MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL

Anonymous ACL submission

Abstract

Recent LLM-based Text-to-SQL methods usually suffer from significant performance degradation on "huge" databases and complex user questions that require multi-step reasoning. Moreover, most existing methods neglect the crucial significance of LLMs utilizing external tools and model collaboration. To address these challenges, we introduce MAC-SQL, a novel LLM-based multi-agent collaborative framework. Our framework comprises a core decomposer agent for Text-to-SQL generation with few-shot chain-of-thought reasoning, accompanied by two auxiliary agents that utilize external tools or models to acquire smaller sub-databases and refine erroneous SQL queries. The decomposer agent collaborates with auxiliary agents, which are activated as needed and can be expanded to accommodate new features or tools for effective Text-to-SQL parsing. In our framework, We initially leverage GPT-4 as the strong backbone LLM for all agent tasks to determine the upper bound of our framework. We then fine-tune an open-sourced instruction-followed model, SQL-Llama, by leveraging Code Llama 7B, to accomplish all tasks as GPT-4 does. Experiments show that SQL-Llama achieves a comparable execution accuracy of 43.94, compared to the baseline accuracy of 46.35 for vanilla GPT-4. At the time of writing, MAC-SQL+GPT-4 achieves an execution accuracy of 59.59 when evaluated on the BIRD benchmark, establishing a new state-of-the-art (SOTA) on its holdout test set [1].

1 Introduction

Text-to-SQL aims to automate the process of generating Structured Query Language (SQL) queries for databases from natural language text. This long-standing challenge is essential for improving database accessibility without requiring expertise of SQL (Qin et al., 2022; Sun et al., 2023).

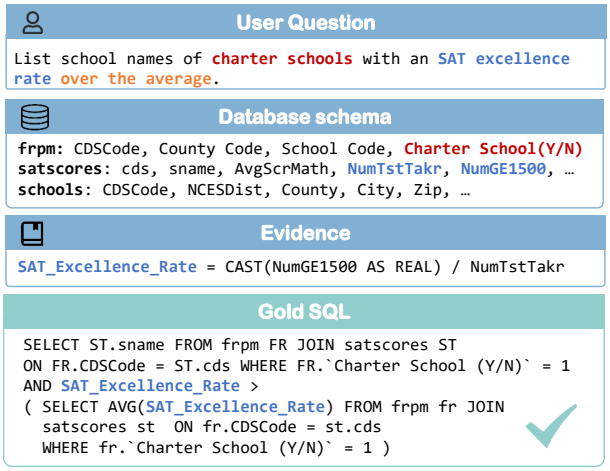

Figure 1: A complex example of Text-to-SQL. In the Gold SQL, we use `SAT_Excellence_Rate` to represent "CAST(NumGE1500 AS REAL)/NumTstTakr" for the sake of brevity.

Over the past decade, research in this field has progressed through three stages. In the initial phase, systems encodes input sequence utilizing pre-trained models, and SQL queries are decoded using either abstract syntax trees (Xu et al., 2017; Guo et al., 2019; Wang et al., 2021) or predefined sketches (He et al., 2019). More recent systems (Raffel et al., 2023; Xie et al., 2022; Scholak et al., 2021) have adopted sequence-to-sequence methodologies. The latest research (Ouyang et al., 2022; OpenAI, 2023; Rozière et al., 2023) has demonstrated the remarkable capabilities of Large Language Models (LLMs) in this task. The success of these models can be ascribed to their emerging abilities (Wei et al., 2023; Brown et al., 2020) and robust reasoning capabilities inherent in LLMs.

Recent research on LLM-based Text-to-SQL (Dong et al., 2023; Pourreza and Rafiei, 2023; Gao et al., 2023) has mainly concentrated on In-Context Learning prompt strategies and supervised fine-tuning using data derived from the target domain. However, these approaches usually

[†]Corresponding Author

[1] https://anonymous.4open.science/r/MAC-SQL-61EC

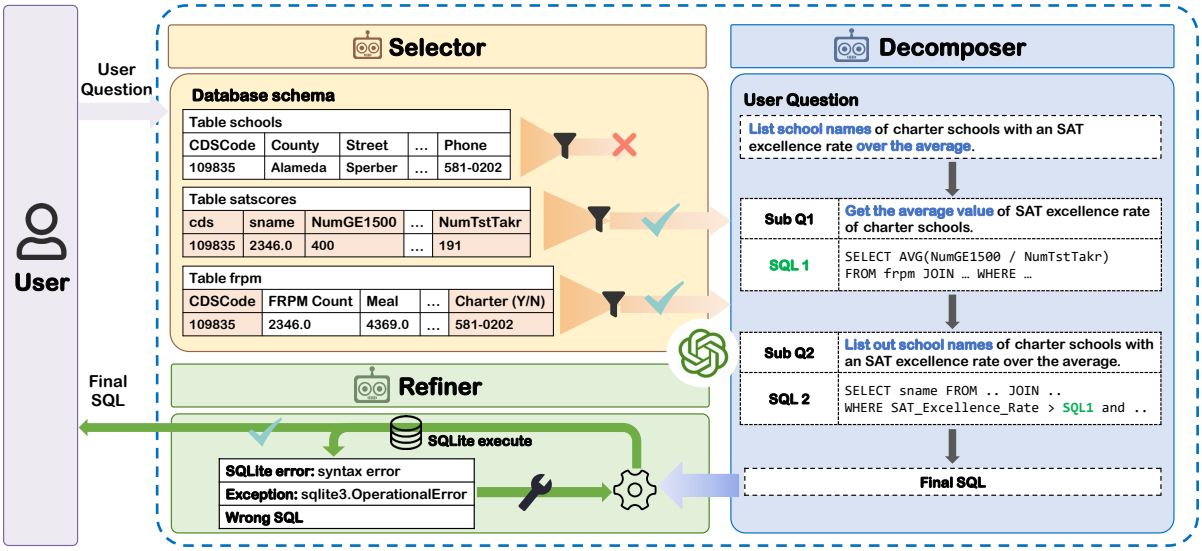

Figure 2: The overview of our MAC-SQL framework, which comprises three agents: (i) the *Selector*, which decomposes a large database into smaller sub-database to mitigate the interference of irrelevant information, and (ii) the *Decomposer*, which breaks down a complex question into simpler sub-questions and resolves them progressively by chain-of-thought reasoning, and (iii) the *Refiner*, which uses an external tool for SQL execution and obtains feedback, then refines faulty SQL queries.

suffer from significant performance degradation on "huge" databases and complex user questions that require multi-step reasoning, as demonstrated in Figure 1. Moreover, most existing methods neglect the crucial significance of LLMs utilizing external tools and model collaboration.

To alleviate the above challenges, we introduce MAC-SQL, a novel LLM-based multi-agent collaborative framework, which exploit LLMs as intelligent agents with different functionalities for effective Text-to-SQL parsing. Our framework comprises a core *Decomposer* agent for Text-to-SQL generation, accompanied by two auxiliary agents, the *Selector* and the *Refiner*, for tool usage and SQL refinement. Specifically, the Decomposer breaks down a complex question into simpler sub-questions and resolves them progressively by chain-of-thought reasoning. When necessary, the Selector decomposes a large database into a smaller sub-database to minimize the interference of irrelevant information, while the Refiner employs an external tool for SQL execution, obtains feedback, and refines erroneous SQL queries.

Furthermore, we have fine-tuned an instruction-followed model, SQL-Llama, by leveraging Code Llama 7B, using agent instruction data from MAC-SQL, thus enabling capabilities in database simplification, question decomposition, SQL generation, and SQL correction. We have also released the agent instruction data, which is derived from the training sets of BIRD and Spider and is generated through multi-agent tasks.

In our experiments, we initially leverage GPT-4 as a strong backbone LLM for all agent tasks determine the upper bound of our MAC-SQL framework on the widely used BIRD and Spider dataset. Experimental results demonstrate that MAC-SQL+GPT-4 achieves an execution accuracy of 59.59 on the holdout test set of BIRD, establishing a new state-of-the-art (SOTA) at the time of writing. Furthermore, We utilize SQL-Llama(7B) to accomplish all tasks in a manner similar to GPT-4. Surprisingly, despite SQL-Llama having an order of magnitude fewer parameters than GPT-4, its execution accuracy reaches 43.94, which is remarkably close to the accuracy of GPT-4 (46.35).

Contribution Our main contributions and results are summarized as follows:

1. We propose MAC-SQL, a novel multi-agent collaborative framework for Text-to-SQL.

2. We introduce an instruction-tuning model, named SQL-Llama(7B), along with its associated agent-instruction dataset.

3. Experimental results demonstrate that MAC-SQL achieves state-of-the-art execution accuracy of 59.59% on the BIRD test set at the time of writing.

2 Task Formulation and Annotations

The problem of Text-to-SQL parsing involves generating an SQL query $\mathcal{Y}$ that corresponds to a given natural language question $\mathcal{Q}$ and is based on a database schema $\mathcal{S}$ and optional external knowledge evidence $\mathcal{K}$. The database schema $\mathcal{S}$ is defined as $\{\mathcal{T}, \mathcal{C}, \mathcal{R}\}$, where $\mathcal{T}$ represents multiple tables $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m\}$, $\mathcal{C}$ represents columns $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$, and $\mathcal{R}$ represents foreign key relations. Here, m and n denote the number of tables and column names, respectively. Finally, the Text-to-SQL task could be formulated as:

$$\mathcal{Y} = f(\mathcal{Q}, \mathcal{S}, \mathcal{K} \mid \boldsymbol{\theta}), \tag{1}$$

where the function $f(\cdot \mid \boldsymbol{\theta})$ can represent a model or neural network with the parameter $\boldsymbol{\theta}$.

Algorithm 1 The algorithm of MAC-SQL

Input: question q, database db, knowledge kg
Output: sql

```
if need simplify to database then
    db = LLM_Selector(q, db, kg)
end if
dbDesc = getDbRepresenation(db, kg)
subQs, subSQLs = LLM_Decomposer(q, dbDesc)
sql = subSQLs[-1]
count = 0
while count < maxTryTimes do
    ok, err = executeAndAnalyze(sql, db)
    if ok then
        return sql
    else
        sql = LLM_Refiner(q, dbDesc, sql, err)
    end if
end while
return sql
```

3 Methodology

3.1 Overview

We introduce MAC-SQL, a novel LLM-based multi-agent collaborative framework, which exploit LLMs as intelligent agents with different functionalities for effective Text-to-SQL parsing. As illustrated in Figure 2, our framework comprises a core *Decomposer* agent for Text-to-SQL generation, accompanied by two auxiliary agents, the *Selector* and the *Refiner*, for tool usage and SQL refinement.

Task Description

```
As an experienced and professional database
administrator, your task is to ...
```

Instruction

```
1. Discard any table schema that is not
related to the user question and evidence.
2. Sort the columns in each relevant table in
 descending order of relevance and keep the
top 6 columns.
3. Ensure that at least 3 tables are included
 in the final output JSON.
4. The output should be in JSON format.
```

Demonstration

```
[DB_ID] banking_system
[Schema] Table schemas of account, client,
loan, district ...
[Foreign keys] ...
[Question]: What is the gender of the
youngest client who opened account in the
lowest average salary branch?
[Evidence]: Later birthdate refers to younger
 age; A11 refers to average salary
[Answer]
```json
{ "account": "keep_all",
  "client": "keep_all",
  "loan": "drop_all",
  "district": ["district_id", "A11", "A2",
  ...] }
```
```

Test Question

```
[DB_ID] {db_id}
[Schema] {desc_str}
[Foreign keys] {fk_str}
[Question] {query}
[Evidence] {evidence}
[Answer]
```

Figure 3: An example of Selector prompt. The specific details are omitted for the sake of brevity.

Specifically, the Decomposer disassembles complex questions into simpler sub-questions and addresses them sequentially through chain-of-thought reasoning. If required, the Selector decomposes a large database into smaller sub-databases to minimize interference from irrelevant information. Meanwhile, the Refiner utilizes an external tool for SQL execution, acquires feedback, and refines any incorrect SQL queries.

We present the detailed algorithm of our framework in Algorithm 1, which introduce the collaboration of three agents. In the following section, a detailed introduction of three agents will be presented.

3.2 Selector

The Selector agent is designed to automatically decompose a large database into smaller sub-databases to minimize interference from irrelevant information. The rationale behind this selection process is crucial, particularly in real-world business contexts where databases comprise numerous

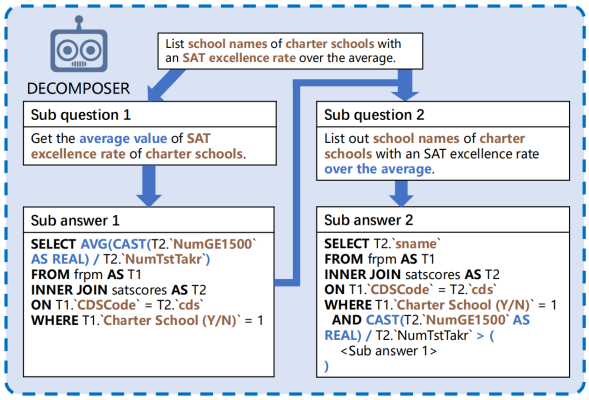

Figure 4: The Decomposer Agent Illustration.

tables and columns. The expansive nature of such schemas may surpass the processing capacity of a LLM within a single api call. Consequently, the Selector is designed to perform an initial filtration of relevant tables and columns to minimize interference from irrelevant information.

Figure 3 illustrates the Selector agent prompt, which is comprised of four components: task description, instruction, demonstrations, and a test example. The figure indicates that the anticipated output is a JSON string enumerating the table selections. These choices can be classified into three categories: "keep_all", "drop_all", or a relevant column list. The JSON output will subsequently be interpreted as the input for the new database schema for the Decomposer agent.

3.3 Decomposer

The primary purpose of the Decomposer is to systematically decompose complex questions into progressively refined sub-questions, which can then be solved individually. The original idea behind this is that, when analyzing error examples, the model encounters difficulty in reasoning and generating entirely accurate SQL for complex questions in a straightforward prompting. As a result, the generated SQL frequently exhibits flaws. Consequently, we naturally consider the approach of decomposing and solving the question, similar to the CoT method.

The Decomposer pattern can be approached in two prompting methods for text-to-SQL parsing: chain-of-thought (CoT) prompting (Wei et al., 2023) and least-to-most prompting (Zhou et al., 2022). The former involves generating thinking and reasoning once to obtain an answer, while the latter incurs higher computational costs to generate each SQL query due to the iterative process.

Due to the inefficiency of the iterative method and the need to determine when to stop, we adopt the CoT approach to generate sub-questions and their corresponding SQL. The specific implementation is as follows: dynamically judging the difficulty of the user's question, if it can be answered by a simple SQL query, then the SQL is generated directly. If the question is more complex, the corresponding SQL is generated starting from the simplest sub-problem, and then gradually broken down to obtain progressive sub-problems until the final SQL corresponding to the question is obtained. As shown in Figure 4, the question is decomposed into multiple sub-problems and solved one by one. In order to enhance LLM's understanding of instructions, we adopt the few-shot approach and conduct experiments on the entire BIRD dev dataset.

3.4 Refiner

The primary function of the Refiner is to detect and automatically rectify SQL errors. In a comprehensive multi-agent collaborative framework, particularly within the context of Text-to-SQL tasks, the inclusion of a Refiner is essential for the inspection and correction of generated answers. For instance, in the ChatDev project, intelligent agents are responsible for conducting overall and functional module testing in addition to overall architectural design and code writing for game software development tasks. Similarly, in Text-to-SQL tasks, the Refiner can be used to make appropriate adjustments due to the different data sets, database schemas, SQL generation styles, and specific inductive biases.

As shown in Figure 2, upon receiving an SQL query, the Refiner first diagnoses the SQL statement to assess its syntactic correctness, execution feasibility, and the retrieval of non-empty results from the database. If the check passes, the result is output as the final answer; otherwise, a correction operation is initiated, as shown in Figure 5. The corrected result then undergoes re-evaluation, and if issues persist, the process is repeated until the result is correct or the maximum number of modifications is reached. The specific correction process involves reasoning based on the original SQL and error feedback information or modification guidance signals to generate the modified result. In general, the core function of the Refiner is to achieve self-checking and self-correction of the model to enhance the overall framework's fault

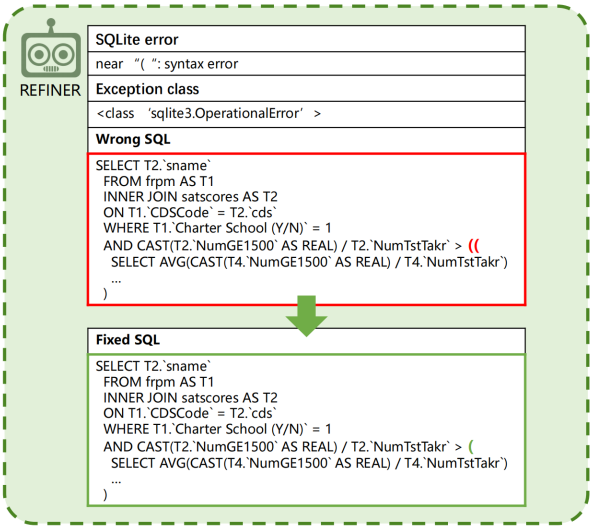

Figure 5: The Refiner Agent Illustration.

tolerance and accuracy.

Furthermore, there is a significant reduction in syntax errors, schema linking, and other errors. The remaining issues are often related to database misunderstandings, evidence misunderstandings, and question understanding, which pose greater challenges. For more abstract problems, they rely more on the reasoning ability of the LLM itself and the level of detailed information provided, thus requiring more optimization at the system level, as the assistance that a single refiner agent can provide is relatively limited.

4 Instruction-tuned SQL-Llama

Our research has been primarily focused on the development of open-source models within the MAC-SQL framework, with the goal of achieving performance levels comparable to closed-source models like GPT-4. In order to achieve this, we have put significant effort into preparing the data for model training and have open-sourced SQL-Llama, a model that has been fine-tuned using three intelligent agent instruction data. This model, based on Code Llama 7B, has undergone supervised fine-tuning using agent instruction data from MAC-SQL, which has enhanced its capabilities in database simplification, question decomposition, SQL generation, and SQL correction.

The process of preparing the data for model training involved the use of the instruction dataset derived from the training sets of BIRD and Spider through multi-agent tasks. This dataset provided a diverse and rich source of data for model training and evaluation. The model training process itself involved extensive fine-tuning and optimization to ensure that SQL-Llama could perform at the desired level within the MAC-SQL framework.

One of the key challenges we encountered during the model training process was the need to balance model complexity with performance. We had to carefully optimize the model architecture and parameters to ensure that it could effectively handle the complexities of database-related tasks while still maintaining high performance levels. Additionally, ensuring the quality and relevance of the instruction dataset for training was crucial, as it directly impacted the performance of the model.

Despite these challenges, our work on instruction-tuned models represents a significant step towards democratizing access to high-performance language models for database-related tasks. By open-sourcing both the model and the instruction dataset, we aim to provide valuable resources for further research and development in this area, ultimately leading to more accessible and effective tools for database query processing and related tasks.

5 Experiments

5.1 Experimental Setup

Datasets The Spider (Yu et al., 2018) dataset is frequently employed for assessing the performance of text-to-SQL parsing across multiple databases, necessitating models to demonstrate adaptability to unfamiliar database structures. The dataset comprises 7,000 question-query pairs in the training set and 1,034 pairs in the development set, encompassing 200 distinct databases and 138 domains. In this study, we assess the efficacy of our framework on the Spider development set, as the test set is not accessible.

The BIRD (Li et al., 2023) dataset released by Alibaba DAMO Academy is a new benchmark for large-scale real databases, containing 95 large-scale databases and high-quality Text-SQL pairs, with a data storage volume of up to 33.4GB spanning 37 professional domains. Unlike Spider, BIRD focuses on massive and real database content, external knowledge reasoning between natural language questions and database content, and new challenges in SQL efficiency when dealing with large databases.

Evaluation Metrics Following BIRD (Li et al., 2023) and Test-suite (Zhong et al., 2020), we con-

sider three metrics, exact match accuracy (EM), execution accuracy (EX) and valid efficiency score (VES) to evaluate text-to-SQL models confronted with real-world scenarios with large database contents. *Exact Match Accuracy (EM)* treats each clause as a set and compares the prediction for each clause to its corresponding clause in the reference query. A predicted SQL query is considered correct only if all of its components match the ground truth. This metric does not take values into account. *Execution Accuracy (EX)* is defined as the proportion of questions in the evaluation set for which the execution results of both the predicted and ground-truth inquiries are identical, relative to the overall number of queries. *Valid Efficiency Score (VES)* is designed to measure the efficiency of valid SQLs generated by models. It is important to note that "valid SQLs" refers to predicted SQL queries whose result sets align with those of the ground-truth SQLs.

Baselines We conduct experiments on both BIRD and Spider dataset and compare our method with the following baseline:

- **GPT-4** (OpenAI, 2023) uses simple zero-shot text-to-SQL prompt for SQL generation.
- **DIN-SQL** (Pourreza and Rafiei, 2023) decomposes the text-to-SQL task into smaller subtasks and designs different prompts for each subtask to instruct GPT-4 to complete each subtask and obtain the final SQL.
- **DAIL-SQL** (Gao et al., 2023) encodes structure knowledge as SQL statements, selects few-shot demonstrations based on their skeleton similarities and removes cross-domain knowledge from examples for token efficiency.
- **C3-SQL** (Dong et al., 2023) first performs schema linking filtering and then directs GPT-4 with a calibration bias prompt designed for Spider using a self-consistency strategy.

5.2 Overall Performance

It is important to note that the experiment utilized the 32k version of GPT-4 and the 16k version of GPT-3.5-Turbo.

BIRD Results In Table 1, we report the performance of our method and baseline methods on the BIRD dataset. It is evident that our method surpasses all LLM-based methods in terms of execution accuracy (EX) and valid efficiency score (VES) on both the development and test sets. Specifically, our method outperforms the second-best method by 4.63% on the development set and by 2.18% on the test set. At the time of writing, MAC-SQL+GPT-4 achieves an execution accuracy of 59.59 when evaluated on the BIRD benchmark, establishing a new state-of-the-art (SOTA) on its holdout test set.

Method	Dev		Test	
	EX	VES	EX	VES
Palm-2	27.38	-	33.04	-
ChatGPT + CoT	36.64	42.30	40.08	56.56
Claude-2	42.70	-	49.02	-
GPT-4	46.35	49.77	54.89	60.77
DIN-SQL + GPT-4	50.72	58.79	55.90	59.44
DAIL-SQL + GPT-4	54.76	56.08	57.41	61.95
SQL-Llama(7B)	32.87	55.67	-	-
MAC-SQL + SQL-Llama(7B)	43.94	57.36	-	-
+ Oracle Schema	51.43	58.24	-	-
MAC-SQL + GPT-3.5-Turbo	50.56	61.25	-	-
+ Oracle Schema	65.78	60.62	-	-
MAC-SQL + GPT-4	**59.39**	**66.39**	**59.59**	**67.68**
+ Oracle Schema	70.28	62.63	-	-

Table 1: Execution accuracy(EX) and Valid efficiency score (VES) on both dev and test set of BIRD dataset. The term "Oracle Schema" refers to the utilization of a ground truth sub-database as the input for the Decomposer, rather than employing the results obtained from the Selector.

Method	EX (Dev)	EX (Test)
C3 + ChatGPT	81.80	82.30
DIN-SQL + GPT-4	82.80	85.30
DAIL-SQL + GPT-4	84.40	**86.60**
SQL-Llama(7B)	65.48	61.63
MAC-SQL + SQL-Llama(7B)	76.25	70.58
MAC-SQL + GPT-3.5-Turbo	80.56	75.53
MAC-SQL + GPT-4	**86.75**	82.80

Table 2: Execution accuracy(EX) on both dev and test set of Spider.

Spider Results Currently, Spider has open-sourced the test set, so we can evaluate our method in both development and test set. As shown in Table 2, for the dev set of Spider (Yu et al., 2018), our method achieves the highest execution accuracy using GPT-4. These results demonstrate the generalization ability of our MAC-SQL framework.

Method	Simple	Mod.	Chall.	All
MAC-SQL + GPT-4	65.73	52.69	40.28	**59.39**
w/o Selector	65.73	52.04	35.14	57.28(↓)
w/o Decomposer	61.51	48.82	38.89	55.54(↓)
w/o Refiner	63.24	44.52	33.33	54.76(↓)

Table 3: Execution accuracy of MAC-SQL ablation study in BIRD dev set. For brevity, the abbreviation "Mod." stands for "Moderate" while "Chall." denotes "Challenging".

Few-shot	BIRD		Spider	
	EX	VES	EM	EX
0-shot	55.54	63.31	58.42	74.22
1-shot	57.26	64.32	59.68	78.35
2-shot	**59.39**	**66.24**	**63.20**	**86.75**

Table 4: Results of MAC-SQL+GPT-4 on the dev set of BIRD and Spider with few-shot evaluation.

5.3 Ablation Study

Table 3 presents the results of an ablation study for the MAC-SQL model in the BIRD dev set. The table lists different variations of the MAC-SQL model, including with and without certain components such as Selector, Decomposer, and Refiner. The other columns represent the accuracy of the model on different levels of difficulty: Simple, Moderate, and Challenging, as well as the overall accuracy (All).

The findings show that the original MAC-SQL + GPT-4 model achieves an accuracy of 65.73% on Simple, 52.69% on Moderate, and 40.28% on Challenging, with an overall accuracy of 59.39%. When removing the Selector component, the accuracy remained the same for Simple, but decreased to 52.04% for Moderate and 35.14% for Challenging, resulting in an overall accuracy of 57.28% (a decrease of 2.11%). Similarly, removing the Decomposer and Refiner components also led to decreases in accuracy across all difficulty levels.

Overall, the ablation study indicates that each component of the MAC-SQL model (Selector, Decomposer, and Refiner) plays a crucial role in achieving high accuracy, as their removal resulted in decreased performance across all difficulty levels.

5.4 Discussion

Impact on the number of demonstrations Table 4 shows evaluation results of MAC-SQL with different number of demonstrations on the BIRD and Spider datasets. As the number of shots increases from 0 to 2, there is a consistent improvement in the performance metrics (EX, VES, and EM) for both BIRD and Spider. This indicates that the model benefits from additional demonstration examples and is able to generalize better with more data. The highest performance is achieved with 2-shot evaluation, indicating that the model is capable of learning effectively from a small number of examples. The high cost of the GPT-4 interface results in a significant consumption of tokens during a full test of the dev set for Spider and BIRD, estimated at approximately 6 million and 10 million tokens, respectively. Due to the cost constraints, our analysis is limited to a maximum of 2-shot, and further experiments involving more shots (e.g., shot $k < 2$) will have to await a more budget-friendly implementation of GPT-4.

5.5 Error Analysis

In order to thoroughly assess the limitations of our method, we begin by choosing two datasets (BIRD and Spider) that contain various types of structured data, as shown in Figure 6.

Figure 6 displays the error type distribution in BIRD and Spider datasets. "Gold Error" is the most common error type, accounting for 30% and 22% in BIRD and Spider, respectively, signifying the significance of gold standard annotations. "Semantic Correct" is another prevalent error type, representing 14% and 22% in BIRD and Spider, respectively, indicating the importance of semantic understanding and correctness. However, "Schema Linking Error" is more frequent in BIRD (2%) than in Spider (8%), demonstrating differences in schema linking errors. This analysis underscores the need for addressing gold standard annotations, semantic correctness, and schema linking in dataset development and evaluation, thereby improving their quality and reliability. The appendix B contains detailed examples of error types.

6 Related Work

LLMs for Text-to-SQL Recent advancements in text-to-SQL tasks using large language models (LLMs) have focused on improving prompt design and developing multi-stage refined frameworks. In the early stages of the emergence of large language models, research efforts were primarily focused on designing high-quality prompts

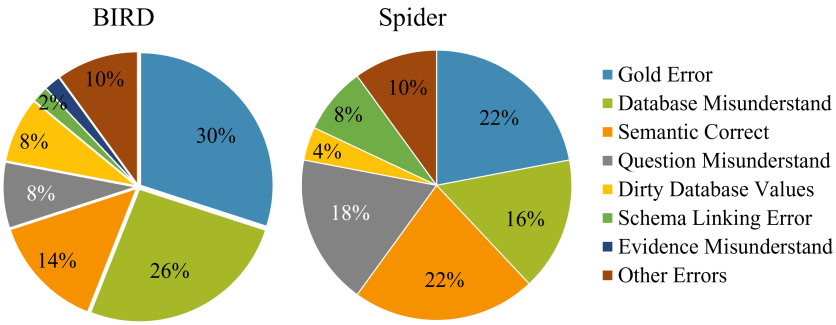

Figure 6: Error Distributions of MAC-SQL on dev set of BIRD and Spider.

to better exploit the potential of LLMs for SQL generation. For example, (Tai et al., 2023) systematically studied how to enhance LLMs' reasoning ability through chain-of-thought style prompting, including the original chain-of-thought prompting and least-to-most prompting. Similarly, (Chang and Fosler-Lussier, 2023) comprehensively investigated the impact of prompt constructions across various settings when constructing the prompt text for text-to-SQL inputs. Additionally, DAIL-SQL (Gao et al., 2023) systematically examined prompt engineering for LLM-based Text-to-SQL methods, including question representations, prompt components, example selections, and example organizations. Later studies, like C3-SQL (Dong et al., 2023), DIN-SQL (Pourreza and Rafiei, 2023), and StructGPT (Jiang et al., 2023), proposed frameworks for simplifying databases, generating SQL, verifying queries, and integrating answers through zero-shot approaches, query decomposition, and specialized interfaces for structured data access.

However, the aforementioned methods have several issues. Firstly, the experiments were conducted solely on the Spider family dataset, failing to demonstrate their generalization to more complex datasets like BIRD, hence limiting their real-world applicability. Secondly, certain methods depend on difficulty level classifiers and customized biases specific to the Spider dataset for error correction, thus lacking the ability to generalize to a broader spectrum of error types.

Thus, we propose a framework centered on multi-agent collaboration that can be utilized for more intricate data scenarios and a broader spectrum of error types for detection and correction.

LLM-based Agents LLM-based agents have been a prominent area of study in both academic and industry communities for an extended period (Wang et al., 2023). Recently, through the acquisition of vast amounts of web knowledge, LLMs have demonstrated remarkable potential in achieving human-level intelligence. This development has led to a surge in research exploring autonomous agents based on LLMs. AutoGPT (Team, 2023) is an open-source implementation of an AI agent and follows a single-agent paradigm in which it augments the AI model with many useful tools, and does not support multi-agent collaboration. Similarly, OpenAgents (Xie et al., 2023) develops three distinct agents, the Data Agent for data analysis, the Plugins Agent for plugin integration, and the Web Agent for autonomous web browsing, each specializing in different domains, similar to OpenAI's ChatGPT Plugins. Additionally, AutoGen (Wu et al., 2023) is an open-source framework that enables developers to build customizable, conversable agents that can operate in various modes, employing combinations of LLMs, human inputs, and tools to accomplish tasks. However, how to apply LLM-based agents to Text-to-SQL parsing remains under-explored.

We fill this gap by proposing a multi-agent collaborative Text-to-SQL framework, which integrates multiple LLM-based agents to collectively interpret SQL queries and address the complexity and diversity of SQL queries encountered in real-world scenarios.

7 Conclusion

In summary, this paper proposes the MAC-SQL framework, which utilizes multi-agent collaboration to address challenges in Text-to-SQL tasks. The framework, along with the open-sourced SQL-Llama model, achieved state-of-the-art execution accuracy on the BIRD dataset. This work presents a novel approach to Text-to-SQL and provides practical guidance for achieving high performance in this domain.

Limitations

There are two limitations of our work. Firstly, we did not extensively engineer the prompts, which may not be optimal. Secondly, this paper reports the fine-tuning results of the 7B CodeLLama model. Although it performs at a comparable level, we believe its performance can be further improved by using larger models.

Ethics Statement

The datasets and models utilized in this paper, and the implementation of the code and the resulting models, are not associated with any ethical concerns.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Shuaichen Chang and Eric Fosler-Lussier. 2023. How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings.

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot text-to-sql with chatgpt.

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.

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.

Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. 2019. X-sql: reinforce schema representation with context.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. 2023. Structgpt: A general framework for large language model to reason over structured data.

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

OpenAI. 2023. Gpt-4 technical report. *ArXiv*.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction.

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

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the limits of transfer learning with a unified text-to-text transformer.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models.

Ruoxi Sun, Sercan O. Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. Sql-palm: Improved large language model adaptation for text-to-sql.

Chang-You Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. Exploring chain-of-thought style prompting for text-to-sql.

AutoGPT Team. 2023. Autogpt: build and use ai agents.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2021. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2023. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2022. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models.

Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, Leo Z. Liu, Yiheng Xu, Hongjin Su, Dongchan Shin, Caiming Xiong, and Tao Yu. 2023. Openagents: An open platform for language agents in the wild.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning.

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 *Proc. of EMNLP*.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-SQL with distilled test suites. In *Proc. of EMNLP*.

Denny Zhou, Nathanael Sch¨arli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

A Prompt Details

A.1 Selector Prompt

```
selector_template = """
As an experienced and professional database administrator, your task is to analyze a
     user question and a database schema to provide relevant information. The
    database schema consists of table descriptions, each containing multiple column
    descriptions. Your goal is to identify the relevant tables and columns based on
    the user question and evidence provided.

[Instruction]:
1. Discard any table schema that is not related to the user question and evidence.
2. Sort the columns in each relevant table in descending order of relevance and keep
     the top 6 columns.
3. Ensure that at least 3 tables are included in the final output JSON.
4. The output should be in JSON format.

Requirements:
1. If a table has less than or equal to 10 columns, mark it as "keep_all".
2. If a table is completely irrelevant to the user question and evidence, mark it as
     "drop_all".
3. Prioritize the columns in each relevant table based on their relevance.

Here is a typical example:

==========
[DB_ID] banking_system
[Schema]
# Table: account
[
  (account_id, the id of the account. Value examples: [11382, 11362, 2, 1, 2367].),
  (district_id, location of branch. Value examples: [77, 76, 2, 1, 39].),
  (frequency, frequency of the acount. Value examples: ['POPLATEK MESICNE', '
      POPLATEK TYDNE', 'POPLATEK PO OBRATU'].),
  (date, the creation date of the account. Value examples: ['1997-12-29',
      '1997-12-28'].)
]
# Table: client
[
  (client_id, the unique number. Value examples: [13998, 13971, 2, 1, 2839].),
  (gender, gender. Value examples: ['M', 'F']. And F:female . M:male ),
  (birth_date, birth date. Value examples: ['1987-09-27', '1986-08-13'].),
  (district_id, location of branch. Value examples: [77, 76, 2, 1, 39].)
]
# Table: loan
[
  (loan_id, the id number identifying the loan data. Value examples: [4959, 4960,
      4961].),
  (account_id, the id number identifying the account. Value examples: [10, 80, 55,
      43].),
  (date, the date when the loan is approved. Value examples: ['1998-07-12',
      '1998-04-19'].),
  (amount, the id number identifying the loan data. Value examples: [1567, 7877,
      9988].),
  (duration, the id number identifying the loan data. Value examples: [60, 48, 24,
      12, 36].),
  (payments, the id number identifying the loan data. Value examples: [3456, 8972,
      9845].),
  (status, the id number identifying the loan data. Value examples: ['C', 'A', 'D',
      'B'].)
]
# Table: district
[
  (district_id, location of branch. Value examples: [77, 76].),
  (A2, area in square kilometers. Value examples: [50.5, 48.9].),
  (A4, number of inhabitants. Value examples: [95907, 95616].),
  (A5, number of households. Value examples: [35678, 34892].),
  (A6, literacy rate. Value examples: [95.6, 92.3, 89.7].),
  (A7, number of entrepreneurs. Value examples: [1234, 1456].),
```

```
  (A8, number of cities. Value examples: [5, 4].),
  (A9, number of schools. Value examples: [15, 12, 10].),
  (A10, number of hospitals. Value examples: [8, 6, 4].),
  (A11, average salary. Value examples: [12541, 11277].),
  (A12, poverty rate. Value examples: [12.4, 9.8].),
  (A13, unemployment rate. Value examples: [8.2, 7.9].),
  (A15, number of crimes. Value examples: [256, 189].)
]
[Foreign keys]
client.`district_id` = district.`district_id`
[Question]
What is the gender of the youngest client who opened account in the lowest average
    salary branch?
[Evidence]
Later birthdate refers to younger age; A11 refers to average salary
[Answer]
```json
{{
  "account": "keep_all",
  "client": "keep_all",
  "loan": "drop_all",
  "district": ["district_id", "A11", "A2", "A4", "A6", "A7"]
}}
```
Question Solved.

==========

Here is a new example, please start answering:

[DB_ID] {db_id}
[Schema]
{desc_str}
[Foreign keys]
{fk_str}
[Question]
{query}
[Evidence]
{evidence}
[Answer]
"""
```

A.2 Decomposer Prompt

```
decompose_template = """
Given a [Database schema] description, a knowledge [Evidence] and the [Question],
    you need to use valid SQLite and understand the database and knowledge, and then
     decompose the question into subquestions for text-to-SQL generation.
When generating SQL, we should always consider constraints:
[Constraints]
- In `SELECT <column>`, just select needed columns in the [Question] without any
    unnecessary column or value
- In `FROM <table>` or `JOIN <table>`, do not include unnecessary table
- If use max or min func, `JOIN <table>` FIRST, THEN use `SELECT MAX(<column>)` or `
    SELECT MIN(<column>)`
- If [Value examples] of <column> has 'None' or None, use `JOIN <table>` or `WHERE <
    column> is NOT NULL` is better
- If use `ORDER BY <column> ASC|DESC`, add `GROUP BY <column>` before to select
    distinct values

==========

[Database schema]
# Table: frpm
[
  (CDSCode, CDSCode. Value examples: ['01100170109835', '01100170112607'].),
  (Charter School (Y/N), Charter School (Y/N). Value examples: [1, 0, None]. And 0:
      N;. 1: Y),
  (Enrollment (Ages 5-17), Enrollment (Ages 5-17). Value examples: [5271.0,
      4734.0].),
```

```
  (Free Meal Count (Ages 5-17), Free Meal Count (Ages 5-17). Value examples:
      [3864.0, 2637.0]. And eligible free rate = Free Meal Count / Enrollment)
]
# Table: satscores
[
  (cds, California Department Schools. Value examples: ['10101080000000',
      '10101080109991'].),
  (sname, school name. Value examples: ['None', 'Middle College High', 'John F.
      Kennedy High', 'Independence High', 'Foothill High'].),
  (NumTstTakr, Number of Test Takers in this school. Value examples: [24305, 4942,
      1, 0, 280]. And number of test takers in each school),
  (AvgScrMath, average scores in Math. Value examples: [699, 698, 289, None, 492].
      And average scores in Math),
  (NumGE1500, Number of Test Takers Whose Total SAT Scores Are Greater or Equal to
      1500. Value examples: [5837, 2125, 0, None, 191]. And Number of Test Takers
      Whose Total SAT Scores Are Greater or Equal to 1500. . commonsense evidence:.
      . Excellence Rate = NumGE1500 / NumTstTakr)
]
[Foreign keys]
frpm.`CDSCode` = satscores.`cds`
[Question]
List school names of charter schools with an SAT excellence rate over the average.
[Evidence]
Charter schools refers to `Charter School (Y/N)` = 1 in the table frpm; Excellence
    rate = NumGE1500 / NumTstTakr


Decompose the question into sub questions, considering [Constraints], and generate
    the SQL after thinking step by step:
Sub question 1: Get the average value of SAT excellence rate of charter schools.
SQL
```sql
SELECT AVG(CAST(T2.`NumGE1500` AS REAL) / T2.`NumTstTakr`)
    FROM frpm AS T1
    INNER JOIN satscores AS T2
    ON T1.`CDSCode` = T2.`cds`
    WHERE T1.`Charter School (Y/N)` = 1
```

Sub question 2: List out school names of charter schools with an SAT excellence rate
     over the average.
SQL
```sql
SELECT T2.`sname`
  FROM frpm AS T1
  INNER JOIN satscores AS T2
  ON T1.`CDSCode` = T2.`cds`
  WHERE T2.`sname` IS NOT NULL
  AND T1.`Charter School (Y/N)` = 1
  AND CAST(T2.`NumGE1500` AS REAL) / T2.`NumTstTakr` > (
    SELECT AVG(CAST(T4.`NumGE1500` AS REAL) / T4.`NumTstTakr`)
    FROM frpm AS T3
    INNER JOIN satscores AS T4
    ON T3.`CDSCode` = T4.`cds`
    WHERE T3.`Charter School (Y/N)` = 1
  )
```

Question Solved.

==========

[Database schema]
Table: account
[
  (account_id, the id of the account. Value examples: [11382, 11362, 2, 1, 2367].),
  (district_id, location of branch. Value examples: [77, 76, 2, 1, 39].),
  (frequency, frequency of the acount. Value examples: ['POPLATEK MESICNE', '
      POPLATEK TYDNE', 'POPLATEK PO OBRATU'].),
  (date, the creation date of the account. Value examples: ['1997-12-29',
```

'1997-12-28'].)
]
Table: client
[
(client_id, the unique number. Value examples: [13998, 13971, 2, 1, 2839].),
(gender, gender. Value examples: ['M', 'F']. And F:female . M:male),
(birth_date, birth date. Value examples: ['1987-09-27', '1986-08-13'].),
(district_id, location of branch. Value examples: [77, 76, 2, 1, 39].)
]
Table: district
[
(district_id, location of branch. Value examples: [77, 76, 2, 1, 39].),
(A4, number of inhabitants . Value examples: ['95907', '95616', '94812'].),
(A11, average salary. Value examples: [12541, 11277, 8114].)
]
[Foreign keys]
account.`district_id` = district.`district_id`
client.`district_id` = district.`district_id`
[Question]
What is the gender of the youngest client who opened account in the lowest average salary branch?
[Evidence]
Later birthdate refers to younger age; A11 refers to average salary

Decompose the question into sub questions, considering [Constraints], and generate the SQL after thinking step by step:
Sub question 1: What is the district_id of the branch with the lowest average salary ?
SQL
```sql
SELECT `district_id`
FROM district
ORDER BY `A11` ASC
LIMIT 1
```

Sub question 2: What is the youngest client who opened account in the lowest average salary branch?
SQL
```sql
SELECT T1.`client_id`
FROM client AS T1
INNER JOIN district AS T2
ON T1.`district_id` = T2.`district_id`
ORDER BY T2.`A11` ASC, T1.`birth_date` DESC
LIMIT 1
```

Sub question 3: What is the gender of the youngest client who opened account in the lowest average salary branch?
SQL
```sql
SELECT T1.`gender`
FROM client AS T1
INNER JOIN district AS T2
ON T1.`district_id` = T2.`district_id`
ORDER BY T2.`A11` ASC, T1.`birth_date` DESC
LIMIT 1
```
Question Solved.

==========

[Database schema]
{desc_str}
[Foreign keys]
{fk_str}
[Question]
{query}
[Evidence]

```
{evidence}

Decompose the question into sub questions, considering [Constraints], and generate
    the SQL after thinking step by step:
"""
```

A.3 Refiner Prompt

```
refiner_template = """
[Instruction]
When executing SQL below, some errors occurred, please fix up SQL based on query and
     database info.
Solve the task step by step if you need to. Using SQL format in the code block, and
    indicate script type in the code block.
When you find an answer, verify the answer carefully. Include verifiable evidence in
     your response if possible.
[Constraints]
- In `SELECT <column>`, just select needed columns in the [Question] without any
    unnecessary column or value
- In `FROM <table>` or `JOIN <table>`, do not include unnecessary table
- If use max or min func, `JOIN <table>` FIRST, THEN use `SELECT MAX(<column>)` or `
    SELECT MIN(<column>)`
- If [Value examples] of <column> has 'None' or None, use `JOIN <table>` or `WHERE <
    column> is NOT NULL` is better
- If use `ORDER BY <column> ASC|DESC`, add `GROUP BY <column>` before to select
    distinct values
[Query]
-- {query}
[Evidence]
{evidence}
[Database info]
{desc_str}
[Foreign keys]
{fk_str}
[old SQL]
```sql
{sql}
```
[SQLite error]
{sqlite_error}
[Exception class]
{exception_class}

Now please fixup old SQL and generate new SQL again.
[correct SQL]
"""
```

B Error Type Examples

Examples of error types can be observed in Figure 7 (next page).

Gold Error

Question	Evidence	Gold SQL	Pred SQL	Error Description
How many male patients have a normal level of both albumin and total protein?	male refers to Sex = 'M'; normal level of both albumin and total protein refers to **ALB > 3.5 and ALB < 5.5** AND TP between 6.0 and 8.5;	SELECT COUNT(T1.ID) FROM Patient AS T1 INNER JOIN Laboratory AS T2 ON T1.ID = T2.ID WHERE T1.SEX = 'M' AND **T2.ALB BETWEEN 3.5 AND 5.5** AND T2.TP BETWEEN 6.0 AND 8.5	SELECT COUNT(*) FROM Laboratory WHERE `ID` IN (SELECT ... AND **`ALB` > 3.5 AND `ALB` < 5.5** AND `TP` >= 6.0 AND `TP` <= 8.5	Gold SQL uses "ALB BETWEEN 3.5 AND 5.5", which is conflict with evidence "3.5 < ALB < 5.5".
How many artists have designed a card **with a black border color** and is available in both "arena" and "mtgo" printing type?	available in both "arena" and "mtgo" refers to availability like '%arena,mtgo%'	SELECT COUNT(CASE WHEN availability LIKE '%arena,mtgo%' THEN 1 ELSE NULL END) FROM cards	SELECT COUNT(DISTINCT `artist`) FROM cards WHERE **`borderColor` = 'black'** AND …	Gold SQL missing the *borderColor* constraint.

Database Misunderstand

Question	Evidence	Gold SQL	Pred SQL	Error Description
What is the total score of the posts edited by Yevgeny and **include the user's website URL**.	"Yevgeny" is the DisplayName; edited refers to LastEditorUserId	SELECT SUM(T1.Score), **T2.WebsiteUrl** FROM posts AS T1 INNER JOIN users AS T2 ON **T1.OwnerUserId = T2.Id** WHERE T2.DisplayName = 'Yevgeny' GROUP BY T2.WebsiteUrl	SELECT ... ON **T1.`LastEditorUserId` = T2.`Id`** WHERE T2.`DisplayName` = 'Yevgeny'	Use wrong db foreign key

Semantic Correct

Question	Evidence	Gold SQL	Pred SQL	Error Description
Which budget allowed the most money for water, chips, and cookies?	budget allowed refers to expense_description; expense_description = 'Water, chips, cookies'; most money refers to	SELECT **T2.budget_id, T1.cost** FROM expense AS T1 INNER JOIN budget AS T2 ON T1.link_to_budget = T2.budget_id WHERE …	SELECT **T1.cost, T2.budget_id** FROM expense AS T1 INNER JOIN budget AS T2 ON …	Same answer, but with different order

Question Misunderstand

Question	Evidence	Gold SQL	Pred SQL	Error Description
Which user has a higher reputation, Harlan or Jarrod Dixon?	"Harlan" and "Jarrod Dixon" are both DisplayName; highest reputation refers to Max(Reputation)	SELECT DisplayName FROM users WHERE **DisplayName IN ('Harlan', 'Jarrod Dixon')** AND Reputation = (SELECT MAX(Reputation) FROM users WHERE **DisplayName IN ('Harlan', 'Jarrod Dixon')**)	SELECT `DisplayName`, `Reputation` FROM users WHERE **`DisplayName` = "Harlan"**	Pred SQL miss out one of the names in the question.

Dirty Database Values

Question	Evidence	Gold SQL	Pred SQL	Error Description
What are the cards for set OGW? State the colour for these cards.	set OGW refers to setCode = 'OGW';	SELECT id, colors FROM cards WHERE id IN (SELECT id FROM set_translations WHERE setCode = 'OGW')	SELECT `name`, `colors` FROM cards WHERE `setCode` = 'OGW' **AND `colors` IS NOT NULL**	Both table *cards* and table *set_translations* have the column *setCode* with inconsistent values.

Wrong Schema Linking

Question	Evidence	Gold SQL	Pred SQL	Error Description
What is the **post ID and the comments** commented in the post titled by "Group differences on a five point Likert item"?	Title = 'Group differences on a five point Likert item';	SELECT **T2.Id**, T1.Text FROM comments AS T1 INNER JOIN **posts AS T2** ON **T1.PostId** = T2.Id WHERE T2.Title = 'Group differences on a five point Likert item'	SELECT **T2.`Id`** AS CommentId, T2.`Text` AS CommentText FROM **posts AS T1** INNER JOIN comments AS T2 ON T1.`Id` = T2.`PostId` WHERE …	Different table join order, use wrong table id

Evidence Misunderstand

Question	Evidence	Gold SQL	Pred SQL	Error Description
Which cities have the top 5 lowest enrollment number for students in grades 1 through 12?	K-12 refers to students in **grades 1 through 12**.	SELECT T2.City FROM frpm AS T1 INNER JOIN schools AS T2 ON T1.CDSCode = T2.CDSCode GROUP BY T2.City ORDER BY SUM(T1.**`Enrollment (K-12)`**) ASC LIMIT 5	SELECT T1.City, ... JOIN frpm ON schools.CDSCode = frpm.CDSCode WHERE **frpm.`Low Grade` = '1' AND frpm.`High Grade` = '12'**) ... ASC LIMIT 5	Misunderstand knowledge *K-12*

Other Errors

Question	Evidence	Gold SQL	Pred SQL	Error Description
What is the grade span offered in the school with the highest longitude?		SELECT GSoffered FROM schools ORDER BY ABS(longitude) DESC LIMIT 1	SELECT `GSoffered` FROM schools WHERE `Longitude` = (SELECT **MAX(`Longitude`)** FROM schools)	No information about ABS(longtitude)

Figure 7: 8 major types of error cases of BIRD are presented. Some cases are shortcuts for better presentation.