algorithm

We utilize GRPO for reinforcement learning, whose core innovation lies in the group-wise relative assessment approach (Shao et al., 2024). Specifically, for each natural language prompt, the policy generates a group of G candidate SQL queries $\{o_1, o_2, \dots, o_G\}$. Each candidate is evaluated by a composite reward function, and advantages are computed relative to the group through normalization $A_i = (r_i - \mu)/\sigma$, where μ and σ are the mean and standard deviation of rewards within the group. This relative comparison provides more stable training signals than absolute reward values.

The policy optimization follows the GRPO objective function:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}\Bigg[\frac{1}{G}\sum_{i=1}^{G} \min\Big(r_i^{\text{ratio}} A_i, \text{clip}\Big(r_i^{\text{ratio}}, 1-\epsilon, 1+\epsilon\Big)A_i\Big) - \beta D_{\text{KL}}\left(\pi_\theta \| \pi_{\text{ref}}\right)\Bigg],$$

where $r_i^{\text{ratio}} = \pi_\theta(o_i \mid \mathbf{v})/\pi_{\theta_{\text{old}}}(o_i \mid \mathbf{v})$ is the importance sampling ratio, ϵ controls the clipping range for update stability, and the KL divergence term with coefficient β prevents excessive deviation from the reference policy π_{ref}.

4 Code-Oriented Reasoning for SQL

CORES internalizes procedural reasoning into LLMs for complex text-to-SQL generation. As illustrated in Figure 2, our approach consists of four components. First, the inference paradigm adopts code-oriented prompting for SQL which decomposes complex user queries into intermediate Python reasoning steps before the final SQL generation. Second, the training phase is supported by ProcQA, a dedicated dataset which aligns these SQL queries with verified and executable Python logic to match the proposed paradigm. Finally, the algorithmic optimization utilizes GRPO alongside a process-supervised reward function which provides dense feedback across formatting, schema grounding, logical structure, and execution correctness. Beyond the end-to-end generation of the model, we present advanced inference strategies for both text-to-SQL and tableQA tasks.

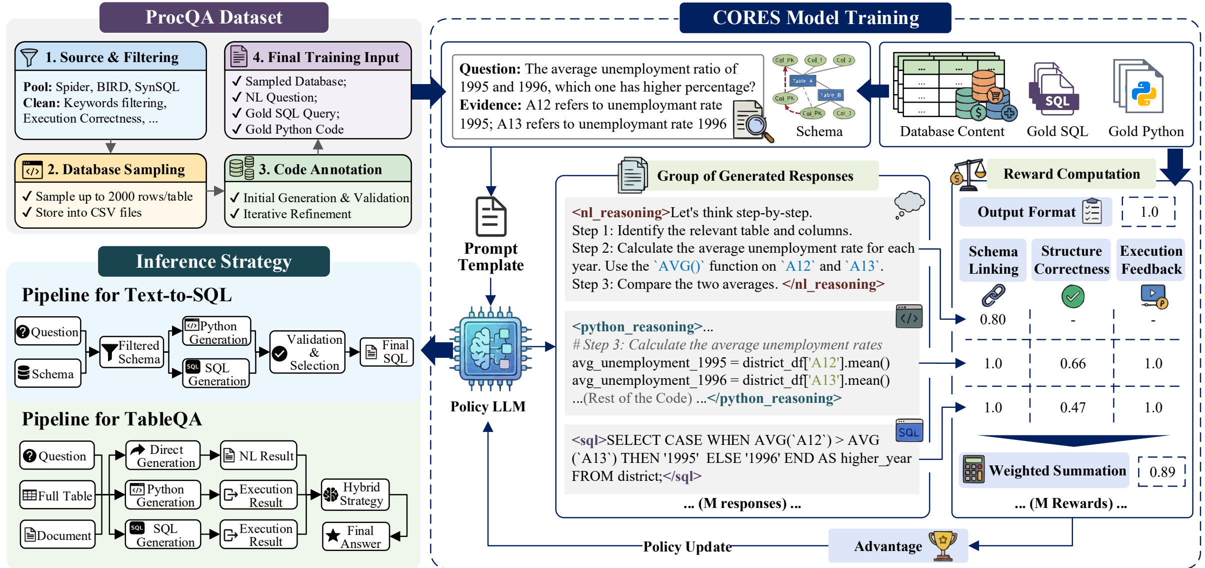

Figure 2: Overview of the CORES framework. The training phase utilizes the constructed ProcQA dataset and the prompt template to decompose complex queries into intermediate Python reasoning. The model is optimized via GRPO with a dense process-supervised reward function encompassing format, schema, structure, and execution signals. During inference, flexible pipelines are employed for text-to-SQL and tableQA tasks.

4.1 Code-Oriented Prompting for SQL

We propose Code-Oriented Prompting for SQL (CoPS), which integrates semantic understanding, procedural reasoning, and declarative expression in a single response. Specifically, the output comprises three generation parts: natural language (NL) reasoning which outlines the high-level analytical plan, Python code reasoning which implements this plan via explicit procedural steps, and a final SQL query which distills the preceding logic into a declarative expression. In this prompt, the high-resource programming language as intermediate representation enhances the performance of the SQL generation (Chi et al., 2025).

The input to CoPS comprises the user question, external evidence (Li et al., 2024b), and a Data Definition Language (DDL) schema representation. During training, the schema is filtered based on the gold SQL and augmented with a controlled set of irrelevant schema items to maintain realistic schema-linking difficulty, which reduces input token consumption while allowing the model concentrates logical reasoning. During inference, we provide the complete database schema. Moreover, we construct a BM25 index over all textual values in the database, retrieve the top-2 values most relevant to the question, and add them to the schema as content examples(Li et al., 2025). Complete prompts are provided in Appendix F.

4.2 ProcQA Construction

As high-quality training data is the cornerstone of model performance, thus, we introduce ProcQA, a benchmark to address the scarcity of aligned complex procedural reasoning in text-to-SQL. Table 1 shows the statistics of ProcQA, which is derived from Spider (Yu et al., 2018), BIRD (Li et al., 2024b), and SynSQL (Li et al., 2025) to provide diverse schema structures and query requirements. From this initial pool, we conduct data filtering, database sampling, and code annotation, whose details are illustrated in Appendix A. Finally, our ProcQA contains 11,879 samples for training, where each sample comprises a question, a sampled database file, a gold SQL query, and the corresponding gold Python code.

Source	# Query			# Database	
	Origin	Complex	Final	Origin	Final
Spider	7,000	3,500	2,059	140	132
BIRD	9,428	4,200	2,566	69	49
SynSQL	36,227	15,000	7,254	398	265
Total	52,655	22,700	**11,879**	607	446

Table 1: Statistics of the ProcQA dataset collection.

4.3 Process-Supervised Reward Design

As shown in Figure 2, a dense and rule-based reward function is designed to decompose supervision across format, schema linking, structure, and

execution correctness, which addresses the sparsity of binary execution signals while avoiding the substantial annotation effort and computational resources required for training a separate process reward model. The total reward R is a weighted aggregation of four components:

$$R = R_{\text{f}} \cdot (R_{\text{schema}} + R_{\text{struct}} + 2R_{\text{exec}})$$

Format Reward (R_{f}) It enforces adherence to the output format requirement. Specifically, R_{f} is set to 1 only if the three output components are strictly enclosed in the `<nl_reasoning>`, `<python_reasoning>`, and `<sql>` tags. Otherwise, R_{f} is set to 0 and R returns 0.

Schema Linking Reward (R_{schema}) Formulated as a weighted summation of F1-scores across NL, Python, and SQL, this reward evaluates whether the model correctly identifies relevant database entities across all reasoning parts, which prevents schema hallucination and enforces consistency:

$$R_{\text{schema}} = \sum_{m \in \{\text{nl, py, sql}\}} \omega_m \cdot \frac{2 \cdot |E^m_{\text{pred}} \cap E_{\text{gold}}|}{|E^m_{\text{pred}}| + |E_{\text{gold}}|}$$

where $\omega_{\text{nl}} = 0.5$, $\omega_{\text{py}} = \omega_{\text{sql}} = 1.0$, E_{gold} denotes the gold SQL schema items, and E^m_{pred} represents entities extracted from the model's output at generation part m. Specifically, for NL, we extract schema mentions against the database schema to identify referenced tables and columns. For Python, we extract table names from `read_csv` path arguments and column names from DataFrame[2] access patterns such as bracket indexing `df['col']`. For SQL, we parse table references and column identifiers using SQL parsing tools[3].

Structure Reward (R_{struct}) As shown in Figure 3, this reward evaluates structural correctness by capturing the proper sequence of algorithmic operations independent of variable names, which addresses cases where schema linking succeeds but the structure remains flawed. We formulate it as the summation of the arithmetic means of the sequence similarity metric and the operation-based F1-score for Python and SQL:

$$R_{\text{struct}} = \sum_{m \in \{\text{py,sql}\}} \frac{1}{2} \left(\text{Sim}(S^m_p, S^m_g) + \text{F1}(O^m_p, O^m_g) \right)$$

[2] https://pypi.org/project/pandas/
[3] https://pypi.org/project/sql-metadata/

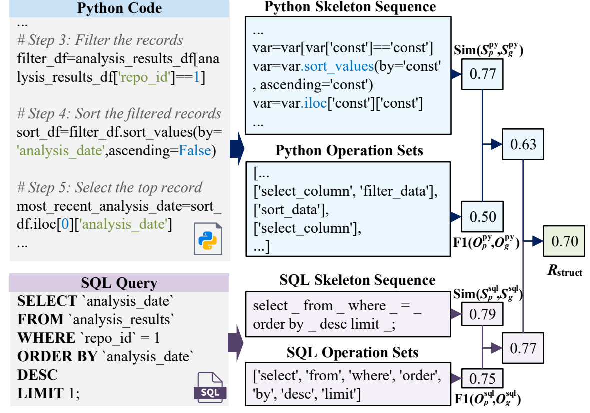

Figure 3: An example of structure reward computation.

where S^m_p and S^m_g represent the structural skeletons of the generated and gold code, and O^m_p and O^m_g denote the corresponding sets of operations.

For Python, we derive the skeletons by replacing variable identifiers with `VAR` and constant values with `CONST`. The function $\text{Sim}(\cdot)$ computes the similarity score using the Gestalt pattern matching algorithm (Ratcliff et al., 1988), which recursively locates the longest contiguous matching subsequences between normalized code steps. Concurrently, we obtain the operation sets O^{py} by mapping code segments into skeleton such as `FILTER_DATA` and `SORT_DATA`. We then calculate the F1-score based on the overlap between the generated and gold operation sets. Regarding SQL, we construct query skeletons by removing schema names and literals. We apply the same $\text{Sim}(\cdot)$ function to capture global query patterns between the generated and gold skeletons. The operator score extracts specific SQL keywords, such as `GROUP BY` and `CASE WHEN`, and calculates the F1-score. This formulation incentivizes the model to replicate the structural complexity of gold solutions and prevents logically disordered queries that merely reference correct entities. We provide the details of Python mapping rules and SQL keywords in Appendix B.1.

Execution Reward (R_{exec}) This part provides the validation of both the Python code and the final SQL output through actual database queries. Since correctness can only be assessed when code executes successfully, we adopt a formulation as:

$$R_{\text{exec}} = \sum_{m \in \{\text{py,sql}\}} 0.25 \cdot R^m_{\text{ex}} + R^m_{\text{cor}}$$

where the $R^m_{\text{ex}} \in \{0, 1\}$ denotes execution success and $R^m_{\text{cor}} \in \{0, 1\}$ denotes the execution result

matches the gold reference. This formulation penalizes the Python code or SQL query that execute without runtime errors but yield incorrect outputs.

4.4 Inference Strategy

Text-to-SQL Pipeline Beyond directly generating SQL with CoPS generation mode, we integrate our model into the PI-SQL framework to achieve further improvements, which employs Python as an intermediate representation and has been proven more effective for lightweight LLMs than various advanced baselines (Chi et al., 2025). Therefore, this integration naturally leverages the strengths of CORES to maximize performance.

TableQA Pipeline Generalization capabilities on tableQA tasks are also necessary in practical scenarios. We employ three generation strategies for the tableQA task: directly generating the final answer, generating and executing Python code, and generating and executing a SQL query. Moreover, a hybrid strategy is proposed to select the optimal answer based on these outputs, which maximizes information utilization (Algorithm 1 with detailed analysis in Appendix C).

Algorithm 1: Best Answer Selection

Input : Model outputs $\mathcal{A} = [a_{\text{dir}}, a_{\text{py}}, a_{\text{sql}}]$
Output : Selected answer a^*

1. $\mathcal{L} \leftarrow \{x \in \mathcal{A} \mid x \neq \emptyset \wedge \neg \text{IsError}(x)\}$;
2. **if** $\mathcal{L} = \emptyset$ **then return** *None*;
3. $\mathcal{N} \leftarrow \{x \in \mathcal{L} \cap \{a_{\text{py}}, a_{\text{sql}}\} \mid \texttt{IsNumber}(x)\}$;
4. **if** $\mathcal{N} \neq \emptyset$ **then**
5.   **if** $|Unique(\mathcal{N})| = 1$ **then return** $\mathcal{N}[0]$;
6.   **if** `IsNumber` (a_{dir}) **then**
7. $v_{\text{ref}} \leftarrow \texttt{ExtractNum}(a_{\text{dir}})$;
8.   **return** $\arg\min_{n \in \mathcal{N}} |n - v_{ref}|$
9.   **return** $\mathcal{N}[0]$
10. **return** *MajorityVote($\mathcal{L}$)*

5 Experiments

5.1 Experimental Setup

Datasets we evaluate on six text-to-SQL benchmarks and three tableQA benchmarks (Appendix D.1). For text-to-SQL, we adopt (1) **ProcQA**, **LogicCat** (Liu et al., 2025a) and **Archer** (Zheng et al., 2024) for complex reasoning evaluation including arithmetic, hypothetical reasoning, and multi-step reasoning; (2) **BIRD** (Li et al., 2024b) (Dev set), **Spider** (Yu et al., 2018) (Dev and Test sets) and **Spider-Realistic** (Deng et al., 2021) for multi-domain databases and real-world queries evaluation. For tableQA, we use (1) **TableBench** (Wu et al., 2025) for queries with multiple reasoning types including numerical computation and fact checking; (2) **HiTab** (Cheng et al., 2022) for hierarchical table reasoning with complex structures; and (3) **FinQA** (Chen et al., 2021) for financial numerical reasoning over both documents and tables.

Evaluation Metric We use **Execution Accuracy (EX)** (Li et al., 2024b) for text-to-SQL tasks, where queries are correct only if Python/SQL execution results match the gold SQL. For tableQA tasks, we employ **Exact Match (EM) accuracy** (Lei et al., 2025), which requires generated answers to exactly match ground-truth after normalization.

Baselines We compare CORES against various models, categorized into three groups (Appendix D.2). (1) **Commercial LLMs**, including GPT-4o-mini evaluated under advanced prompting strategies of PI-SQL (Chi et al., 2025); (2) **Open-source pre-trained LLMs**, including Qwen2.5-Coder-3B/7B-Instruct (Hui et al., 2024), DeepSeek-R1-Distill-Qwen-7B (Shao et al., 2024), Meta-Llama-3.1-8B-Instruct (Dubey et al., 2024), OpenCoder-8B-Instruct (Huang et al., 2025), and Granite-3.1-8B-Instruct (Mishra et al., 2024). (3) **Open-source fine-tuned LLMs**, including SQL-R1-3B/7B (Ma et al., 2025), XiYanSQL-QwenCoder-3B (Liu et al., 2025b), CscSQL-Grpo-XiYanSQL-3B/7B (Sheng and Xu, 2025), Think2SQL-7B (Papicchio et al., 2025), RewardSQL-7B (Zhang et al., 2025c), OmniSQL-7B (Li et al., 2025), Arctic-Text2SQL-R1-7B (Yao et al., 2025).

Implementation Details We implement our experiments using the `verl` framework (Sheng et al., 2025) for efficient reinforcement learning. CORES-3B/7B models are fine-tuned on Qwen2.5-Coder-3B-Instruct and OmniSQL-7B, respectively. More details can be found in Appendix D.3.

5.2 Performance on Text-to-SQL Tasks

Table 2 demonstrates that CORES establishes a new SOTA across diverse benchmarks, particularly excelling in complex reasoning tasks while maintaining robust generalization capabilities. First, regarding complex reasoning, our model exhibits substantial improvements over base models. Specifically, on the ProcQA benchmark, CORES-3B outperforms Qwen2.5-Coder-3B-Instruct by 15.73% and 14.07% in SQL and Python accuracy, respectively, while CORES-7B further amplifies these gains, sur-

Model/Method	Complex Reasoning				General Evaluation				AVG
	ProcQA-SQL	ProcQA-Python	LogicCat	Archer	BIRD-Dev	Spider-Dev	Spider-Test	Spider-Realistic	
Commercial LLMs for reference									
GPT-4o-mini	40.93	33.47	21.67	18.27	56.26	78.72	78.95	78.35	50.83
GPT-4o-mini + PI-SQL	44.47	47.87	19.68	25.00	64.54	78.24	80.81	77.36	54.75
Open-source LLMs (~3B)									
Qwen2.5-Coder-3B-Instruct	23.07	13.00	12.07	6.73	36.57	61.51	61.95	52.17	33.38
SQL-R1-3B	35.80	0.00	12.93	8.65	47.00	71.95	73.68	65.55	39.45
XiYanSQL-QwenCoder-3B	29.13	12.40	15.49	10.58	44.59	73.02	72.75	64.96	40.37
CscSQL-Grpo-XiYanSQL-3B	33.87	9.00	15.11	**11.54**	52.22	78.24	77.13	71.65	43.60
CORES-3B	38.80	27.07	17.97	8.65	52.22	**80.85**	**82.25**	**76.97**	48.10
CORES-3B-PI	**40.60**	**33.27**	**18.44**	10.58	**59.19**	79.88	81.14	76.18	**49.91**
Open-source LLMs (~7B)									
DeepSeek-R1-Distill-Qwen-7B	13.53	1.27	7.79	5.77	19.30	46.91	46.81	41.14	22.82
OpenCoder-8B-Instruct	21.73	7.80	11.03	5.77	33.96	59.09	58.73	52.95	31.38
Granite-3.1-8B-Instruct	23.00	16.60	12.64	5.77	31.23	65.38	64.04	62.40	35.13
Meta-Llama-3.1-8B-Instruct	27.00	14.27	12.93	8.65	39.31	67.89	68.19	63.98	37.78
Qwen2.5-Coder-7B-Instruct	31.47	21.40	15.30	17.31	48.31	77.27	77.78	73.23	45.26
RewardSQL-7B	39.27	0.00	15.02	11.54	60.37	80.17	78.95	73.23	44.82
Think2SQL-7B	37.13	23.47	18.44	17.31	55.87	78.63	80.95	75.98	48.47
OmniSQL-7B	50.93	0.00	19.58	23.08	63.17	81.24	82.58	76.38	49.62
SQL-R1-7B	51.07	0.00	19.77	23.08	63.30	81.43	82.53	77.17	49.79
CscSQL-Grpo-XiYanSQL-7B	38.93	24.13	13.46	18.06	58.34	84.04	83.37	80.51	50.11
Arctic-Text2SQL-R1-7B	50.93	0.00	19.20	23.08	**68.58**	**89.75**	**89.47**	**86.61**	53.45
CORES-7B	51.27	40.67	**23.57**	21.15	66.04	85.98	86.77	83.66	57.39
CORES-7B-PI	**52.20**	**51.40**	23.48	**24.04**	68.25	87.81	86.49	85.43	**59.89**

Table 2: Main results on text-to-SQL benchmarks for complex reasoning and general evaluation, where CORES-3B/7B-PI refers to the integration of CORES in to PI-SQL framework.

passing its 7B counterpart by 19.80% and 19.27%. Second, CORES effectively preserves dual reasoning capabilities in both SQL and Python. While SQL-focused models such as OmniSQL, SQL-R1, and Arctic-Text2SQL-R1 achieve competitive execution accuracy in SQL generation, they completely lose generic coding skills, scoring zero on Python generation tasks. In contrast, CORES maintains the procedural reasoning capacity of Python to support an alternative to text-to-SQL for obtaining final execution results for user questions. Finally, the strong Python capability of CORES allows it to leverage the PI-SQL framework. Due to our model's aligned Python capabilities, integrating CORES into this framework yields further gains, increasing the average performance of the 3B and 7B variants by 2.08% and 2.50% respectively, ultimately achieving a state-of-the-art average score of 59.89% with CORES-7B-PI.

5.3 Performance on TableQA Tasks

As shown in Table 3, CORES demonstrates strong generalization across TableQA benchmarks by effectively leveraging diverse reasoning strategies. Our analysis reveals that the optimal generation strategies across direct answer, Python code, and SQL query, varies significantly depending on the tableQA tasks. Specifically, for TableBench, which demands arithmetic reasoning, solutions via Python and SQL significantly outperforms direct natural language thought. In contrast, HiTab involves hierarchical table structures where SQL queries often incur execution errors, whereas directly answering and Python-based solutions demonstrate superior flexibility. Moreover, FinQA requires numerical derivation over heterogeneous data sources, which renders Python the most effective solution for extracting information from both semi-unstructured tables and unstructured text.

Upon these observations, our hybrid strategy integrates these three generation outputs, which leverages the strengths of distinct reasoning paths. In particular, CORES-3B attains a 9.35% improvement with hybrid rules over its best individual strategy (SQL) on TableBench. This integration capability stands in sharp contrast to specialized models like OmniSQL-7B, which suffer from catastrophic forgetting in non-SQL tasks, dropping to negli-

Model	TableBench				HiTab				FinQA			
	Dir	PY	SQL	Mix	Dir	PY	SQL	Mix	Dir	PY	SQL	Mix
Commercial LLMs for reference												
GPT-4o-mini	62.20	73.17	59.55	78.05	74.75	70.39	29.86	74.56	56.23	60.59	25.81	62.07
Open-source LLMs (~3B)												
Qwen2.5-Coder-3B-Instruct	39.63	50.20	47.15	64.63	51.14	33.08	18.18	47.10	37.31	40.28	6.02	42.46
SQL-R1-3B	**43.90**	44.51	54.27	66.67	50.88	32.45	23.99	47.16	**37.40**	40.80	9.42	42.28
XiYanSQL-QwenCoder-3B	39.02	49.80	52.44	64.84	52.21	45.39	23.48	53.91	37.31	41.76	12.38	43.24
CscSQL-Grpo-XiYanSQL-3B	30.28	51.42	56.30	**68.90**	52.40	40.66	23.86	51.33	6.89	41.76	13.16	42.89
CORES-3B	40.24	**53.66**	**58.54**	67.89	**53.47**	**46.34**	**24.62**	**54.86**	36.27	**43.33**	**14.21**	**43.85**
Δ vs. Qwen2.5-Coder-3B-Instruct	*+0.61*	*+3.46*	*+11.39*	*+3.26*	*+2.33*	*+13.26*	*+6.44*	*+7.76*	*-1.04*	*+3.05*	*+8.19*	*+1.39*
Open-source LLMs (~7B)												
Qwen2.5-Coder-7B-Instruct	53.86	62.40	57.93	73.98	66.48	54.10	33.52	64.39	49.61	50.92	13.51	53.62
OmniSQL-7B	0.00	2.03	62.40	62.40	0.06	0.76	26.58	26.83	0.00	1.83	13.51	14.39
SQL-R1-7B	0.00	10.37	62.80	62.40	0.76	3.72	30.30	31.88	0.17	4.88	16.65	19.97
Arctic-Text2SQL-R1-7B	0.20	17.68	**67.68**	68.70	0.00	12.63	**33.78**	38.57	0.44	12.73	**20.14**	27.64
CORES-7B	**39.23**	**61.99**	67.48	**70.93**	**52.90**	**52.15**	33.27	**57.51**	**45.77**	**47.95**	16.74	**49.08**
Δ vs. OmniSQL-7B	*+39.23*	*+59.96*	*+5.08*	*+8.53*	*+52.84*	*+51.39*	*+6.69*	*+30.68*	*+45.77*	*+46.12*	*+3.23*	*+34.69*

Table 3: Performance comparison on tableQA benchmarks under different solving strategies. Dir: direct answer generation; PY: Python-based solving; SQL: SQL-based solving; Mix: our hybrid rules across three strategies.

gible accuracy on TableBench-PY and HiTab-PY. Moreover, CORES-3B significantly outperforms its pre-trained base model, Qwen2.5-Coder-3B-Instruct, by 13.26% on HiTab-PY and 11.39% on TableBench-SQL. These results validate that our training framework effectively enhances domain-specific Text-to-SQL reasoning without compromising generalization to TableQA.

Method	Proc-SQL	Proc-PY
CORES-3B (Full)	**38.80**	**27.07**
w/o Training	23.07	13.00
w/o Schema Linking Reward	35.93	25.00
w/o Structure Reward	35.67	26.93
w/o Execution Reward	30.07	20.93
Execution Reward Only	35.80	24.87

Table 4: Ablation study of reward components for SQL and Python generation on ProcQA.

5.4 Ablation Study

Table 4 validates the necessity of our dense reward design, where removing specific process rewards degrades performance across SQL and Python generation. Specifically, the removal of schema linking rewards notably degrades SQL accuracy, which validates the necessity of precise entity grounding. The absence of structure rewards impairs logic synthesis, as evidenced by the decline in Python performance. The significant gap between the full model and the execution-only baseline underscores the insufficiency of sparse outcome-based feedback.

5.5 Statistic Analysis

As illustrated in Figure 4, we conduct a multi-dimensional capability analysis of CORES against baselines. For text-to-SQL, CORES establishes a comprehensive advantage across six critical dimensions for complex reasoning, which maintains robustness across these diverse reasoning requirements. Meanwhile, the result reveals that CORES effectively generalize to tableQA, which facilitates superior handling of numeric reasoning and diverse table scenarios that require procedural reasoning.

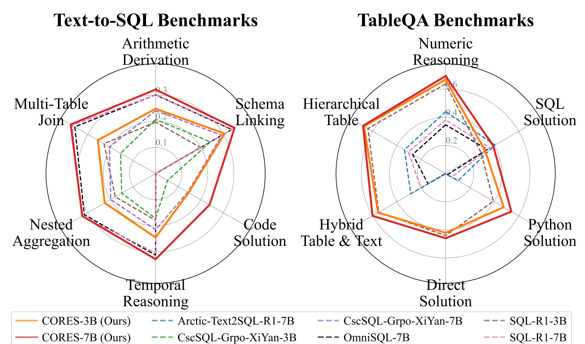

Figure 4: Analysis comparing CORES with baselines.

6 Conclusion

In this paper, we introduced CORES that addresses the limitations of complex reasoning and catas-

trophic forgetting in specialized text-to-SQL models. Our framework leverages Python as a procedural reasoning pivot and dense process reward in GRPO training to enhance model's capacity, which achieves strong performance in text-to-SQL while preserve generalization to tableQA. These findings suggest that aligning code with querying offers a promising direction for developing data agents.

Acknowledgment

This work is supported by NSFC No.62302503, Innovation Research Foundation of NUDT No.ZK23-15, the Open Research Fund from State Key Laboratory of High Performance Computing of China Grant No.202401-09, and Young Elite Scientists Sponsorship Program by CAST No.YESS20240767.

Limitations

While the CORES framework demonstrates strong capabilities in unifying SQL generation and tabular reasoning, we acknowledge three limitations regarding our design choices and experimental scope. First, our approach exclusively utilizes Python as the intermediate representation to guide procedural reasoning. We prioritize Python due to its extensive presence in pre-training corpora, but this implies that we have not investigated the potential of other programming languages or symbolic intermediate forms which might offer alternative advantages in logical expression. Second, the model is fine-tuned solely on text-to-SQL datasets, and its capability on tableQA tasks relies entirely on generalization. This configuration reflects our specific industrial requirement where SQL generation takes precedence over tabular reasoning, which means that our performance on tableQA benchmarks may not equal that of models which are supervised directly on tableQA corpus. Third, we restrict our experiments to model sizes of 3B and 7B parameters. This decision is driven by the constraints of computational resources and the practical necessity for deploying lightweight models in local auditing environments, which leaves the scalability of our approach to larger foundation models for future work.

References

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Wenhu Chen, Hanwen Zha, Zhiyu Chen, Wenhan Xiong, Hong Wang, and William Yang Wang. 2020. Hybridqa: A dataset of multi-hop question answering over tabular and textual data. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1026–1036.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan R Routledge, and 1 others. 2021. Finqa: A dataset of numerical reasoning over financial data. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3697–3711.

Zhoujun Cheng, Haoyu Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. 2022. Hitab: A hierarchical table dataset for question answering and natural language generation. In *Proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 1094–1110.

Yongdong Chi, Hanqing Wang, Yun Chen, Yan Yang, Jian Yang, Zonghan Yang, Xiao Yan, and Guanhua Chen. 2025. Pi-sql: Enhancing text-to-sql with fine-grained guidance from pivot programming languages. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 25120–25144.

Chris J Date. 2011. *SQL and relational theory: how to write accurate SQL code*. " O'Reilly Media, Inc.".

Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-grounded pretraining for text-to-sql. In *The 2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics*.

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

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

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*.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, and 1 others. 2025. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33167–33193.

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*.

Ruochun Jin, Xiyue Wang, Dong Wang, Haoqi Zheng, Yunpeng Qi, Silin Yang, and Meng Zhang. 2025. Talon: A multi-agent framework for long-table exploration and question answering. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 27385–27401.

Rahul Kumar, Amar Raja Dibbu, Shrutendra Harsola, Vignesh Subrahmaniam, and Ashutosh Modi. 2024. Booksql: A large scale text-to-sql dataset for accounting domain. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 497–516.

Fangyu Lei, Jinxiang Meng, Yiming Huang, Tinghong Chen, Yun Zhang, Shizhu He, Jun Zhao, and Kang Liu. 2025. Reasoning-table: Exploring reinforcement learning for table reasoning. *arXiv preprint arXiv:2506.01710*.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, and 1 others. 2025. 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. 2024a. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024b. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

Tao Liu, Hongying Zan, Yifan Li, Dixuan Zhang, Lulu Kong, Haixin Liu, Jiaming Hou, Aoze Zheng, Rui Li, Yiming Qiao, and 1 others. 2025a. Logiccat: A chain-of-thought text-to-sql benchmark for multi-domain reasoning challenges. *arXiv preprint arXiv:2505.18744*.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and 1 others. 2025b. Xiyan-sql: A novel multi-generator framework for text-to-sql. *arXiv preprint arXiv:2507.04701*.

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*.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, and 1 others. 2024. Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*.

Simone Papicchio, Simone Rossi, Luca Cagliero, and Paolo Papotti. 2025. Think2sql: Reinforce llm reasoning capabilities for text2sql. *arXiv preprint arXiv:2504.15077*.

Mohammadreza Pourreza and Davood Rafiei. 2024. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36.

Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, and 1 others. 2022. A survey on text-to-sql parsing: Concepts, methods, and future directions. *arXiv preprint arXiv:2208.13629*.

John W Ratcliff, David E Metzener, and 1 others. 1988. Pattern matching: The gestalt approach. *Dr. Dobb's Journal*, 13(7):46.

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*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.

Lei Sheng and Shuai-Shuai Xu. 2025. Csc-sql: Corrective self-consistency in text-to-sql via reinforcement learning. *arXiv preprint arXiv:2505.13271*.

Aofeng Su, Aowen Wang, Chao Ye, Chen Zhou, Ga Zhang, Gang Chen, Guangcheng Zhu, Haobo Wang, Haokai Xu, Hao Chen, and 1 others. 2024. Tablegpt2: A large multimodal model with tabular data integration. *arXiv preprint arXiv:2411.02059*.

Yize Sui, Jing Ren, Wenjing Yang, Ruochun Jin, Liyang Xu, Xiyao Liu, and Ji Wang. 2025. Elastic robust unlearning of specific knowledge in large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

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*.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

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.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, and 1 others. 2025. Tablebench: A comprehensive and complex benchmark for table question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 25497–25506.

Saisai Yang, Qingyi Huang, Jing Yuan, Liangyu Zha, Kai Tang, Yuhang Yang, Ning Wang, Yucheng Wei, Liyao Li, Wentao Ye, and 1 others. 2025. Tablegpt-r1: Advancing tabular reasoning through reinforcement learning. *arXiv preprint arXiv:2512.20312*.

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, and 1 others. 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*, pages 3911–3921.

Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024a. Finsql: Model-agnostic llms-based text-to-sql framework for financial analysis. In *Companion of the 2024 International Conference on Management of Data*, pages 93–105.

Meng Zhang, Kexin Ma, Liyang Xu, Kedi Zhang, Yuanxi Peng, and Ruochun Jin. 2025a. Clear: A parser-independent disambiguation framework for nl2sql. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 1–14. IEEE.

Tianshu Zhang, Xiang Yue, Yifei Li, and Huan Sun. 2024b. Tablellama: Towards open large generalist models for tables. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6024–6044.

Xiaokang Zhang, Sijia Luo, Bohan Zhang, Zeyao Ma, Jing Zhang, Yang Li, Guanlin Li, Zijun Yao, Kangli Xu, Jinchang Zhou, and 1 others. 2025b. Tablellm: Enabling tabular data manipulation by llms in real office usage scenarios. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10315–10344.

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

Danna Zheng, Mirella Lapata, and Jeff Pan. 2024. 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)*, pages 94–111.

Alex Zhuang, Ge Zhang, Tianyu Zheng, Xinrun Du, Junjie Wang, Weiming Ren, Stephen W Huang, Jie Fu, Xiang Yue, and Wenhu Chen. 2024. Structlm: Towards building generalist models for structured knowledge grounding. *arXiv preprint arXiv:2402.16671*.

A Dataset Construction

Data Filtering Our training dataset is derived from Spider (Yu et al., 2018), BIRD (Li et al., 2024b), and SynSQL (Li et al., 2025), which provide diverse schema structures and query requirements. We apply four rigorous filtering rules: (1) The special keywords in gold SQL skeletons with frequencies below a predefined threshold are discarded. These keywords encompass arithmetic operators (e.g., `+`, `-`, `*`, `/`), aggregation clauses (e.g., `GROUP BY`, `HAVING`), functions (e.g., `MAX`, `ROW_NUMBER`), and conditional expressions (e.g.,

CASE WHEN, IF). (2) Questions exceeding 50 words are removed to mitigate potential ambiguity. (3) Gold SQL queries trigger runtime errors on the original database are excluded. (4) Gold SQL queries yielding non-informative execution results are filtered out, which denotes empty result sets, results consisting solely of NULL values, or trivial scalar outputs such as [[None, 0], [None]].

Database Sampling We randomly sample up to 2,000 rows per table from the remaining samples, which significantly mitigates the potential latency problem caused by runtime database access during the reward computation (Weng et al., 2025). The sampled tables are stored in CSV files, where both SQL and Python can easily query.

Code Annotation As shown in Figure 5, each sample is augmented with a gold Python code that mirrors the logical reasoning of the corresponding SQL query. Initially, we utilize a lightweight LLM, gpt-4o-mini, to generate preliminary Python code with step-wise comments. Then, a powerful reasoning LLM, o3-mini, is employed to iteratively refine the code over three rounds to match the gold SQL execution results. Finally, samples that fail to yield consistent results are excluded. This strict filtering guarantees the consistency between procedural Python reasoning and declarative SQL.

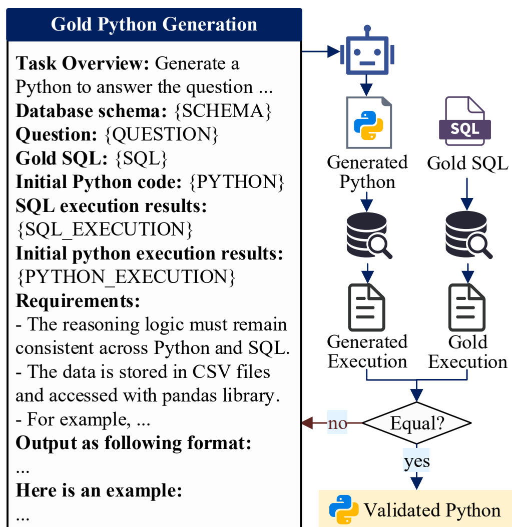

Figure 5: Pipeline for gold Python generation.

B Reward Design Details

B.1 Structure Reward Details

For the computation of structure reward R_{struct}, we describe the mapping rules to abstract Python code into operation sets and list the specific SQL keywords used for operator F1-score evaluation.

Abstract Operation	Python Functions
LOAD_DATA	read_csv, read_excel, read_sql
CREATE_DATAFRAME	DataFrame, Series
MERGE_DATA	merge, concat, join
GROUP_AGGREGATE	groupby, pivot_table, value_counts, agg, aggregate
AGGREGATE_DATA	sum, mean, max, min, count, size
FEATURE_ENGINEERING	apply, map, astype, to_datetime
DATA_TRANSFORMATION	reset_index, drop, rename
SORT_DATA	sort_values
SELECT_ROW	loc, iloc, head, tail
FILTER_DATA	isin, query, where

Table 5: Mapping rules from Python function calls to abstract operations.

Python Operation Mapping In order to construct the operation set O^{py}, we traverse the abstract syntax tree[4] of the generated Python code. We identify function calls and attribute accesses primarily associated with the pandas library and map them to high-level abstract operations. This abstraction ensures that semantically similar operations are treated equivalently. The detailed mapping rules are presented in Table 5.

SQL Keyword Set The operation set O^{sql} represents the structural function of the query. We extract keywords from the generated SQL and compare them against the gold SQL. To avoid penalizing trivial differences such as alias naming, we focus on functional keywords encompassing clauses, joins, aggregations, and functions. The complete list of involved keywords is categorized in Table 6.

B.2 Process Reward Algorithm

Algorithm 2 provides the pseudo code for our multi-dimensional rule-based process reward function. As detailed in Section 4.3, this mechanism aggregates feedback from format validity, schema linking alignment, structural logic, and execution correctness to derive a comprehensive scalar reward for policy optimization.

C Hybrid Algorithm Details

When all three reasoning pathways (Direct, Python, SQL) produce different numerical results, we extract numerical values from the Direct reasoning output as a reference and select the execution result with the minimum absolute difference to this reference. The intuition is that when multiple reason-

[4] https://pypi.org/project/astToolkit/

Category	Keywords
Core	SELECT, FROM, WHERE, AS, DISTINCT, LIMIT, OFFSET
Joins	JOIN, ON, LEFT, RIGHT, INNER, OUTER, FULL, CROSS
Grouping & Sorting	GROUP, BY, HAVING, ORDER, ASC, DESC
Logic & Conditions	AND, OR, NOT, IN, IS, NULL, TRUE, FALSE, LIKE, GLOB, BETWEEN, EXISTS
Set Operations	UNION, ALL, INTERSECT, EXCEPT
Control Flow	CASE, WHEN, THEN, ELSE, END
CTEs	WITH, RECURSIVE
Aggregation	COUNT, SUM, AVG, MIN, MAX
Functions	CAST, COALESCE, IFNULL, NULLIF, LENGTH, LOWER, UPPER, SUBSTR, TRIM, REPLACE, ROUND, ABS, RANDOM
Date & Time	CURRENT_DATE, CURRENT_TIMESTAMP, DATE, DATETIME, STRFTIME
Window Functions	OVER, PARTITION, ROW_NUMBER, RANK, DENSE_RANK

Table 6: Categorized list of SQL keywords.

ing pathways yield conflicting outputs, the direct reasoning path often provides a reliable numerical estimate. By minimizing the distance to this estimate, we reduce the impact of numerical precision issues while maintaining consistency across reasoning modalities.

To validate this heuristic design, we compare it against alternative selection strategies in Table 7. Results show that our approach consistently outperforms fixed selection strategies (always choosing Direct/Python/SQL) and random selection across all three TableQA benchmarks. This demonstrates that adaptive selection based on numerical proximity is more robust than any single fixed strategy.

Selection Method	TableBench	HiTab	FinQA
Our Heuristic Strategy	**67.89**	**54.86**	43.85
Fixed Direct Selection	62.80	51.77	39.93
Fixed Python Selection	65.85	52.40	**44.64**
Fixed SQL Selection	63.62	48.30	30.51
Random Selection	64.23	50.95	38.36

Table 7: Heuristic Strategy vs. Alternative Selection Methods with CORES-3B

D Experiment Setting Details

D.1 Dataset Details

We provide detailed descriptions of the nine cross-domain benchmarks used in our evaluation.

ProcQA is our proposed benchmark to evaluate text-to-SQL on complex procedural queries requiring multi-step reasoning. It contains 1500 examples spanning aggregation with filtering, nested sub-

Algorithm 2: Process Reward Algorithm

Input : Generated output $\mathcal{O}$, Gold SQL q_g, Gold Python p_g, Database $\mathcal{D}$
Output : Total reward $R \in [0, 1]$

// (1) Format Reward
1 $(o_{\text{nl}}, o_{\text{py}}, o_{\text{sql}}) \leftarrow \texttt{ParseTags}(\mathcal{O})$;
2 **if** *all tags* <nl_reasoning>, <python_reasoning>, <sql> *present* **then**
3 $R_{\text{f}} \leftarrow 1$;
4 **else**
5 **return** 0;

// (2) Schema Linking Reward
6 $E_{\text{gold}} \leftarrow \texttt{ExtractEntities}(q_g)$;
7 $\omega_{\text{nl}} \leftarrow 0.5$; $\omega_{\text{py}} \leftarrow 1.0$; $\omega_{\text{sql}} \leftarrow 1.0$;
8 $R_{\text{schema}} \leftarrow 0$;
9 **foreach** $m \in \{nl, py, sql\}$ **do**
10 $E^m_{\text{pred}} \leftarrow \texttt{ExtractEntities}(o_m)$;
11 $R_{\text{schema}} \leftarrow R_{\text{schema}} + \omega_m \cdot \texttt{F1}(E^m_{\text{pred}}, E_{\text{gold}})$;

// (3) Structure Reward
12 $R_{\text{struct}} \leftarrow 0$;
13 **foreach** $(o_m, g_m) \in \{(o_{py}, p_g), (o_{sql}, q_g)\}$ **do**
14 $S_p \leftarrow \texttt{Skeleton}(o_m)$; $S_g \leftarrow \texttt{Skeleton}(g_m)$;
15 $O_p \leftarrow \texttt{ExtractOps}(o_m)$; $O_g \leftarrow \texttt{ExtractOps}(g_m)$;
16 $R_{\text{struct}} \leftarrow R_{\text{struct}} + \frac{1}{2}\left(\texttt{Sim}(S_p, S_g) + \texttt{F1}(O_p, O_g)\right)$;

// (4) Execution Reward
17 $R_{\text{exec}} \leftarrow 0$;
18 **foreach** $(o_m, g_m) \in \{(o_{py}, p_g), (o_{sql}, q_g)\}$ **do**
19 $(r_m, \mathit{success}) \leftarrow \texttt{Execute}(o_m, \mathcal{D})$;
20 $(r_g, _) \leftarrow \texttt{Execute}(g_m, \mathcal{D})$;
21 **if** *success* **then**
22 $R^m_{\text{ex}} \leftarrow 1$;
23 $R^m_{\text{cor}} \leftarrow \mathbf{1}[r_m = r_g]$;
24 **else**
25 $R^m_{\text{ex}} \leftarrow 0$;
26 $R^m_{\text{cor}} \leftarrow 0$;
27 $R_{\text{exec}} \leftarrow R_{\text{exec}} + 0.25 \cdot R^m_{\text{ex}} + R^m_{\text{cor}}$;

// Total Reward Aggregation & Normalization
28 $R_{\text{raw}} \leftarrow R_{\text{schema}} + R_{\text{struct}} + 2 \cdot R_{\text{exec}}$;
29 $R \leftarrow R_{\text{f}} \cdot (R_{\text{raw}}/9.5)$;
30 **return** R

queries, and multi-table joins with complex conditions. Each example is paired with aligned Python that decompose complex SQL queries into interpretable procedural steps.

LogicCat (Liu et al., 2025a) is a multi-step reasoning text-to-SQL benchmark targeting logical reasoning capabilities, where each example includes chain-of-thought decomposing questions into logical steps. We select 1,051 examples emphasizing arithmetic and hypothetical reasoning with no-empty execution results of gold SQL.

Archer (Zheng et al., 2024) is a bilingual text-to-SQL benchmark with 101 complex test queries for arithmetic reasoning, commonsense reasoning requiring world knowledge, and hypothetical reasoning about counterfactual scenarios.

BIRD (Li et al., 2024b) is a large-scale real-world text-to-SQL benchmark across 95 databases with 37 domains. We use the development set containing 1,534 examples. BIRD features external knowledge evidence and realistic database characteristics including dirty values and ambiguous column names that challenge existing systems.

Spider (Yu et al., 2018) is a papular large-scale text-to-SQL benchmark, split into dev set with 1034 examples and test set with 2147 examples across 200 databases spanning 138 domains.

Spider-Realistic (Deng et al., 2021) removes explicit column name mentions from Spider questions, which requires models to infer relevant columns from context rather than lexical matching. The benchmark contains 508 examples.

TableBench (Wu et al., 2025) is a comprehensive tableQA benchmark with the fact verification and numerical reasoning requirements. We employ 493 questions provided from Lei et al. (2025).

HiTab (Cheng et al., 2022) is a hierarchical tableQA benchmark which features complex hierarchical structures with multi-level headers and tree-structured row/column indices. We use the test set with 1,584 examples.

FinQA (Chen et al., 2021) is a large-scale tableQA dataset for numerical reasoning over financial data with 1,147 test examples. Each instance requires multi-step numerical reasoning over both documents and tables.

D.2 Baseline Details

We compare CORES against methods including closed-source LLMs with advanced prompting, open-source pre-trained LLMs, and fine-tuned specialized models.

PI-SQL (Chi et al., 2025) employs multi-stage generation to enhance GPT-4o-mini[5], which generates SQL through the diverse Python intermediate verification and guidance.

Qwen2.5-Coder (Hui et al., 2024) is a code-specialized model pre-trained on diverse code and natural language data, available in 3B and 7B variants with strong performance on code generation.

DeepSeek-R1-Distill-Qwen (Shao et al., 2024) is distilled from DeepSeek-R1, preserving chain-of-thought reasoning patterns beneficial for complex SQL generation while enabling efficient inference.

Meta-Llama-3.1 (Dubey et al., 2024) offers balanced computational efficiency and model capacity with improved pre-training data curation compared to predecessors.

OpenCoder (Huang et al., 2025) is an open-source code generation model trained on publicly available repositories with careful data filtering to ensure quality and diversity.

Granite-3.1 (Mishra et al., 2024) is designed for enterprise applications including code generation and structured data manipulation, optimized for production deployment.

OmniSQL (Li et al., 2025) is a strong SFT model based on Qwen2.5-Coder, fine-tuned on a massive corpus of synthetic data enhanced with chain-of-thought reasoning.

SQL-R1 (Ma et al., 2025) applies reinforcement learning with execution-based feedback, fune-tuned based on Qwen2.5-Coder and OmniSQL for 3B and 7B model, respectively.

XiYanSQL-QwenCoder (Liu et al., 2025b) combines supervised fine-tuning with GRPO to achieve significant text-to-SQL performance without relying on explicit reasoning steps.

CscSQL-Grpo-XiYanSQL (Sheng and Xu, 2025) trains XiYanSQL-QwenCoder using GRPO with execution accuracy and format consistency as the reward functions.

Think2SQL (Papicchio et al., 2025) introduces explicit reasoning steps through structured chain-of-thought during GRPO fine-tuning with fine-grained process reward design.

RewardSQL (Zhang et al., 2025c) trains a reward model on SQL queries formatted in common table expression (CTE) with execution-based labels to guide generation of GRPO training.

Arctic-Text2SQL-R1 (Yao et al., 2025) employs Arctic training architecture for GRPO training and leverage simple execution-only reward.

D.3 Implementation Details

All training and evaluation processes are conducted on a server equipped with two NVIDIA A800 80GB GPUs. We perform full-parameter fine-tuning on both 3B and 7B models for 2 epochs using the GRPO algorithm directly. For the 3B model, we initialize from Qwen2.5-Coder-3B-Instruct, explicitly bypassing the preliminary SFT stage. In contrast, for the 7B model, we select OmniSQL-7B to leverage its superior SQL performance; however, to address its poor instruction-following capability, we implement a cold start phase using 500 samples with CoPS-style prompts before RL training. A KL divergence penalty with a coefficient $\beta = 0.001$ is

[5] https://platform.openai.com/docs/models

Model/Method	Complex Reasoning				General Evaluation				AVG
	ProcQA-SQL	ProcQA-Python	LogicCat	Archer	BIRD-Dev	Spider-Dev	Spider-Test	Spider-Realistic	
Qwen2.5-Coder-3B-Base	23.07	13.00	12.07	6.73	36.57	61.51	61.95	52.17	33.38
Qwen2.5-Coder-3B-SFT	31.53	26.07	12.26	6.73	44.46	71.37	70.66	70.28	41.54
Qwen2.5-Coder-3B-CORES	38.80	27.07	17.97	8.65	52.22	80.85	82.25	76.97	48.10
Qwen2.5-Coder-7B-Base	31.47	21.40	15.30	17.31	48.31	77.27	77.78	73.23	45.26
Qwen2.5-Coder-7B-SFT	48.93	35.47	21.29	18.27	62.45	80.27	81.88	80.31	53.61
Qwen2.5-Coder-7B-CORES	51.27	40.67	23.57	21.15	66.04	85.98	86.77	83.66	57.39

Table 8: Comparison of Base, SFT, and CORES on Qwen2.5-Coder models.

applied to stabilize policy updates and the learning rate is set to 1×10^{-6} . The maximum sequence lengths for both prompts and generation are restricted to 1,024 tokens.

E Additional Experimental Results

E.1 Comparison with Standard SFT

We fine-tune Qwen2.5-Coder-(3B/7B) on ProcQA using standard supervised fine-tuning (SFT) as a baseline. As shown in Table 8, the proposed CORES significantly outperforms SFT across all benchmarks and model sizes, which validates that our RL-based approach with verifiable code-based rewards provides a more effective training signal than standard SFT, especially for complex reasoning tasks. Moreover, SFT requires gold standard answers for loss computation, where we therefore use GPT-4o-mini to complete the SFT datasets.

E.2 Impact of Prompting Strategies

Table 9 presents a comparative analysis of direct generating SQL without any reasoning (Direct), chain-of-thought (CoT) (Wei et al., 2022), program-of-thoughts (PoT) (Chen et al., 2022), and the proposed CoPS on the BIRD and ProcQA benchmarks, which indicates that the direct prompting method outperforms reasoning-based strategies, including CoT and our CoPS. The lower performance with complex reasoning suggests that pre-trained models without specific tuning fail to utilize the intermediate steps effectively. In contrast, the results in Table 2 demonstrate that our CORES framework, which applies CoPS during training, achieves better performance than methods based on CoT (such as SQL-R1) or direct generation (such as XiYanSQL-QwenCoder). This comparison highlights that incorporating code logic into the training process provides a more effective signal for model improvement than using it solely for prompting.

Model	Prompt	BIRD	ProcQA
Qwen2.5-Coder-3B	Direct	42.76	29.87
	CoT	38.33	26.00
	PoT	33.18	20.73
	CoPS	35.92	22.93
Qwen2.5-Coder-7B	Direct	54.11	36.00
	CoT	50.91	33.73
	PoT	47.26	32.53
	CoPS	48.50	31.20

Table 9: Comparison of different prompting strategies for SQL generation on BIRD and ProcQA.

E.3 Performance Analysis by Difficulty Level

Table 10 provides a detailed performance across different difficulty levels for the BIRD and ProcQA benchmarks. On BIRD, CORES achieves consistent improvements over both the base Qwen models and the specialized OmniSQL baseline across all difficulty categories, particularly in the Moderate and Challenge subsets. Similarly, in ProcQA-SQL generation, our method demonstrates superior accuracy in handling complex queries compared to the baselines. Regarding Python generation, the results highlight that OmniSQL fails to generate executable code, while CORES recover and even enhances the coding capability of the base model.

E.4 Evaluation of Python Solution

We further evaluate the quality of the intermediate Python reasoning on datasets beyond ProcQA to assess its correlation with the final SQL output. Table 11 presents the execution accuracy for both the generated Python code and the resulting SQL across six diverse benchmarks. The results indicate that Python generation accuracy generally tracks the trend of SQL performance, which validates the stability of our code-oriented approach. We also observe that the final SQL accuracy frequently exceeds the Python accuracy, suggesting that the model remains robust and can generate correct SQL

(a) Results on BIRD.

Method	Simple	Moderate	Challenge	Overall
Qwen-3B	44.86	24.57	22.07	36.57
CORES-3B	60.00	41.16	37.93	52.22
Δ vs. Qwen	*+15.14*	*+16.59*	*+15.86*	*+15.65*
Qwen-7B	56.22	37.50	32.41	48.31
OmniSQL-7B	69.95	54.74	46.90	63.17
CORES-7B	71.78	59.70	49.66	66.04
Δ vs. OmniSQL	*+1.83*	*+4.96*	*+2.76*	*+2.87*

(b) Results on ProcQA.

Method	Sim.	Mod.	Com.	H.C.	Overall
		SQL Generation			
Qwen-3B	43.20	25.87	15.20	8.00	23.07
CORES-3B	56.53	43.73	34.67	20.27	38.80
Δ vs. Qwen	*+13.33*	*+17.86*	*+19.47*	*+12.27*	*+15.73*
Qwen-7B	53.33	34.93	24.00	13.60	31.47
OmniSQL-7B	61.33	53.60	45.07	32.00	48.00
CORES-7B	67.20	60.00	47.20	30.67	51.27
Δ vs. OmniSQL	*+5.87*	*+6.40*	*+2.13*	*-1.33*	*+3.27*
		Python Generation			
Qwen-3B	29.87	11.73	7.73	2.67	13.00
CORES-3B	48.00	25.07	19.20	5.87	24.53
Δ vs. Qwen	*+18.13*	*+13.34*	*+11.47*	*+3.20*	*+11.53*
Qwen-7B	39.73	25.07	14.67	6.13	21.40
OmniSQL-7B	0.00	0.00	0.00	0.00	0.00
CORES-7B	60.27	49.33	37.07	16.00	40.67
Δ vs. OmniSQL	*+60.27*	*+49.33*	*+37.07*	*+16.00*	*+40.67*

Table 10: Detailed performance comparison by difficulty level on BIRD and ProcQA datasets. Abbreviations: Sim.: Simple, Mod.: Moderate, Chal.: Challenging, Com.: Complex, H.C.: Highly Complex.

queries even when the intermediate Python step contains minor execution errors.

E.5 Comparison with Table-tuned LLMs

Table 12 compares CORES against state-of-the-art 7B-parameter models explicitly tuned for tabular tasks, including TableLlama (Zhang et al., 2024b), StructLM (Zhuang et al., 2024), and TableGPT2 (Su et al., 2024). The results demonstrate that our framework achieves superior generalization across diverse benchmarks, outperforming the baselines on BIRD, TableBench, and FinQA. Notably, CORES-7B exceeds TableGPT2-7B on the BIRD benchmark by a margin of 18.65%, which underscores its advantage in handling complex logic. While TableGPT2 shows slightly higher accuracy on HiTab, our model maintains a consistent lead on the remaining datasets, suggesting that our training strategy effectively balances code generation and reasoning capabilities.

Dataset	CORES-3B		CORES-7B	
	Python	SQL	Python	SQL
Archer	12.50	10.58	31.73	24.04
LogicCat	20.53	18.44	23.19	23.48
BIRD	46.94	59.19	58.67	68.25
Spider-Dev	67.89	79.88	79.88	87.81
Spider-Test	64.56	81.14	81.46	86.49
Spider-Realistic	62.20	76.18	80.51	85.43

Table 11: Performance Comparison between intermediate Python generation and final SQL generation for CORES-3B and CORES-7B.

Model	BIRD	TableBench	HiTab	FinQA
TableLlama-7B	–	–	63.19	–
StructLM-7B	22.30	42.02	–	27.30
TableGPT2-7B	47.39	–	**63.89**	38.36
CORES-3B	52.22	67.89	54.86	43.85
CORES-7B	**66.04**	**70.93**	57.51	**49.08**

Table 12: Performance comparison with instruction-tuned LLMs. –: results not reported in original papers or official baselines.

E.6 Analysis of Catastrophic Forgetting

As shown in Table 13, we test larger SQL-specialized models on the ProcQA benchmark, which illustrates that even models with significantly larger parameter counts suffer severe performance catastrophic forgetting on Python reasoning tasks.

E.7 Training Process Analysis

To validate the stability of our reinforcement learning framework, we visualize the reward trajectories during the GRPO training phase in Figure 6. The total reward R is normalized to ensure the optimization objective remains bounded within the unit interval $[0, 1]$. The curves exhibit a consistent upward trend for both model scales, which indicates effective optimization of the policy against the proposed process reward functions. Specifically, CORES-3B demonstrates rapid adaptation in the initial steps, rising from a near-zero baseline to convergence, whereas CORES-7B benefits from its warm-started initialization to achieve a higher terminal reward. The steady convergence observed in both smoothed trajectories suggests that the KL divergence penalty successfully mitigates training instability, ensuring that the model internalizes the reasoning pivot without deviating from the language distribution.

Model	ProcQA		Δ	
	SQL	Python	SQL	Python
Qwen2.5-Coder-14B	39.00	33.73	—	—
OmniSQL-14B	49.60	5.07	**+10.60**	**-28.66**
OmniSQL-32B	48.47	11.20	**+9.47**	**-22.53**
SQL-R1-14B	42.53	30.13	**+3.53**	**-3.60**

Table 13: Catastrophic forgetting analysis on larger models. Δ indicates the performance change.

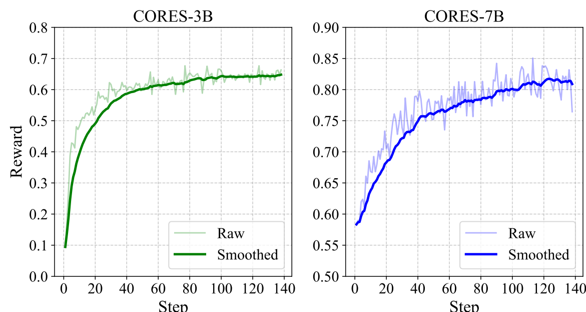

Figure 6: Reward convergence trajectories for CORES-3B and CORES-7B during GRPO training. The solid lines represent the smoothed reward, while the transparent lines indicate raw fluctuations.

F Prompts

Figure 7 and 8 are the prompts for CORES to complete text-to-SQL and tableQA tasks, respectively.

G Case Study

Figure 9 illustrates how our framework resolves complex queries where baseline models often fail. The Python intermediate step clarifies the logical dependencies, which allows the model to produce a valid SQL query that accurately joins the tables.

Figure 10 demonstrates the effectiveness of Python reasoning for tableQA generalization in hybrid data scenarios. The code-based solution are more flexible than SQL-based solution, meanwhile, ensures arithmetic accuracy.

Figure 11 reveals the catastrophic forgetting issues observed in the Baseline model. Due to overfitting on specific SQL reasoning traces, baselines lose the flexibility to follow strict formatting instructions or switch reasoning modes, often erroneously applying SQL generation patterns to non-SQL tasks such as TableQA.

Task Overview:
You are a database expert skilled in text-to-SQL with `SQLite` syntax.
Your task is to generate an correct SQL query to answer the question based on the given schema.

Database schema:
{SCHEMA}
This schema describes the database's structure, including tables, columns, primary keys, foreign keys.

Question:
{QUESTION}

To solve the task, you need complete the following steps:
1. First, provide step-by-step reasoning using Natural Language (NL).
2. Then, based on the NL reasoning, provide stepwise reasoning using Python code with comments.
  - The data is stored in CSV files and accessed with the pandas library.
  - For example, step 1: Import necessary libraries; step 2: Load CSV files (path: `**{DB_ID}**/table_name.csv`); step 3: ...; step N: Assign the result to variable `ans: DataFrame` (i.e., ans = xxx).
3. Finally, based on the natural language and python reasoning, generate the SQL query.

Requirements:
- The reasoning logic must remain consistent across natural language, python code, and SQL.
- When writing SQL, use the original column names provided in the schema, and enclose them with backticks (`) for quoting.

Output as following format:

<nl_reasoning>
Write natural language reasoning here.
</nl_reasoning>

<python_reasoning>
```python
Write python reasoning here.
```
</python_reasoning>

<sql>
```sql
Write final SQL here.
```
</sql>

Figure 7: The prompt of our proposed Code-Oriented Prompting for SQL (CoPS).

Directly Answering

Task Overview:
You are a expert data analyst skilled in table question answering.
Your task is to give a direct answer for the question based on the given tables and text.

{DATASET_INPUT_TEMPLATE}

To solve the task, you need complete the following steps:
1. Understand the table content and the context, and solve the question step by step.
- All data are directly provided in the Table Content and the context.
- Scanning through the data and list the used data one by one in detail.
- Verify the accuracy of your calculations.
2. Provide the final answer.
- The answer should be short and simple. It can be a number, a word, or a phrase in the table, but not a full sentence or a statement.

Requirements:
- You should manually calculating based on the provided table data.
- DO NOT generate any SQL queries or code. Simply look at the table content and perform the reasoning.

Output format:

<think>
Step 1: Understand the question.
Step 2: Scanning through the data.
Step 3: Performing calculation process.
...
Therefore, ...
</think>

<answer>
Write your answer here.
</answer>

Python-based Solving

Task Overview:
You are an expert data analyst skilled in table question answering.
Your task is to generate Python code to answer the question based on the given tables.

{DATASET_INPUT_TEMPLATE}

To solve the task, you need to complete the following steps:
1. Generate the step-by-step natural language reasoning to clarify which values are required.
- All data are provided in above Table Content and Text. You need extract the key information from both of them.
- Do not write any SQL queries. You are thinking about writing a Python code.
2. Based on the reasoning, generate Python code with stepwise comments to answer the question.
- **DO NOT** try to read external CSV files.
- Instead, extract the relevant data from the Table Content into a **list of dictionaries**.
- The final result must be assigned to the variable `ans`.

Output in the following format:

<nl_reasoning>
Write natural language reasoning here.
</nl_reasoning>

<python_code>
```python
# Extract the relevant data from the table content into a list of dictionaries.
data = ...
# ... rest of your code ...
ans = ...
```
</python_code>

SQL-based Solving

Task Overview:
You are an expert data analyst skilled in table question answering and familiar with `SQLite` syntax.
Your task is to generate an SQL query to answer the question based on the given tables.

{DATASET_INPUT_TEMPLATE}

To solve the task, you need complete the following steps:
1. First, provide step-by-step reasoning using Natural Language (NL).
- All data are provided in above Table Content and texts. You need extract the key information.
- The Table Content may be hierarchical.
2. Then, based on the natural language and python reasoning, generate the SQL query.
- The table names are corresponded to their Table ID. Default is Table 0.
- Use the original column names provided in the schema, and enclose them with backticks (`) for quoting.

Output in the following format:

<nl_reasoning>
Write natural language reasoning here.
</nl_reasoning>

<sql>
```sql
Write final SQL here.
```
</sql>

Figure 8: The prompts for tableQA tasks, divided into three generation strategies.

Input:

Question: From 2000 to 2005, what percentage of drivers who were born before 1985 and the lap numbers were over 50?

Schema: drivers: [driverId, dob, ...], results: [raceId, driverId, laps, ...], races: [raceId, year, ...] ...

Baseline Output: Hallucinating the temporal constraint ``From 2000 to 2005'' as a filter

Think: To translate the natural language question into an executable SQLite query, let's break it down step by step.
1. Filter drivers: `dob < 1985` & `laps > 50`.
2. Filter total drivers: `dob` between 2000-2005.
3. Calc Percentage: Count(Condition 1) / Count(Condition 2).

SQL: SELECT (CAST(COUNT(CASE WHEN d.dob < '2000-01-01' AND r.laps > 50 THEN 1 ELSE NULL END) AS REAL) * 100.0) / COUNT(CASE WHEN d.dob BETWEEN '2000-01-01' AND '2005-12-31' THEN 1 ELSE NULL END) AS percentage FROM drivers AS d JOIN results AS r ON d.driverId = r.driverId;

CORES Output: Leverage Python to correctly identify the filter of season year

<nl_reasoning>
1. Join `drivers` and `results` on `driverId`.
2. Filter Context: `races.year` in [2000, 2005].
3. Condition A: `dob < 1985` AND `laps > 50`.
4. Ops: Count(A) / Count(Total in Context) * 100.
</nl_reasoning>

<python_reasoning>
```
# Step 1: Import necessary libraries
import pandas as pd

# Step 2: Load CSV files into DataFrames
drivers = pd.read_csv('formula_1/drivers.csv')
results = pd.read_csv('formula_1/results.csv')
races = pd.read_csv('formula_1/races.csv')

# Step 3: Merge the results and races DataFrames on 'raceId'
merged_df = pd.merge(results, races, on='raceId')

# Step 4: Merge the merged DataFrame with the drivers DataFrame on 'driverId'
merged_df = pd.merge(merged_df, drivers, on='driverId')

# Step 5: Filter the DataFrame for races between 2000 and 2005
filtered_df = merged_df[(merged_df['year'] >= 2000) & (merged_df['year'] <= 2005)]

# Step 6: Count the total number of drivers in the filtered DataFrame
total_drivers = filtered_df.shape[0]

# Step 7: Count the number of drivers who were born before 1985 and have laps > 50
matching_drivers = filtered_df[(filtered_df['dob'].str[:4].astype(int) < 1985) & (filtered_df['laps'] > 50)].shape[0]

# Step 8: Calculate the percentage
percentage = (matching_drivers / total_drivers) * 100 if total_drivers > 0 else 0

# Step 9: Assign the result to variable ans as a DataFrame
ans = pd.DataFrame({'percentage': [percentage]})
```
</python_reasoning>

<sql>
```
SELECT CAST(SUM(CASE WHEN SUBSTR(d.dob, 1, 4) < '1985' AND r.laps > 50 THEN 1 ELSE 0 END) AS REAL) *
100.0 / COUNT(*) FROM drivers d JOIN results r ON d.driverId = r.driverId JOIN races ra ON r.raceId = ra.raceId WHERE
ra.year BETWEEN 2000 AND 2005;
```
</sql>

Figure 9: A case study on the Text-to-SQL task. The baseline model hallucinates a filter condition on the wrong table due to schema ambiguity. CORES leverages a Python reasoning trace to clarify the join path and temporal constraints, which guides the generation of the correct SQL query.

Input:

Question: What is the percentage increase in cash flows from operations from 2015 to 2016?

Text: A regular quarterly dividend of $ 0.29 per common share is payable on march 31 , 2017 to shareholders of record as of thef close of business on march 17 , 2017. Cash flows from operations cash flows from operations were $ 1925 million , $ 1131 million and $ 1165 million in 2016 , 2015 and 2014 respectively. Our net cash provided by operating activities consists primarily of net earnings...

Table:
type_of_obligations	total	payments_due_in_less_than_1_year	payments_due_in_13_years	payments_due_in_35_years	payments_due_in_more_than_5_years
long-term debt (1)	$ 10591	$ 332	$ 1573	$ 2536	$ 6150
interest (2)	2829	381	706	595	1147
operating leases	401	96	158	82	65
data processing and maintenance	557	242	258	35	22
other contractual obligations (3)	51	17	17	16	1
total	$ 14429	$ 1068	$ 2712	$ 3264	$ 7385

Directly Answering: 70.55% ✗ **(Arithmetic error due to model hallucination.)**

Step 1: Understand the question. The question asks for the percentage increase in cash flows from operations ...
Step 2: Scanning through the data. The relevant data from the table is: Cash flows from operations in 2016 ...
Step 3: Performing calculation process.
The formula for percentage increase is: \[\text{Percentage Increase} = \left(\frac{\text{Value in 2016} - \text{Value in 2015}}{\text{Value in 2015}} \right) \times 100 \]
...
Then, calculating the percentage:
\[\left(\frac{794}{1131} \right) \times 100 \approx 70.55\% \]
Therefore, the percentage increase in cash flows from operations from 2015 to 2016 is approximately 70.55%.

Python-based Solution: 70.20% ✓ ***(Closest to direct answer, so we select it.)***

```
# Step 1: Extract the relevant data from the table content into a list of dictionaries.
data = [
    {"year": 2016, "cash_flows_from_operations": 1925},
    {"year": 2015, "cash_flows_from_operations": 1131}
]

# Step 2: Calculate the percentage increase in cash flows from operations from 2015 to 2016.
# We use a simple for loop to sum the values for each year.
values = [0, 0]
for record in data:
    if record["year"] == 2016:
        values[0] = record["cash_flows_from_operations"]
    elif record["year"] == 2015:
        values[1] = record["cash_flows_from_operations"]

# Step 3: Compute the percentage increase.
percentage_increase = ((values[0] - values[1]) / values[1]) * 100

# Step 4: Assign the result to the variable ans.
ans = percentage_increase
```

SQL-based Solution: 373.80% ✗ **(Heterogeneous table formats hinder SQL performance.)**

```
SELECT CAST((CAST(REPLACE(`payments_due_in_13_years`, '$', '') AS REAL) -
CAST(REPLACE(`payments_due_in_less_than_1_year`, '$', '') AS REAL)) AS REAL) * 100 /
CAST(REPLACE(`payments_due_in_less_than_1_year`, '$', '') AS REAL) AS percentage_increase FROM table_0;
```

Figure 10: A case study on the TableQA task involving heterogeneous data. The SQL-based solution fails to handle the irregular table format, and the direct LLM output contains arithmetic errors. CORES solves the problem by extracting data into a Python structure and performing the calculation programmatically.

Text-to-SQL Instructions

Task: You need directly generate the SQL query without any explanations.

Output as following format:
<sql>
```sql
Write final SQL here.
```
</sql>

Task: Your task is to generate an correct Python code to answer the question based on the given schema.

Output as following format:
<python_reasoning>
```python
Write python reasoning here.
```
</python_reasoning>

Baseline: Do not follow the instruction, always output the similar content. ✗

To translate the natural language question into an executable SQLite query, let's break it down step by step:

Step 1: Understand the Question
Step 2: Identify Relevant Tables and Columns
Step 3: Break Down the Query
Step 4: Write the SQL Query
Step 5: Verify the Query
Final SQL Query:

```sql
SELECT Name, Age FROM singer WHERE Song_Name = "Gentleman";
```

This query will return the names and ages of all singers who released the song "Gentleman".

TableQA Instructions

Task: Your task is to directly answer the question based on the given tables.

Output in the following format:
<think>
Write your step-by-step thought here.
</think>

<answer>
Here is your answer.
</answer>

Task: Your task is to generate Python code to answer the question based on the given tables.

Output in the following format:
<nl_reasoning>
Write natural language reasoning here.
</nl_reasoning>

<python_code>
Here is the Code.
</python_code>

Baseline: Do not follow the instruction, lose generalization to tableQA input. ✗

To determine the net change in net revenue during 2015 for Entergy Corporation, we need to focus on the difference between the 2015 net revenue and the 2014 net revenue. The relevant columns in the table are `2014 net revenue` and `2015 net revenue`. The difference between these two values will give us the net change in net revenue for 2015.
Here is the step-by-step reasoning:
1. Identify the columns for 2014 and 2015 net revenue from the table."
2. Calculate the difference between the 2015 net revenue and the 2014 net revenue."
3. Present the result as the net change in net revenue for 2015.
The SQL query to achieve this is as follows:
```sql
SELECT (2015_net_revenue - 2014_net_revenue) AS net_revenue_change_2015 FROM ( SELECT
   SUM(CASE WHEN col0 = '2014 net revenue' THEN amount ELSE 0 END) AS 2014_net_revenue,
   SUM(CASE WHEN col0 = '2015 net revenue' THEN amount ELSE 0 END) AS 2015_net_revenue
 FROM   table) AS revenue_data;
```

Figure 11: A case study illustrating catastrophic forgetting in the baseline model. In the text-to-SQL task, the model ignores the instruction to output direct SQL, instead defaulting to a verbose reasoning trace. In the tableQA task, the model loses generalization capability, hallucinating an SQL-based solution for a problem.