DART-SQL: Enhancing Text-to-SQL Parsing through Question Rewriting and Execution-Guided Refinement

Anonymous ACL submission

Abstract

Large Language Model (LLM)-based approach has become the mainstream for Text-to-SQL task and achieves remarkable performance. In this paper, we augment the existing prompt engineering methods by exploiting the database content and execution feedback. Specifically, we introduce DART-SQL, which comprises two key components: (1) Question Rewriting: DART-SQL rewrites natural language questions by leveraging database content information to eliminate ambiguity. (2) Execution-Guided Refinement: DART-SQL incorporates database content information and utilizes the execution results of the generated SQL to iteratively refine the SQL. We apply this framework to the two LLM-based approaches (DAIL-SQL and C3) and test it on four widely used benchmarks (Spider-dev, Spider-test, Realistic and DK). Experiments show that our framework for DAIL-SQL and C3 achieves an average improvement of 12.41% and 5.38%, respectively, in terms of the execution accuracy (EX) metric.

1 Introduction

Text-to-SQL is crafted to convert natural language questions into executable SQL queries, creating a user-friendly interface for relational databases that enhances both usability and query efficiency (Wang et al., 2022). It advances multiple facets of data management, including database accessibility and web design flexibility.

Recent researches (Rajkumar et al., 2022; Liu and Tan, 2023; Nan et al., 2023; **?**; Chang and Fosler-Lussier, 2023; Sun et al., 2023) prominently focus on the utilization of emerging LLMs, employing zero-shot and few-shot learning techniques to harness the knowledge and generalization capabilities of the LLM. The DIN-SQL approach, introduced by Pourreza and Rafiei (2023), breaks down intricate Text-to-SQL task into smaller subtasks, enhancing the performance of the LLM in the reasoning process. When combined with GPT-4, DIN-SQL achieves an impressive score of 85.3 on the Spider-test dataset. Building upon the foundations laid by DIN-SQL, Dong et al. (2023) develop C3, facilitated by Clear Prompting (CP), Calibration with Hints (CH), and Consistent Output (CO). C3 achieves equivalent performance to DIN-SQL through Zero-Shot Learning. Further advancements in this domain are exemplified by the DAIL-SQL approach proposed by Gao et al. (2023). DAIL-SQL adopts a comprehensive approach, constructing prompts from three distinct perspectives: problem description, sample selection, and sample description. Guided by clear mission interpretation, high relevance, and a strategic SC (strategy and clarification) strategy, DAIL-SQL achieves the noteworthy feat of securing the second-highest score on the Spider Leaderboard.

However, most popular benchmarks like Spider (Yu et al., 2019) and WikiSQL (Zhong et al., 2017) focus on the database schema, with only a few rows of data in the database (Li et al., 2023c). There exists a significant gap between academic research and real-world applications, resulting in existing approaches that place more emphasis on database schema over database content information. The inherent fuzziness in natural language also poses a challenge when linking the entity names referenced in the question and the corresponding data in the database. For instance, when addressing specific conditions like the representation of gender, database entries may employ diverse representation such as '0','f' and others, whereas natural language questions often use the term 'female'. This inherent fuzziness introduces uncertainty that can significantly impact the execution efficiency of the generated SQL statements. Simultaneously, it is noteworthy that existing methods often overlook the valuable feedback generated after the execution of SQL statements. There is a lot of scope to exploit the feedback to further fine-tune the generated SQL statements.

In this work, we present a unified framework, called DART-SQL[1], which is designed to mine database content information, eradicate ambiguity in natural language questions and iteratively enhance Text-to-SQL performance by incorporating execution information. In detail, DART-SQL leverages the power of database content information through a multifaceted approach. Initially, it rewrites natural language questions by seamlessly integrating database content information, ensuring the revised questions align with the database and eliminate ambiguity. Additionally, DART-SQL introduces database content data into prompts, thereby enhancing Text-to-SQL performance. Finally, DART-SQL employs an AI agent to evaluate and analyze the results of SQL execution, progressively refining the SQL statement with the feedback from the AI agent. Through this iterative correction process, we achieve the self-training effect, enhancing the overall efficacy of DART-SQL. Experiments show that our framework for DAIL-SQL and C3 achieves an average improvement of 12.41% and 5.38%, respectively, in terms of the EX metric. Besides, Our framework DART-SQL achieves an EX score of 82.5 when applied to DAIL-SQL, placing sixth on the Spider Leaderboard[2].

In summary, our main contributions are:

- We observe that both existing research and benchmarks have very few database content information, which differ from real-world application significantly .
- We introduce DART-SQL, a solution that not only tackles the three aforementioned shortcomings but also effectively enhances Text-to-SQL performance.
- Extensive experiments demonstrate that our method can be built on top of a range of existing strong methods and improve their performance on Text-to-SQL task

2 Related Work

2.1 Template Matching

In the early stages, Text-to-SQL is approached using rule-based approach (Mahmud et al., 2015). Methods in this category utilize the strong rule-based feature of the SQL to design different templates for different query problems, then select the template according to the problem, and then derive the corresponding template SQL statements.

2.2 Seq2Seq Models

Owing to the inherent alignment between Text-to-SQL task and the N -> M machine translation task, many researchers adopt Seq2Seq models to address Text-to-SQL task. They first build an encoder to understand the database table structure and the input user query problem, such as the LSTM-based method (Yu et al., 2018), and the Transformer-based method (Kelkar et al., 2020), and then use a decoder to predict the target SQL statement, like the sketch-based method (Yu et al., 2018), and the generation-based method (Huang et al., 2021).

2.3 Pre-trained Models

The utilization of pre-trained language models can augment the parsing capability of Text-to-SQL. As the textual data, tabular data and SQL queries are heterogeneous, it is non-trivial but important to develop a joint reasoning framework over the three types of data (Qin et al., 2022). Qi et al. (2022) incorporate various associations such as schema encoding, schema linking, and syntactic dependencies of a problem into a unified relational representation by constructing ternary groups. Li et al. (2023a) proposes a ranking-enhanced encoding and skeleton-aware decoding framework to decouple the schema linking and the skeleton parsing . The MIGA framework, proposed by Fu et al. (2023), introduces a two-stage unified multi-task generation approach that promotes compatibility between dialogue-based Text-to-SQL methods and generative pre-trained language models. Li et al. (2023b) introduce GRAPHIX-T5, which excels in capturing relational structural information while preserving the robust contextual encoding abilities inherited from the pre-trained T5 model.

2.4 Large Language Models

LLMs have excellent performance and generalization ability, and their outstanding performance in zero-shot and few-shot learning has attracted the attention of the industry. Liu et al. (2023) provide an initial comprehensive analysis of ChatGPT's Text-to-SQL capabilities, revealing its powerful capabilities in Text-to-SQL. Recent researches have focused on exploring the appropriate form of prompt

[1]DART-SQL: Database-Aware Refinement and Transformation for Text-to-SQL with Large Language Models

[2]The top five methods on Spider Leaderboard are based on GPT-4. Due to resource limitations, we apply GPT-3.5 in our study, which hinders the performance of our framework.

organization to maximize the performance of large models in Text-to-SQL.

2.4.1 Table Representation

The table and column name associated with the entered user query question is an important part of the prompt in Text-to-SQL. The key to table representation is to select only the database tables and columns that will be used to generate the target SQL. The study by Pourreza and Rafiei (2023) focuses on identifying and extracting pertinent columns and tables from the database schema based on a chain-of-thought template. Meanwhile, Dong et al. (2023) simplify the table representation by schema linking which recalls relevant tables and columns filters out irrelevant tables and columns, thereby reducing the number of tokens required for hinting and improving context clarity. Also, how database information is displayed in the prompt can affect the performance of the model. For example, Nan et al. (2023) find that describing table structure information as CREATE statements can yield improvements.

2.4.2 Demonstration Selection

The demonstration provided in the input prompt can improve the performance of the LLM in Text-to-SQL. Poesia et al. (2022) point out that presentation retrieval based on similarity is effective in a cross-domain environment, while Levy et al. (2023) demonstrate that diverse demonstrations benefit in-context compositional generalization. In terms of the basis for selecting demonstration, it is mainly divided into those based on the similarity of the input question (Guo et al., 2023), those based on the output SQL (Nan et al., 2023), and those that consider both the input question and the output SQL (Gao et al., 2023).

3 Methodology

In this section, we present our method that leverages database contents to enhance the natural language questions and to provide the LLM with feedback using execution results. This enables the model to learn from its own mistakes and improve its SQL generation. Our framework can be applied on top of various existing prompt designs to improve the performance. These prompts usually consist of at least two components: the schema of the database and a few examples of natural language questions along with their corresponding SQL SELECT statements. We omit the details of these prompt designs as they are not the main focus of our study. Readers can refer to the related works on 2.4 for more information.

In a nutshell, our framework consists of the following steps: ①**Question Rewriting**: We choose several rows from the table in the corresponding database related to the question as relevant database content information to rewrite the natural language question. ②**Context-Aware Prompt Generation**: For each table, we select a few rows relevant to the question, and sample some rows from the remaining ones. Such content information is then combined with an initial prompt generated by an existing prompt design to form a new context-aware prompt. ③**Execution-Guided Refinement**: The LLM takes the finalized context-aware prompt as input, and generates an SQL query. Our framework then executes it locally and offers feedback to assist the LLM to refine the statement.

3.1 Question Rewriting

To eliminate ambiguity in natural language questions and gain deeper insights into the database, we introduce an additional stage "Question Rewriting" to the conventional prompting method, which optimizes the natural language questions by incorporating database information.

To clarify the "Question Rewriting" task for the LLM and make effective use of the database content information, we design the following prompt:

Instruction. In this section, we give a clear definition of the "Question Rewriting" task. The details are as follows: "Formulate natural language queries for database data based on the provided information. Ensure the questions you rewrite are clear, concise, and aligned with the data specifications within the database. If you think the sentence is clear enough, you can return to the original without rewriting it."

Examples. In this section, we summarize the three specifications for the model to follow when rewriting questions and illustrate them with concrete examples. Firstly, we instruct the model on how to rewrite questions exhibiting fuzziness through the specification 1. Secondly, Specification 2 serves as a reminder for the model not to omit key information during the rewrite process. For instance, database content information like "Note that 1 stands for yes, and 0 stands for NO in the tables" in the original question might be omitted after rewriting. Finally, through Specification 3, we

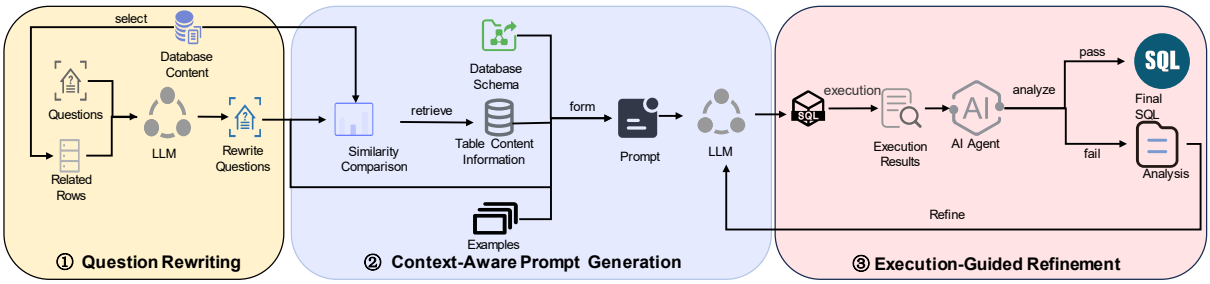


Figure 1: Overview of DART-SQL.

ensure that the model avoids making meaningless or potentially harmful modifications to questions that are already sufficiently clear.

Input. In this section, we take the initial natural language question and related database content information as input. Previous methods focused on schema and structure but neglected effective use of database content. To address this, we add the first K (due to prompt length limitations, we set $K = 5$) data information for all tables corresponding to the question. Surprisingly, thanks to the LLMs' summarization and generalization capabilities, the model mostly identified the relevant table and refined the problem based on specific database statements, ensuring consistency in entity names, cases, abbreviations, and filter criteria for attribute names.

The Question Rewriting module greatly reduces problem ambiguity. Due to the limited context length of the LLM, the original Text-to-SQL task prompt includes only limited database information. By introducing the additional Question Rewriting task, we can effectively incorporate underutilized database content information into the natural language problems during rewriting. This allows for the utilization of more database information within the constraints of a limited context length.

3.2 Context-Aware Prompt Generation

After rewriting the question, the initial prompt can be generated using an existing prompt design. Our framework then refines the prompt to become the context-aware prompt by adding additional portions of the table content Intuitively, this refinement aims to reduce value errors by providing the LLM with examples of the data format in the database. Moreover, these data samples also complement the database schema, assisting the LLM to understand the database structure.

At most N rows of each table are included in the prompt. If a table has more than N rows, half of the rows are selected based on their relevance to the question, and the other half are sampled from the remaining rows. We set $N = 15$ for our method.

To calculate the relevance of a row inside a table to a question Q_i, we first convert the row into a JSON dictionary D_i, with column names as keys and the cell contents as values. The dictionary and the question are tokenized respectively, resulting in $D_i = \{D_{i1}, ..., D_{i|D_i|}\}$ and $Q_i = \{Q_{i1}, ..., Q_{i|Q_i|}\}$. Next, we compute the sentence embeddings of D_i and Q_i:

$$E(D_i) = \frac{1}{|D_i|} \sum_{k=1}^{|D_i|} \text{GloVe}(D_{ik})$$

$$E(Q_i) = \frac{1}{|Q_i|} \sum_{k=1}^{|Q_i|} \text{GloVe}(Q_{ik})$$

where $\text{GloVe}(x)$ refers to the GloVe embedding (Pennington et al., 2014) of token x. Finally, the relevance is calculated as the cosine similarity between $E(D_i)$ and $E(Q_i)$.

3.3 Execution-Guided Refinement

After receiving the processed prompt, the LLM then outputs a single SELECT statement to answer the question. We then start a self-refinement procedure that executes the statement on the database and ask the LLM to analyze the result, until it thinks the results are correct or the procedure is repeated 3 times. The full process is shown in Algorithm 1.

In the refinement process, the SELECT statement is executed. If the execution is successful, the results (truncated at $M = 15$ rows) are analyzed by the LLM. Otherwise, the LLM receives the exception information and outputs a new statement addressing the issue. If the exception is related to column names, including ambiguity and no-such-

Algorithm 1 Self-Refinement Algorithm

Input: initial SELECT statement SQL_0
Output: refined statement SQL

```
SQL ← SQL_0
for i = 1 to 3 do
    clear the interaction history, excluding the prompt and the latest generated SQL
    res ← Execute(SQL_0)
    if res is a column-name error then
        c ← wrong column name in res
        msg ← list of tables containing c
        SQL ← LLM-SQL(msg, res)
    else if res is an exception then
        SQL ← LLM-SQL(res)
    else                          ▷ res stores execution results
        keep at most M rows in res
        flag ← LLM-Judge(res)
        if flag == True then
            return SQL
        else
            SQL ← LLM-SQL()
        end if
    end if
end for
return SQL
```

column errors, we additionally provide the LLM with a list of tables containing the column.

Furthermore, the LLM is forbidden from using certain keywords including LEFT JOIN and SELECT $*$. If the output SQL statement contains such keywords, the LLM is immediately asked to regenerate it. The rationale is that upon manually analyzing the final outputs, most statements with these keywords produce incorrect results. This suggests that the LLMs we test are not proficient at utilizing these SQL features.

4 Experiments

4.1 Experimental Setup

Dataset. We evaluate DART-SQL on three datasets, **Spider** (Yu et al., 2019), **Spider-DK** (Gan et al., 2021), and **Spider-Realistic** (Deng et al., 2021). Spider is a large-scale cross-domain Text-to-SQL dataset. It comprises 10181 annotated questions, corresponding to 5693 distinct complex SQL queries and 200 databases with multiple tables. The training, development, and test sets of Spider include 8659, 1034, and 2147 examples and 146, 20, and 40 databases respectively. Our approach selects pertinent few-shot samples from the training set. Spider-DK and and Spider-Realistic are more challenging variants of the original Spider dataset. Spider-DK features 535 instances and emphasizes the importance of domain knowledge, while Spider-Realistic consists of 508 instances from Spider with explicit mentions of column names removed.

Metric. Following previous studies, we use exact-set-match accuracy (**EM**) and execution accuracy (**EX**) to evaluate our framework. EM breaks down the predicted SQL statement and the ground-truth into two sequences of SQL clauses, and checks if they are identical after removing the values in the statements. EX compares the execution results of the predicted statement and the ground-truth on the corresponding database. The test suite we use was introduced in (Zhong et al., 2020).

Model and hyper-parameter. We use the 0613 version of GPT-3.5-turbo-16k as our LLM via the OpenAI API for all experiments, including the baseline methods[3]. We set the number of chat completion choices n to 1 and the temperature T to 0.0. When self-consistency (Wang et al., 2023) is enabled, we set $n = 5$ and $T = 1.0$ for SQL generation instead, and leave the parameters unchanged for text generation.

Base prompt. We select DAIL-SQL and C3 to generate the initial prompt. DAIL-SQL includes the entire database schema in the prompt, and selects few-shot examples based on both the questions and the pre-generated queries. C3 uses the LLM to perform zero-shot schema linking, incorporates guidelines into the conversation history, and features execution-based self-consistency.

4.2 Overall Performance

Table 1 and 2 display the results of our framework as well as the baseline methods on the datasets. According to the results, our framework brings noticeable performance improvement to both prompt designs in terms of execution accuracy. When DAIL-SQL is used as the base prompt, DART-SQL drastically outperforms the baseline method by 15.2% in EM and 10.4% in EX on the Spider-Realistic dataset, while also providing satisfying performance boost on other datasets. When combined with C3, DART-SQL achieves 73.8% in EX, outperforming both the baseline and the separated modules in DART-SQL. This indicates that DART-

[3]The baseline methods are rerun with the authors' open-source repositories on GPT-3.5, and the results might differ from those reported in the original paper.

	Dev		Test		Realistic		DK	
Base Prompt: DAIL-SQL	EM	EX	EM	EX	EM	EX	EM	EX
baseline	62.1	76.2	**61.4**	76.9	42.1	68.9	41.7	61.5
+SC	62.1	79.4	60.3	79.3	44.1	70.3	41.1	63.6
RW	**63.9**	79.9	60.3	79.3	47.8	71.5	**44.7**	66.5
+SC	61.7	80.9	**61.2**	79.6	42.3	71.7	39.4	64.1
CAP+ER	61.3	**84.0**	**61.0**	81.6	55.9	**80.7**	**45.0**	70.7
+SC	59.3	**83.5**	59.9	**83.0**	53.3	78.1	43.6	**72.0**
DART-SQL	62.1	**83.1**	**62.0**	81.8	**57.3**	79.3	**44.7**	**71.4**
+SC	59.3	**83.1**	59.9	**82.5**	53.3	79.3	41.1	70.7

Table 1: Results using DAIL-SQL as the base prompt. SC means self-consistency while RW, CAP, and ER refer to question rewriting, context-aware prompt generation, and execution-guided refinement introduced in Section 3.1, 3.2, and 3.3 respectively. For each metric on a dataset, the highest result is both bold and underlined, while the results that are at most 1 percentage lower than it are also marked bold.

	Dev		Test		Realistic		DK	
Base Prompt: C3	EM	EX	EM	EX	EM	EX	EM	EX
baseline	**46.4**	81.3	**46.7**	79.7	44.3	77.6	**40.6**	66.4
RW	**47.3**	80.9	**47.2**	79.5	**47.4**	78.5	**39.6**	69.0
CAP+ER	**47.0**	**84.4**	**47.6**	**83.0**	43.9	**82.9**	37.8	72.1
DART-SQL	**46.4**	83.0	46.3	**82.1**	45.7	81.7	38.1	**73.8**

Table 2: Results using C3 as the base prompt.

SQL has superior skills in utilizing domain knowledge to solve challenging problems in the datasets. On the other hand, self-consistency only boosts the performance in EX on the test set of Spider when using DAIL-SQL, and negatively affect the EM scores on all datasets, which means enabling self-consistency is unnecessary for our framework.

4.3 Analysis of DART-SQL Components

We conduct an ablation study on the development set of Spider with DAIL-SQL as our base prompt, and record the results in Table 3. In each experiment, one or more components of DART-SQL is disabled to validate their necessity. In addition, we perform a hyper-parameter analysis on our framework in Figure 2, regarding the modules of table content selection and execution-guided refinement.

4.3.1 Example Selection

We test the performance of DART-SQL under the zero-shot scenario, i.e. all examples of question-query pairs offered by DAIL-SQL are removed. Here, DART-SQL is used as a replacement of DAIL-SQL, providing a 2.4% increase in EX, which indicates that the LLM has the ability to learn from database patterns and refine the query statement based on the execution results. However, as the training and development sets both come from the Spider dataset so they share the same design principles and distribution, the removal of few-shot samples still degrade the overall performance, not to mention the drastic decrease of 19.9% in EM scores.

As the number of examples in the prompt increases, according to Figure 2a, the performance boost depends on S, the number of samples, being insignificant when $S > 3$. This indicates that to maintain a similar level of performance, we should offer at least 3 examples, achieving 82.6% in EX.

4.3.2 Question Rewriting

In Table 3, it is evident that the EM score experiences a decline of 0.8% when the Question Rewriting module is removed. Although ablation experiments are exclusively conducted in the development set of Spider, referencing Table 1 and 2, we observe that, whether DAIL-SQL or C3 is used as the base prompt, the majority of the four datasets exhibit improvement when this module is utilized. These results indicate that the module effectively mitigates ambiguity in natural language problems by incorporating database content information relevant to the original question, aligning the generated SQL statement format more closely with reality,

consequently leading to increased EM scores. Additionally, Table 1 and 2 reveal some improvement in the Question Rewriting module for EX. Through these experiments, we establish the efficacy and versatility of the Question Rewriting module.

4.3.3 Context-Aware Prompt Generation

As can be seen in Table 3, removing the module of context-aware prompt generation decreases the EX score by 2.7% while giving a similar EM score. Without the content samples in each table, the LLM is unaware of the data formats in the database until execution time, and its understanding of the database structure is also negatively affected.

Furthermore, we design experiments to study the impact on EX scores of different N, the maximum number of rows offered to the LLM in each table in the database. According to Figure 2b, even the contents of a single row in each table makes a significant contribution of 2.0% to the EX score. As N gradually increases, the EX score also increases slowly. Since most tables in the Spider dataset contain no more than 15 rows, we can infer that adding more rows has rather limited impact on the performance.

4.3.4 Execution-Guided Refinement

In Table 3, when the refinement module is disabled, the EX score decreases by 2.5% while the EM score maintains a similar level. If we additionally switch off the content selection module, the EX score further decreases by 0.7%. In other words, when the two modules are combined, the overall performance boost is larger than those offered by them separately. We can conclude that the LLM can better utilize the execution results to refine the query with the help of database contents.

We design two sets of experiment on M, the number of rows in the execution results and R, the maximum number of revisions by the LLM, respectively. As we can see in Figure 2c, 3 rows are barely enough for the self-refinement process, while the performance suffers greatly from diminishing returns when $M > 7$. As for the maximum number of revisions, in Figure 2d, the LLM is capable of correcting quite a few mistakes when $R = 1$, receiving another increase of 0.6% if R is increased to 3.

4.4 Result Analysis

Here we compare the correctness of output results of C3 as the baseline method, and DART-SQL with C3 as the base prompt design, on the Spider-DK dataset, where these methods achieve 66.4% and 73.8% execution accuracy respectively. Table 4 displays the results.

Base Prompt: DAIL-SQL	**Dev** (EM)	**Dev** (EX)
DART-SQL	62.1	83.1
w/o RW	61.3	84.0
w/o CAP	62.2	80.4
w/o ER	62.4	80.6
w/o RW+CAP	61.3	80.2
w/o RW+ER	61.9	80.4
w/o CAP+ER	63.9	79.9
w/o DART-SQL (baseline)	62.1	76.2
w/o DAIL-SQL (0-shot)	42.2	78.6

Table 3: Ablation study on the dev split of the Spider dataset.

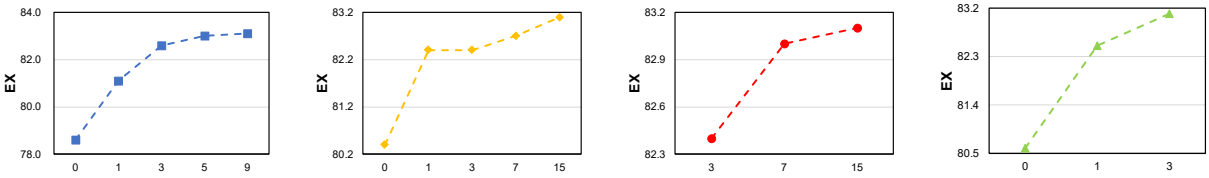
Overall, our framework addresses 66 errors but introduces 25 new mistakes compared to the baseline method. About half of the fixed errors are wrong values, since C3 provides very limited information on the database contents via schema linking, while we directly include them in our prompt, in addition to sending feedback to the LLM with execution results.

For other errors, condition errors include bugs in the WHERE or HAVING clauses, while table/-column errors refer to mistakes in table or column names, or selecting extra columns. These bugs are much more difficult to fix than value errors, requiring deep understanding of the question and the database schema. Thanks to the question rewriting module, the clarity of the question is noticeably improved for the LLM, while the other two modules are helpful in illustrating the structure of the database. As a result, our framework also makes progress in addressing these errors.

4.5 Case Study

To explore how each module in DART-SQL encourages the LLM to generate SQL statements of higher quality, we conduct a case study on the 254th question in the Spider-Realistic dataset as shown in Figure 3.

We first generate an SQL statement with the baseline method, DAIL-SQL. The original question is included in the prompt, and as a result of the incorrect grammar in addition to the absence of database contents, it presents difficulty for the LLM to understand the question and the database



(a) Number of few-shot samples (S)

(b) Number of rows in the content samples (N)

(c) Number of rows in each execution results (M)

(d) Maximum number of iterations (R)

Figure 2: Hyper-parameter analysis of DART-SQL.

Question: List the name and date the battle that has lost Lettice and HMS Atalanta

Query by DAIL:
```
SELECT T1.name, T1.date FROM battle AS T1
    JOIN ship AS T2 ON T1.id = T2.lost_in_battle
    WHERE T2.name = "Lettice" AND T2.ship_type = "HMS Atalanta"
```
❌ **Execution Result:** None

Rewritten Question: List the name and date of the battle in which Lettice and HMS Atalanta were lost.

Content Samples (Table: ship)

lost_in_battle	id	name	tonnage	ship_type	location
8	1	Lettice	t	Brig	English Channel
8	7	HMS Atalanta	225	8 gun Brig	Mid-Atlantic

Initial Query by DART:
```
SELECT T1.name, T1.date FROM battle AS T1
    JOIN ship AS T2 ON T1.id = T2.lost_in_battle
    WHERE T2.name = "Lettice" OR T2.name = "HMS Atalanta"
```
❌ **Execution Result:**
```
('Siege of Constantinople', '1235')
('Siege of Constantinople', '1235')
```

Revised Query by DART:
```
SELECT T1.name, T1.date FROM battle AS T1
    JOIN ship AS T2 ON T1.id T2.lost_in_battle
    WHERE T2.name = "Lettice"
    INTERSECT SELECT T1.name, T1.date FROM battle AS T1
    JOIN ship AS T2 ON T1.id = T2.lost_in_battle
    WHERE T2.name = "HMS Atalanta"
```
✔ **Execution Result:**
```
('Siege of Constantinople', '1235')
```

Figure 3: Case study of DART-SQL. The modified part in the question is marked in dark red. The differences in the three SQL statements are underlined. For simplicity, we only list some relevant rows in the content samples.

structure. With the DAIL-SQL prompt, the LLM mistakenly considers "HMS Atalanta" as a ship type, and returns a wrong query.

In the following step, we use DART-SQL to solve the problem. The question is rephrased to improve clarity, and parts of the relevant contents are integrated into the prompt. As shown in the Content Samples table in Figure 3, the two entity names are actually ship names, and the LLM generates an initial query, correctly addressing the issue in the baseline method. However, there exists one more problem that the generated statement allows duplicated rows. Once the LLM receives the results, it detects the mistake and corrects it by utilizing the INTERSECT operator, as shown in the revised query. The final execution result is correct and identical to the ground-truth output.

5 Conclusion

In this paper, we propose a novel Text-to-SQL framework, DART-SQL, which addresses the issue of insufficient utilization of database contents, ambiguity in the natural language question, and lack of self-refinement based on execution results.

Base Prompt: C3	**Fixed**	**Introduced**
Total	66	25
Values	34	2
Conditions	6	5
Table/column	20	16
Others	6	2

Table 4: Number of errors fixed and introduced by our framework compared to the baseline method C3, on the Spider-DK dataset.

Our experiment on the Spider dataset and two of its variants shows significant improvement over DAIL-SQL and C3. Furthermore, we substantiated the validity and generality of our framework through a comprehensive series of experiments. Overall, our study presents a practical solution to the task of Text-to-SQL. It opens up a new research direction for those entering the field, and we anticipate more in-depth exploration in this direction in future research endeavors.

Limitations

Due to resource limitations, we are unable to conduct our research on other state-of-the-art LLMs, such as GPT-4-turbo, PaLM, and Llama, or on other base prompt designs like DIN-SQL. Our feedback to the LLM also only checks a few common issues, which can be enhanced with SQL static tools. We expect future researches to provide more helpful insights on the generated query for the LLM to further increase Text-to-SQL performance.

Ethics Statement

Our experimentation leverages the OpenAI platform to assess the efficacy of our approach, utilizing their GPT-3.5-turbo-16k model. It's worth noting that large models heavily depend on significant computing power, contributing to substantial power consumption and increased carbon dioxide emissions.

References

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. *arXiv preprint arXiv:2305.11853*.

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

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*.

Yingwen Fu, Wenjie Ou, Zhou Yu, and Yue Lin. 2023. Miga: a unified multi-task generation framework for conversational text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 12790–12798.

Yujian Gan, Xinyun Chen, and Matthew Purver. 2021. Exploring underexplored limitations of cross-domain text-to-sql generalization.

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*.

Chunxi Guo, Zhiliang Tian, Jintao Tang, Pancheng Wang, Zhihua Wen, Kang Yang, and Ting Wang. 2023. Prompting gpt-3.5 for text-to-sql with desemanticization and skeleton retrieval.

Junyang Huang, Yongbo Wang, Yongliang Wang, Yang Dong, and Yanghua Xiao. 2021. Relation aware semi-autoregressive semantic parsing for nl2sql.

Amol Kelkar, Rohan Relan, Vaishali Bhardwaj, Saurabh Vaichal, Chandra Khatri, and Peter Relan. 2020. Bertrand-dr: Improving text-to-sql using a discriminative re-ranker.

Itay Levy, Ben Bogin, and Jonathan Berant. 2023. Diverse demonstrations improve in-context compositional generalization.

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.

Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023b. Graphix-t5: Mixing pretrained transformers with graph-aware layers for text-to-sql parsing. *arXiv preprint arXiv:2301.07507*.

Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, et al. 2023c. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint arXiv:2305.03111*.

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

Xiping Liu and Zhao Tan. 2023. Divide and prompt: Chain of thought prompting for text-to-sql. *arXiv preprint arXiv:2304.11556*.

Tanzim Mahmud, K. M. Azharul Hasan, Mahtab Ahmed, and Thwoi Hla Ching Chak. 2015. A rule based approach for nlp based query processing. In *2015 2nd International Conference on Electrical Information and Communication Technologies (EICT)*, pages 78–82.

Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. 2023. Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies. *arXiv preprint arXiv:2305.12586*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *arXiv preprint arXiv:2304.11015*.

Jiexing Qi, Jingyao Tang, Ziwei He, Xiangpeng Wan, Yu Cheng, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. 2022. Rasat: Integrating relational structures into pretrained seq2seq model for text-to-sql. *arXiv preprint arXiv:2205.06983*.

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

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498*.

Ruoxi Sun, Sercan O Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. Sql-palm: Improved large language modeladaptation for text-to-sql. *arXiv preprint arXiv:2306.00739*.

Lihan Wang, Bowen Qin, Binyuan Hui, Bowen Li, Min Yang, Bailin Wang, Binhua Li, Jian Sun, Fei Huang, Luo Si, et al. 2022. Proton: Probing schema linking information from pre-trained language models for text-to-sql parsing. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1889–1898.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018. Typesql: Knowledge-based type-aware neural text-to-sql generation. *arXiv preprint arXiv:1804.09769*.

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. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-sql with distilled test suite. In *The 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

A Prompt Examples

In the appendix, we demonstrate an example of the prompt in each step used in our framework.

A.1 Question Rewriting

Listing 1: Example prompt for Question Rewriting.

```
/*Instructions:
Formulate natural language queries for database data based on the
   provided information. Ensure the questions you rewrite are clear,
   concise, and aligned with the data specifications within the
   database. If you think the sentence is clear enough, you can
   return to the original without rewriting it.

/*Here are some basic examples: */

/*Example 1: Note that rewrite question aligned with the data
   specifications within the database*/

/*Given the initial value from Table:
['Table countries: ', [(1, 'usa', 1), (2, 'germany', 2), (3, 'france
   ', 2), (4, 'japan', 3), (5, 'italy', 2)]]*/
/*Rewrite the following question:What is the total number of car
   makers in Italy?*/
/*results:Can you provide information on the number of car
   manufacturers in \"italy\"?*/

...

/*Example 2: Note that you can not miss the information in original
   question*/
/*Given the initial value from Table:
['Table Dogs: ...]*/
/*Rewrite the following question:What are the dog name, age and
   weight of the dogs that were abandoned? Note that 1 stands for yes
   , and 0 stands for no in the tables.*/
/*results:What are the dog name, age and weight of the dogs in the
   Table \"Dogs\" that were abandoned? Note that 1 stands for yes,
   and 0 stands for no in the tables.*/

...

/*Example 3: If you think the sentence is clear enough, you can
   return to the original without rewriting it*/
/*Given the initial value from Table:['Table cars_data: ', [(1, '18',
    8, 307.0, '130', 3504, 12.0, 1970), (2, '15', 8, 350.0, '165',
   3693, 11.5, 1970), (3, '18', 8, 318.0, '150', 3436, 11.0, 1970),
   (4, '16', 8, 304.0, '150', 3433, 12.0, 1970), (5, '17', 8, 302.0,
   '140', 3449, 10.5, 1970)]]
*/
Rewrite the following question: Among the cars that do not have the
   minimum horsepower , what are the make ids and names of all those
```

```
   with less than 4 cylinders ?*/
/*results:Among the cars that do not have the minimum horsepower ,
   what are the make ids and names of all those with less than 4
   cylinders ?*/

...

/*Here is the input:*/
/*Given the initial value from Table:['Table singer: ', [(1, 'Liliane
    Bettencourt', 1944.0, 30.0, 'France'), (2, 'Christy Walton',
   1948.0, 28.8, 'United States'), (3, 'Alice Walton', 1949.0, 26.3,
   'United States'), (4, 'Iris Fontbona', 1942.0, 17.4, 'Chile'), (5,
    'Jacqueline Mars', 1940.0, 17.8, 'United States')]]
['Table song: ', [(1, \"Do They Know It's Christmas\", 1, 1094000.0,
   1.0), (2, \"F**k It (I Don't Want You Back)\", 1, 552407.0, 1.0),
   (3, 'Cha Cha Slide', 2, 351421.0, 1.0), (4, 'Call on Me', 4,
   335000.0, 1.0), (5, 'Yeah', 2, 300000.0, 1.0)]]
*/
Rewrite the following question: What are the names of the singers who
    are not French citizens?
results:
```

A.2 Context-Aware Prompt Generation

We only present examples with DAIL-SQL as the base prompt. For simplicity, only one examples as well as the schema and the content of a single table is listed.

Listing 2: Example prompt with selected contents.

```
/* Some SQL examples are provided based on similar problems: */
/* Answer the following: which us city has the highest population
   density */
SELECT city_name FROM city WHERE population  =  ( SELECT MAX (
   population ) FROM city );

-- ...

/* Given the following database schema: */
CREATE TABLE "countries" (
    "CountryId" INTEGER PRIMARY KEY,
    "CountryName" TEXT,
    "Continent" INTEGER,
    FOREIGN KEY (Continent) REFERENCES continents(ContId)
)

-- ...

/* With at most 15 example rows in each table: */
INSERT INTO car_makers (Id, Maker, FullName, Country) VALUES /* 7
   rows omitted */
(1, 'amc', 'American␣Motor␣Company', '1'),
(5, 'ford', 'Ford␣Motor␣Company', '1'),
(4, 'gm', 'General␣Motors', '1'),
```

```
(15, 'peugeaut', 'Peugeaut', '3'),
(20, 'triumph', 'Triumph', '7'),
(19, 'toyota', 'Toyota', '4'),
(8, 'nissan', 'Nissan␣Motors', '4'),
(12, 'mazda', 'Mazda', '4'),
(16, 'renault', 'Renault', '3'),
(23, 'hyundai', 'Hyundai', '8'),
(18, 'subaru', 'Subaru', '4'),
(21, 'volvo', 'Volvo', '6'),
(6, 'chrysler', 'Chrysler', '1'),
(22, 'kia', 'Kia␣Motors', '8'),
(14, 'opel', 'Opel', '2');

-- ...

/* Answer the following: Which model has the highest miles per gallon
    in terms of gasoline consumption? */
SELECT
```

A.3 Execution-Guided Refinement

Listing 3: Example prompt for a column-name error.

```
/* Received the following exception: no such column: Model
In fact, the following 2 table(s) contain column "model": model_list,
    car_names.
Please analyze the cause. */
```

Listing 4: Example prompt for other exceptions.

```
/* Received the following exception: no such function: YEAR
Please analyze the cause. */
```

Listing 5: Example prompt for a successful execution.

```
/* Here are the execution results (total 5 rows):
('Balmoor', 1)
('Glebe Park', 1)
('Recreation Park', 1)
('Somerset Park', 2)
("Stark's Park", 1)

Judge whether the results are correct, and you selected the exact
   number of columns asked by the question.
Output Yes if you agree with both, or No otherwise. Follow the answer
    with your analysis of the results, and the meanings of the
   columns you selected and those requested by the question. */
```

Listing 6: Prompt asking the LLM to refine the query.

```
/* Revise the SQL query to correct any mistakes: */
SELECT
```