Divide and Conquer: Harnessing Small Agents for Schema Extraction in Text-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, the first multi-agent framework to incorporate schema linking within an NL-to-SQL system. 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 iterative refinement process, which enhances both schema extraction and SQL generation. In experiments, we found that critic-based refinement significantly improved column recall by 4.7% and execution accuracy by 28.2% for NL-to-SQL generation, including in open-source small LLMs such as Llama-8B, Mistral-7B, and Granite-8B. Further analysis, we observe that our LiteMARS agent achieves comparable performance to large language models like Deepseekcoder-33B while being less resource-intensive in terms of GPU required and memory consumption.

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 Natural Language to SQL (NL-to-SQL) systems an attractive solution. Toward 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., 2018b; Li et al., 2023b; 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 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). However, question decomposition alone also does not address the schema-linking problem head-on.

In this work, we propose *LiteMARS* agent, the

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

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) SQL generation stage. Importantly, LiteMARS leverages two critics for performing iterative refinement within this pipeline. The *Schema Critic* agent reviews the extracted schema and provides execution feedback to refine 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.

To summarize, our key contributions are as follows:

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

2. Our multi-agent framework achieves an improvement of 7% in execution accuracy with LLama-3.1-8B, 2.5% with Mistral-7B, and 8.83% with Granite-8B models for Schema Linking. Additionally, this includes an 5.12% increase from *QD* + *Schema Critic* and a 3.71% boost through *SQL Critic*.

3. Furthermore, we find that our novel *Schema Critic* agent detects and corrects SQL errors based on execution feedback in the SQL generation phase, leading to an 8.83% improvement in execution accuracy for models with limited SQL generation capabilities, providing a robust solution for environments with limited resources.

2 Methodology

In Figure 1, 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 for improving schema linking. SQL Critic and SQL generation agents are used to improve NL-to-SQL generation. In the figure, we show how different agents interact in a collaborative way to improve NL-to-SQL generation.

2.1 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.

2.2 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. 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:

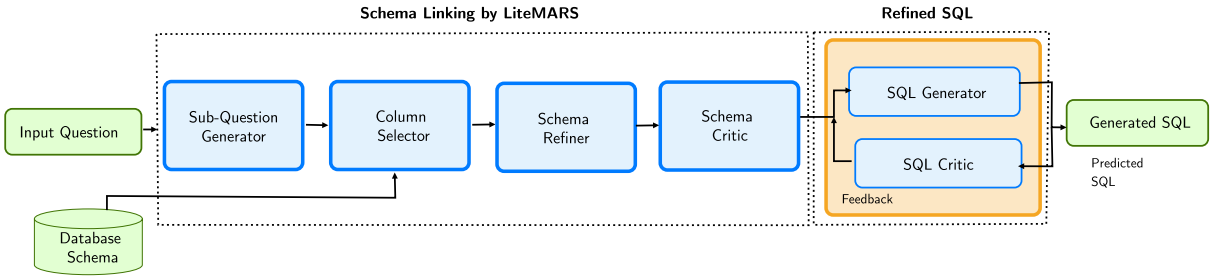

Figure 1: Overview of our proposed `LiteMARS Agent` for NL-to-SQL Generation

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

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

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

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

2.3 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. The Schema Refiner takes as input the pruned schema D and the original question q, and outputs a refined schema $'$ where:

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

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

2.4 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.

2.5 SQL Generator

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

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

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

2.6 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 $\hat{S}$, and the execution error e (if any) as inputs, and produces a corrected SQL $\hat{S}_{\text{final}}$ by identifying potential mistakes in the SQL query:

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

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

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.

3 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. (2024)). 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 smaller language models (<15B parameters) 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., 2018a) 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 achieves higher performance than the model double or triple of 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. (2023a)). 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 in NL-to-SQL: 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 has gained traction (Wu et al. (2023); Shinn et al. (2023); Zhou et al. (2023); Xu et al. (2023)) across various NLP tasks. Cheng et al. (2024) introduced HaluAgent, an LLM-based autonomous agent framework for hallucination detection using relatively smaller LLMs. In the NL-to-SQL domain, Wang et al. (2024) leveraged an agent-based approach in their MAC-SQL framework, allowing more flexible and robust query generation.

4 Experiments, Results and Analysis

4.1 Setup

The performance of our pipeline was evaluated using a selection of recent Small and Large Language Models from diverse model families, including Meta-Llama, DeepSeek-Coder, IBM-Granite, and Mistral. The Small Language Models comprised Meta-Llama-3-8B-Instruct, Llama-3.1-8B-Instruct, deepseek-coder-7b-instruct-v1.5, granite-8b-code-instruct-128k, and Mistral-7B-Instruct-v0.3. The Large Language Models included Meta-Llama-3-70B-Instruct, Llama-3.1-70B-Instruct, and Deepseek-coder-33b-instruct.

4.2 Datasets and Metrics

Dataset: Our evaluation utilized two challenging and widely recognized datasets: Spider and BIRD. Spider, introduced by Yu et al. (2018a), 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,

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	0.2727	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., 2018a) Dataset with different SLMs with QD and Critic Agents. **SC**: SQL Critic.

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.

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 generation 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. **w/ Oracle:** This is a variation of vanilla setting in which we provide the oracle schema extracted from the ground truth SQL present in the dataset. This allows us to assess the maximum execution accuracy that the model can achieve when the ideal schema is utilized.

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 a variation of our `LiteMARS 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.

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., 2018a), using multiple Small LLMs integrated with Question 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 with the full pipeline (LiteMARS). For SQL generation and SQL Critic, the Deepseek-coder-7b-instruct-v1.5 model was used across all experiments.

4.4.1 Impact of Question Decomposition and Schema Critic

Table 1 highlights the significant improvements brought by the LiteMARS pipeline when evaluated on the BIRD dataset. The execution accuracy, a key metric reflecting the model's ability to generate correct SQL queries, showed substantial gains across all tested models when using the LiteMARS configuration. For instance, with Llama-3.1-8B, the execution accuracy improved from 0.2501 in the Vanilla setup to 0.3206 in the full LiteMARS pipeline—a marked improvement of over 28.2%. Similarly, Mistral-7B showed an increase from 0.2727 to 0.2985 (9.2%) and Granite-8B from 0.1421 to 0.2304 (62.0%).

Moreover, the LiteMARS pipeline also significantly boosted column-related metrics, critical for ensuring that the model correctly identifies and links relevant database columns when generating SQL queries. With the addition of Natural Language Query Decomposition and Schema Critic, Llama-3.1-8B achieved a column recall of 0.8295, showing a noticeable improvement from 0.7926 in the Vanilla configuration (4.7%). This increase in column recall suggests that LiteMARS enables more accurate identification of the necessary columns, a crucial factor for improving SQL query correctness, especially when working with structured databases. Precision similarly saw a rise from 0.7340 in the Vanilla setup to 0.7231 with LiteMARS w/o SC, and maintained the same precision in the full LiteMARS pipeline, indicating that LiteMARS effectively reduces the likelihood of incorrect or irrelevant column selections.

4.4.2 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 Llama-3.1-8B model achieved an execution accuracy of 0.5386 with the full LiteMARS pipeline, a significant jump from 0.4528 in the Vanilla configuration (18.9%). Similarly, Mistral-7B exhibited an increase from 0.4314 to 0.4971 (66.5%), and Granite-8B improved from 0.3600 to 0.4305 (43.1%) after the addition of SQL Critic.

In addition to execution accuracy, column recall saw a notable improvement from 0.6468 in the Vanilla setup to 0.7073 with the full LiteMARS pipeline, reflecting its superior ability to handle schema linking and column selection. These improvements, both in execution accuracy and column-related metrics, highlight the robustness of the full LiteMARS pipeline in significantly enhancing the performance of Small LLMs like Llama-3.1-8B. Furthermore, the comparison between LiteMARS with and without the SQL Critic (SC) emphasizes the added value of incorporating the SC stage in refining and critiquing the SQL generation, further boosting the model's ability to generate correct SQL queries and accurately recall columns.

In summary, the results from both datasets 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 Text-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.

Table 3 shows the effects of Question 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 (w/ QD + Critic), 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 from QD alone (0.1982) compared to the Critic (0.1724), while Mistral-7B and Granite-8B exhibit the most significant enhancements when both QD and Critic are applied. Mistral-7B's performance, for

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

example, improves from 0.1034 (Vanilla) to 0.1508 (w/ QD + Critic).

For **challenging queries**, both QD and the Critic agent contribute substantially to the models' performance. LLaMA-3.1-8B's accuracy increases from 0.1103 (Vanilla) to 0.1586 (w/ QD) but shows a slight decrease with the addition of the Critic to 0.1517. Mistral-7B and Granite-8B both experience upward trends, with Granite-8B demonstrating a significant improvement from 0.0965 (w/ QD) to 0.1448 (w/ QD + Critic), highlighting the effectiveness of the Critic step in addressing more complex queries.

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, while still providing improvements for challenging ones.

In conclusion, Question Decomposition and Critic provide complementary benefits in improving both the SQL generation and schema linking processes. Larger models such as LLaMA-3.1-70B show less dependence on decomposition yet still gain from the Critic, while smaller models like Mistral-7B and Granite-8B benefit significantly from both techniques.

4.5 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.

Specifically, when we provide the complete schema to Deepseek-coder-33B, we attain an execution accuracy of 34.46%. In contrast, our LiteMARS setup with LLaMA-3.1-8B yields a competitive execution accuracy of 32.06%. This result is particularly noteworthy as it demonstrates that smaller models, when augmented with an effective framework like LiteMARS, can perform at levels close to their larger counterparts.

The implications of this finding are significant: it suggests that smaller models can be a viable option for NL-to-SQL tasks, offering a balance between resource efficiency and performance. Consequently, our research advocates for the use of optimized small agents, as they can deliver robust performance without the computational overhead associated with larger models.

4.6 Qualitative Analysis of the Effectiveness of Schema Critic

The Schema Critic demonstrates significant effectiveness in refining the SQL generation process, particularly in handling complex queries such as retrieving the lowest grade for the District Special Education Consortia School with a specific 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 first sub-question identifies the relevant school district identification number, while the second establishes the connection between the district and the school's name. The final sub-question 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

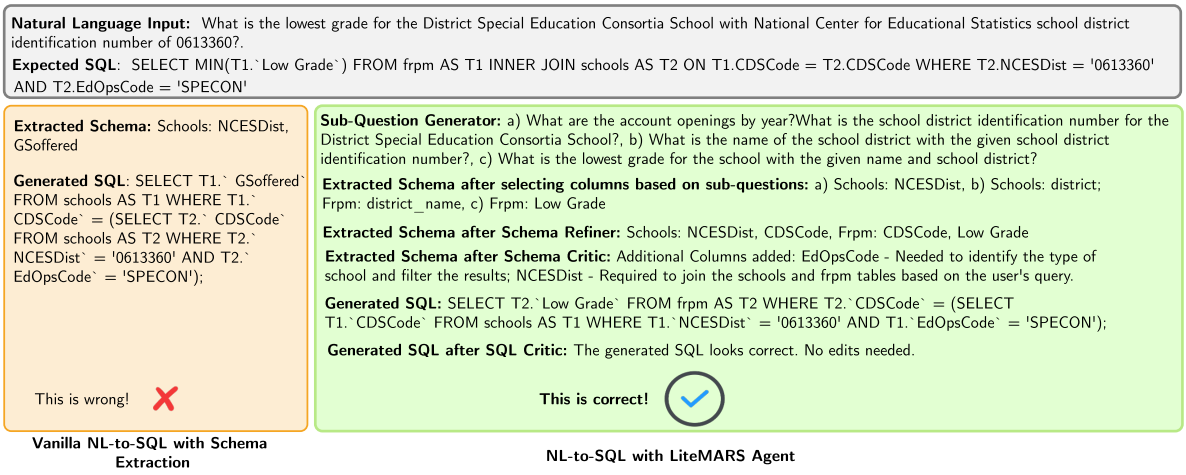

Figure 2: An example showing how `LiteMARS Agent` improves Schema Linking for NL-to-SQL Generation

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, paving the way for advancements in SQL generation capabilities.

6 Limitations

In this section we elaborate on the limitations of our work. First and foremost, the performance of our model is highly dependent on the quality and completeness of the training data. 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 Question Decomposition (QD) and the Critic agent has shown to improve performance, the overhead 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*.

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.

Xiaoxue Cheng, Junyi Li, Wayne Xin Zhao, Hongzhi Zhang, Fuzheng Zhang, Di Zhang, Kun Gai, and Ji-Rong Wen. 2024. Small agent can also rock! empowering small language models as hallucination detector. *arXiv preprint arXiv:2406.11277*.

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*.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. 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. 2023b. 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)*.

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.

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. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv 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.

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. 2018a. 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.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, Yanlai Ma, James Li, Irene Zhang, and Dragomir Radev. 2018b. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3911–3921.

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*.

A Example Appendix

A.1 Performance Comparison with and without Question Decomposition

In Table 4, we compare the performance of several models on the Bird dataset, both with and without Question Decomposition (QD). The results highlight the effectiveness of the QD method, particularly in improving execution accuracy and column recall.

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

Metrics	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	**0.2727**	**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 on the Bird dataset. **QD**: Question Decomposition. All modules utilize the same model.

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.

A.2 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.

A.3 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