Divide and Conquer: Harnessing Small Agents for Schema Extraction in NL-to-SQL Generation

Anonymous ACL submission

Abstract

Large Language Models have shown remarkable promise in various code generation tasks, particularly in SQL generation. However, much like other structured query generation tasks, SQL generation presents the unique challenge of extracting the correct schema to achieve good performance. Despite being a critical component of the process, the problem of schema extraction has received little attention, especially when it comes to Small Language Models [1]. In this paper, we propose LiteMARS: Lite Multi-Agent Recall Oriented System, to the best of our knowledge, the first multi-agent framework to incorporate schema linking that leverages question decomposition for the task of Natural Language to SQL (NL-to-SQL). LiteMARS operates as a multi-agent pipeline with three key stages: *natural language query decomposition*, *schema linking*, and *SQL generation*. Notably, LiteMARS introduces a novel critic-based one-step refinement process, enhancing schema extraction and SQL generation. In experiments, we found that critic-based refinement significantly improved column recall by 26.6% and execution accuracy by 73.4% for NL-to-SQL generation. Further analysis shows that our LiteMARS agent achieves comparable performance to Large Language Models like DeepSeek-Coder-33B.

1 Introduction

Many organizations and businesses today would not exist without gathering and maintaining large amounts of data. SQL is one of the most common ways to query these data. However, manually writing SQL to navigate the complexities of data schemas is both time-consuming and error-prone, making NL-to-SQL systems an attractive solution. Towards this end, LLMs have recently shown great success in a variety of translation and code generation tasks, including the NL-to-SQL task (Chen et al., 2021; Shi et al., 2020; Yu et al., 2018; Li et al., 2023c; Zhong et al., 2017).

However, unlike typical translation and code generation tasks, the task of generating SQL brings a unique and important challenge that is unaddressed by the current NL-to-SQL language model agents. This challenge is *schema linking* i.e., the process of identifying and extracting relevant columns from the database schema given the natural language prompt. In an NL-to-SQL task, accurate schema linking is crucial to ensure that the final SQL query is both relevant and executable (Chen et al., 2020).

Although several related efforts and directions exist, none address the question of schema linking satisfactorily: (1) One candidate approach is to provide the entire schema to the model's prompt. However, in most language models, especially the smaller models, the context window size is limited. As such, it is often infeasible to include the entire schema within the model's prompt. (2) Some recent works (Maamari et al., 2024) have argued that schema linking is unnecessary for models with large context windows and advanced SQL generation capabilities. However, this approach is not applicable for smaller models with fewer parameters (e.g., Llama-3.1-8B or Mistral-7B) that lack the flexibility and size of proprietary models such as GPT-4 and Gemini (Achiam et al., 2023; Team et al., 2023). (3) Some works have approached this problem tangentially by performing question decomposition to break down complex questions into manageable sub-questions (Wang et al., 2024; Pourreza and Rafiei, 2024a). They applied question decomposition during the SQL generation stage rather than the schema linking stage, and question decomposition alone does not directly tackle the schema-linking problem.

[1]In this work, we define language models with fewer than 10 billion parameters as Small Language Models (SLMs).

In this work, we propose `LiteMARS` agent, the first multi-agent framework that leverages schema linking for the NL-to-SQL task. `LiteMARS` consists of a multi-agent pipeline consisting of (1) a natural language query decomposition stage, (2) a schema linking stage, and (3) a SQL generation stage. Importantly, `LiteMARS` leverages two critics for performing refinement within this pipeline. The *Schema Critic* agent reviews the extracted schema and refines the schema within the schema linking stage. The *SQL Critic* agent reviews the generated SQL to refine it within the SQL generation stage. In experiments, we note that this critic-based refinement produces significantly improved column recall and execution accuracy for NL-to-SQL generation, even for small open-source models with limited parameters.

Our contributions are:

1. `LiteMARS`, the first agent that integrates natural language query decomposition (*QD*), a schema refinement module (*Schema Refiner*), and a schema critiquing mechanism (*Schema Critic*) at the Schema Linking stage.

2. Multi-agent framework that achieves an improvement of 28.1% in execution accuracy with Llama-3.1-8B, 73.4% with Mistral-7B, and 62.1% with Granite-8B models over their Vanilla counterpart (refer 4.3). For Mistral-7B, this includes a 54.6% improvement from *QD* + *Schema Critic* alone.

3. Detection and correction of SQL errors by our novel SQL Critic agent, based on execution feedback in the SQL generation phase, result in a 19.2% improvement in execution accuracy for models with limited SQL generation capabilities, providing a robust solution.

2 Related Work

SLM for NL-to-SQL: Recent years have witnessed a significant shift towards using large language models (LLMs) for NL-to-SQL tasks, with impressive results (Gao et al. (2023); Pourreza and Rafiei (2024a); Dong et al. (2023); Sun et al. (2023); Pourreza et al. (2024a)). However, the computational costs and potential privacy concerns associated with these models have motivated research into more efficient alternatives. Li et al. (2024a) introduced CodeS, a series of SLMs specifically designed for SQL generation. Their work demonstrated that SLMs, when properly trained, can achieve competitive performance on challenging benchmarks like Spider (Yu et al., 2018) and BIRD (Li et al., 2024b). Similarly, Pourreza and Rafiei (2024b) proposed DTS-SQL, which employs supervised fine-tuning on smaller models to narrow the performance gap with larger models. With the latest SLM, which achieves higher performance than the model double or triple its size (Team (2024); Microsoft (2024)), we can see a growing trend towards developing more efficient and accessible NL-to-SQL solutions using it, which aligns with our work's objectives.

Schema Linking Techniques: Schema linking, the task of mapping natural language elements to database schema components, has long been considered a critical step in NL-to-SQL pipelines (Lewis (2019)). Traditional approaches often relied on string matching or learned embeddings to identify relevant schema elements (Guo et al. (2019); Bogin et al. (2019); Wang et al. (2019); Li et al. (2023b)). More recently, LLM-based methods have shown promise in this area. Talaei et al. (2024) and Pourreza and Rafiei (2024b) proposed using LLMs for hierarchical retrieval of schema components. However, Maamari et al. (2024) challenged the conventional wisdom surrounding schema linking, arguing that for state-of-the-art LLMs, schema linking might be unnecessary or even detrimental when the entire schema fits within the model's context window.

Query Decomposition: Query decomposition has emerged as a powerful technique to improve NL-to-SQL performance, especially for complex queries. Pourreza and Rafiei (2024a) introduces query classification and decomposition module that decomposes based on the task complexity. Further, Wang et al. (2024) proposes an agent-based framework that decomposes complex questions into simpler sub-questions before SQL generation. While these works primarily focus on decomposing the natural language query or the overall task, our approach uniquely applies decomposition techniques to the schema linking phase, offering a novel perspective on this crucial step.

LLM-based Agents: The concept of LLM-based agents, where language models are employed as autonomous or semi-autonomous entities capable of reasoning, planning, and interacting with their environment, has gained significant traction across various NLP tasks (Qian et al. (2023); Zhou et al. (2023); Xu et al. (2023)).

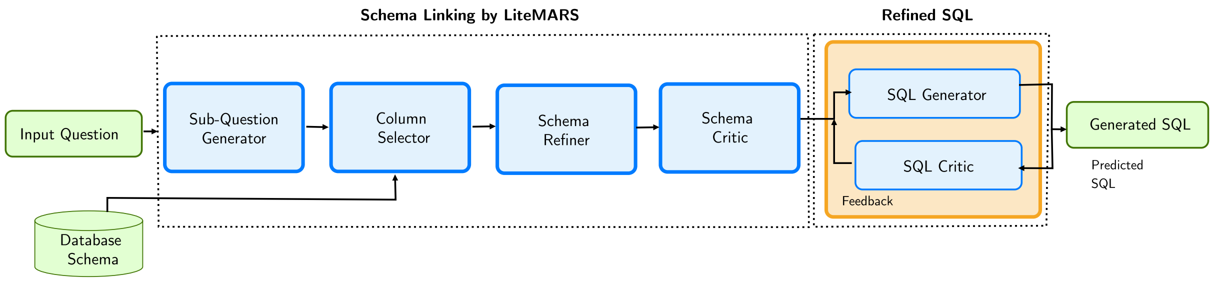


Figure 1: Overview of our proposed `LiteMARS Agent` for NL-to-SQL Generation.

Initially, most agents were constructed using large, proprietary models (Li et al. (2023a); Wu et al. (2023); Zhou et al. (2024)) to facilitate complex instruction following. More recently, however, SLM-based agents (Liang et al. (2024); Zeng et al. (2023); Chen et al. (2024)) have emerged due to their competitiveness with closed models. Nevertheless, the application of multi-agent frameworks in the NL-to-SQL domain remains relatively unexplored. While Wang et al. (2024) introduced an early agent-based approach, it did not fully exploit the potential benefits of query decomposition techniques or incorporate critique mechanisms for the schema linking phase. Our work addresses this gap by incorporating a sub-question generator and a schema critique agent, enabling reflection (Shinn et al. (2023); Madaan et al. (2024); Asai et al. (2023)) and refinement of the schema extracted by the schema refiner module.

3 Methodology

We propose `LiteMARS` agent into NL-to-SQL generation pipeline consisting of six modules: Natural Language Query Decomposition, Column Selector, Schema Refiner, Schema Critic, SQL Generator, and SQL Critic. We use Natural Language Query Decomposition, Column Selector, Schema Refiner, and Schema Critic to improve schema linking. SQL Critic and SQL Generator agents are used to improve NL-to-SQL generation. In the figure 1, we show how different agents interact in a collaborative way to improve NL-to-SQL generation.

Before delving into the technical details of our method, we first outline the Problem Statement:

Task: To convert a given Natural Language (NL) Query to its corresponding SQL Query that can be executed on a relational database.

Input: NL Query (q) and a database ($\mathcal{D}$).

Output: SQL Query ($\hat{S}$) corresponding to the NL Query.

3.1 Sub-Question Generator

Sub-question generator step involves breaking down the original natural language question q into simpler sub-questions $q_1, q_2, \ldots, q_n$, allowing the model to focus on retrieving the relevant schema components for each sub-question. Formally, given a natural language query q, we decompose it into a set of sub-questions $\{q_i\}_{i=1}^{n}$, where each sub-question q_i addresses a specific aspect of q. The decomposition can be expressed as:

$$\mathcal{QD}(q) = \{q_1, q_2, \ldots, q_n\}$$

Here, $\mathcal{QD}$ represents the question decomposition function. The key idea is to improve the **column recall rate** (the proportion of relevant columns selected from the total relevant columns) by ensuring that each sub-question focuses on a smaller portion of the aspects of the given question.

3.2 Column Selector

The Column Selector module retrieves a set of relevant columns for each sub-question generated in the previous stage. The goal is to maximize the column recall rate, without overly penalizing the selection of irrelevant columns with carefully designed prompts (refer Figure 4, 5). Let C_i denote the set of columns selected for sub-question q_i, and let $\mathcal{C}$ represent the set of all columns in the database schema. The selected schema can be represented as:

$$D = \bigcup_{i=1}^{n} C_i$$

where D is the selected schema.

Let $\mathcal{S}$ be the column selection function, which retrieves all columns $c \in \mathcal{C}$ that are relevant to the sub-question q_i. Then according to the context window size of different Language Models, the column selection task can be defined as follows.

For small context windows: When working with small models (e.g., Llama-3-8B), the context window might not accommodate the entire schema. In such cases, the model iterates over each table $T_j \in \mathcal{T}$, where $\mathcal{T}$ represents the set of all tables in the database. The task of the model is then to select relevant columns C_i^j from table T_j for sub-question q_i:

$$C_i^j = \mathcal{S}(q_i, T_j)$$

$$C_i = \bigcup_{j:T_j \in \mathcal{T}} C_i^j$$

For large context windows: For models with larger context windows (e.g., Llama-3.1-8B), it is feasible to provide the entire schema to the model. In this case, the module selects relevant columns from the entire schema rather than table-specific columns:

$$C_i = \mathcal{S}(q_i, \mathcal{C})$$

3.3 Schema Refiner

The Schema Refiner prunes the set of columns selected by the Column Selector. Given the pruned schema, the refiner removes columns that are not essential for the final SQL generation thereby aiding in increasing column precision rate. The Schema Refiner takes as input the selected schema D and the original question q, and outputs a refined schema D' where:

$$D' = \mathcal{R}(D, q)$$

Here, $\mathcal{R}$ represents the refinement function that removes unnecessary columns.

3.4 Schema Critic

The Schema Critic Agent reviews the extracted schema and assesses the columns that were not initially selected to determine if any relevant columns were mistakenly omitted. This additional step is aimed at improving the column recall, ensuring that all necessary columns for SQL generation are included. The process of critiquing serves as a safeguard against potential omissions by the Schema Refiner. The input to this agent is D', the schema produced by the Schema Refiner, and the output is $\hat{D}$, the further refined schema, defined as:

$$\hat{D} = \mathcal{C}_{sch}(D', q)$$

Where $\mathcal{C}_{sch}$ represents the schema Critic function based on the input schema D' and the query q.

3.5 SQL Generator

Using the refined schema $\hat{D}$, the SQL Generator produces the SQL query $\hat{S}_{\text{gen}}$ based on the original question q. Formally, the SQL generation process can be defined as:

$$\hat{S}_{\text{gen}} = \mathcal{G}(q, \hat{D})$$

where $\mathcal{G}$ is the SQL generation function. The generated SQL $\hat{S}_{\text{gen}}$ is then executed against the database, and any execution error e is recorded.

3.6 SQL Critic

To improve the performance further, we introduce a SQL Critic agent which takes the original question q, the refined schema $\hat{D}$, the generated SQL $\hat{S}_{\text{gen}}$, and the execution error e (if any) as inputs, and produces a corrected SQL $\hat{S}$ by identifying potential mistakes in the SQL query:

$$\hat{S} = \mathcal{C}_{sql}(q, \hat{D}, \hat{S}_{\text{gen}}, e)$$

where $\mathcal{C}_{sql}$ is the SQL Critic function that modifies $\hat{S}_{\text{gen}}$ to generate $\hat{S}$, the final predicted SQL.

It is important to note that our experiments revealed higher scores when using the refined queries generated by the SQL Critic agent, even in cases where the SQL query produced by the SQL Generator was free of execution errors. Consequently, all the results presented in this work are based on this setting.

The methodology described above mimics human reasoning in complex tasks like NL-to-SQL by incorporating Question Decomposition at the Schema Linking stage. This approach increases column recall and improves the overall execution accuracy, especially for Small Language Models with low SQL generation capabilities.

4 Experiments, Results and Analysis

4.1 Datasets

Our evaluation utilized two challenging and widely recognized datasets: Spider and BIRD. Spider, introduced by Yu et al. (2018), is a large-scale cross-domain NL-to-SQL dataset comprising 8,650 training examples, 1,034 development examples, and 2,147 holdout test examples. These span 200 databases across 138 domains. BIRD, developed by Li et al. (2024b), is a more recent benchmark dataset consisting of 9,428 training instances, 1,534 development instances, and 1,789 concealed test instances. It covers 95 large databases totaling 33.4 GB and encompasses over 37 professional domains, including blockchain, hockey, healthcare, and education. Notably, BIRD features more complex SQL queries compared to Spider. All our evaluations were conducted on the development sets of both Spider and BIRD datasets.

4.2 Metrics

We evaluated our pipeline's performance using the official metric of Execution Accuracy (EX) for both Spider and BIRD evaluation datasets. EX compares the execution output of the predicted SQL query against that of the ground truth SQL query on given database instances. This metric provides a more precise estimate of model performance as it accounts for the possibility of multiple valid SQL queries for a given question. To assess the performance of the schema-linking module, we employ two additional metrics: Column Precision and Column Recall. Column Precision measures the accuracy of the retrieved columns, while Column Recall evaluates the completeness of the retrieval with respect to the golden oracle schema.

4.3 Baselines and Variations

In all the following experiments, we utilized the Deepseek-coder-7b-instruct-v1.5 for both SQL Generator and the SQL Critic Agent. The following settings were compared to evaluate the performance of our approach:

1. **Vanilla:** In this setting, we employ only the Schema Linking without incorporating Natural Language Query Decomposition or any Critic agents (schema or SQL). This serves as our baseline to understand the performance of Schema Linking abilities of the Small Language Models in isolation.

2. **Vanilla + QD:** In this setting, we incorporate Natural Language Query Decomposition over the Vanilla setting. This serves as our baseline for understanding the importance of Schema Critic in our pipeline (refer Appendix B).

3. **LiteMARS w/o SQL Critic:** In LiteMARS agent, we employ the sub-question generator for NL query decomposition alongside the Schema Critic. This setting aims to investigate the benefits of incorporating both query decomposition and schema Critic on execution accuracy.

4. **LiteMARS:** This is the complete agent in which we integrate the SQL Critic agent. This addition allows us to analyze the combined effect of Query Decomposition, Schema Critic, and SQL Critic on the overall performance of the system. Here, as the only difference from the previous setting is in the addition of SQL Critic, the schema linking related metrics (Column Precision and Column Recall) remains the same.

Refer to Appendix C for hyperparameters and implementation details. For comparisons with SoTA refer Appendix D.

4.4 Results

The following section presents the results obtained from experiments conducted on two datasets, BIRD (Li et al., 2024b) and Spider 1.0 (Yu et al., 2018), using multiple SLMs integrated with Query Decomposer (QD) and Critic Agents. We compared the performance of the models in terms of Execution Accuracy, Column Precision, and Column Recall across different configurations: Vanilla (base model), LiteMARS without SQL Critic (LiteMARS w/o SC), and LiteMARS (the full pipeline). For SQL generation and SQL Critic, the Deepseek-coder-7b-instruct-v1.5 model was used across all experiments.

4.4.1 Comparison with the Oracle Baseline

As we previously stated, our primary focus is on enhancing the schema linking capabilities of open-source small language models. To evaluate this, we compare LiteMARS with an oracle baseline. In the oracle baseline, we supply the ideal schema derived directly from the ground truth SQL, providing an upper-bound estimate of LiteMARS's potential. Using this oracle schema with Deepseek-coder-7b-instruct-v1.5 for SQL generation achieves an Execution Accuracy (EX) of **0.3906** on the BIRD dataset. As shown in Table 1, employing Llama-3.1-8B for schema linking yields an EX of **0.3643**, closely approaching the oracle baseline. Similarly, Mistral-7B and

Models	Execution Accuracy			Column Precision			Column Recall		
	Vanilla	LiteMARS w/o SC	LiteMARS	Vanilla	LiteMARS w/o SC	LiteMARS	Vanilla	LiteMARS w/o SC	LiteMARS
Llama-3.1-8B	0.2501	0.2980	**0.3206**	**0.7340**	0.7231	0.7231	0.7926	**0.8295**	**0.8295**
Mistral-7B	0.1721	0.2661	**0.2985**	0.6743	**0.7324**	**0.7324**	0.6183	**0.7832**	**0.7832**
Granite-8B	0.1421	0.1933	**0.2304**	0.4244	**0.6010**	**0.6010**	0.6105	**0.7015**	**0.7015**

Table 1: Experiments on BIRD (Li et al., 2024b) Dataset with different SLMs with QD and Critic Agents. **SC**: SQL Critic.

Models	Execution Accuracy			Column Precision			Column Recall		
	Vanilla	LiteMARS w/o SC	LiteMARS	Vanilla	LiteMARS w/o SC	LiteMARS	Vanilla	LiteMARS w/o SC	LiteMARS
Llama-3.1-8B	0.4528	0.4971	**0.5386**	**0.5205**	0.4501	0.4501	0.6468	**0.7073**	**0.7073**
Mistral-7B	0.2985	0.4314	**0.4971**	0.4424	**0.4886**	**0.4886**	0.4954	**0.6414**	**0.6414**
Granite-8B	0.3007	0.3600	**0.4305**	0.2894	**0.3425**	**0.3425**	0.4753	**0.5319**	**0.5319**

Table 2: Experiments on Spider 1.0 (Yu et al., 2018) Dataset with different SLMs with QD and Critic Agents. **SC**: SQL Critic.

Granite-8B demonstrate comparable performance highlighting our approach.

4.4.2 Impact of Query Decomposition and Schema Critic

Table 1 presents the significant improvements achieved by the LiteMARS pipeline when evaluated on the BIRD dataset. Execution accuracy, a critical metric indicative of the model's capability to generate correct SQL queries, demonstrated consistent gains across all evaluated models with the integration of Query Decomposition (QD) and Schema Critic (SC) within the pipeline. For example, in the case of Llama-3.1-8B, execution accuracy increased from 0.2501 in the Vanilla setup to 0.2980 in the LiteMARS w/o SC configuration—an improvement exceeding 19.1%. Similarly, Mistral-7B exhibited a rise from 0.1721 to 0.2661 (54.6%), while Granite-8B improved from 0.1421 to 0.1933 (36.0%). Detailed comparisons of the impact of QD alone are provided in Appendix A.

In addition to execution accuracy, the LiteMARS pipeline significantly enhanced column-related metrics, which are crucial for accurately identifying and linking relevant database columns during SQL generation. It is important to note that in LiteMARS, since the only addition to the LiteMARS w/o SC configuration is the SQL Critic (SC), the column-related metrics remain unchanged. With the incorporation of Natural Language Query Decomposition and Schema Critic, Mistral-7B achieved a column recall of 0.7832, a substantial improvement over the Vanilla configuration's 0.6183 (54.6%). Comparable trends were observed for Llama-3.1-8B (4.6% increase) and Granite-8B (14.9% increase). These results underscore the pipeline's efficacy in improving column recall, which is essential for generating correct SQL queries.

However, in terms of column precision, which measures the model's tendency to select incorrect or irrelevant columns, a slight decrease was observed in some configurations. For instance, in Llama-3.1-8B, column precision dropped from 0.7340 in the Vanilla setup to 0.7231 in the LiteMARS w/o SC configuration. Despite this reduction, the pipeline achieved higher execution accuracy, highlighting the importance of column recall rate.

4.4.3 Impact of SQL Critic

Table 2 corroborates these findings with results from the Spider 1.0 dataset, demonstrating that the LiteMARS pipeline is equally effective across different datasets. On Spider, the Granite-8B model achieved an execution accuracy of 0.4305 with the full LiteMARS pipeline, a significant jump from 0.3600 in the LiteMARS w/o SC configuration (19.5%). Similarly, Mistral-7B exhibited an increase from 0.4314 to 0.4971 (15.2%), and Llama-3.1-8B improved from 0.4971 to 0.5386 (8.3%) after the addition of SQL Critic.f

In summary, the results from both datasets

Models	Simple			Moderate			Challenging		
	Vanilla	LiteMARS w/o SC	LiteMARS	Vanilla	LiteMARS w/o SC	LiteMARS	Vanilla	LiteMARS w/o SC	LiteMARS
QD/SL: Llama-3.1-8B	0.3059	0.3113	**0.3643**	0.1831	**0.1982**	0.1724	0.1103	**0.1586**	0.1517
QD/SL: Mistral-7B	0.2194	0.2594	**0.3135**	0.1034	0.1250	**0.1508**	0.0896	**0.1241**	0.1103
QD/SL: Granite-8B	0.1772	0.2454	**0.2875**	0.0862	0.1185	**0.1443**	0.0965	0.1034	**0.1448**

Table 3: Impact of Query Decomposition (QD) and Critic Agent on Solving Complex Queries on Bird (Li et al., 2024b) dataset. **SC**: SQL Critic.

clearly demonstrate that the full `LiteMARS` pipeline is highly effective at improving execution accuracy, column recall, and precision, establishing it as a comprehensive solution for enhancing the performance of small-scale models in NL-to-SQL tasks. These gains are not only consistent across different datasets but also significant enough to illustrate the advantage of using a fully integrated pipeline over baseline and partially optimized configurations.

4.5 Impact of Query Decomposition and Critic agents on Query Complexity

Table 3 shows the effects of Query Decomposition (QD) and the Critic agent on addressing queries of varying complexity—classified as Simple, Moderate, and Challenging, utilizing the same three Small Language Models (SLMs): Llama-3.1-8B, Mistral-7B, and Granite-8B, on the BIRD dataset. For **simple queries**, the integration of QD followed by the Critic agent yields notable enhancements across all models. For instance, Llama-3.1-8B demonstrates an increase in performance from 0.3059 (Vanilla) to 0.3643 (`LiteMARS`), underscoring the effectiveness of both decomposition and the Critic in refining simpler queries. In the case of **moderate queries**, results present a more varied picture. Llama-3.1-8B shows a slight advantage without SQL Critic, `LiteMARS` w/o SC (0.1982) compared to the `LiteMARS` (0.1724), while Mistral-7B and Granite-8B exhibit the most significant enhancements when SQL Critic is applied. Mistral-7B's performance, for example, improves from 0.1034 (Vanilla) to 0.1508 (`LiteMARS`).

For **challenging queries**, both QD and the Critic agent significantly enhance the performance of models when Granite-8B is employed at the schema linking stage. However, for Llama-3.1-8B and Mistral-7B, a decrease in performance is observed upon incorporating the SQL Critic agent into the pipeline. This decline in performance for challenging and moderate queries might have been mitigated by utilizing the initial SQL query, provided it had no execution errors. Nevertheless, the refined SQL consistently delivered higher overall execution accuracy.

Overall, this table emphasizes the complementary role of both QD and the Critic in managing queries of increasing complexity, with more substantial gains observed in simpler and moderately complex cases.

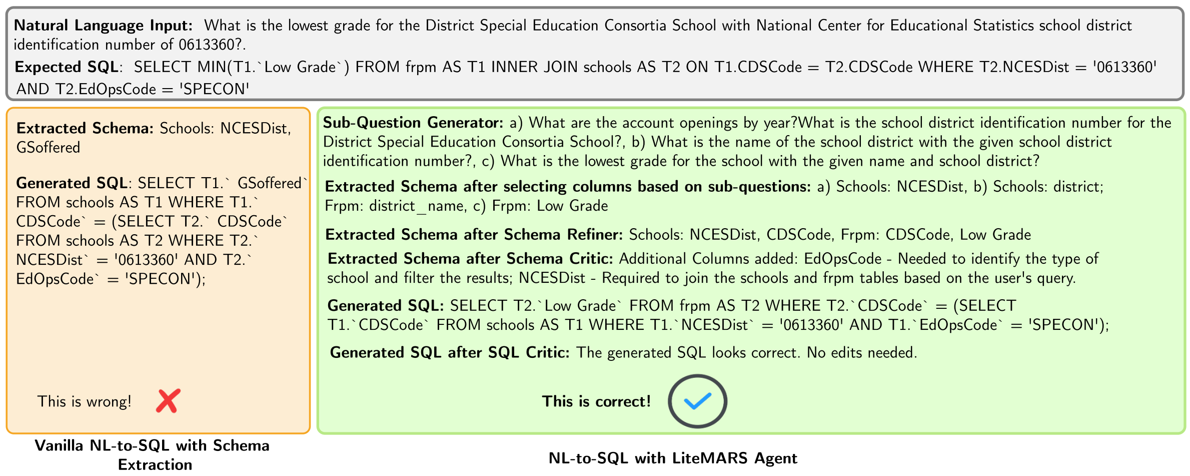
4.6 Impact of Using Small Agents vs. Larger Models on NL-to-SQL

In our experiments, we observed that the Llama-3.1-8B model, when integrated with our `LiteMARS` framework, achieves an execution accuracy comparable to that of the significantly larger Deepseek-coder-33B model. This finding highlights the efficiency and effectiveness of our proposed approach, even when utilizing smaller models with fewer parameters.

When providing the complete schema to DeepSeek-Coder-33B, we achieve an execution accuracy of 34.46%. In comparison, our `LiteMARS` framework with Llama-3.1-8B achieves a competitive execution accuracy of 32.06%. This outcome is particularly significant as it highlights that smaller models, when integrated with an effective framework like `LiteMARS`, can approach the performance levels of much larger models. Notably, in this setup, the entire schema is directly supplied to the model without schema linking, and the prompt used for the SQL Generator is employed to produce the SQL query.

4.7 Qualitative Analysis of the Effectiveness of Schema Critic

The Schema Critic demonstrates significant effectiveness in refining the Schema linking process, particularly in handling complex queries such as retrieving the lowest grade for the District Special Education Consortia School with a specific


Natural Language Input: What is the lowest grade for the District Special Education Consortia School with National Center for Educational Statistics school district identification number of 0613360?.

Expected SQL: SELECT MIN(T1.`Low Grade`) FROM frpm AS T1 INNER JOIN schools AS T2 ON T1.CDSCode = T2.CDSCode WHERE T2.NCESDist = '0613360' AND T2.EdOpsCode = 'SPECON'

Extracted Schema: Schools: NCESDist, GSoffered

Generated SQL: SELECT T1.` GSoffered` FROM schools AS T1 WHERE T1.` CDSCode` = (SELECT T2.` CDSCode` FROM schools AS T2 WHERE T2.` NCESDist` = '0613360' AND T2.` EdOpsCode` = 'SPECON');

This is wrong! ✗

Sub-Question Generator: a) What are the account openings by year?What is the school district identification number for the District Special Education Consortia School?, b) What is the name of the school district with the given school district identification number?, c) What is the lowest grade for the school with the given name and school district?

Extracted Schema after selecting columns based on sub-questions: a) Schools: NCESDist, b) Schools: district; Frpm: district_name, c) Frpm: Low Grade

Extracted Schema after Schema Refiner: Schools: NCESDist, CDSCode, Frpm: CDSCode, Low Grade

Extracted Schema after Schema Critic: Additional Columns added: EdOpsCode - Needed to identify the type of school and filter the results; NCESDist - Required to join the schools and frpm tables based on the user's query.

Generated SQL: SELECT T2.`Low Grade` FROM frpm AS T2 WHERE T2.`CDSCode` = (SELECT T1.`CDSCode` FROM schools AS T1 WHERE T1.`NCESDist` = '0613360' AND T1.`EdOpsCode` = 'SPECON');

Generated SQL after SQL Critic: The generated SQL looks correct. No edits needed.

This is correct! ✓

Vanilla NL-to-SQL with Schema Extraction
NL-to-SQL with LiteMARS Agent

Figure 2: An example showing how `LiteMARS Agent` improves Schema Linking for NL-to-SQL Generation

National Center for Educational Statistics (NCES) school district identification number (refer to Figure 2). By employing a series of sub-questions, the model effectively breaks down the primary query into manageable components. For instance, the second sub-question identifies the relevant school district identification number, while the last establishes the connection between the district and the school's name. The final sub-question also zeroes in on retrieving the minimum grade, guiding the SQL Generator to accurately construct the query:

SELECT T2.'Low Grade' FROM frpm AS T2 WHERE T2.'CDSCode' = (SELECT T1.'CDSCode' FROM schools AS T1 WHERE T1.'NCESDist' = '0613360' AND T1.'EdOpsCode' = 'SPECON');

The role of the Schema Critic becomes particularly evident when assessing the SQL query's accuracy and schema alignment. The Critic not only confirms that the predicted SQL accurately reflects the user's intent but also verifies that it adheres to the established schema by appropriately referencing the necessary tables and columns. For example, the inclusion of the 'EdOpsCode' column is justified as it is crucial for filtering results specific to the type of school queried. Additionally, the Critic evaluates the relationships between the 'schools' and 'frpm' tables, ensuring that the primary and foreign key constraints are respected. By generating constructive feedback and highlighting any potential improvements, such as enhancing query readability with table aliases, the Schema Critic reinforces the overall robustness and clarity of the SQL generation process, leading to higher execution accuracy and reliability in complex querying scenarios.

5 Conclusion and Future Work

This paper presents an effective approach to enhancing Natural Language to SQL (NL-to-SQL) tasks through the integration of Question Decomposition (QD) and a Critic agent within a small language model framework. Our experiments demonstrated that the Llama-3.1-8B model, augmented by the `LiteMARS` framework, achieves execution accuracies comparable to larger models such as Deepseek-coder-33B, with our model recording an execution accuracy of 32.06% compared to 34.46% for the larger model. These results highlight the potential of smaller models to perform competitively in NL-to-SQL applications, while also showcasing the effectiveness of combining QD and Critic strategies to improve query handling across various complexity levels.

As a future work, we will focus on enhancing the performance of our approach. We aim to incorporate additional contextual information to improve accuracy in complex queries and explore the integration of reinforcement learning techniques to optimize the Critic agent's feedback mechanism. Additionally, we will evaluate the generalizability of our model across diverse datasets beyond the BIRD and Spider datasets. This research advocates for a shift towards more resource-efficient language models that do not compromise performance.

Limitations

In this section we elaborate on the limitations of our work. First and foremost, while the BIRD and Spider datasets provide a robust foundation, the model may struggle with rare query patterns or domain-specific languages that are underrepresented in these datasets. This limitation suggests that the model's effectiveness may not generalize well to all real-world applications.

Additionally, while integrating Query Decomposition (QD) and the Critic agent has shown to improve performance, the overhead (in terms of latency) associated with these processes might need to be optimized in scenarios requiring real-time SQL generation. Future research should explore optimizing these techniques to minimize processing time while retaining accuracy. Lastly, our experiments primarily focus on a few specific small language models. While results indicate competitive performance, further studies are needed to evaluate the scalability of our approach across a broader range of models and architectures, ensuring its applicability in various contexts.

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*.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *arXiv preprint arXiv:2310.11511*.

Ben Bogin, Matt Gardner, and Jonathan Berant. 2019. Global reasoning over database structures for text-to-sql parsing. *arXiv preprint arXiv:1908.11214*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Wenhu Chen, Xiang Lin, Yuwei Ma, Xinyi Li, Raymond Mooney, and William Yang Wang. 2020. Language models are few-shot table reasoners. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7485–7498.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. 2024. Agent-flan: Designing data and methods of effective agent tuning for large language models. *arXiv preprint arXiv:2403.12881*.

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*.

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

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. *arXiv preprint arXiv:1905.08205*.

M Lewis. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023a. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023b. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 13067–13075.

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.

Xinya Li, Hanjie Sun, Alexey Protasov, Yuan Shi, and Lei Qiu. 2023c. Crescendo: Improving text-to-sql semantic parsing via multi-granularity schema linking. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*.

Xuechen Liang, Meiling Tao, Yinghui Xia, Tianyu Shi, Jun Wang, and JingSong Yang. 2024. Cmat: A multi-agent collaboration tuning framework for enhancing small language models. *arXiv preprint arXiv:2404.01663*.

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

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.

Microsoft. 2024. Discover the new multi-lingual, high-quality phi 3.5 slms. Accessed: 2024-10-16.

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

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

Mohammadreza Pourreza and Davood Rafiei. 2024a. Din-sql: decomposed in-context learning of text-to-sql with self-correction. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Mohammadreza Pourreza and Davood Rafiei. 2024b. Dts-sql: Decomposed text-to-sql with small large language models. *ArXiv*, abs/2402.01117.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 6:3.

Tao Shi, Xi Victoria Lin, Tatsunori Yeh, Huan Sun, Chin-Yew Lee, and Xiang Ren. 2020. Learning compositional representations for few-shot natural language to sql task. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 487–498.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.(2023). *arXiv preprint cs.AI/2303.11366*.

Ruoxi Sun, Sercan Ö Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, et al. 2023. Sql-palm: Improved large language model adaptation for text-to-sql (extended). *arXiv preprint arXiv:2306.00739*.

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, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

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

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

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. 2023. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2023. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 744–755.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

Pei Zhou, Jay Pujara, Xiang Ren, Xinyun Chen, Heng-Tze Cheng, Quoc V Le, Ed H Chi, Denny Zhou, Swaroop Mishra, and Huaixiu Steven Zheng. 2024. Self-discover: Large language models self-compose reasoning structures. *arXiv preprint arXiv:2402.03620*.

A Performance Comparison with and without Query Decomposition

In Table 4, we compare the performance of several models on the BIRD dataset, both with and without Query Decomposition (QD). The results highlight the effectiveness of the QD method, particularly in improving execution accuracy and column recall. Please note that w/o QD is the Vanilla setting mentioned in the baselines section **??**.

For instance, using Llama-3-70B, QD boosts execution accuracy by 33.74%, demonstrating the substantial impact of employing a decomposition approach at the schema linking stage. This method enhances the model's ability to decompose complex natural language questions and link them to the appropriate data schema, resulting in more accurate SQL generation.

One key observation from the results is the importance of column recall in generating correct SQL queries. Column recall measures the model's ability to correctly identify all relevant columns required to form the SQL query. Missing even a single column typically results in an incorrect SQL query, and the tables confirm that higher column recall generally correlates with better execution accuracy.

Interestingly, we see that as model size increases, the ability to recall the correct columns also improves. Though Llama-3.1-8B achieves a column recall score comparable to that of the larger Llama-3-70B, its execution accuracy is lower. This suggests that Llama-3.1-8B's SQL generation capabilities are limited despite its strong schema linking, indicating a gap in reasoning or query construction. In contrast, the Llama-3.1-70B model shows minimal improvement with QD, likely because it already exhibits strong reasoning abilities, requiring less decomposition to perform well.

In some cases, precision without QD is higher, indicating that while the model selected fewer columns, they were more likely to be correct. However, this also means that some important columns were missed, explaining why the precision score was not always accompanied by a higher execution accuracy. This highlights the trade-off between precision and recall in SQL generation tasks.

B Performance Comparison with and without Schema Critic

Table 5 demonstrates that incorporating the Schema Critic improves performance across all models. Granite-8B exhibits the most significant gains in Execution Accuracy (EX) and Column Precision, with improvements of 21.19% and 15.73%, respectively. For Column Recall, Mistral benefits the most, achieving a 24.52% improvement. In contrast, the newer Llama-3.1-8B model shows comparatively smaller gains. To make it clear the w/o SchC is our **Vanilla + QD** setting that we have mentioned in the section **??** and w/ SchC is the **LiteMARS w/o SC** setting

C Implementation Details

The context length varied among the models: Llama v3 models feature an 8k context length, while Llama v3.1 models extend this to a maximum of 128k. The Deepseek-coder language models offer context lengths of 4k and 16k for their Small and Large versions, respectively. The ibm-granite-8b model supports an extensive context length of 128k, and the mistral-7b accommodates a 32k context limit.

All experiments were conducted using Nvidia Tesla A100 GPUs with a batch size of 4. We implemented the Fully Sharded Data Parallel (FSDP) technique available in the HuggingFace accelerator with default settings to optimize our computational resources and enhance training efficiency. We employed greedy decoding for reproducibility, i.e., do sample parameter is set to False.

Models	Execution Accuracy		Column Precision		Column Recall	
	w/o QD	w/ QD	w/o QD	w/ QD	w/o QD	w/ QD
Llama-3-8B	0.1621	**0.1738**	0.7421	**0.7674**	0.6425	**0.6697**
Llama-3.1-8B	0.2464	**0.2626**	**0.7340**	0.7221	0.7926	**0.8206**
Llama-3-70B	0.2924	**0.3900**	**0.6310**	0.5567	0.7692	**0.8314**
Llama-3.1-70B	0.4970	**0.5000**	0.8700	**0.8942**	0.8470	**0.8563**

Table 4: Comparison of models with and without Question Decomposition (QD) on the BIRD dataset using the Llama family of models. **QD** refers to Question Decomposition. In this experiment, the same model was used across all modules. Notably, for Llama-3.1-8B, the scores for the w/o QD setup in this table differ from the Vanilla scores in Table 1, as the latter utilized the DeepSeek-Coder-7B model for SQL generation.

Models	Execution Accuracy		Column Precision		Column Recall	
	w/o SchC	w/ SchC	w/o SchC	w/ SchC	w/o SchC	w/ SchC
Llama-3.1-8B	0.2626	**0.2980**	0.7221	**0.7231**	0.8206	**0.8295**
Mistral-7B	0.2449	**0.2661**	0.6919	**0.7324**	0.6590	**0.8206**
Granite-8B	0.1595	**0.1933**	0.5193	**0.6010**	0.6273	**0.7015**

Table 5: Comparison of models with and without Schema Critic on the BIRD dataset. **SchC**: Schema Critic. All modules utilize the same model.

D Comparison of State-of-the-Art Methods on the BIRD

From Table 6, it is evident that proprietary models excel in SQL generation from natural language queries and their corresponding databases. Among these, CHASE-SQL (Pourreza et al., 2024b), a Gemini-based model (Team et al., 2023), achieves the highest execution accuracy of 74.46%, significantly outperforming other GPT-4-based models. This result aligns with expectations, given the extensive number of parameters and the large-scale data used for training such proprietary systems. Notably, CHESS (Talaei et al., 2024) employs a multi-agent framework similar to ours but leverages much larger models like GPT-4 and Llama-3-70B for its experiments. Meanwhile, Distillery (Maamari et al., 2024), built on GPT-4o, highlights the limitations of schema linking in large models, emphasizing its inefficiency in certain scenarios.

Fine-tuned models demonstrate competitive performance as well, owing to their task-specific optimization for SQL generation. For instance, DTS-SQL (Pourreza and Rafiei, 2024b), which shares a decomposition-based approach similar to ours, divides the NL-to-SQL generation task into two key subtasks: Schema Linking and SQL Generation. By fine-tuning these modules independently, DTS-SQL achieves superior results compared to general-purpose models. This modular design reflects the advantages of tailoring models for specific components of the SQL generation pipeline.

While fine-tuned models and proprietary systems dominate in terms of execution accuracy, our approach aims to bridge this gap by improving reasoning capabilities in smaller, open-source models using their pretrained knowledge.

Category	Model	Execution Accuracy (EX)
Fine-tuned Models	SFT CodeS-15B (Li et al., 2024a)	58.47%
	DTS-SQL + DeepSeek 7B (Pourreza and Rafiei, 2024b)	55.80%
Proprietary-based Models	CHASE-SQL + Gemini (Pourreza et al., 2024b)	**74.46%**
	CHESS + GPT-4 (Talaei et al., 2024)	68.31%
	Distillery + GPT-4o (Maamari et al., 2024)	67.21%
Ours	`LiteMARS` + Llama-3.1-8B	32.06%

Table 6: Comparison of State-of-the-Art Methods on the BIRD Dev Dataset based on Execution Accuracy (EX)

E Prompts Used for Various Modules

The following figures illustrate the prompts used across different modules in our framework.

[Task]: Given the following question, decompose it into minimal, linear, and dependent sub-questions. Each sub-question should logically depend on the previous one, and the entire sequence should be suitable for generating an SQL query. The output should be in JSON format for easy parsing.

[Example]

{example}

Now decompose the following question into the minimal number of linear and dependent sub-questions, following the specified JSON output (start with ```json and end with ```) format similarly:

Question: {question}

Figure 3: Sub-question Prompt

~~~
[Task]: From the provided JSON content of the database schema, extract
ALL the relevant columns from the tables that will be useful for
answering the questions. Use Hint for additional context which helps in
disambiguating the column names.

Questions: {sub_questions}
Hint: {evidence}
Database Schema: {json_schema}
Primary Keys: {primary_keys}
Foreign Keys: {foreign_keys}

[Output Format]:
1. Output only a JSON object with the relevant columns, in the following
format (start with ```json and end with ```):
    ```json
    [
        {
            "first": "first sub question",
            "relevant_schema": {
                "table_name_1": ["relevant original_column_name 1",
"relevant original_column_name 2", ...],
                "table_name_2": ["relevant original_column_name 1",
"relevant original_column_name 2", ...],
                ...
            }
        },
        {
            "second": "second sub-question",
            "relevant_schema": {
                "table_name_1": ["relevant original_column_name 1",
"relevant original_column_name 2", ...],
                "table_name_2": ["relevant original_column_name 1",
"relevant original_column_name 2", ...],
                ...
            }
        },
        ...
    ]
    ```

[Instructions]:
1. Use "column_name", "column_description", "value_description", and
"sample_values" to determine which columns are necessary.
2. Double check the generated JSON is syntactically correct.
3. Ensure the selected columns are consistent with the given table
content, i.e., make sure the selected columns exist in the table.
4. Ensure the columns names are exactly as sepcified in the schema.
Especially columns names which contains whitespaces and brackets.
5. Ensure that the necessary relationships between tables are maintained
using primary and foreign keys.
6. Do not include any additional text or information.


[Response]:
~~~

Figure 4: Column Selector Prompt

~~~
[Task]: From the provided JSON content of the table, extract ALL the
relevant columns from the table that will be useful for answering the
questions. Use Hint for additional context which helps in disambiguating
the column names.

Questions: {sub_questions}
Hint: {evidence}
Table Content: {json_schema}
Primary Keys: {primary_keys}

[Output Format]:
1. Output only a JSON object with the relevant columns, in the following
format (start with ```json and end with ```):
    ```json
    [
        {
            "question": "first sub-question",
            "relevant_columns": ["relevant original_column_name 1",
"relevant original_column_name 2", ...],
            "reasoning": "a single line reasoning for why the columns are
relevant."
        },
        {
            "question": "second sub-question",
            "relevant_columns": ["relevant original_column_name 1",
"relevant original_column_name 2", ...],
            "reasoning": "a single line reasoning for why the columns are
relevant."
        },
        ...
    ]
    ```
2. If no columns are relevant return an empty list [] in
relevant_columns.

[Instructions]:
1. Use "column_name", "column_description", "value_description", and
"sample_values" to determine which columns are necessary.
2. Double check the generated JSON is syntactically correct. Only add
commas (,) when you are required to separate data in JSON.
3. Ensure the selected columns are consistent with the given table
content, i.e., make sure the selected columns exist in the table.
4. Ensure the column names are exactly as specified in the schema,
especially those that contain whitespaces and brackets.
5. Strictly adhere to the above output format. Do not include any
additional text or information.


[Response]:
~~~

Figure 5: Column Selector Table Prompt

[Task]: Refine the schema to include only the necessary tables and columns required to generate an SQL query for the given natural language question.

Question: {question}
Hint: {evidence}
Database Schema: {json_schema}
Primary Keys: {primary_keys}
Foreign Keys: {foreign_keys}

[Instructions]:

1. Identify Required Columns: Use "column_description", "value_description", and "sample_values" to determine which columns are necessary.
2. Select Relevant Tables: Choose tables that contain the identified columns. Ensure that the necessary relationships between tables are maintained using primary and foreign keys.
3. Prune Unnecessary Elements: Remove tables and columns from the schema that are not required for generating the SQL query.
4. Make sure the refined schema is consistent with the given schema, i.e., make sure the columns exist in the provided schema.
5. Ensure the column names are exactly as specified in the schema. Especially column names that contain whitespaces and brackets.
6. Ensure the generated JSON is syntactically correct. The property name should be enclosed in double quotes.

[Output Format]:
Output the refined schema, ensuring only the required tables and columns are included, in the following JSON format (start with ```json and end with ```):

~~~
    ```json
    {
        "reasoning": "reasoning for why tables and/or columns were
removed in a single line.",
        "relevant_schema":{
            "relevant_table_1": ["relevant original_column_name_1",
"relevant original_column_name_2", ...],
            "relevant_table_2": ["relevant original_column_name_1",
"relevant original_column_name_2", ...],
            ...
        }
    }
    ```
~~~

[Response]:

Figure 6: Schema Refiner Prompt

[Task]: Decompose the given Question into meaningful sub-questions and generate corresponding sub-SQL queries until the SQL corresponding to the Question is generated. Generate the sqlite query based on the given question, hint, and database schema. Use primary and foreign keys to ensure proper joins.

[Instructions]:
1. Use the question to determine the main objective of the query.
2. Use the hint to disambiguate column names and ensure the correct columns are selected.
3. Identify the root verb, subject, object, named entities, and key phrases and use that to construct the query conditions.
4. Use the primary and foreign keys to define the necessary joins between tables.
5. Ensure that the column names and table names with whitespaces or special characters are surrounded by backticks (`). Example: `County Name`.
6. Refer to the Schema Information for details regarding the tables and column names.
7. Only include relevant tables and columns needed for generating the query.
8. If you are doing a logical operation on a column, such as mathematical operations and sorting, make sure to filter null values within those columns.
9. Construct and output the SQL query.
10. Strictly adhere to the below output format. Do not include any additional text or information.

[Output format]:
1. Output in the following JSON format (start with ```json and end with ```):
~~~
    ```json
    {
        "first": {
            "question": <sub-question 1>,
            "SQL": "sub-SQL 1 in a single line"
            },
        "second": {
            "question": <sub-question 2>,
            "SQL": "sub-SQL 2 in a single line"
            },
        ...
    }
    ```
~~~
2. Ensure the generated JSON is syntactically correct. The property name should be enclosed in double quotes. Only add commas (,) when you are required to separate data in JSON.

[Example]:
{example}

Now solve the following Question similarly:

Question: {question}
Hint: {evidence}
Database Schema:
{schema}

Output:

Figure 7: SQL Generator Prompt

[Task]: Verify whether the predicted SQL query is appropriate for answering the provided question. Conduct a thorough evaluation based on the schema, hint, and key relationships.

[Inputs]:

- **Question**: {question}
- **Schema**: {pred_schema}
- **Hint** (**Evidence**): {evidence}
- **Predicted SQL**: {pred_sql}
- **Sqlite3 Error**: {pred_error}

[Instructions]:

1. SQL Query Accuracy: Analyze the predicted SQL query to determine if it correctly answers the question based on the given schema and question context.
2. Schema Alignment: Verify that the SQL query properly references the correct tables and columns from the schema. Ensure that the query covers all necessary attributes, joins, and conditions.
3. Key Relationships: Check that the predicted SQL respects primary and foreign key constraints, ensuring proper relationships between tables.
4. Hint Validation: Compare the predicted SQL to the provided hint and ensure that the query reflects any key information or requirements mentioned in the hint.
5. Feedback Generation: Provide thorough feedback covering:
   - Correct elements in the SQL query.
   - Any missing or incorrect components (e.g., tables, joins, conditions).
   - Suggestions for improving the SQL to better match the question.

[Feedback]:

Figure 8: SQL Critic Agent Prompt

```
**Task:**
Analyze the input and determine if any columns from the available schema
should be added to the filtered schema to improve SQL generation
accuracy. Provide a brief justification for each suggested addition.

**Input:**
1. Available schema: [List of columns not in the filtered schema]
2. Filtered schema: [List of columns already selected]
3. User question: [The natural language query]
4. Evidence: [Any relevant context or information]
5. Primary and foreign keys: [List of primary and foreign key
relationships]

**Guidelines:**
1. Focus on columns that are likely to be necessary for the SQL query
based on the user question and available information.
2. Consider relationships between tables when suggesting columns.
3. Provide concise justifications, typically one sentence per column.

**Output format:**
Return a JSON object with the following structure:
{
  "additional_columns": [
    {
      "column_name": "string",
      "justification": "string"
    },
    ...
  ]
}

If no additional columns are needed, return an empty list for
"additional_columns".

Example output:
{example}

Now, based on the input provided, suggest any additional columns that
should be included in the schema without generating any preamble.

**Input:**
Available schema: {available_schema}
Filtered schema: {filtered_schema}
Primary Keys:{primary_keys}
Foreign Keys:{foreign_keys}
User question: {question}
Evidence: {evidence}
Output:
```

Figure 9: Schema Critic Prompt