Boosting Text-to-SQL through Multi-grained Error Identification

Bo Xu[1], **Shufei Li**[1], **Hongyu Jing**[1], **Ming Du**[1], **Hui Song**[1,*],
Hongya Wang[1] and **Yanghua Xiao**[2]
[1]School of Computer Science and Technology, Donghua University,
[2]Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University,
xubo@dhu.edu.cn, {2222699, 2232833}@mail.dhu.edu.cn ,
{duming, songhui, hywang}@dhu.edu.cn, shawyh@fudan.edu.cn

Abstract

Text-to-SQL is a technology that converts natural language questions into executable SQL queries, allowing users to query and manage relational databases more easily. In recent years, large language models have significantly advanced the development of text-to-SQL. However, existing methods often overlook validation of the generated results during the SQL generation process. Current error identification methods are mainly divided into self-correction approaches based on large models and feedback methods based on SQL execution, both of which have limitations. We categorize SQL errors into three main types: system errors, skeleton errors, and value errors, and propose a multi-grained error identification method. Experimental results demonstrate that this method can be integrated as a plugin into various methods, providing effective error identification and correction capabilities.

1 Introduction

Text-to-SQL is a technology designed to convert natural language questions into executable SQL queries, offering users a more intuitive and user-friendly interface for querying and managing relational databases, thereby enhancing database usability and query efficiency (Deng et al., 2022).

In recent years, large language models (LLMs) like ChatGPT and GPT-4 have achieved remarkable success across various natural language processing tasks (Ouyang et al., 2022; Achiam et al., 2023). By leveraging in-context learning techniques, researchers have effectively harnessed the knowledge of LLMs (Dong et al., 2023).

Despite significant progress in generating SQL, existing methods often overlook the critical step of validating the generated results. Unlike natural language, SQL is a language with very strict syntax, and any non compliant part of a statement will be rejected by the database. To address this issue, current methods based on large language models primarily follow a generation-identification-correction framework: first generating the SQL, then detecting any errors, and finally correcting these errors. These methods can be classified into two major categories, with the key difference lying in their approach to error identification, as illustrated in Figure 1. The first category employs the large language model itself to detect errors in the generated SQL (Pourreza and Rafiei, 2024; Wang et al., 2024). However, because these models are not specifically designed for SQL syntax validation, they often struggle to accurately identify precise SQL errors. The second category relies on SQL execution engines, such as MySQL, to determine the presence of errors based on the execution results (Chen et al., 2023). Nonetheless, this approach is limited to detecting system errors and cannot identify a broader range of error types.

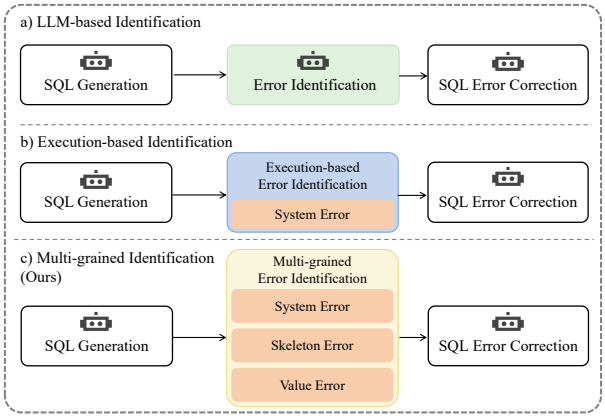

Figure 1: Comparison of different text-to-SQL methods based on the generation-identification-correction framework, highlighting the differences in error identification approaches.

Corresponding Author: Hui Song. The work reported in this paper is partially supported by the Fundamental Research Funds for the Central Universities 2232023D-19 and the NSF of Shanghai under grant number 22ZR1402000.

Proceedings of the 31st International Conference on Computational Linguistics, pages 4282–4292
January 19–24, 2025.

To address the above challenges, we propose a multi-grained error identification method designed to enhance text-to-SQL generation results. We categorize SQL errors into three main types: system errors, skeleton errors, and value errors. As illustrated in Figure 2, **System Error** arises when the predicted SQL query contains invalid syntax that prevents it from being executed by the SQL engine. **Skeleton Error** occurs when the predicted SQL query structural mismatches the expected query after removing specific value-related components. **Value Error** is identified when the values used in the predicted SQL query do not align with the values expected by the database. Specifically, to identify system errors, we employ an SQL executor, which flags these errors based on execution failures. To identify skeleton errors, we introduce a skeleton matching model that measures the structural similarity between the question and the generated SQL to identify mismatches. To identify value errors, we interact with the database, comparing the predicted values with the actual values in the relevant columns to identify inconsistencies. Once errors are identified, our framework outputs the error type along with detailed error information. This information is then used to guide a large language model in correcting the identified errors, ensuring a more accurate SQL query generation.

Our main contributions are as follows:

- We systematically study the errors that occur during SQL generation, categorizing them into three distinct types and proposing a multi-grained identification method. This approach provides detailed error information that facilitates subsequent model correction.

- We introduce a skeleton matching module based on contrastive learning to identify skeleton errors by comparing the semantic similarity between the query and the SQL skeleton. To enhance the encoder's representation capability, we incorporate hard negative examples into the learning process.

- Experiments demonstrate that our approach can serve as a versatile plugin, enhancing the performance of various methods across different models. Through ablation study, we highlight the importance of each module within the method.

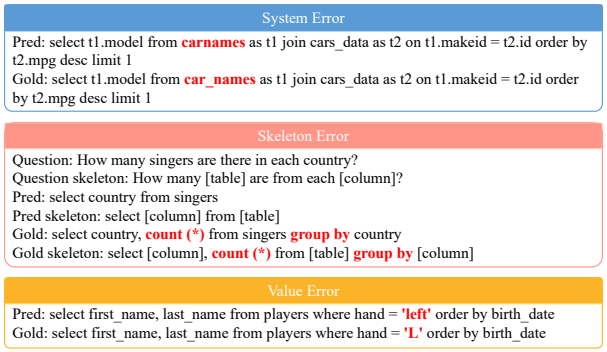

Figure 2: An example illustrating three types of errors, with each error highlighted in red for easy identification.

2 Overview

In this section, we first define our problem and then introduce our framework.

2.1 Problem Formulation

The text-to-SQL task can be divided into three sequential subtasks:

1) **SQL Generation**: Given a natural language question q and a relational database $\mathcal{D}$ comprised of tables $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ and columns $\mathcal{C} = \{c_1^1, c_2^1, ..., c_j^i, ..., c_{m_n}^n\}$, where c_j^i represents the j-th column of table T_i, the objective is to generate an initial SQL query s. Thus, the input for this subtask is q and $\mathcal{D}$, and the output is s.

2) **Error Identification**: Given the natural language question q, the relational database $\mathcal{D}$, and the generated SQL query s, this subtask aims to determine whether s is correct or erroneous. If the query is erroneous, the output should include specific error type and information. Therefore, the input consists of q, $\mathcal{D}$, and s, and the output is a binary indication of the correctness of s along with the relevant error type and information, if applicable.

3) **SQL Error Correction**: If the SQL query s is identified as erroneous in the previous subtask, this final subtask addresses the correction of the query. Given the natural language question q, the relational database $\mathcal{D}$, the initial SQL query s, the error type, and the error information, the objective is to generate a corrected SQL query. Thus, the input for this subtask comprises q, $\mathcal{D}$, s, the error type and the error information, and the output is the corrected SQL query s'.

In this paper, we focus on the **Error Identification** subtask. Specifically, we define three error types, namely *System Error*, *Skeleton Error* and

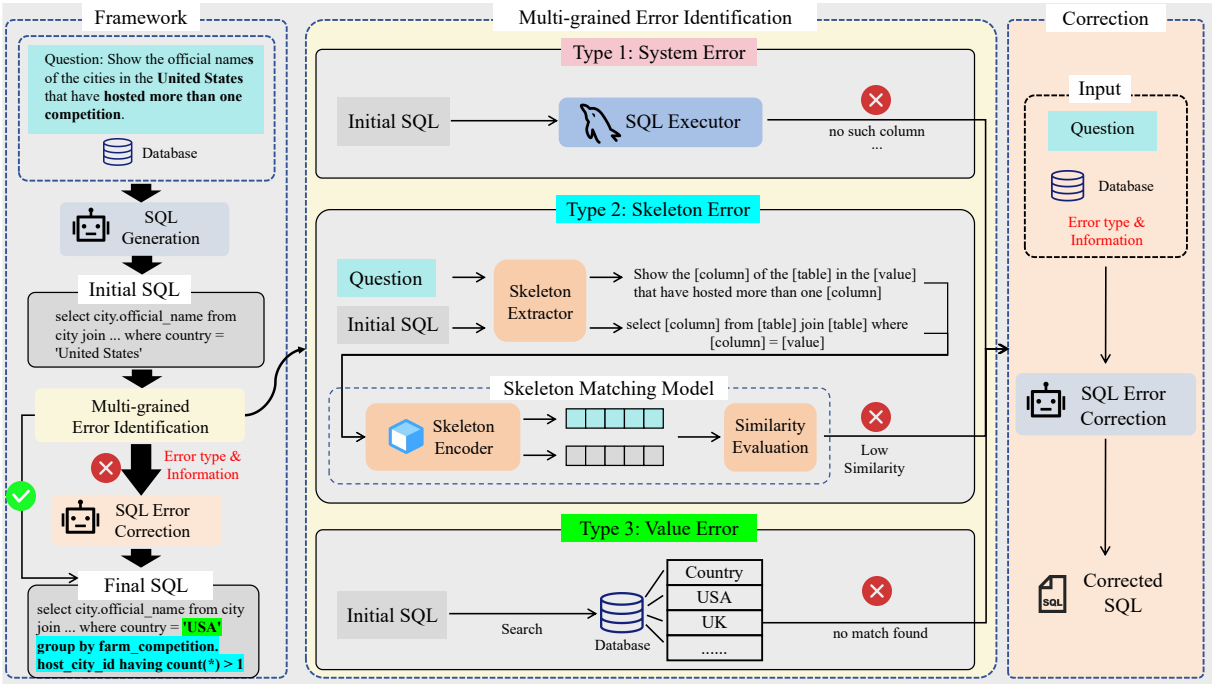

Figure 3: The Generation-Identification-Correction Text-to-SQL Framework with Multi-grained Error Identification Method.

Value Error.

2.2 Framework

The overall text-to-SQL framework, as illustrated in Figure 3, consists of three stages: SQL generation, multi-grained error identification, and SQL error correction. As mentioned above, the focus of this paper is on error identification. The SQL generation and SQL error correction stages can directly utilize existing large language models. The multi-grained error identification stage is composed of three modules: the system error identification module, the skeleton error identification module, and the value error identification module. Each module is responsible for identifying specific types of SQL errors and providing corresponding error information.

The entire process is as follows: we employ a large language model to generate an initial SQL query based on the given natural language question and the database schema. The generated query then undergoes identification through three distinct modules. First, the system error identification module employs an SQL executor to identify any syntax errors. Second, the skeleton error identification module utilizes a skeleton matching model based on contrastive learning to determine whether the SQL skeleton aligns with the question. Finally, the value error identification module checks for discrepancies between the values in the question and the database. By leveraging these three error identification modules, we hierarchically identify errors in the SQL query. Finally, specific error information is provided to the SQL error correction models for correcting the identified errors.

3 Method

In this section, we provide a detailed introduction to the text-to-SQL framework. The prompt details for the large language models used in SQL generation and SQL error correction can be found in Appendix A.

3.1 SQL Generation Stage

The objective of this stage is to take a question and a database schema as input and output SQL. This is a modular component that can be replaced with any text-to-SQL method based on large language models. These methods include using in-context learning to directly leverage open-source and closed-source large language models, or fine-tuning open-source large language models using a text-to-SQL dataset. The process is defined as follows:

$$s = LLM(q, D), \tag{1}$$

where q is the question, D is the database schema and s is the generated SQL.

3.2 Error Identification Stage

This stage will introduce the modules for identifying system errors, skeleton errors, and value errors separately.

3.2.1 System Error Identification

This module primarily checks for syntax errors in the generated SQL. Specifically, we determine the presence of syntax errors by analyzing the results of SQL execution. In simple terms, any SQL that fails to execute correctly is identified as having a syntax error. These errors may include incorrect table aliases, mistakes in joining columns, column name errors, or improper use of keywords. To facilitate correction, we categorize all these issues under system errors and collect the corresponding error information.

3.2.2 Skeleton Error Identification

This module is designed to detect inconsistencies in intent between the SQL skeleton and the question skeleton. To achieve this, we employ contrastive learning to help the model understand the underlying relationships between the intended question skeleton and the SQL skeleton. Once these hidden correspondences are learned, we use a similarity threshold to determine whether the question skeleton and the SQL skeleton align.

We first the skeletons of the question x and the SQL query y by abstracting specific table names, column information, and values, a process achieved through matching with the database schema. As illustrated in Figure 3, skeletonizing the question and SQL query allows the model to focus more on their underlying structure. After constructing the skeletons, we use contrastive learning to train a skeleton matching model. The goal is to bring the skeletons of similar questions and SQL queries closer together in the representation space while pushing apart those with different skeletons. We believe this method enhances the alignment between the skeletons of questions and SQL queries. For instance, the phrase "more than" in a question might correspond to the "having count" keyword in SQL, reflecting similar underlying structures.

Specifically, we treat the question skeletons and SQL skeletons within the same question-SQL pair as having the same label. By using the supervised contrastive learning (Khosla et al., 2020) loss function, we bring the vector representations of the skeletons with the same intent closer together. Additionally, to better capture the corresponding relationships between skeletons, we introduce hard negative samples to help the model learn more challenging semantic scenarios. For each SQL skeleton, we select a variant that has one more or one fewer common keyword as a hard negative sample. The overall training objective is as follows:

$$\mathcal{L}_{out}^{sup} = \sum_{i \in I} \frac{-1}{|P(i)|} \sum_{p \in P(i)} \log \frac{exp(x_i \cdot y_p / \tau)}{\sum_{a \in A(i)} \exp(x_i \cdot y_a / \tau) + N}, \tag{2}$$

Where x_i represents the i-th question skeleton in the batch, y_p represents the p-th positive SQL skeleton in the batch, $P(i)$ represents the set of indices for all positive samples in the batch, and N represents the hard negative samples.

Finally, skeleton errors are identified by evaluating the similarity between the question skeleton and the SQL skeleton. If the similarity score falls below a predefined threshold τ, the SQL is deemed to have a skeleton error.

3.2.3 Value Error Identification

This module primarily addresses discrepancies between the values referenced in a question and those stored in the database schema. For example, if a question requires filtering by gender using `Male` or `Female`, but the database stores these values as `M` or `F`, a mismatch arises. If SQL generation only considers the query information and overlooks these differences in the schema, it may result in the SQL query returning an empty result set. This module aims to identify and correct such inconsistencies to prevent errors during query execution.

Specifically, we only examine SQL queries that involve filtering values. If the SQL query returns an empty result set, we compare the filter values against the corresponding column in the database. If the filter values are present in the relevant column, we consider the SQL query correct; if the filter values are not found in the corresponding column, we mark the SQL query as potentially erroneous. For queries with multiple filter values, each value is checked individually; if any value is missing from the database, the query is flagged as potentially erroneous.

In reality, the SQL query returning an empty result set can also be correct. This module uniformly treats it as a value error. If it is correct, the subsequent error correction module will not modify the SQL; if it is incorrect, the subsequent error correction module will modify it based on the values in the corresponding column of the database.

3.3 Error Correction Stage

At this stage, we pass the information obtained from error identification to the respective correction models to perform SQL correction. In line with the SQL generation stage, this is a modular component that can be replaced with any text-to-SQL method based on large language models. Below, we will provide a detailed introduction to the method of fine-tuning open-source large language models using text-to-SQL datasets. The general process is outlined in Appendix A.

3.3.1 System Errors and Skeleton Error Correction

The error correction model takes as input a problem, database schema, error information, and erroneous SQL, and outputs a corrected SQL query. To train the model, we randomly selected 1,000 examples from the training set, following the same procedures as those used for the SQL generation module. After training, we used the model to predict the remaining data in the training set and identified SQL queries as erroneous if their execution results differed from the Gold SQL results. We collected a total of 4,921 erroneous SQL queries by repeatedly sampling and identifying discrepancies.

We categorize the erroneous datasets into two distinct types of errors: system errors and skeleton errors. The specific details for prompt correction are provided in Appendix A. Similar to SQL generation model, our error correction training dataset contains four elements: $\mathcal{Z}_c = \{(q_i, \mathcal{S}_i, e_i, s_i, s_i')\}_{i=1,..,N}$, where $q_i, \mathcal{S}_i$, e_i, s_i and s_i' represent the natural language question, the corresponding database schema, the error information, the erroneous SQL, and the gold SQL, respectively. Our goal of fine-tuning the SQL error correction model $\mathcal{M}_c$ is to maximize the conditional language modeling objective:

$$\max_{\mathcal{M}_c} \sum_{(q,\mathcal{S},e,s,s')\in\mathcal{Z}_c} \sum_{t=1}^{|s'|} P_{\mathcal{M}_c}(s_t'|q, \mathcal{S}, e, s, s'_{<t}), \tag{3}$$

Where s_t' represents the t-th token of the expected SQL. By providing incorrect guidance and faulty SQL, the model is instructed to correct the erroneous SQL based on the type of error identified.

3.3.2 Value Error Correction

To correct value errors, we trained a value error correction model using the information available from the training set. We manually cleaned and processed the data in the training set where there was a gap between the required values in the questions and the value ranges available in the database. Based on the discrepancies between the question values and the database value ranges, we constructed erroneous SQL queries, resulting in a total of 247 entries. To ensure that cases where the question value does not fall within the database value range, but an empty output is considered correct, were properly handled, we included scenarios where the output itself is empty. This resulted in a total of 532 entries.

To address the gap between the required values and those present in the database, we randomly select 30 values from each filtering column in the SQL queries. The training set $\mathcal{Z}_D$, similar to our SQL error correction model, consists of $\{(q_i, v_i, s_i, s_i')\}_{i=1,..,N}$, where q_i, v_i, s_i and s_i' represent the natural language question, the values of columns filtered by SQL, the erroneous SQL, and the corrected SQL, respectively. Our goal of fine-tuning the value error correction model $\mathcal{M}_D$ is to maximize the conditional language modeling objective:

$$\max_{\mathcal{M}_D} \sum_{(q,v,s,s')\in\mathcal{Z}_D} \sum_{t=1}^{|s'|} P_{\mathcal{M}_D}(s_t'|q, v, s, s'_{<t}), \tag{4}$$

Where s_t' represents the t-th token of the expected SQL.

We employ the value error correction model to address the gap between the question values and the database values, working in conjunction with the value error identification module for comprehensive evaluation. We input the question, the SQL column values being filtered, and the erroneous SQL query into the model. The model then processes this input to either correct the SQL query to align with the appropriate value range or determine that no correction is necessary.

4 Experiment

Our experiments consist of two parts. The first part evaluates the performance of the multi-grained error identification method (MGEI). The second part assesses the overall improvement in text-to-SQL performance after existing methods incorporate our approach. This section presents the experimental settings and results.

Methods	precision(%)	recall(%)	F1(%)
Codellama	32.2	**75.5**	45.2
ChatGPT	65.0	53.3	58.6
DIN-SQL	75.3	54.5	63.2
MAC-SQL	75.5	51.6	61.4
DEA-SQL	69.1	56.4	62.1
PURPLE	**100**	37.4	54.4
MGEI	94.1	49.4	**64.8**

Table 1: Comparison of our method with previous approaches in terms of identification error on the Spider dataset.

Methods	EM	EX
Fine-tuning		
CodeLlama-13b-Instruct	72.7	75.0
+ MGEI + CodeLlama	**73.2**	**78.6**
CodeLlama-13b-Python	70.6	72.3
+ MGEI + CodeLlama	**72.3**	**76.5**
In-context Learning		
DIN-SQL (GPT4)	54.3	76.8
+ MGEI + GPT4	**56.0**	**81.4**
DAIL-SQL (ChatGPT)	26.7	74.4
+ MGEI + ChatGPT	**27.2**	**76.0**

Table 2: Performance of MGEI as a plugin across different methods. All results are obtained by running the code released by the author.

4.1 Experimental Setup

4.1.1 Data

We evaluated our method using the Spider dataset, a comprehensive cross-domain collection for Text-to-SQL tasks. Each instance comprises a natural language question tailored to a specific database, along with its corresponding SQL query. We opted for the Spider development subset for our evaluation, given that the test subset has not been made publicly available.

We use SQL queries generated by the fine-tuned CodeLlama-13b-Instruct model on the Spider validation set as the dataset for evaluating error identification performance. This dataset includes a total of 257 erroneous SQL queries. The detailed distribution of error types is illustrated in Figure 4.

4.1.2 Evaluation Metrics

When evaluating the model's ability to identify SQL errors, it is crucial not only to detect as many errors as possible but also to ensure that the detected errors are indeed accurate. Therefore, we use precision, recall, and the F1 score as our evaluation metrics. Meanwhile, we follow the previously established research methodology, employing both exact-set-match accuracy (EM) and execution accuracy (EX) for evaluation purposes. Following previous work (Zhong et al., 2020), we utilize the evaluation scripts available at https://github.com/taoyds/test-suite-sql-eval.

4.1.3 Parameter Settings

We performed all experiments using 2 Nvidia RTX A6000 GPUs with PyTorch 2.0.0. The parameters for our method were configured as follows: we utilized the CodeLlama-13b-Instruct and CodeLlama-13b-Python models (Roziere et al., 2023) across all modules. For fine-tuning the LLMs, we employed LoRA (Hu et al., 2021) for efficient adaptation, setting the parameters to r = 16 and α = 64, with a learning rate of 5×10^{-4}. In the skeleton error identification module, we set a threshold τ of 0.3 to filter out SQL queries with mismatched skeletons.

4.1.4 Baselines

When comparing the performance of error identification, we conducted a comprehensive comparison by testing the correction modules of various methods, including CodeLlama-13b-Instruct, ChatGPT in a zero-shot setting, and other approaches such as DIN-SQL, MAC-SQL (Wang et al., 2024), DEA-SQL (Xie et al., 2024), and PURPLE (Ren et al., 2024). The prompts used for error identification of SQLs directly with CodeLlama-13b-Instruct and ChatGPT are shown in Appendix B. For other self-correction methods, including DIN-SQL, MAC-SQL, and DEA-SQL, we compare their output with the initial SQL; if the outputs are different, the SQL is considered erroneous. PURPLE only recognizes SQL that produce execution errors; therefore, we categorize system errors as erroneous SQL. Each method's ability to identify errors was individually assessed and compared against our own approach.

To evaluate the effectiveness of our method as a plugin, we used DIN-SQL and DAIL-SQL to generate initial SQL queries. DIN-SQL explores breaking down complex text-to-SQL tasks into smaller sub-tasks. DAIL-SQL compared existing prompt engineering methods, including question representation, example selection, and example organization, and based on these comparisons, proposed a new integrated solution designed to overcome the limitations of current methods. For DAIL-SQL, we utilized ChatGPT in a zero-shot scenario to generate SQL based on the code provided by the au-

Method	Easy	Medium	Hard	Extra Hard	All
CodeLlama-13b-Instruct + MGEI + CodeLlama-13b-Instruct					
No Error Identification	87.5	84.3	62.6	45.2	75.0
w/ System Error Identification	87.5	86.3	64.9	45.2	76.4
w/ System and Skeleton Error Identification	88.7	86.1	64.9	45.2	76.6
w/ System, Skeleton and Value Error Identification	91.5	87.7	67.2	47.0	78.6
DAIL-SQL (ChatGPT) + MGEI + ChatGPT					
No Error Identification	89.9	79.8	62.6	48.8	74.4
w/ System Error Identification	90.3	80.5	63.8	50.0	75.1
w/ System and Skeleton Error Identification	90.3	80.5	63.8	50.0	75.1
w/ System, Skeleton and Value Error Identification	92.3	80.5	63.8	52.4	76.0

Table 3: Ablation study on the execution accuracy of the MGEI module across different text-to-SQL methods.

thors. Additionally, we fine-tuned the CodeLlama-13b-Instruct and CodeLlama-13b-Python models following the DAIL-SQL-SFT format to produce initial SQL queries. This approach allowed us to test the effectiveness of our method across multiple models.

4.2 Effectiveness

Firstly, we evaluate the model's ability to identify errors. Then, we demonstrate the performance improvements after the model corrects these errors.

4.2.1 The Performance of Identifying Errors

We compared the performance of MGEI with six other methods in detecting errors on the Spider dataset. Table 1 presents the results of each method across three metrics: precision, recall, and F1-score.

Firstly, from the perspective of precision, there are significant differences in the accuracy of error identification across the models. Among them, PURPLE stands out with a precision of 100%, the highest among all, due to its focus solely on correcting system errors. However, the MGEI method also demonstrates exceptional performance, achieving a precision of 94.1%. This indicates that MGEI is highly accurate in identifying errors. In contrast, Codellama has the lowest precision at just 32.2%, highlighting its significant shortcomings in accurately detecting errors.

Secondly, from a recall perspective, the Codellama method excels at identifying as many errors as possible, achieving a recall of 75.5%, which indicates its ability to detect the majority of errors. However, despite its high recall, Codellama's precision is lower, suggesting that while it identifies a large number of errors, it also introduces a significant number of false positives. In comparison, MGEI has a recall of 49.4%, which is moderate among the methods tested. ChatGPT shows a similar recall rate at 53.3%, close to that of MGEI, while other methods such as DIN-SQL (54.5%) and MAC-SQL (51.6%) also have comparable recall rates, though none surpass Codellama.

Finally, when evaluating the overall F1-score, MGEI leads with a score of 64.8%, highlighting its excellent balance between precision and recall. Although CodeLlama achieves the highest recall, its F1-score is only 45.2%, indicating that its lower precision significantly impacts its overall performance. On the other hand, while PURPLE boasts exceptionally high precision, its low recall results in an F1-score of just 54.4%, preventing it from surpassing MGEI in overall performance. Other methods, such as DIN-SQL (63.2%), MAC-SQL (61.4%), and DEA-SQL (62.1%), also fall short of MGEI in F1-score. This demonstrates the effectiveness and comprehensiveness of our method in identifying the three types of errors.

4.2.2 The Performance of Improvement

Table 2 shows the performance of our method compared to the baseline methods across different setups. Our method consistently improves execution accuracy (EX) in all cases. When integrated with CodeLlama-13b-Instruct, our method increases EX by 3.6%, achieving 78.6%. Similarly, with CodeLlama-13b-Python, our method yields a 4.2% boost in EX. Notably, when combined with DIN-SQL+GPT4, our approach enhances both EM and EX, with a significant 4.6% increase in EX. Although the improvement is more modest when paired with DAIL-SQL+ChatGPT, our method still achieves higher scores, demonstrating its effectiveness as a universal plugin across various models and prompts.

4.3 Ablation Study

In the ablation study, we examine the impact of different modules in our method by testing them on the CodeLlama-13b-Instruct and DAIL-SQL (ChatGPT) models, focusing on execution accuracy. The results are summarized in Table 3.

Starting with the CodeLlama-13b-Instruct model, the baseline performance of generating SQL without any error identification (Initial SQL) achieves an accuracy of 75.0%. When the system error identification module is introduced, there is a noticeable improvement, with the overall accuracy rising to 76.4%. Adding the skeleton error identification further enhances the accuracy slightly to 76.6%. The most significant improvement is observed when the value error identification is included, raising the overall accuracy to 78.6%, with particularly strong gains in the hard and extra hard categories. A similar pattern is evident for the DAIL-SQL (ChatGPT) model. The baseline (Initial SQL) starts with an accuracy of 74.4%, which increases to 75.1% after integrating the system error identification module. The performance remains consistent with the addition of the skeleton error identification module, and with the full integration of all three identification modules, the accuracy rises to 76.0%, with the most notable improvements seen in the easy and extra hard categories. These results underscore the critical role of each module in enhancing the overall execution accuracy of our method.

5 Related Work

In this section, we review and summarize the studies that are most closely related to our research.

5.1 Text-to-SQL

Early approaches primarily relied on carefully designed rules and templates(Popescu et al., 2004), which were only effective in simple database scenarios. However, as databases became more complex, creating a specific rule or template for each scenario proved increasingly challenging. In recent years, the advancement of deep neural networks has significantly propelled the progress of text-to-SQL tasks by enabling the automatic mapping of user queries to corresponding SQL commands(Sutskever et al., 2014; Guo et al., 2019; Xu et al., 2017). simultaneously, large-scale datasets like Spider (Yu et al., 2018) and BIRD (Li et al., 2024b) have been released. Subsequently, pre-trained language models (PLMs), which offer superior semantic parsing capabilities, have emerged as the new paradigm in text-to-SQL systems(Yin et al., 2020). Recently, methods based on large language models (LLMs) that utilize in-context learning and fine-tuning paradigms have become mainstream, achieving state-of-the-art accuracy in text-to-SQL tasks(Li et al., 2024a; Jiang et al., 2023a).

5.2 Self-correction

There has been extensive research on improving the responses of large language models (LLMs) during inference, including in areas like arithmetic reasoning, code generation, and question answering (Gao et al., 2023; Shinn et al., 2024; Brown et al., 2020). Some studies have proposed self-correction methods, where the LLM generates feedback for itself through prompting to correct its responses(Gou et al., 2023; Jiang et al., 2023b). Additionally, other research has explored the use of external information to enhance feedback, such as leveraging external tools like code executors(Chen et al., 2023), external knowledge from search engines(Jiang et al., 2023c), or additional information from sources like Wikipedia(Yu et al., 2023). However, recent studies have also reported negative results, indicating that LLMs may not always be capable of self-correction(Valmeekam et al., 2023; Stechly et al., 2023). Therefore, we evaluate SQL errors from multiple contextual perspectives to ensure accuracy.

6 Conclusion

In this paper, we systematically examined SQL generation errors, categorizing them into three types, and proposed a multi-grained identification method that provides detailed error information for effective correction. We introduced a skeleton matching module based on contrastive learning to detect skeleton errors, enhancing representation with hard negative examples. Our experiments validate the versatility of our approach as a plugin, improving the performance of both open-source and closed-source large language models. Ablation study further underscore the significance of each module in our method. This innovative classification and modular design offer a new perspective for more accurate SQL error handling.

Limitation

Firstly, our approach primarily focuses on basic syntax design, such as MySQL, without addressing specialized syntax for multiple databases or complex constructs like window functions. This limitation reduces the method's ability to handle complex queries in multi-database environments. Secondly, although the method is intended to be widely used as a plugin, it may encounter compatibility issues when integrated with existing systems, particularly in environments with custom or legacy SQL generation tools. This often requires additional customization or adjustments, increasing the complexity of integration and maintenance costs.

References

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent advances in text-to-SQL: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea. International Committee on 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*.

Luyu Gao, Zhuyun Dai, Panupong Pasupat, Anthony Chen, Arun Tejasvi Chaganty, Yicheng Fan, Vincent Y Zhao, Ni Lao, Hongrae Lee, Da-Cheng Juan, et al. 2023. Rarr: Researching and revising what language models say, using language models. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.

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. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy. Association for Computational Linguistics.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Xin Zhao, and Ji-Rong Wen. 2023a. StructGPT: A general framework for large language model to reason over structured data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9237–9251, Singapore. Association for Computational Linguistics.

Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2023b. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*.

Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023c. Active retrieval augmented generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7969–7992.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *Neural Information Processing Systems,Neural Information Processing Systems*.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024a. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024b. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. 2004. Modern natural language interfaces to databases: Composing

statistical parsing with semantic tractability. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 141–147.

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

Tonghui Ren, Yuankai Fan, Zhenying He, Ren Huang, Jiaqi Dai, Can Huang, Yinan Jing, Kai Zhang, Yifan Yang, and X Sean Wang. 2024. Purple: Making a large language model a better sql writer. *arXiv preprint arXiv:2403.20014*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Kaya Stechly, Matthew Marquez, and Subbarao Kambhampati. 2023. Gpt-4 doesn't know it's wrong: An analysis of iterative prompting for reasoning problems. *arXiv preprint arXiv:2310.12397*.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. 2023. Can large language models really improve by self-critiquing their own plans? *arXiv preprint arXiv:2310.08118*.

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

Yuanzhen Xie, Xinzhou Jin, Tao Xie, MingXiong Lin, Liang Chen, Chenyun Yu, Lei Cheng, ChengXiang Zhuo, Bo Hu, and Zang Li. 2024. Decomposition for enhancing attention: Improving llm-based text-to-sql through workflow paradigm. *arXiv preprint arXiv:2402.10671*.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426.

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

Wenhao Yu, Zhihan Zhang, Zhenwen Liang, Meng Jiang, and Ashish Sabharwal. 2023. Improving language models via plug-and-play retrieval feedback. *arXiv preprint arXiv:2305.14002*.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-sql with distilled test suites. *arXiv preprint arXiv:2010.02840*.

A Prompt

A.1 Correction Process

We abstract and generalize the error correction process, which takes as input a problem, a database, error information, and the erroneous SQL, and outputs the corrected SQL. It is defined as follows:

$$s' = LLM(q, D, e, s), \tag{5}$$

where q is the question, D is the database schema, e is the error information, s is the generated SQL and s' is the corrected SQL.

A.2 prompt Examples

Table 4 provides examples of all prompts, including SQL generation, system error correction, skeleton error correction, value error correction, and error identification (ChatGPT & CodeLlama).

B Experimental settings

B.1 Error types

Figure 4 illustrates the distribution of different types of errors in the error identification dataset.

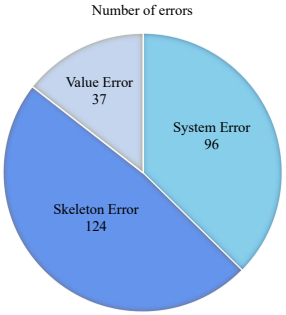

Figure 4: This is the distribution of error types in our error identification dataset.

SQL Generation
Write a sql to answer the question "How many singers do we have?" ### Input: concert(concert_id, concert_name, theme, stadium_id, year) singer(singer_id, name, country, song_name, song_release_year, age, is_male) singer_in_concert(concert_id, singer_id) stadium(stadium_id, location, name, capacity, highest, lowest, average) ### SQL: select
System Error Correction
You are a database expert. I will provide a question, the database schema, and non-executable SQL. There are syntax errors in non-executable SQL, such as s̈yntax errorsänd n̈o such column.̈ Please correct it. Question: "Which model saves the most gasoline? That is to say, have the maximum miles per gallon." ### Input: car_makers(id, maker, fullname, country) car_names(makeid, model, make) cars_data(id, mpg, cylinders, edispl, horsepower, weight, accelerate, year) continents(contid, continent) countries(countryid, countryname, continent) model_list(modelid, maker, model) Non-executable SQL: select model_list.model from model_list join cars_data on model_list.modelid = cars_data.modelid order by cars_data.mpg desc limit 1
Skeleton Error Correction
You are a database expert. I will provide a question, the database schema, and skeleton error SQL. The syntax of the SQL is correct, but the execution results do not match the answer to the question. Do not alter any value-specific parts, focus solely on correcting the structural elements related to SQL keywords. Please correct it. Question: "What is the maximum capacity and the average of all stadiums ?" ### Input: concert(concert_id, concert_name, theme, stadium_id, year) singer(singer_id, name, country, song_name, song_release_year, age, is_male) singer_in_concert(concert_id, singer_id) stadium(stadium_id, location, name, capacity, highest, lowest, average) Skeleton error SQL: select max (capacity), avg (average) from stadium
Value Error Correction
You are a database expert. I will provide a question, the expected values for the columns in the SQL query, and an SQL query. Focus solely on verifying and correcting the values used in the SQL query to ensure they match the expected values in the database. Pay attention to discrepancies such as case sensitivity, abbreviations, and any other issues that might cause a mismatch between the SQL values and the expected values. If there are no discrepancies, return the original SQL query. Otherwise, correct the erroneous SQL query. Question: "What is allergy type of a cat allergy?" ### column value: allergy:[Anchovies, Bee Stings, Cat, Dog, Eggs, Grass Pollen, Milk, Nuts, Ragweed, Rodent, Shellfish, Soy, Tree Pollen, Wheat] SQL: select allergytype from allergy_type where allergy = ‘cat‘
Error identification (ChatGPT & Codellama)

You are a database expert. I will provide a question, database schema, and SQL query,
and you will determine if the SQL query can answer the question.
# Question: "How many singers do we have?" 	
# Database schema: 	
concert(concert_id, concert_name, theme, stadium_id, year)
singer(singer_id, name, country, song_name, song_release_year, age, is_male)
singer_in_concert(concert_id, singer_id)
stadium(stadium_id, location, name, capacity, highest, lowest, average)
# SQL: select count (*) from singer 	
# Only respond with "Yes" or "No," no explanation needed. 	

Table 4: Examples of prompt templates.