DSMR-SQL: Enhancing Text-to-SQL with Dual-Strategy SQL Generation and Multi-Role SQL Selection

Anonymous ACL submission

Abstract

Recent advancements in Large Language Models (LLMs) have markedly improved SQL generation. Nevertheless, existing approaches typically rely on single-model designs, limiting their capacity to effectively handle complex user queries. In addition, current methods often face difficulties in selecting the optimal SQL from multiple candidates. To mitigate these limitations, this study presents **DSMR-SQL**, a two-stage framework consisting of: (**1**) **Dual-Strategy SQL Generation:** DSMR-SQL aims to produce a broader spectrum of SQL queries by using multiple models with two strategies: Supervised Fine-Tuning and In-Context Learning; (**2**) **Multi-Role SQL Selection:** DSMR-SQL seeks to identify the SQL most aligning with user intent by introducing a collaborative framework involving three roles (i.e., Proposer, Critic, Summarizer). Extensive experiments on various datasets substantiate the efficacy of DSMR-SQL in enhancing SQL generation.

1 Introduction

Two heads are better than one.

- Proverb

SQL queries are essential for optimizing data retrieval efficiency across multiple databases. Such data has been implemented in critical domains, including healthcare analytics (Mendhe et al., 2024) and financial systems (Zhang et al., 2024a). While technical professionals possess specialized expertise in crafting SQL, the emergence of natural language interfaces to databases (NLIDBs) has enabled non-technical users to effortlessly access structured data (Deng et al., 2022). This enhanced accessibility has catalyzed significant advancements in text-to-SQL systems that automatically translate natural language (NL) queries into valid SQL statements.

Recent breakthroughs in large language models (LLMs) (Achiam et al., 2023) have revolutionized text-to-SQL methodologies, particularly through the implementation of *In-Context Learning (ICL)* and *Fine-Tuning (FT)*. In particular, ICL allows models to utilize prompt engineering to cope with unseen scenarios effectively (Pourreza and Rafiei, 2024a; Gao et al., 2024; Lee et al., 2024). In contrast, FT customizes models for domain-specific tasks using curated datasets, with an emphasis on Supervised Fine-Tuning (SFT) (Li et al., 2024a; Pourreza and Rafiei, 2024b; Zhang et al., 2024a). Despite their promise, current approaches commonly encounter the following limitations that impede their broader adoption:

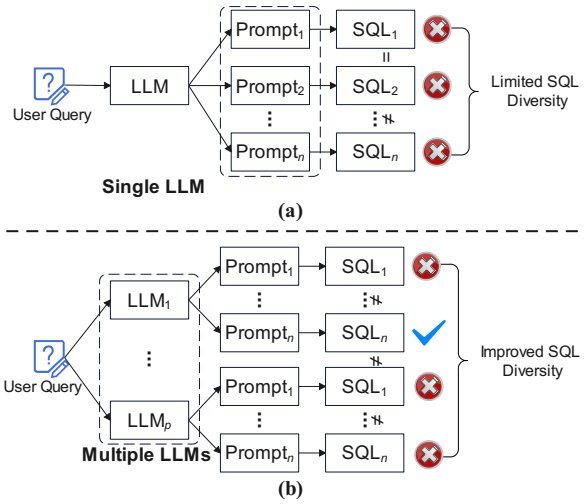


Figure 1: Illustrations of two different system designs. (a) Limited SQL diversity for Single-Model design. (b) Improved SQL diversity for Multi-Model design.

Limitation 1: Limited SQL Diversity caused by Single-Model Designs. As depicted in Figure 1, existing methods often rely on a single model for SQL generation, which can be categorized into *Single-Model Single-Prompt* and *Single-Model Multi-Prompt* designs (Pourreza and Rafiei, 2024a; Lee et al., 2024).

(a) ***Single-Model Single-Prompt Designs:*** Notably, LLMs are highly sensitive to the structure

and content of semantically identical prompts, leading to inconsistent SQL outputs (Lu et al., 2022; Jang and Lukasiewicz, 2023). Moreover, using a single prompt inherently narrows the search space for potential SQL solutions, thereby overlooking alternative SQL formulations that may better reflect actual user intent. To mitigate these issues, self-consistency (Cheng et al., 2024) introduces variability through high-temperature sampling and selects the SQL with the most consistent execution results (Gao et al., 2024; Mao et al., 2024; Talaei et al., 2024). However, increased temperature can introduce model hallucinations, thereby undermining overall model performance (Renze and Guven, 2024). Meanwhile, the SQL diversity achieved by self-consistency remains insufficient for handling highly complex user queries (Lee et al., 2024).

(b) ***Single-Model Multi-Prompt Designs:*** Notably, approaches like MCS-SQL (Lee et al., 2024) have attempted to expand the solution space by generating multiple SQL queries from diverse prompts. Nonetheless, the SQL diversity of a single model still remains limited, which stems from the tendency of LLMs to follow specific syntactic and semantic patterns when producing SQL (Ji et al., 2023; Yin et al., 2023). This behavior is influenced by specific training data distributions and inherent model architectures (Jiang et al., 2024a,b). For instance, GPT-series models tend to prefer "LEFT JOIN" over "JOIN" when constructing SQL queries (Liu et al., 2023). As a result, the generated SQL candidates may exhibit structural similarities and struggle to capture actual user intent, even when diverse prompts are utilized in a single model. In particular, MCS-SQL (Lee et al., 2024) used five different prompts and high-temperature sampling to generate 100 SQL candidates for each user query to achieve competitive performance. This showcases an over-reliance on exhaustive exploration as a means to compensate for the limited SQL diversity in single-model designs.

Limitation 2: Insufficient SQL Selection Mechanisms. Notably, identifying the optimal SQL from diverse candidates remains a significant challenge. In particular, existing approaches predominantly rely on two strategies: *Consistency Voting* and *Simple SQL Selection*. To be specific, *Consistency Voting* determines the optimal SQL by choosing the one with the most frequent execution results (Gao et al., 2024; Talaei et al., 2024). In contrast, *Simple SQL Selection* employs a single LLM as a judge to rank and select the most appropriate SQL (Lee et al., 2024; Li and Xie, 2024). However, *Consistency Voting* fails when none of the SQL candidates yield consistent execution results, and even the most consistent result may be erroneous due to shared underlying errors (Pourreza et al., 2024). Besides, *Simple SQL Selection* is susceptible to inherent model biases and positional answer biases (Wang et al., 2023; Shi et al., 2024; Zhang et al., 2024b), which may result in errors when ranking complex SQL queries. Meanwhile, it overlooks the iterative nature of human reasoning, which typically involves multiple rounds of revision and refinement (Zheng et al., 2024; Madaan et al., 2024). Accordingly, despite their user-friendly nature, the above methods often lead to incorrect SQL selection results, which hinder the reliability of text-to-SQL systems in real-world applications.

In light of the above limitations, we raise the following research question: *How to leverage LLMs to generate diverse SQL candidates while improving the reliable selection of the optimal one?*

This study introduces **DSMR-SQL**, a framework developed to improve SQL generation using **D**ual-**S**trategy reasoning and **M**ulti-**R**ole SQL selection. (**1**) **Dual-Strategy SQL Generation (DSG):** DSMR-SQL combines SFT and ICL using multiple models to produce diverse SQL, which alleviates the limitations of single-model settings. By harnessing the strengths of these two strategies, DSG increases the possibility that the correct SQL is available in the candidate set; (**2**) **Multi-Role SQL Selection (MRS):** DSMR-SQL employs a multi-role framework (i.e., Proposer, Critic, and Summarizer) to identify the optimal SQL. This collaborative mechanism simulates human-like reasoning to improve SQL selection, where the final SQL is determined through iterative refinement. Extensive experiments were conducted on several datasets, showcasing the effectiveness of DSMR-SQL in improving SQL generation.

In summary, this work offers the following contributions: (**1**) This study highlights critical limitations in current SQL generation approaches, particularly in producing diverse SQL candidates and reliably selecting the optimal one. This motivated us to develop more effective methods to handle these issues; (**2**) We present DSMR-SQL, a framework integrating dual-strategy reasoning and multi-role SQL selection to improve SQL generation; (**3**) Extensive experiments across various datasets confirm the effectiveness of DSMR-SQL in enhancing model reasoning and generating high-quality SQL.

2 Related Work

2.1 SQL Generation Approaches

Recent LLM-based SQL generation techniques widely adopt *In-Context Learning (ICL)* and *Fine-Tuning (FT)*. In particular, prompt engineering has emerged as an effective approach to enhance SQL generation due to its flexibility to deal with unfamiliar scenarios (Pourreza and Rafiei, 2024a; Gao et al., 2024; Wang et al., 2024). However, closed-source LLMs are often associated with high monetary costs and are characterized by inherent model biases and output instability, thereby diminishing their practical reliability (Liu et al., 2023; Turpin et al., 2024). In contrast, the growing adoption of open-source LLMs has catalyzed research on fine-tuning these models for SQL generation owing to the stability of SFT (Zhang et al., 2024a; Pourreza and Rafiei, 2024b; Li et al., 2024a). Despite their potential, open-source models still struggle with maintaining robustness in complex scenarios and generalizing effectively due to their reliance on the curated training data. Inspired by these advancements, this study combines the strengths of ICL and SFT to promote diverse SQL generation, thereby increasing the likelihood that the correct SQL is included in the candidate set.

2.2 SQL Selection Techniques

In the literature, *Consistency Voting* and *Simple SQL Selection* have been introduced for selecting SQL from multiple candidates. In particular, *Consistency Voting* enables the selection of the SQL with the most frequently appearing execution results. For instance, approaches such as C3 (Dong et al., 2023), DAIL-SQL (Gao et al., 2024), MetaSQL (Fan et al., 2024), and PURPLE (Ren et al., 2024) reduced output noise by selecting the SQL with the most consistent execution results. Additionally, *Simple SQL Selection* utilizes a single LLM to assess and identify the optimal SQL from multiple candidates. For instance, LEVER (Ni et al., 2023) and Li et al. (Li and Xie, 2024) used ranking techniques to select the most suitable SQL. MCS-SQL (Lee et al., 2024) generated diverse SQL candidates using varied prompts and applied multiple-choice selection to determine the final SQL. Different from the above techniques, this study introduces multiple roles to mimic human-like problem-solving processes and iteratively improve SQL selection.

3 Methodology

This section introduces DSMR-SQL, a two-stage framework designed to enhance SQL generation. As depicted in Figure 2, the framework comprises two primary stages: (**1**) **Dual-Strategy SQL Generation (DSG):** Multiple SQL candidates are generated by combining SFT and ICL. This dual-strategy approach leverages the complementary strengths of both techniques to improve SQL diversity; (**2**) **Multi-Role SQL Selection (MRS):** A collaborative framework with three roles is applied to iteratively identify the most suitable SQL. The details of these stages are elaborated as follows.

3.1 Dual-Strategy SQL Generation (DSG)

As illustrated in the left part of Figure 2, DSG combines the stability of SFT with the flexibility of ICL to mitigate the limitations of single-model reliance (Liu et al., 2023; Turpin et al., 2024).

Specifically, the process begins with using open-source LLMs to perform SFT on task-specific datasets, enabling models to produce SQL candidates that adhere to the syntactic and semantic patterns in the training data. This alignment provides a stable foundation for SQL generation, reducing output instability and inherent model biases in ICL-based methods. To achieve this, DSG adopts the CodeS framework (Li et al., 2024a) and utilizes several open-source models for SQL generation. Despite its advantages, the fixed reasoning of SFT inherently limits its capacity to further explore alternative SQL solutions.

To overcome this limitation, ICL-based methods are further integrated to enhance reasoning flexibility. Typically, humans solve problems using various methods: some rely on quick intuitive thinking, others follow methodical rules, while some adopt flexible reasoning without rigid guidelines (Qi et al., 2024). Inspired by these strategies, DSG implements diverse reasoning processes to mimic these human-like approaches, thereby enhancing the reasoning capacity of closed-source LLMs. An illustrative example of the ICL-based strategy implemented in DSG is presented in Figure 5 from Appendix L, which includes:

(1) **Direct SQL Generation:** LLMs generate SQL directly without providing explanations, prioritizing simplicity and speed. This approach is akin to "fast thinking" (Lin et al., 2024) and is particularly effective for straightforward user queries with minimal logical reasoning. The key prompt for this

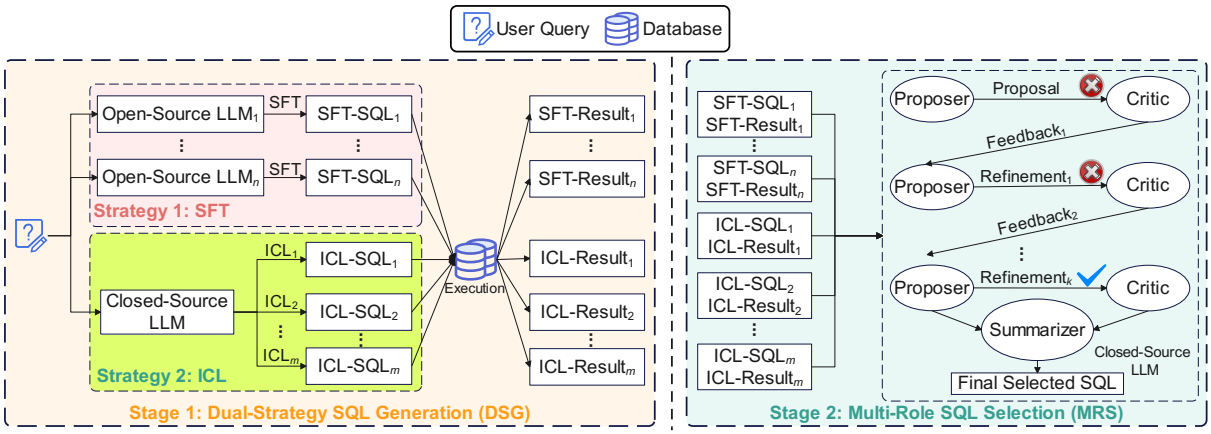


Figure 2: Illustration of our proposed DSMR-SQL. Specifically, the framework consists of two stages, including Dual-Strategy SQL Generation (DSG) and Multi-Role SQL Selection (MRS).

method is: "*Please directly generate SQL queries with no explanations*."

(2) **Strict Step-by-Step Reasoning:** LLMs adopt a structured and methodical approach to improve SQL accuracy. Each step involves a detailed analysis of specific aspects (e.g., user query, database schema, etc.), ensuring that the generated SQL satisfies all requirements and undergoes rigorous validation. The key prompt for this method is: "*Please strictly obey the following steps to generate high-quality SQL queries*."

(3) **Flexible Reasoning:** LLMs engage in flexible reasoning processes without rigid guidelines, relying on contextual understanding of the given problem. This approach is also effective in addressing highly complex user queries. The key prompt for this method is: "*Please generate high-quality SQL queries with your detailed reasoning*."

By integrating the stability of SFT with the diverse reasoning of ICL, a wide range of SQL candidates are generated by DSG. These SQL queries broaden the solution space, thereby increasing the probability of incorporating the correct SQL. Subsequently, the SQL candidates are executed in the databases to retrieve execution results, which serve as input for the next SQL selection process. The detailed prompts for each ICL-based method are provided in Appendix E, F, and G.

3.2 Multi-Role SQL Selection (MRS)

While DSG generates diverse SQL candidates, selecting the most suitable one remains a significant challenge. Therefore, inspired by the multi-agent debate (Liang et al., 2023), this study introduces a Multi-Role SQL Selection (MRS) framework, which assigns distinct roles in a single LLM to facilitate accurate SQL selection. As depicted in the right part of Figure 2, MRS employs the following roles to determine the optimal SQL based on the given SQL candidates and their respective execution results:

(1) **Proposer:** The Proposer formulates a clear proposal involving the chosen SQL and its detailed reasoning, serving as the foundation for subsequent critique and refinement.

(2) **Critic:** The Critic assesses the Proposer's reasoning process. It delivers detailed feedback to the Proposer to pinpoint errors or suggest improvements, fostering a rigorous evaluation process.

(3) **Summarizer:** The Summarizer consolidates the Proposer's reasoning and the Critic's feedback to finalize the optimal SQL, ensuring that the SQL selection process is accurate and logically coherent.

Importantly, MRS operates as an iterative process, wherein the Proposer refines its reasoning through several rounds of feedback from the Critic. This iterative mechanism ensures that each SQL is selected with a high degree of precision. The complete procedure is outlined in Algorithm 1, which is detailed as follows:

The process commences with the Proposer presenting a chosen SQL and its associated reasoning. The reasoning is then analyzed by the Critic, who identifies potential flaws and offers constructive feedback to improve SQL selection. Based on this feedback, the reasoning process is refined by the Proposer before being returned to the Critic. This iterative cycle continues until a consensus is reached between the Proposer and Critic on the reasoning process and the selected SQL. Once a consensus is achieved, the Summarizer reviews the finalized reasoning and feedback, consolidating them into

the final selected SQL.

This iterative collaboration between the Proposer and Critic imitates human-like problem-solving processes, where the Critic's feedback refines the solutions from the Proposer. Consequently, MRS facilitates a more reliable and robust SQL selection. An illustrative example of this iterative process is given in Appendix N, and the detailed prompt for MRS is shown in Appendix H.

Algorithm 1 Multi-Role SQL Selection (MRS)

Input: User query Q, provided hints H, SQL candidates $\{S_k\}$, and their execution results $\{E_k\}$
Output: Final selected SQL S^*

R_1, S_1 = Proposer(Q, H, $\{S_k\}$, $\{E_k\}$)
▷ R_1, S_1 are the initial reasoning process and selected SQL
for $i = 1$ to N **do**
  F_i = Critic(Q, H, R_i, S_i)
  ▷ N is the number of iterations, which is not limited to a certain value
  ▷ F_i is the i-th negative feedback, requiring the Proposer to refine its reasoning
  R_{i+1}, S_{i+1} = Proposer(R_i, F_i)
  ▷ R_{i+1}, S_{i+1} are the refined reasoning process and selected SQL
  F_{i+1} = Critic(Q, H, R_{i+1}, S_{i+1})
  if F_{i+1} is positive **then**
    Return F_{i+1}, R_{i+1}, S_{i+1}
  end if
end for
S^* = Summarizer(F_{i+1}, R_{i+1}, S_{i+1})
return S^* ▷ Final selected SQL

4 Experiments

This section evaluates the performance of DSMR-SQL on multiple datasets. Extensive experiments were conducted to answer the following questions: **RQ1**. How does DSMR-SQL perform compared with previous LLM-based approaches in SQL generation? **RQ2**. What is the contribution of each module in DSMR-SQL to its overall effectiveness?

4.1 Experimental Setup

Datasets. In this study, the efficacy of DSMR-SQL was assessed on Spider (Yu et al., 2018), BIRD (Li et al., 2024b), Spider-DK (Gan et al., 2021b), Spider-Realistic (Deng et al., 2021), and Spider-Syn (Gan et al., 2021a). Further details about the datasets are given in Appendix A.

Implementation Details. In this study, Llama-3.2-3B-Instruct (Touvron et al., 2023) and StableCode-3B (Pinnaparaju et al., 2024) were utilized to generate SQL through SFT. Additionally, GPT-4o (Hurst et al., 2024) and Gemini-1.5-Pro (Team et al., 2024) were employed to generate diverse SQL via ICL, which were then used in MRS. Notably, the modular design of DSMR-SQL ensures its adaptability with various LLMs, thereby extending its applicability beyond the models used in this study. Further details can be found in Appendix B.

Evaluation Metrics. In this study, the official evaluation scripts from Spider[1] were used to assess Spider and its variant datasets, which include Execution Accuracy (EX) (Yu et al., 2018) and Test-suite Accuracy (TS)[2] (Zhong et al., 2020). For BIRD, its official evaluation scripts were employed[3], involving EX and Valid Efficiency Score (VES) (Li et al., 2024b). The definitions of these metrics are provided in Appendix C.

4.2 Overall Performance (RQ1)

To validate the efficacy of the proposed DSMR-SQL framework in SQL generation, a detailed evaluation was conducted across multiple datasets. Descriptions of the compared approaches are provided in Appendix D, with performance results detailed in Tables 1 and 2. Notably, DSMR-SQL consistently outperformed other LLM-based approaches across diverse datasets. For instance, DSMR-SQL achieved an EX of 89.7% and a TS of 84.5% on Spider-Dev using GPT-4o, surpassing methods like MCS-SQL (Lee et al., 2024), which generated 100 SQL candidates for each user query.

Additionally, the efficacy of each stage within DSMR-SQL was analyzed in Tables 3 and 4, highlighting the impact of its modular design on overall performance. For instance, during the first stage on Spider-Dev, using GPT-4o for systematic step-by-step reasoning (i.e., $\text{ICL}_{\text{GPT-4o2}}$) yielded the highest individual performance among SQL candidates, with an EX of 87.4% and a TS of 81.2%. Building on this foundation, the incorporation of MRS further improved performance, elevating the EX to 89.7% and the TS to 84.5%. This resulted in an increase of 2.3% in EX and 3.3% in TS compared to the standalone $\text{ICL}_{\text{GPT-4o2}}$. These results underscored the synergistic effectiveness of DSG

[1] https://yale-lily.github.io/spider
[2] TS is not reported for Spider-Test and Spider-DK due to the absence of test suites for these datasets.
[3] https://bird-bench.github.io/

Method	*Spider-Dev*		*Spider-Test*	*BIRD-Dev*	
	EX	*TS*	*EX*	*EX*	*VES*
In-Context Learning					
DIN-SQL + GPT-4 (Pourreza and Rafiei, 2024a)	82.8	74.2	85.3	50.7	58.8
DAIL-SQL + GPT-4 (Gao et al., 2024)	83.5	76.2	86.6	54.8	56.1
DEA-SQL + GPT-4 (Xie et al., 2024b)	85.4	-	87.1	52.4	-
TA-SQL + GPT-4 (Qu et al., 2024)	85.0	-	-	56.2	-
MAG-SQL + GPT-4 (Xie et al., 2024a)	85.3	-	85.6	61.1	-
PTD-SQL + GPT-4 (Luo et al., 2024)	85.7	-	-	57.0	57.7
MAC-SQL + GPT-4 (Wang et al., 2024)	86.8	-	82.8	59.4	66.4
MCS-SQL + GPT-4 (Lee et al., 2024)	89.5	-	89.6	63.4	64.8
PURPLE + GPT-4o (Ren et al., 2024)	87.8	83.3	-	63.0	-
Fine-Tuning					
DTS-SQL (Pourreza and Rafiei, 2024b)	85.5	-	84.4	55.8	60.3
CodeS-15B (Li et al., 2024a)	84.9	79.4	-	58.5	59.9
SENSE-13B (Yang et al., 2024)	84.1	83.5	86.6	55.5	-
Ours (In-Context Learning + Fine-Tuning)					
DSMR-SQL + Gemini-1.5-Pro	89.4	83.7	89.3	66.4	70.9
DSMR-SQL + GPT-4o	**89.7**	**84.5**	**89.7**	**67.2**	**71.8**

Table 1: Performance of different methods on the Spider-Dev, Spider-Test, and BIRD-Dev datasets. Note that "-" indicates that the result was not reported in the original paper.

Method	*Spider-DK*	*Spider-Realistic*		*Spider-Syn*	
	EX	*EX*	*TS*	*EX*	*TS*
SENSE-13B (Yang et al., 2024)	80.2	84.1	76.6	77.6	70.2
CodeS-15B (Li et al., 2024a)	70.7	83.1	75.6	77.0	69.4
PURPLE + GPT-4o (Ren et al., 2024)	75.3	79.9	-	74.0	-
DSMR-SQL + Gemini-1.5-Pro	80.0	86.4	77.8	81.3	72.7
DSMR-SQL + GPT-4o	**80.4**	**87.0**	**78.3**	**82.2**	**73.8**

Table 2: Performance of different methods on the Spider-variant datasets, including Spider-DK, Spider-Realistic, and Spider-Syn. Note that "-" indicates that the result was not reported in the original paper.

and MRS, demonstrating the framework's capacity to enhance SQL generation.

4.3 Analysis of DSMR-SQL (RQ2)

We further analyzed the performance of DSMR-SQL on Spider-Dev, focusing on the contributions of Dual-Strategy SQL Generation (DSG) and Multi-Role SQL Selection (MRS).

4.3.1 Dual-Strategy SQL Generation (DSG)

The performance of DSG was evaluated under varying numbers of SQL candidates and different LLM configurations. The results in Figure 3 and Appendix M revealed the following key insights:

(1) **Impact of the Number of SQL Candidates:** We compared the performance of DSMR-SQL using different numbers of SQL candidates. Note that MRS was not applied when only one SQL was considered. As shown in Figure 3(a), both EX and TS showed steady growth with the increasing number of SQL candidates, showcasing the benefits of integrating multiple SQL within a collaborative framework. Accordingly, DSG effectively mitigated the limitations of single-model settings. Notably, EX improved from 86.5% to 89.7%, and TS increased from 80.2% to 84.5% with the rise in the number of SQL queries. However, the trend of improvement began to plateau as the candidate pool expanded. Based on this observation, five SQL queries for each user question were selected in this study.

(2) **Effect of Dual-Strategy Configuration:** The performance of DSMR-SQL was compared using SFT only, ICL only, and a combination of SFT and ICL. As shown in Figure 3(b), the integration of SFT and ICL achieved the highest performance, with an EX of 88.6% and a TS of 82.9%,

Method			*Spider-Dev*		*Spider-Test*	*BIRD-Dev*	
			EX	***TS***	***EX***	***EX***	***VES***
Stage 1 – DSG	SFT	Llama-3.2-3B-Instruct	84.8	79.5	85.8	57.0	57.8
		StableCode-3B	86.0	80.7	85.1	58.7	60.5
	ICL	$\text{ICL}_{\text{Gemini-1.5-Pro1}}$	87.0	80.4	86.6	62.5	67.4
		$\text{ICL}_{\text{Gemini-1.5-Pro2}}$	86.7	81.1	87.1	63.1	67.9
		$\text{ICL}_{\text{Gemini-1.5-Pro3}}$	86.7	80.7	86.8	63.4	70.2
		$\text{ICL}_{\text{GPT-4o1}}$	87.2	79.7	87.2	63.0	68.0
		$\text{ICL}_{\text{GPT-4o2}}$	87.4	81.2	87.5	63.4	70.4
		$\text{ICL}_{\text{GPT-4o3}}$	87.1	80.0	87.1	63.8	70.6
Stage 2 – MRS	Gemini-1.5-Pro		89.4	83.7	89.3	66.4	70.9
	GPT-4o		**89.7**	**84.5**	**89.7**	**67.1**	**71.8**

Table 3: Experimental results of different modules in DSMR-SQL on the Spider-Dev, Spider-Test, and BIRD-Dev datasets. Note that DSG and MRS indicate Dual-Strategy SQL Generation and Multi-Role SQL Selection, respectively. Moreover, the subscripts in ICL represent three reasoning processes mentioned in DSG for each model.

Method			*Spider-DK*	*Spider-Realistic*		*Spider-Syn*	
			EX	***EX***	***TS***	***EX***	***TS***
Stage 1 – DSG	SFT	Llama-3.2-3B-Instruct	72.5	81.3	76.6	75.3	68.6
		StableCode-3B	75.7	79.9	75.0	74.5	68.8
	ICL	$\text{ICL}_{\text{Gemini-1.5-Pro1}}$	78.5	85.4	75.2	78.6	70.1
		$\text{ICL}_{\text{Gemini-1.5-Pro2}}$	77.4	84.8	76.8	80.2	71.2
		$\text{ICL}_{\text{Gemini-1.5-Pro3}}$	78.7	84.3	76.6	80.5	71.6
		$\text{ICL}_{\text{GPT-4o1}}$	77.6	84.8	74.8	79.4	70.6
		$\text{ICL}_{\text{GPT-4o2}}$	79.6	83.5	75.8	80.9	73.8
		$\text{ICL}_{\text{GPT-4o3}}$	78.7	85.2	74.8	81.4	73.0
Stage 2 – MRS	Gemini-1.5-Pro		80.0	86.4	77.8	81.3	72.7
	GPT-4o		**80.3**	**87.0**	**78.3**	**82.2**	**73.8**

Table 4: Experimental results of different modules in DSMR-SQL on the Spider-DK, Spider-Realistic, and Spider-Syn datasets. Note that DSG and MRS indicate Dual-Strategy SQL Generation and Multi-Role SQL Selection, respectively. Moreover, the subscripts in ICL represent three reasoning processes mentioned in DSG for each model.

surpassing ICL alone by 0.9% in EX and 1.3% in TS. This improvement underscored the synergistic effect of combining both strategies to optimize the generation of diverse SQL candidates by capitalizing on their respective strengths. This dual-strategy setting is also crucial for the subsequent MRS in effectively identifying the optimal SQL.

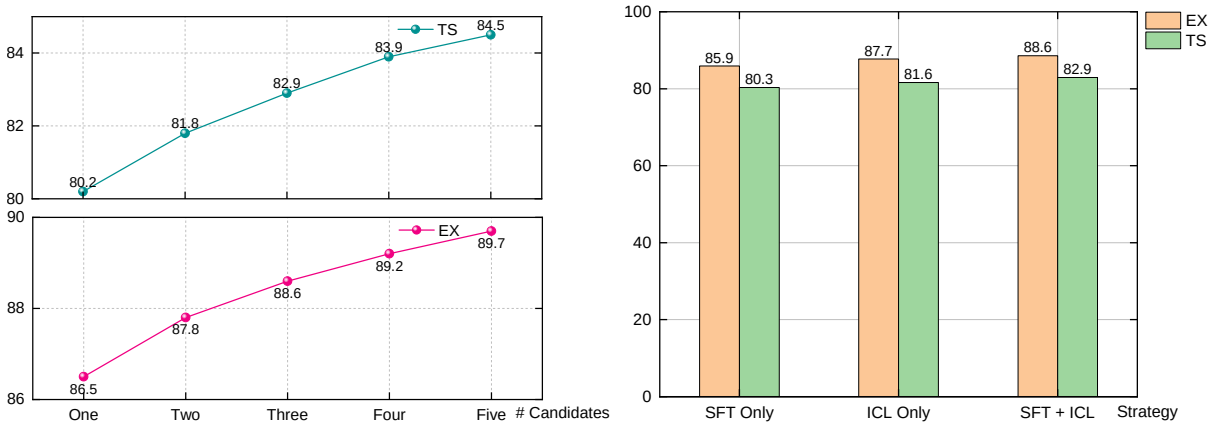
4.3.2 Multi-Role SQL Selection (MRS)

As illustrated in Figure 4, the proposed MRS was compared with Consistency-Voting (CV), Direct SQL Selection (DS), and SQL Selection with Explanation (SE). Specifically, CV selected the SQL candidate via majority voting on execution results across all options. DS employed LLMs to directly identify the most accurate SQL candidate without providing explanations. SE used LLMs to select the optimal SQL, incorporating explanatory reasoning into the selection process. The results in Table 5 indicated that MRS outperformed CV, DS, and SE. For instance, when using GPT-4o, MRS achieved the highest performance, surpassing SE by 0.7% in EX and 1.1% in TS on Spider-Dev. These findings highlighted MRS's capacity to leverage multi-role decision-making processes, facilitating more accurate and reliable identification of the optimal SQL. The detailed prompts regarding CV, DS, and SE are provided in Appendix I, J, and K.

4.3.3 Case Study

To offer a clear illustration of how DSMR-SQL enhances the generation of higher-quality SQL, a case study was conducted on the 463rd question from Spider-Dev, as described in Appendix N.

Specifically, the question required the name and rank points of the winner who won the most times.



(a) Performance comparisons among different numbers of SQL candidates.

(b) Performance comparisons among using SFT only, using ICL only, and using both SFT and ICL.

Figure 3: The performance of DSG with varying numbers of SQL candidates and different strategy settings on the Spider-Dev dataset. Note that the experimental results are based on open-source models and GPT-4o.

Method		*Spider-Dev*	
		EX	***TS***
Gemini-1.5-Pro	CV	87.8	81.6
	DS	88.3	82.5
	SE	88.7	82.8
	MRS	**89.4**	**83.7**
GPT-4o	CV	88.2	82.1
	DS	88.5	82.6
	SE	89.0	83.4
	MRS	**89.7**	**84.5**

Table 5: Results of using Multi-Role SQL Selection (MRS), Consistency-Voting (CV), Direct SQL Selection (DS), and SQL Selection with Explanation (SE) on the Spider-Dev dataset.

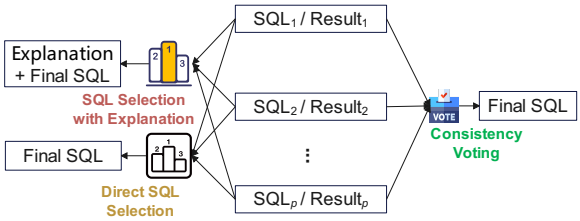


Figure 4: Illustration of Consistency-Voting (CV), Direct SQL Selection (DS), and SQL Selection with Explanation (SE).

To address this, five SQL queries were produced. The first two SQL were generated via SFT, while the remaining three SQL were produced using ICL.

(1) The first two SQL queries correctly retrieved the winner name and rank points by grouping records based on the winner ID or winner name, which aligned with the query intent.

(2) The third and fourth SQL candidates deviated from the user intent by unnecessarily separating the winner's first and last names, which introduced irrelevant complexity.

(3) The fifth SQL grouped records by both the winner name and rank points. However, grouping by rank points risked producing incorrect results if a player's rank points varied across matches. This fragmentation would split the total win counts into separate groups based on different rank points.

Notably, simple SQL selection strategies (i.e., CV, DS, and SE) favored the overly complex third and fourth SQL candidates. In contrast, MRS employed the collaboration of various roles to comprehensively analyze the provided SQL queries. Through its iterative reasoning process, MRS ultimately identified the SQL query that most accurately aligned with the user intent.

5 Conclusion

This study introduces DSMR-SQL, a SQL generation framework leveraging dual-strategy reasoning and multi-role SQL selection. Experiments on several datasets demonstrate the framework's superior performance, and extensive analyses were conducted to further validate its effectiveness. Overall, this study presents a pragmatic solution for improving model reasoning and accurately selecting the optimal SQL. It creates opportunities for researchers in this field to further explore and refine SQL generation techniques.

Limitations

Despite the promising results achieved by DSMR-SQL, there are several limitations that deserve further attention. (**1**) **Human Reasoning Processes:** While our framework partially mimics human reasoning behaviors, significant gaps remain in fully understanding and incorporating advanced human reasoning processes into SQL generation. In-depth research into these processes could be crucial for enhancing the model's performance; (**2**) **Candidate Coverage:** In this study, only five SQL candidates are used for selection. However, this restricted number might fail to sufficiently encompass the full SQL query space, particularly in complex datasets like BIRD. Therefore, dynamically adjusting the number of SQL candidates to optimize SQL generation performance offers a compelling avenue for future research.

Ethical Considerations

In this study, the efficacy of the proposed DSMR-SQL is assessed using both open-source and closed-source LLMs. It is important to highlight that these models demand substantial computational resources, which leads to a rise in carbon dioxide emissions and high energy consumption. Besides, the datasets we utilized in this study are openly available, and there are no data privacy concerns.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Furui Cheng, Vilém Zouhar, Simran Arora, Mrinmaya Sachan, Hendrik Strobelt, and Mennatallah El-Assady. 2024. Relic: Investigating large language model responses using self-consistency. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–18.

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.

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

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

Yuankai Fan, Zhenying He, Tonghui Ren, Can Huang, Yinan Jing, Kai Zhang, and X. Sean Wang. 2024. Metasql: A generate-then-rank framework for natural language to SQL translation. In *40th IEEE International Conference on Data Engineering, ICDE 2024*, pages 1765–1778.

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

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.

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. In *International Conference on Very Large Data Bases (VLDB)*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Myeongjun Jang and Thomas Lukasiewicz. 2023. Consistency analysis of chatgpt. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 15970–15985.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Wenzhao Jiang, Jindong Han, Hao Liu, Tao Tao, Naiqiang Tan, and Hui Xiong. 2024b. Interpretable cascading mixture-of-experts for urban traffic congestion prediction. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5206–5217.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2024. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. *arXiv preprint arXiv:2405.07467*.

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, et al. 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.

Zhenwen Li and Tao Xie. 2024. Using llm to select the right sql query from candidates. *arXiv preprint arXiv:2401.02115*.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2023. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*.

Bill Yuchen Lin, Yicheng Fu, Karina Yang, Faeze Brahman, Shiyu Huang, Chandra Bhagavatula, Prithviraj Ammanabrolu, Yejin Choi, and Xiang Ren. 2024. Swiftsage: A generative agent with fast and slow thinking for complex interactive tasks. *Advances in Neural Information Processing Systems*, 36.

Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S Yu. 2023. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability. *arXiv preprint arXiv:2303.13547*.

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? *arXiv preprint arXiv:2408.05109*.

Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8086–8098.

Ruilin Luo, Liyuan Wang, Binghuai Lin, Zicheng Lin, and Yujiu Yang. 2024. Ptd-sql: Partitioning and targeted drilling with llms in text-to-sql. *arXiv preprint 2409.14082*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Wenxin Mao, Ruiqi Wang, Jiyu Guo, Jichuan Zeng, Cuiyun Gao, Peiyi Han, and Chuanyi Liu. 2024. Enhancing text-to-sql parsing through question rewriting and execution-guided refinement. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 2009–2024.

Dinesh Mendhe, Akriti Dogra, Prabha Shreeraj Nair, S Punitha, KS Preetha, and S BG Tilak Babu. 2024. Ai-enabled data-driven approaches for personalized medicine and healthcare analytics. In *2024 Ninth International Conference on Science Technology Engineering and Mathematics (ICONSTEM)*, pages 1–5. IEEE.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, et al. 2024. Stable code technical report. *arXiv preprint arXiv:2404.01226*.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint 2410.01943*.

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

Mohammadreza Pourreza and Davood Rafiei. 2024b. Dts-sql: Decomposed text-to-sql with small large language models. *arXiv preprint arXiv:2402.01117*.

Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. 2024. Mutual reasoning makes smaller llms stronger problem-solvers. *arXiv preprint arXiv:2408.06195*.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! A novel and effective strategy for mitigating hallucinations in text-to-sql generation. In *Findings of the Association for Computational Linguistics, ACL 2024*, pages 5456–5471.

Tonghui Ren, Yuankai Fan, Zhenying He, Ren Huang, Jiaqi Dai, Can Huang, Yinan Jing, Kai Zhang, Yifan Yang, and Xiaoyang Sean Wang. 2024. Purple:

Making a large language model a better sql writer. *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 15–28.

Matthew Renze and Erhan Guven. 2024. The effect of sampling temperature on problem solving in large language models. *arXiv preprint arXiv:2402.05201*.

Lin Shi, Chiyu Ma, Wenhua Liang, Weicheng Ma, and Soroush Vosoughi. 2024. Judging the judges: A systematic investigation of position bias in pairwise comparative assessments by llms. *arXiv preprint arXiv:2406.07791*.

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*.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel Bowman. 2024. Language models don't always say what they think: unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems*, 36.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. 2024. Mac-sql: A multi-agent collaborative framework for text-to-sql. *arXiv preprint arXiv:2312.11242*.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhifang Sui. 2023. Large language models are not fair evaluators. *arXiv preprint arXiv:2305.17926*.

Wenxuan Xie, Gaochen Wu, and Bowen Zhou. 2024a. Mag-sql: Multi-agent generative approach with soft schema linking and iterative sub-sql refinement for text-to-sql. *arXiv preprint arXiv:2408.07930*.

Yuanzhen Xie, Xinzhou Jin, Tao Xie, Matrixmxlin Matrixmxlin, Liang Chen, Chenyun Yu, Cheng Lei, Chengxiang Zhuo, Bo Hu, and Zang Li. 2024b. Decomposition for enhancing attention: Improving llm-based text-to-sql through workflow paradigm. In *Findings of the Association for Computational Linguistics, ACL 2024*, pages 10796–10816.

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.

Zhangyue Yin, Qiushi Sun, Qipeng Guo, Jiawen Wu, Xipeng Qiu, and Xuan-Jing Huang. 2023. Do large language models know what they don't know? In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 8653–8665.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 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.

Hengran Zhang, Ruqing Zhang, Jiafeng Guo, Maarten de Rijke, Yixing Fan, and Xueqi Cheng. 2024b. Are large language models good at utility judgments? In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1941–1951.

Li Zheng, Hao Fei, Fei Li, Bobo Li, Lizi Liao, Donghong Ji, and Chong Teng. 2024. Reverse multi-choice dialogue commonsense inference with graph-of-thought. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19688–19696.

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

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*.

Pan Zhou, Xingyu Xie, Zhouchen Lin, and Shuicheng Yan. 2024. Towards understanding convergence and generalization of adamw. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

A Appendix A: Description of the Experimental Datasets

This section provides detailed information on the datasets used in this study. Additional details on database and query complexity are presented in Table 6 (Liu et al., 2024). These datasets are publicly available under the CC BY-SA 4.0 license, permitting modifications and inclusion of additional annotations on the original datasets.

Spider: Spider (Yu et al., 2018) serves as a comprehensive dataset, comprising 11840 NL questions in English and 6448 unique SQL queries across 138 distinct domains. It contains 8659 samples in the training set, 1034 in the development set, and 2147 in the test set[4].

Spider-Variant Datasets: Spider-DK (Gan et al., 2021b) highlights the role of domain-specific knowledge in SQL generation with 535 samples in English[5]. Spider-Realistic (Deng et al., 2021) focuses on challenging queries that omit explicit column references, which includes 508 samples in English[6]. Spider-Syn (Gan et al., 2021a) is a variant of Spider, in which schema-specific terms in NL queries are replaced with synonyms. It provides 1034 samples for model evaluation in English[7].

BIRD: BIRD (Li et al., 2024b) has gained widespread attention for incorporating additional complexities, such as complex SQL functions and operations not found in Spider. It contains 9428 training samples and 1534 samples in the development set. The NL queries in BIRD are in English[8].

B Appendix B: Description of the Implementation Details

In this study, DSMR-SQL was implemented using PyTorch (Paszke et al., 2019). Besides, SFT and ICL were combined to improve SQL generation.

(1) **SFT:** Specifically, Llama-3.2-3B-Instruct[9] (Touvron et al., 2023) and StableCode-3B[10] (Pinnaparaju et al., 2024) were utilized for SQL generation via SFT. The former excels in query comprehension, while the latter specializes in code generation. Moreover, the CodeS framework (Li et al., 2024a) was employed to perform SFT, with the learning rate, batch size, and number of epochs set to 5×10^{-6}, 2, and 4, respectively. In addition, the AdamW optimizer (Zhou et al., 2024) with momentum parameters $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 1 \times 10^{-8}$ was adopted. Other parameter settings remained the same as in the original CodeS[11]. All experiments in SFT were conducted on four NVIDIA GeForce A100 40GB GPUs.

(2) **ICL:** Specifically, GPT-4o (Hurst et al., 2024) and Gemini-1.5-Pro (Team et al., 2024) were employed to generate diverse SQL queries via ICL[12]. Notably, two SQL candidates generated by the open-source LLMs and three SQL candidates generated by GPT-4o (or Gemini-1.5-Pro) were adopted for MRS within GPT-4o (or Gemini-1.5-Pro). Besides, the number of chat completion choices (n) and the temperature (T) were set to 1 and 0, respectively. The number of SQL candidates was determined based on the experiments in Section 4.3.1(1). It is worth noting that the modular design of DSMR-SQL enables the utilization of different numbers of open-source and closed-source LLMs for multiple SQL generation.

C Appendix C: Description of the Evaluation Metrics

Evaluation metrics are crucial for gauging the efficacy of text-to-SQL systems from a quantitative perspective. In this study, we used the following metrics to assess the model's performance.

(1) **Execution Accuracy (EX)** (Yu et al., 2018) assesses the execution outcomes of predicted SQL queries against the ground truth. This metric is valuable for verifying the functional correctness of SQL queries.

(2) **Test-suite Accuracy (TS)** (Zhong et al., 2020) is proposed to gauge the semantic correctness of text-to-SQL systems by constructing a compact test suite from a substantial collection of databases. This approach enables the differentiation between fully correct and nearly correct SQL queries. During the evaluation phase, TS measures the model's ability to correctly execute SQL queries across these databases, thereby establishing a stringent upper bound for semantic accuracy.

(3) **Valid Efficiency Score (VES)** (Li et al., 2024b) quantifies the execution efficiency of SQL queries by simultaneously considering the accuracy and execution efficiency of SQL outputs.

[4] https://yale-lily.github.io/spider
[5] https://github.com/ygan/Spider-DK
[6] https://zenodo.org/records/5205322#.YTts_o5Kgab
[7] https://github.com/ygan/Spider-Syn
[8] https://bird-bench.github.io/
[9] https://hf-mirror.com/meta-llama/Llama-3.2-3B-Instruct
[10] https://hf-mirror.com/stabilityai/stable-code-3b
[11] https://github.com/RUCKBReasoning/codes
[12] We utilized this website to employ closed-source LLMs: https://gpt.zhizengzeng.com/

Dataset	*Database Complexity*		*Query Complexity*			
	# Databases	*# Tables*	*# Tables*	*# Selects*	*# Agg*	*# Math Comp*
Spider	206	1056	1.83	1.17	0.54	0
Spider-DK	169	887	1.71	1.16	0.54	0
Spider-Realistic	166	876	1.79	1.21	0.50	0
Spider-Syn	166	876	1.68	1.17	0.59	0
BIRD	80	611	2.07	1.09	0.61	0.27

Table 6: Additional information of the experimental datasets. Note that *# Agg* and *# Math Comp* indicate the number of aggregation functions and mathematical computations, respectively.

D Appendix D: Description of the Compared Methods

We compared various LLM-based SQL generation methods in the literature, which are divided into two categories.

(1) **In-Context Learning-based Approaches:**

- DIN-SQL (Pourreza and Rafiei, 2024a) segmented the SQL generation process into four distinct modules, namely schema linking, classification and decomposition, SQL generation, and self-correction.
- DAIL-SQL (Gao et al., 2024) incorporated structural knowledge through skeleton similarity-based few-shot prompt selection and improved reasoning efficiency by restricting cross-domain specific terms in the representation.
- DEA-SQL (Xie et al., 2024b) implemented a structured workflow for SQL generation, involving gathering database information, identifying query types, devising solution strategies, generating SQL syntax, conducting initial self-checks, and reviewing past errors to mitigate repetitive mistakes.
- TA-SQL (Qu et al., 2024) introduced Task Alignment (TA) to reduce model hallucinations.
- MAG-SQL (Xie et al., 2024a) adopted the Least-to-Most Prompting approach (Zhou et al., 2022), incrementally generating each sub-question by adding conditions to the previous one.
- PTD-SQL (Luo et al., 2024) utilized query group partitioning to strengthen LLM's reasoning abilities.
- MAC-SQL (Wang et al., 2024) introduced a novel LLM-based multi-agent collaborative framework to enhance tool utilization and agent collaboration, involving the Selector, Decomposer, and Refiner.
- MCS-SQL (Lee et al., 2024) generated various candidate SQL using diverse prompts, followed by filtering and multiple-choice selection to finalize the output.
- PURPLE (Ren et al., 2024) adopted trained classifiers to link questions with database schemas and reduced irrelevant information via pattern pruning, thereby improving schema linking efficiency in SQL generation.

(2) **Fine-Tuning-based Approaches:**

- DTS-SQL (Pourreza and Rafiei, 2024b) was composed of two sub-tasks, namely schema linking and SQL generation. A two-stage fine-tuning approach was implemented to effectively align the performance of the open-source LLM with that of the closed-source LLM.
- CodeS (Li et al., 2024a) built a BM25 index from database content, retrieving top-k values from relevant columns to enhance semantic representation for SQL generation. Moreover, it introduced a bi-directional data augmentation approach to automatically generate a diverse set of (NL, SQL) pairs.
- SENSE (Yang et al., 2024) proposed a synthetic data approach that combines strong data from larger and high-performing models with weak data produced by smaller and less well-aligned models.

E Appendix E: Prompt for Direct SQL Generation

Prompt for Direct SQL Generation

From now on, you are an excellent Database Analyst, who has expertise in understanding complex database schemas, tables, relationships, and can perform detailed analyses to extract required information. Answer the following question while staying in strict accordance with the nature of the provided identity.

[**Instructions**]
Please directly generate the SQL queries according to the provided evidence and the following [**Hints**] with no explanations.

[**Hints**]
- SELECT: Only select columns explicitly mentioned in the user's question. Avoid unnecessary columns or values.
- Capitalization: Pay closer attention to the capitalization in the question. You MUST maintain the original capitalization from the question in the generated SQL!
- FROM/JOIN: Only include tables which are explicitly mentioned in the question. You MUST NOT use LEFT JOIN!
- Fuzzy matching (LIKE): Used to match similar strings.
- Exact matching (=): Used for precise matching.
- INTERSECT: Used to obtain the intersection of two query results.
- UNION: Used to merge two query results.
- Never use 'ǁ' or any other method to concatenate strings in the 'SELECT' clause. Rather output the columns as they are.
- BETWEEN ... AND ...: If the question requires selecting values within a specified range, you should use 'BETWEEN ... AND ...' operator.
- ASC and DESC: If the question does not explicitly mention specific order such as ascending or descending order, you MUST NOT add 'ASC' or 'DESC' in SQL queries.
- Column and Table Selection: Remember to only include columns and tables explicitly requested in the query. Remember to select the most correct tables in the 'FROM' clause.
- Aggregation: When using 'MAX' or 'MIN', please perform joins before selecting these aggregates. DO NOT use 'MAX' or 'MIN' in the 'WHERE' clause! Always use the 'ORDER BY' clause in combination with 'LIMIT X'.
- GROUP BY: When the question does not explicitly and clearly perform calculations on grouped data, you MUST NOT add 'GROUP BY' clause.
- DISTINCT: When the question does not explicitly and clearly search for unique values from a column, you should not add 'DISTINCT' clause.
- Order and Grouping: Use 'GROUP BY <column>' before 'ORDER BY <column> ASC/DESC' to ensure distinct grouped results. Pay more attention to the content after 'ORDER BY'.
- LIMIT: Apply 'LIMIT' only when the question specifies a retrieval count.
- WHERE: Double-check that all 'WHERE' clauses accurately represent the conditions needed to filter the data as per the question's requirements.

Some example questions and corresponding SQL queries are provided based on similar problems:
{**Examples**}

Sqlite SQL tables, with their properties:
{**Database Schema**}

Here are some data information about database references:

{**Example Database Values**}

Foreign key information of Sqlite SQL tables, used for table joins:
{**Foreign Key Information**}

Some evidence can help you generate high-quality SQL:
{**Evidence (Only in the BIRD dataset)**}

The input question is:
{**Question**}

Please read the given examples, the input question, the [**Hints**], and the database schema again. Let's take a deep breath and think step by step, you must think more steps. Please remember to stay in strict accordance with the nature of the provided identity. If you can generate high-quality SQL queries satisfying the question demands, I will give you one million dollars.

F Appendix F: Prompt for Strict Step-by-Step Reasoning

Prompt for Strict Step-by-Step Reasoning

From now on, you are an excellent Database Analyst, who has expertise in understanding complex database schemas, tables, relationships, and can perform detailed analyses to extract required information. Answer the following question while staying in strict accordance with the nature of the provided identity.

[**Instructions**]
Please strictly obey the following steps to generate high-quality SQL queries:

[**Steps**]
- Step 1: Read the Question: You need to understand the primary focus and specific details of the question. Ensure each part of the SQL statement aligns with the user's query intent.
- Step 2: Analyze the Database Schema: Understand the relation between the database and the question accurately.
- Step 3: Analyze the Given Examples: Understand the similar examples provided to you for better SQL generation.
- Step 4: Analyze the Given Evidence: Understand the given evidence provided to you below for better SQL generation.
- Step 5: Analyze the Given Hints: Understand all the given hints provided to you below for better SQL generation.
- Step 6: Finalize the SQL query: Construct correct SQLite SQL corresponding to the given question.
- Step 7: Validation and Syntax Check: Before finalizing, verify that generated SQL query is coherent with the database schema, all referenced columns exist in the referenced table, all joins are correctly formulated, aggregation logic is accurate, and the SQL syntax is correct.

You need to explain your detailed reasoning step by step. Note that you MUST obey the following [**Hints**] to generate SQL queries.

[**Hints**]
- SELECT: Only select columns explicitly mentioned in the user's question. Avoid unnecessary columns or values.

- Capitalization: Pay closer attention to the capitalization in the question. You MUST maintain the original capitalization from the question in the generated SQL!
- FROM/JOIN: Only include tables which are explicitly mentioned in the question. You MUST NOT use LEFT JOIN!
- Fuzzy matching (LIKE): Used to match similar strings.
- Exact matching (=): Used for precise matching.
- INTERSECT: Used to obtain the intersection of two query results.
- UNION: Used to merge two query results.
- Never use 'll' or any other method to concatenate strings in the 'SELECT' clause. Rather output the columns as they are.
- BETWEEN ... AND ...: If the question requires selecting values within a specified range, you should use 'BETWEEN ... AND ...' operator.
- ASC and DESC: If the question does not explicitly mention specific order such as ascending or descending order, you MUST NOT add 'ASC' or 'DESC' in SQL queries.
- Column and Table Selection: Remember to only include columns and tables explicitly requested in the query. Remember to select the most correct tables in the 'FROM' clause.
- Aggregation: When using 'MAX' or 'MIN', please perform joins before selecting these aggregates. DO NOT use 'MAX' or 'MIN' in the 'WHERE' clause! Always use the 'ORDER BY' clause in combination with 'LIMIT X'.
- GROUP BY: When the question does not explicitly and clearly perform calculations on grouped data, you MUST NOT add 'GROUP BY' clause.
- DISTINCT: When the question does not explicitly and clearly search for unique values from a column, you should not add 'DISTINCT' clause.
- Order and Grouping: Use 'GROUP BY <column>' before 'ORDER BY <column> ASC/DESC' to ensure distinct grouped results. Pay more attention to the content after 'ORDER BY'.
- LIMIT: Apply 'LIMIT' only when the question specifies a retrieval count.
- WHERE: Double-check that all 'WHERE' clauses accurately represent the conditions needed to filter the data as per the question's requirements.

Some example questions and corresponding SQL queries are provided based on similar problems:
{**Examples**}

Sqlite SQL tables, with their properties:
{**Database Schema**}

Here are some data information about database references:
{**Example Database Values**}

Foreign key information of Sqlite SQL tables, used for table joins:
{**Foreign Key Information**}

Some evidence can help you generate high-quality SQL:
{**Evidence (Only in the BIRD dataset)**}

The input question is:
{**Question**}

Please read the given examples, the input question, the [**Hints**], and the database schema again. Let's take a deep breath and think step by step, you must think more steps. Please remember to stay in strict accordance with the nature of the provided identity. If you can generate high-quality

SQL queries satisfying the question demands, I will give you one million dollars.

G Appendix G: Prompt for Flexible Reasoning

Prompt for Flexible Reasoning

From now on, you are an excellent Database Analyst, who has expertise in understanding complex database schemas, tables, relationships, and can perform detailed analyses to extract required information. Answer the following question while staying in strict accordance with the nature of the provided identity.

[**Instructions**]
Please generate high-quality SQL queries with your detailed reasoning. Note that you MUST refer to the provided evidence and obey the following [**Hints**] to generate SQL queries.

[**Hints**]
- SELECT: Only select columns explicitly mentioned in the user's question. Avoid unnecessary columns or values.
- Capitalization: Pay closer attention to the capitalization in the question. You MUST maintain the original capitalization from the question in the generated SQL!
- FROM/JOIN: Only include tables which are explicitly mentioned in the question. You MUST NOT use LEFT JOIN!
- Fuzzy matching (LIKE): Used to match similar strings.
- Exact matching (=): Used for precise matching.
- INTERSECT: Used to obtain the intersection of two query results.
- UNION: Used to merge two query results.
- Never use 'ǁ' or any other method to concatenate strings in the 'SELECT' clause. Rather output the columns as they are.
- BETWEEN ... AND ...: If the question requires selecting values within a specified range, you should use 'BETWEEN ... AND ...' operator.
- ASC and DESC: If the question does not explicitly mention specific order such as ascending or descending order, you MUST NOT add 'ASC' or 'DESC' in SQL queries.
- Column and Table Selection: Remember to only include columns and tables explicitly requested in the query. Remember to select the most correct tables in the 'FROM' clause.
- Aggregation: When using 'MAX' or 'MIN', please perform joins before selecting these aggregates. DO NOT use 'MAX' or 'MIN' in the 'WHERE' clause! Always use the 'ORDER BY' clause in combination with 'LIMIT X'.
- GROUP BY: When the question does not explicitly and clearly perform calculations on grouped data, you MUST NOT add 'GROUP BY' clause.
- DISTINCT: When the question does not explicitly and clearly search for unique values from a column, you should not add 'DISTINCT' clause.
- Order and Grouping: Use 'GROUP BY <column>' before 'ORDER BY <column> ASC/DESC' to ensure distinct grouped results. Pay more attention to the content after 'ORDER BY'.
- LIMIT: Apply 'LIMIT' only when the question specifies a retrieval count.
- WHERE: Double-check that all 'WHERE' clauses accurately represent the conditions needed to filter the data as per the question's requirements.

Some example questions and corresponding SQL queries are provided based on similar problems:
{**Examples**}

Sqlite SQL tables, with their properties:

{**Database Schema**}

Here are some data information about database references:
{**Example Database Values**}

Foreign key information of Sqlite SQL tables, used for table joins:
{**Foreign Key Information**}

Some evidence can help you generate high-quality SQL:
{**Evidence (Only in the BIRD dataset)**}

The input question is:
{**Question**}

Please read the given examples, the input question, the [**Hints**], and the database schema again. Let's take a deep breath and think step by step, you must think more steps. Please remember to stay in strict accordance with the nature of the provided identity. If you can generate high-quality SQL queries satisfying the question demands, I will give you one million dollars.

H Appendix H: Prompt for Multi-Role SQL Selection

Prompt for Multi-Role SQL Selection

You are an AI language model employing iterative reasoning through three distinct roles, each encapsulated within specific XML tags:
<proposer>...</proposer>
<critic>...</critic>
<summarizer>...</summarizer>

[**Roles and Responsibilities**]
<proposer>
1. Objective: Select the best SQL query to answer the question based on the given SQL and their corresponding execution results. Propose one or more reasoning steps towards solving the given problem.
2. Instructions:
(1) Generate clear and concise propositions that advance the reasoning process.
(2) Build upon previous valid propositions and consider any critiques provided.
<critic>
1. Objective: Critically evaluate the proposer's reasoning steps and the selection of the best SQL based on the execution results.
2. Instructions:
(1) Analyze the propositions for logical consistency and accuracy.
(2) Provide detailed natural language critiques highlighting any errors or areas for improvement.
<summarizer>
1. Objective: Synthesize the agreed propositions and the corresponding feedback, outputting the final SQL solution.
2. Instructions:
(1) Review the agreed propositions and critiques.
(2) Extract and organize the valid reasoning steps.
(3) Present the final answer.

[**Process Flow**]
1. Iteration Begins: The <proposer> presents one or more reasoning steps to select the best SQL.
2. Critical Evaluation: The <critic> analyzes these steps, providing natural language critiques and suggesting refinements.
3. Assessment and Synthesis: The <summarizer> consolidates the agreed propositions and critiques to output the final selected SQL.
4. Repeat: This cycle continues, with the <proposer> refining or adding propositions based on the <critic>'s feedback, until the <proposer> and the <critic> agree on the propositions.

[**Formatting Guidelines**]
1. Clarity: Ensure each reasoning step and critique is easy to understand.
2. Logical Progression: Each proposition should logically follow from previous ones, considering any critiques.
3. Tags: Always encapsulate your output within the correct XML tags.
4. Natural Language: Use detailed explanations in critiques to provide meaningful feedback.
5. Note that the content in the <summarizer> is the final SQL only with no explanations.
You MUST adhere to the following format to output results!!!

[**Example Interaction**]
<proposer>
[Proposer's reasoning step 1]
</proposer>
<critic>
[Critic's detailed natural language critique 1]
</critic>
<proposer>
[Proposer's reasoning step 2]
</proposer>
<critic>
[Critic's detailed natural language critique 2]
</critic>
... (continue)
<summarizer>
[Final selected SQL only]
</summarizer>

Note that you MUST obey the following [**Hints**] to select SQL queries.

[**Hints**]
- SELECT: Only select columns explicitly mentioned in the user's question. Avoid unnecessary columns or values.
- Capitalization: Pay closer attention to the capitalization in the question. You MUST maintain the original capitalization from the question in the generated SQL!
- FROM/JOIN: Only include tables which are explicitly mentioned in the question. You MUST NOT use LEFT JOIN!
- Fuzzy matching (LIKE): Used to match similar strings.
- Exact matching (=): Used for precise matching.
- INTERSECT: Used to obtain the intersection of two query results.
- UNION: Used to merge two query results.
- Never use 'll' or any other method to concatenate strings in the 'SELECT' clause. Rather output

the columns as they are.
- BETWEEN ... AND ...: If the question requires selecting values within a specified range, you should use 'BETWEEN ... AND ...' operator.
- ASC and DESC: If the question does not explicitly mention specific order such as ascending or descending order, you MUST NOT add 'ASC' or 'DESC' in SQL queries.
- Column and Table Selection: Remember to only include columns and tables explicitly requested in the query. Remember to select the most correct tables in the 'FROM' clause.
- Aggregation: When using 'MAX' or 'MIN', please perform joins before selecting these aggregates. DO NOT use 'MAX' or 'MIN' in the 'WHERE' clause! Always use the 'ORDER BY' clause in combination with 'LIMIT X'.
- GROUP BY: When the question does not explicitly and clearly perform calculations on grouped data, you MUST NOT add 'GROUP BY' clause.
- DISTINCT: When the question does not explicitly and clearly search for unique values from a column, you should not add 'DISTINCT' clause.
- Order and Grouping: Use 'GROUP BY <column>' before 'ORDER BY <column> ASC/DESC' to ensure distinct grouped results. Pay more attention to the content after 'ORDER BY'.
- LIMIT: Apply 'LIMIT' only when the question specifies a retrieval count.
- WHERE: Double-check that all 'WHERE' clauses accurately represent the conditions needed to filter the data as per the question's requirements.

The input question is:
{**Question**}

The candidate SQL are:
Candidate SQL 1: {**Candidate SQL1**}. Execution result 1: {**Execution Result1 from Candidate SQL1**}.
Candidate SQL 2: {**Candidate SQL2**}. Execution result 2: {**Execution Result2 from Candidate SQL2**}.
Candidate SQL 3: {**Candidate SQL3**}. Execution result 3: {**Execution Result3 from Candidate SQL3**}.
Candidate SQL 4: {**Candidate SQL4**}. Execution result 4: {**Execution Result4 from Candidate SQL4**}.
Candidate SQL 5: {**Candidate SQL5**}. Execution result 5: {**Execution Result5 from Candidate SQL5**}.

Please read the input question, the given [**Hints**], the candidate SQL and corresponding execution results again. Let's take a deep breath and think step by step, you must think more steps. Please remember to stay in strict accordance with the nature of the provided identity. If you can select the best SQL satisfying the question demands, I will give you one million dollars.

I Appendix I: Prompt for Consistency Voting

Prompt for Consistency Voting

From now on, you are an excellent Database Analyst, who has expertise in understanding complex database schemas, tables, relationships, and can perform detailed analyses to extract required information. Answer the following question while staying in strict accordance with the nature of the provided identity.

[**Instructions**]
You have been provided with several responses to the latest user query and their execution results.

Your task is to select the SQL query with the highest execution result consistency, which means the SQL execution result appears the most frequently.
You need to explain your detailed reasoning step by step. You MUST adhere to the following format to output results.

[**Example Interaction**]
<Reasoning>
[Detailed reasoning process]
</Reasoning>
<SQL>
[Final selected SQL]
</SQL>

Input question is:
{**Question**}.

The candidate SQL are:
Candidate SQL 1: {**Candidate SQL1**}. Execution result 1: {**Execution Result1 from Candidate SQL1**}.
Candidate SQL 2: {**Candidate SQL2**}. Execution result 2: {**Execution Result2 from Candidate SQL2**}.
Candidate SQL 3: {**Candidate SQL3**}. Execution result 3: {**Execution Result3 from Candidate SQL3**}.
Candidate SQL 4: {**Candidate SQL4**}. Execution result 4: {**Execution Result4 from Candidate SQL4**}.
Candidate SQL 5: {**Candidate SQL5**}. Execution result 5: {**Execution Result5 from Candidate SQL5**}.

Please read the given candidate SQL and corresponding execution results again. Let's take a deep breath and think step by step, you must think more steps. Please remember to stay in strict accordance with the nature of the provided identity. If you can complete this task well, I will give you one million dollars.

J Appendix J: Prompt for Direct SQL Selection

Prompt for Direct SQL Selection

From now on, you are an excellent Database Analyst, who has expertise in understanding complex database schemas, tables, relationships, and can perform detailed analyses to extract required information. Answer the following question while staying in strict accordance with the nature of the provided identity.

[**Instructions**]
You have been provided with several responses to the latest user query and their execution results. Your task is to select the most suitable SQL query satisfying user needs. You only need to output SQL only with no explanations.

Note that you MUST obey the following [**Hints**] to select SQL queries.

[**Hints**]
- SELECT: Only select columns explicitly mentioned in the user's question. Avoid unnecessary

columns or values.
- Capitalization: Pay closer attention to the capitalization in the question. You MUST maintain the original capitalization from the question in the generated SQL!
- FROM/JOIN: Only include tables which are explicitly mentioned in the question. You MUST NOT use LEFT JOIN!
- Fuzzy matching (LIKE): Used to match similar strings.
- Exact matching (=): Used for precise matching.
- INTERSECT: Used to obtain the intersection of two query results.
- UNION: Used to merge two query results.
- Never use 'ǁ' or any other method to concatenate strings in the 'SELECT' clause. Rather output the columns as they are.
- BETWEEN ... AND ...: If the question requires selecting values within a specified range, you should use 'BETWEEN ... AND ...' operator.
- ASC and DESC: If the question does not explicitly mention specific order such as ascending or descending order, you MUST NOT add 'ASC' or 'DESC' in SQL queries.
- Column and Table Selection: Remember to only include columns and tables explicitly requested in the query. Remember to select the most correct tables in the 'FROM' clause.
- Aggregation: When using 'MAX' or 'MIN', please perform joins before selecting these aggregates. DO NOT use 'MAX' or 'MIN' in the 'WHERE' clause! Always use the 'ORDER BY' clause in combination with 'LIMIT X'.
- GROUP BY: When the question does not explicitly and clearly perform calculations on grouped data, you MUST NOT add 'GROUP BY' clause.
- DISTINCT: When the question does not explicitly and clearly search for unique values from a column, you should not add 'DISTINCT' clause.
- Order and Grouping: Use 'GROUP BY <column>' before 'ORDER BY <column> ASC/DESC' to ensure distinct grouped results. Pay more attention to the content after 'ORDER BY'.
- LIMIT: Apply 'LIMIT' only when the question specifies a retrieval count.
- WHERE: Double-check that all 'WHERE' clauses accurately represent the conditions needed to filter the data as per the question's requirements.

The input question is:
{**Question**}.

The candidate SQL are:
Candidate SQL 1: {**Candidate SQL1**}. Execution result 1: {**Execution Result1 from Candidate SQL1**}.
Candidate SQL 2: {**Candidate SQL2**}. Execution result 2: {**Execution Result2 from Candidate SQL2**}.
Candidate SQL 3: {**Candidate SQL3**}. Execution result 3: {**Execution Result3 from Candidate SQL3**}.
Candidate SQL 4: {**Candidate SQL4**}. Execution result 4: {**Execution Result4 from Candidate SQL4**}.
Candidate SQL 5: {**Candidate SQL5**}. Execution result 5: {**Execution Result5 from Candidate SQL5**}.

Please read the input question, the [**Hints**], the candidate SQL and corresponding execution results again. Please remember to stay in strict accordance with the nature of the provided identity. If you can select the best SQL satisfying the question demands, I will give you one million dollars.

K Appendix K: Prompt for SQL Selection with Explanation

Prompt for SQL Selection with Explanation

From now on, you are an excellent Database Analyst, who has expertise in understanding complex database schemas, tables, relationships, and can perform detailed analyses to extract required information. Answer the following question while staying in strict accordance with the nature of the provided identity.

[**Instructions**]
You have been provided with several responses to the latest user query and their execution results. Your task is to select the most suitable SQL query satisfying user needs.
You need to explain your detailed reasoning step by step. You MUST adhere to the following format to output results.

[**Example Interaction**]
<Reasoning>
[Detailed reasoning process]
</Reasoning>
<SQL>
[Final selected SQL]
</SQL>

Note that you MUST obey the following [**Hints**] to select SQL queries.

[**Hints**]
- SELECT: Only select columns explicitly mentioned in the user's question. Avoid unnecessary columns or values.
- Capitalization: Pay closer attention to the capitalization in the question. You MUST maintain the original capitalization from the question in the generated SQL!
- FROM/JOIN: Only include tables which are explicitly mentioned in the question. You MUST NOT use LEFT JOIN!
- Fuzzy matching (LIKE): Used to match similar strings.
- Exact matching (=): Used for precise matching.
- INTERSECT: Used to obtain the intersection of two query results.
- UNION: Used to merge two query results.
- Never use 'ǁ' or any other method to concatenate strings in the 'SELECT' clause. Rather output the columns as they are.
- BETWEEN ... AND ...: If the question requires selecting values within a specified range, you should use 'BETWEEN ... AND ...' operator.
- ASC and DESC: If the question does not explicitly mention specific order such as ascending or descending order, you MUST NOT add 'ASC' or 'DESC' in SQL queries.
- Column and Table Selection: Remember to only include columns and tables explicitly requested in the query. Remember to select the most correct tables in the 'FROM' clause.
- Aggregation: When using 'MAX' or 'MIN', please perform joins before selecting these aggregates. DO NOT use 'MAX' or 'MIN' in the 'WHERE' clause! Always use the 'ORDER BY' clause in combination with 'LIMIT X'.
- GROUP BY: When the question does not explicitly and clearly perform calculations on grouped data, you MUST NOT add 'GROUP BY' clause.
- DISTINCT: When the question does not explicitly and clearly search for unique values from a column, you should not add 'DISTINCT' clause.

- Order and Grouping: Use 'GROUP BY <column>' before 'ORDER BY <column> ASC/DESC' to ensure distinct grouped results. Pay more attention to the content after 'ORDER BY'.
- LIMIT: Apply 'LIMIT' only when the question specifies a retrieval count.
- WHERE: Double-check that all 'WHERE' clauses accurately represent the conditions needed to filter the data as per the question's requirements.

The input question is:
{**Question**}.

The candidate SQL are:
Candidate SQL 1: {**Candidate SQL1**}. Execution result 1: {**Execution Result1 from Candidate SQL1**}.
Candidate SQL 2: {**Candidate SQL2**}. Execution result 2: {**Execution Result2 from Candidate SQL2**}.
Candidate SQL 3: {**Candidate SQL3**}. Execution result 3: {**Execution Result3 from Candidate SQL3**}.
Candidate SQL 4: {**Candidate SQL4**}. Execution result 4: {**Execution Result4 from Candidate SQL4**}.
Candidate SQL 5: {**Candidate SQL5**}. Execution result 5: {**Execution Result5 from Candidate SQL5**}.

Please read the input question, the [**Hints**], the candidate SQL and corresponding execution results again. Let's take a deep breath and think step by step, you must think more steps. Please remember to stay in strict accordance with the nature of the provided identity. If you can select the best SQL satisfying the question demands, I will give you one million dollars.

L Appendix L: An Example of ICL-based Methods

Question: Find the name and rank points of the winner who won the most times.

Direct SQL Generation:

```
SELECT p.first_name, p.last_name, m.winner_rank_points FROM matches AS m JOIN players AS p ON m.winner_id = p.player_id GROUP BY m.winner_id ORDER BY COUNT(*) DESC LIMIT 1;
```

Strict Step-by-Step Reasoning:

<Reasoning>

Step 1: Read the Question: The question asks for the name and rank points of the winner who has won the most matches.

Step 2: Analyze the Schema: The `matches` table contains information about match winners, including their `winner_name` and `winner_rank_points`. The `winner_id` in the `matches` table references the `player_id` in the `players` table, which can be used to join these tables if needed.

Step 3: Analyze the Given Examples: The examples demonstrate the use of `GROUP BY` to count occurrences and `ORDER BY` with `LIMIT 1` to find the maximum.

Step 4: Analyze the Given Hints: The hints emphasize using `GROUP BY` for counting, `ORDER BY` for sorting, and `LIMIT 1` to retrieve the top result.

Step 5: Finalize the SQL Query: The query will group by `winner_name` and `winner_rank_points`, count the number of wins, and order the results in descending order of the count. The top result will be retrieved using `LIMIT 1`.

Step 6: Validation and Syntax Check: The query is validated to ensure it adheres to the schema, uses correct syntax, and satisfies the question's requirements.

</Reasoning>

<SQL>

```
SELECT winner_name, winner_rank_points FROM matches GROUP BY winner_name, winner_rank_points ORDER BY COUNT(*) DESC LIMIT 1;
```

</SQL>

Flexible Reasoning:

<Reasoning>

To solve this problem, I need to determine the winner who has won the most matches. The relevant data can be obtained from the `matches` table using the `winner_id`, which references the `player_id` in the `players` table to get the name of the player. Additionally, I need the rank points of this winner, which can be retrieved from the `matches` table where `winner_rank_points` are stored. The approach is to join the `matches` table with the `players` table to get the winner's name and count the number of wins for each winner. I will order by the count of wins to find the top winner and limit the result to one.

</Reasoning>

<SQL>

```
SELECT p.first_name, p.last_name, m.winner_rank_points FROM matches AS m JOIN players AS p ON m.winner_id = p.player_id GROUP BY m.winner_id ORDER BY COUNT(*) DESC LIMIT 1;
```

</SQL>

Figure 5: An illustrative example of using three different reasoning processes in ICL-based methods.

M Appendix M: Additional Experimental Results

N Appendix N: Case Study of DSMR-SQL

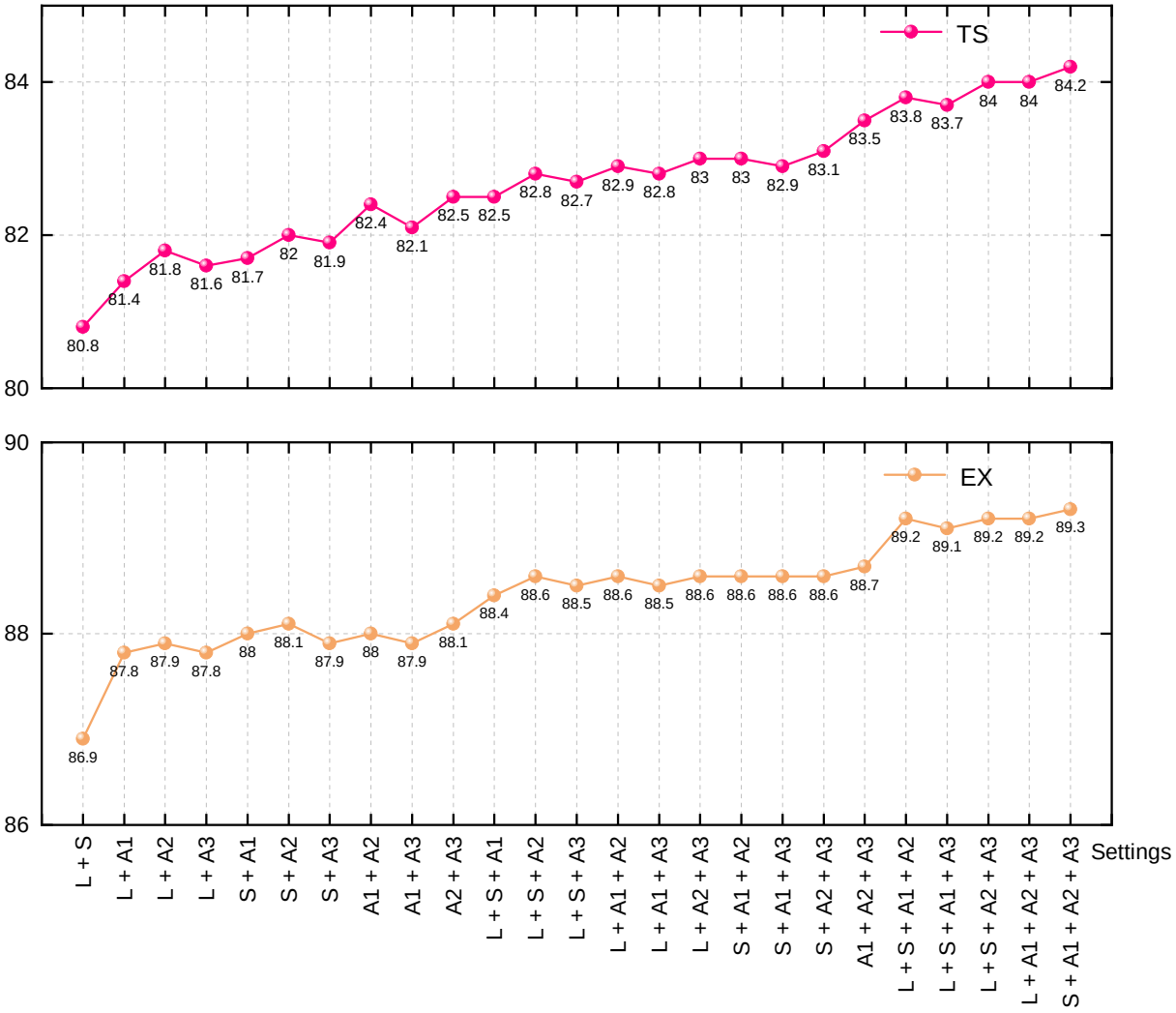


Figure 6: Additional experimental results via using SFT and ICL on the Spider-Dev dataset. For simplicity, L and S indicate Llama-3.2-3B-Instruct and StableCode-3B, respectively. A1, A2, and A3 represent the three reasoning processes in the ICL-based strategy, respectively.

Question: Find the name and rank points of the winner who won the most times.

SQL Candidates:

(1) SELECT winner_name, winner_rank_points FROM matches GROUP BY winner_id ORDER BY COUNT(*) DESC LIMIT 1

(2) SELECT winner_name , winner_rank_points FROM matches GROUP BY winner_name ORDER BY count(*) DESC LIMIT 1

(3) SELECT p.first_name, p.last_name, m.winner_rank_points FROM matches AS m JOIN players AS p ON m.winner_id = p.player_id GROUP BY m.winner_id ORDER BY COUNT(*) DESC LIMIT 1

(4) SELECT p.first_name, p.last_name, m.winner_rank_points FROM matches AS m JOIN players AS p ON m.winner_id = p.player_id GROUP BY m.winner_id ORDER BY COUNT(*) DESC LIMIT 1

(5) SELECT winner_name, winner_rank_points FROM matches GROUP BY winner_name, winner_rank_points ORDER BY COUNT(*) DESC LIMIT 1

Execution Result:

('Angelique Kerber', 3715)

('Angelique Kerber', 3715)

('Angelique', 'Kerber', 3715)

('Angelique', 'Kerber', 3715)

('Elena Vesnina', 2340)

Consistency-Voting (CV) / Direct SQL Selection (DS) / SQL Selection with Explanation (SE):

SELECT p.first_name, p.last_name, m.winner_rank_points FROM matches AS m JOIN players AS p ON m.winner_id = p.player_id GROUP BY m.winner_id ORDER BY COUNT(*) DESC LIMIT 1

Execution Result:

('Angelique', 'Kerber', 3715)

Multi-Role SQL Selection (MRS):

SELECT winner_name, winner_rank_points FROM matches GROUP BY winner_id ORDER BY COUNT(*) DESC LIMIT 1

Execution Result:

('Angelique Kerber', 3715)

<proposer>
The question asks for the name and rank points of the winner who won the most times.
Analysis of the provided SQL queries:
1. **sql1**: This query groups by `winner_id` and orders by the count of matches won. It returns the `winner_name` and `winner_rank_points` of the winner with the most wins. The result is `('Angelique Kerber', 3715)`.
2. **sql2**: This query groups by `winner_name` and orders by the count of matches won. It returns the `winner_name` and `winner_rank_points` of the winner with the most wins. The result is `('Angelique Kerber', 3715)`.
3. **sql3**: This query joins the `matches` and `players` tables to retrieve the `first_name` and `last_name` of the winner, along with their `winner_rank_points`. It groups by `winner_id` and orders by the count of matches won. The result is `('Angelique', 'Kerber', 3715)`.
4. **sql4**: This query joins the `matches` and `players` tables to retrieve the `first_name` and `last_name` of the winner, along with their `winner_rank_points`. It groups by `winner_id` and orders by the count of matches won. The result is `('Angelique', 'Kerber', 3715)`.
5. **sql5**: This query groups by both `winner_name` and `winner_rank_points`. However, grouping by `winner_rank_points` is unnecessary and could lead to incorrect results if a player has multiple rank points across matches. The result is `('Elena Vesnina', 2340)`.
Conclusion:
The best SQL query is **sql3** or ** sql4**, since they provide additional details (first and last name), thereby making the information more comprehensive.
</proposer>
<critic>
For **sql3** and **sql4**, While they provide additional details (first and last name), the question only asks for the name and rank points, making this query overly detailed.
Given the above, **sql1** is actually the most robust and accurate query, as it groups by `winner_id` to uniquely identify winners and avoids potential issues with duplicate names. While **sql2** returns the same result as **sql1**, it is less robust in scenarios with duplicate names.
Recommendation: The best SQL query is **sql1**, not **sql3** or **sql4**.
</critic>
<proposer>
Based on the critic's feedback, I acknowledge that **sql1** is better. Therefore, I revise my conclusion and propose that **sql1** is the best SQL query for this question.
</proposer>
<critic>
The revised proposition is correct. **sql1** is indeed the best SQL query, as it groups by `winner_id` to uniquely identify winners and avoids potential issues with duplicate names. The reasoning is now complete and accurate.
</critic>
<summarizer>
SELECT winner_name, winner_rank_points FROM matches GROUP BY winner_id ORDER BY COUNT(*) DESC LIMIT 1;
</summarizer>

Figure 7: Case study of DSMR-SQL. Notably, consistency-voting (CV) and simple SQL selection techniques (i.e., DS and SE) select the wrong SQL, while the proposed MRS chooses the correct SQL satisfying user intent.