MCTS-SQL: An Effective Framework for Text-to-SQL with Monte Carlo Tree Search

Shuozhi Yuan[1] , **Liming Chen**[1] , **Miaomiao Yuan**[2] , **Jin Zhao**[1] , **Haoran Peng**[1] , **Wenming Guo**[3]
[1]China Telecom Digital Intelligence
[2]Institute of Computing Technology, Chinese Academy of Sciences
[3]Beijing University of Posts and Telecommunication
{yuansz,chenlm3}@chinatelecom.cn, yuanmiaomiao@ict.ac.cn , guowenming@bupt.edu.cn

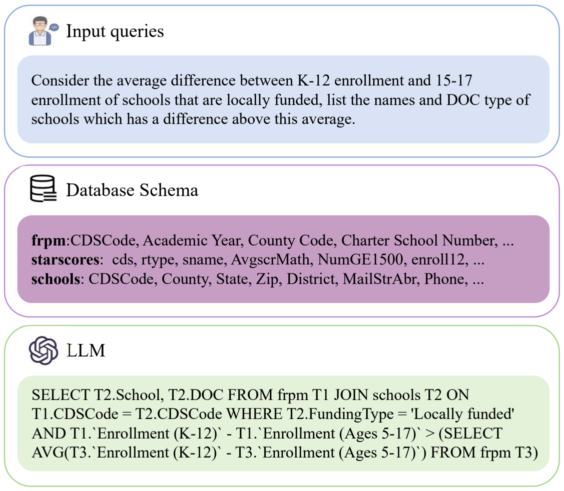
Abstract

Text-to-SQL is a fundamental and longstanding problem in the NLP area, aiming at converting natural language queries into SQL, enabling non-expert users to operate databases. Recent advances in LLM have greatly improved text-to-SQL performance. However, challenges persist, especially when dealing with complex user queries. Current approaches (e.g., COT prompting and multi-agent frameworks) rely on the ability of models to plan and generate SQL autonomously, but controlling performance remains difficult. In addition, LLMs are still prone to hallucinations. To alleviate these challenges, we designed a novel **MCTS-SQL** to guide SQL generation iteratively. The approach generates SQL queries through Monte Carlo Tree Search (MCTS) and a heuristic self-refinement mechanism are used to enhance accuracy and reliability. Key components include a schema selector for extracting relevant information and an MCTS-based generator for iterative query refinement. Experimental results from the SPIDER and BIRD benchmarks show that MCTS-SQL achieves state-of-the-art performance. Specifically, on the BIRD development dataset, MCTS-SQL achieves an Execution (EX) accuracy of **69.40%** using GPT-4o as the base model and a significant improvement when dealing with challenging tasks, with an EX of **51.48%**, which is **3.41%** higher than the existing method.

1 Introduction

Text-to-SQL aims at converting natural language queries to SQL, which plays a critical role in data analytics and supports a wide range of applications (e.g., business intelligence and automated reporting). Recent advances in LLM have significantly improved the accuracy and stability of text-to-s SQL systems.

However, real-world Text-to-SQL applications face with significant challenges, mainly due to user queries' random, often ambiguous nature (as seen in Figure 1). Text-to-SQL is challenging for three main reasons. First, natural language has ambiguous expressions and rich semantics, making it hard to map them to specific database tables and fields. Second, extracting meaningful information from natural language requires strong understanding capabilities. Lastly, generating accurate SQL queries that meet often vague requirements demands both semantic understanding and syntactic precision.



Consider the average difference between K-12 enrollment and 15-17 enrollment of schools that are locally funded, list the names and DOC type of schools which has a difference above this average.

frpm:CDSCode, Academic Year, County Code, Charter School Number, ...
starscores: cds, rtype, sname, AvgscrMath, NumGE1500, enroll12, ...
schools: CDSCode, County, State, Zip, District, MailStrAbr, Phone, ...

```
SELECT T2.School, T2.DOC FROM frpm T1 JOIN schools T2 ON
T1.CDSCode = T2.CDSCode WHERE T2.FundingType = 'Locally funded'
AND T1.`Enrollment (K-12)` - T1.`Enrollment (Ages 5-17)` > (SELECT
AVG(T3.`Enrollment (K-12)` - T3.`Enrollment (Ages 5-17)`) FROM frpm T3)
```

Figure 1: An example of Text-to-SQL, where the user's queries are highly complex, leading to challenges for the LLM in accurately selecting the appropriate fields and solving tasks such as table selection and joins.

To tackle these challenges, existing algorithms have employed various strategies, typically falling into categories such as rule-based methods[Kang *et al.*, 2012; Hammami *et al.*, 2021], sequence-to-sequence models[Zhong *et al.*, 2017; Katsogiannis-Meimarakis and Koutrika, 2021], and pre-trained language models[Katsogiannis-Meimarakis and Koutrika, 2023; Guo and Gao, 2019]. Early Text-to-SQL systems relied on rule-based approaches, which employed handcrafted grammar rules and heuristics. Subsequently, sequence-to-sequence models and pre-trained language models were introduced, enabling more accurate and flexible SQL generation. In the past three years, the advent of LLMs has significantly advanced Text-to-SQL, particularly in zero-shot and few-shot settings where models can generate SQL without extensive training examples.Despite these advancements, challenges remain, such as handling ambiguous queries and optimizing SQL for both performance and correctness. Most methods use COT or multi-agent frameworks, but due to the hallucination in LLMs, the ability of these self-planning strategies to solve complex queries is difficult to control.

A promising way to alleviate these limitations is to design a controllable chain of thought reasoning path, rather than allowing LLMs to generate it autonomously. Recent researches[Pitanov *et al.*, 2023; Li *et al.*, 2023; Chen *et al.*, 2024] have shown that Monte Carlo Tree Search(MCTS) is effective in complex decision-making and optimization tasks, making it a promising tool for Text-to-SQL. Through multiple iterations, MCTS explicitly selects the path with the highest probability to solve complex problems. In this paper, we integrate MCTS with Text-to-SQL, developing a more robust framework of tackling challenging SQL generation tasks that current LLMs struggle with. Furthermore, MCTS can effectively alleviate the mapping of ambiguous semantics in natural language to database fields through continuous attempts and feedback, thereby enhancing the alignment between the generated SQL and the user's vague intent.

However, MCTS typically requires a large number of additional tokens, which is unnecessary for simple SQL generation tasks. Therefore, our framework is designed with fast-slow thinking modules, which comprise a direct SQL generation component and an MCTS-based Refiner. Only when the SQL generation module fails to generate a valid query, the MCTS-based Refiner is employed for iterative optimization. The system performs a self-evaluation of SQL generation errors and, based on the results, applies Monte Carlo Tree Search to choose a better path to refine the solution step by step. Additionally, we propose a selector module, which partitions large databases into smaller sub-databases, identifying the most relevant data tables and columns.

The main contributions of our proposed MCTS-SQL can be summarized as follows:

- To our best knowledge, we are the first to apply Monte Carlo Tree Search(MCTS) to the Text-to-SQL domain, capitalizing on its superior ability to optimize decision-making in complex and challenging queries.
- We introduce MCTS-SQL, an innovative Text-to-SQL framework that integrates a dual-process thinking model, comprising a fast SQL generation module and an MCTS-based refinement process, with an auxiliary selector to optimize the overall performance.
- Through comprehensive experiments, we highlight the complementary strengths of LLMs and MCTS, demonstrating substantial improvements in handling complex Text-to-SQL tasks.
- In our experiments, we conducted extensive testing on the widely used Spider and BIRD datasets. The results demonstrate that MCTS-SQL, based on GPT-4, achieved a query execution accuracy of **69.40%** on the holdout dev set of BIRD, establishing state-of-the-art performance, particularly with significant improvements on challenging tasks, where it reaches **51.48%**, outperforming existing methods by **3.41%**.

2 Related Work

In this section, we provide an overview of related work on Text-to-SQL and Monte Carlo Tree Search, highlighting their relevance and main differences to our proposed research.

2.1 Text-to-SQL

Text-to-SQL aims to bridge natural language queries and structured database queries. Due to its complexity, numerous approaches are proposed to address its challenges. Early systems, such as LUNAR[Kang *et al.*, 2012] and NaLIX[Hammami *et al.*, 2021], employed rule-based methods that manually crafted grammar rules and heuristics. However, these approaches are highly domain-specific, and generalization performance across different tasks or databases is difficult to guarantee.

The deep learning marked a turning point for Text-to-SQL. End-to-end models like Seq2SQL[Zhong *et al.*, 2017] and SQLNet[Katsogiannis-Meimarakis and Koutrika, 2021] directly mapped natural language to SQL but struggled with complex queries, especially those involving nested structures or intricate reasoning. Pre-trained Language Models (PLMs), such as TaBERT[Katsogiannis-Meimarakis and Koutrika, 2023] and BERT-SQL[Guo and Gao, 2019], enhance cross-domain generalization and improve the accuracy of SQL generation. However, handling complex user queries and achieving powerful cross-domain performance remain significant challenges.

Recently, Large Language Models (LLMs) such as GPT-4[Achiam *et al.*, 2023], Palm-2[Anil *et al.*, 2023], and LLaMA[Touvron *et al.*, 2023] have revolutionized Text-to-SQL tasks. These models excel in zero-shot and few-shot settings due to extensive pretraining and advanced reasoning capabilities.[Lee *et al.*, 2024; Talaei *et al.*, 2024; Alp Caferoğlu and Ulusoy, 2024] Techniques like DAIL-SQL[Gao *et al.*, 2023] optimized prompt ngineering, focusing on question representation, prompt structure, and example selection to enhance SQL accuracy with minimal supervision. Frameworks like C3-SQL[Dong *et al.*, 2023], DIN-SQL[Pourreza and Rafiei, 2024], and StructGPT[Jiang *et al.*, 2023] further advanced the field by addressing complex queries through database simplification, query decomposition, and structured data access. Additionally, MAC-SQL[Wang *et al.*, 2024] introduced a collaborative framework integrating decomposer, auxiliary selector, and refiner modules for iterative SQL refinement.

From the analysis of existing methods, it is evident that advancing Text-to-SQL hinges on improving models' ability to reason through complex scenarios. Current approaches often rely on basic COT strategies or multi agent framework to autonomously plan, yet they suffer from hallucination issues. To address this, our work integrates Monte Carlo Tree Search to enhance models' comprehension of ambiguous table structures and queries, significantly improving SQL generation accuracy.

2.2 Monte Carlo Tree Search

MCTS is a widely used tool for planning complex problems, and a large number of downstream experiments have demonstrated its effectiveness. For example, [Pitanov *et al.*, 2023]

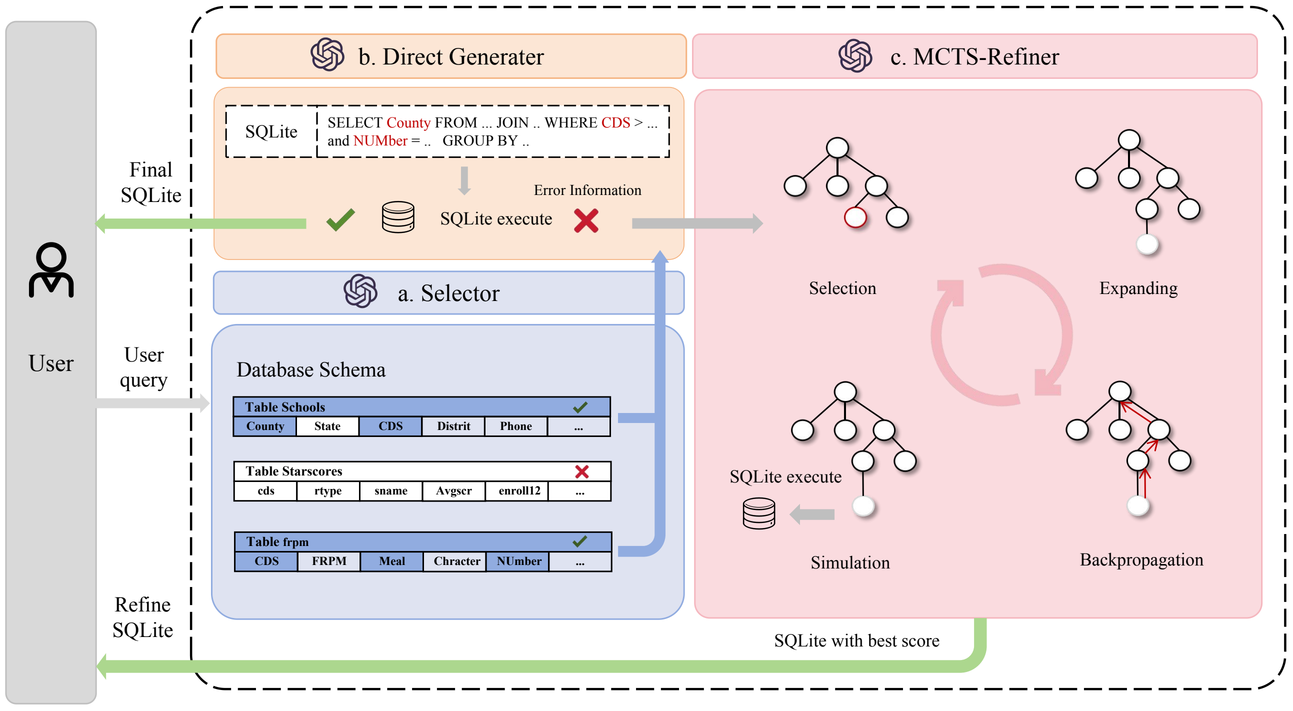


Figure 2: The MCTS-SQL framework is structured around three primary components: the **Selector**, responsible for extracting relevant information across diverse tables and columns, and the generator (consisting of the **Direct Generator** and **MCTS-Refiner**), a hybrid module inspired by the principles of fast and slow thinking. The Generator integrates a direct SQL generation component with an MCTS-based Refiner. In cases where the direct SQL generation fails to address the query effectively, the MCTS-based Refiner performs iterative.

demonstrates its benefits in multi-agent path search, highlighting the advantages over traditional heuristic search methods. Similarly, [Li *et al.*, 2023] use MCTS to effectively address various types of SAT problems. Recently, combining MCTS with large-scale language models has been a great trend. [Chen *et al.*, 2024] proposed IMCTS, an approach designed to enhance the mathematical reasoning capabilities of fine-tuned LLMs. [Xu, 2023] integrated MCTS with a lightweight energy function, demonstrating notable performance improvements. In addition, MCTSr[Di Zhang *et al.*, 2024] introduced systematic exploration and heuristic self-refinement mechanisms, further advancing its applications in complex decision-making tasks.

Extensive experiments across various downstream applications have consistently validated the efficacy of MCTS in addressing complex planning problems. Building on this success, our work pioneers the application of MCTS in the Text-to-SQL domain for the first time. Our approach incorporates a novel ”fast-and-slow thinking” mechanism, comprising a direct SQL generation module and an MCTS-based refinement module. When direct generation fails, the MCTS module critiques and refines SQL outputs by identifying and addressing errors, iteratively exploring optimal solutions under constrained conditions, thereby reducing hallucination effects caused by the model’s autonomous planning.

3 Preliminaries

This section provides the necessary preliminaries and notations for understanding the proposed framework. We start with a concise overview of the Text-to-SQL task, highlighting its relevance to our approach, and then introduce the fundamental concepts of Monte Carlo Tree Search.

3.1 Definition of Text-to-SQL

Let $\mathcal{X} = (Q, S, \mathcal{K})$ represent the input to the Text-to-SQL task, where Q is a natural language question, S is the database schema, and $\mathcal{K}$ optionally incorporates external knowledge to enhance contextual understanding. The database schema S consists of a set of tables $\mathcal{T} = T_1, T_2, \ldots, T_{|\mathcal{T}|}$ and their associated columns $\mathcal{C} = C_1, C_2, \ldots, C_{|\mathcal{C}|}$. Each column C_i is defined within a specific table T_j, providing structural and semantic information crucial for SQL generation. The goal of the Text-to-SQL task is to generate an SQL query $\mathcal{Y}$ that precisely captures the semantic intent of the natural language question Q while adhering to the structural constraints of the schema S.

3.2 Review of MCTS

Monte Carlo Tree Search is a fundamental tool widely used in game AI and complex planning making. It builds search trees, estimates action values through simulation, and solves complex problems step-by-step through continuous exploration. The algorithm relies on the Upper Confidence bounds for

Trees(UCT) to optimize decision efficiency and consists of four key phases: selection, expansion, simulation, and back-propagation.

In the selection phase, the algorithm starts at the root and iteratively selects child nodes based on the UCT until reaching a leaf node. During expansion, if the leaf node is non-terminal, one or more child nodes are added to explore potential future actions. The simulation phase evaluates the newly added nodes by performing stochastic simulations to a terminal state. Finally, in backpropagation, the simulation results results are propagated back through the tree, updating the values of visited nodes and improving future decision-making.

- **Selection**: Traverse the tree from the root by selecting child nodes based on UCT until a leaf node is reached.
- **Expansion**: If the leaf node is not terminal, extend the tree by adding one or more child nodes to represent unexplored actions.
- **Simulation**: Run simulations from the newly added node to a terminal state to estimate its value.
- **Backpropagation**: Propagate simulation outcomes back through the visited nodes to update their values and refine the decision policy.

4 MCTS-SQL Framework

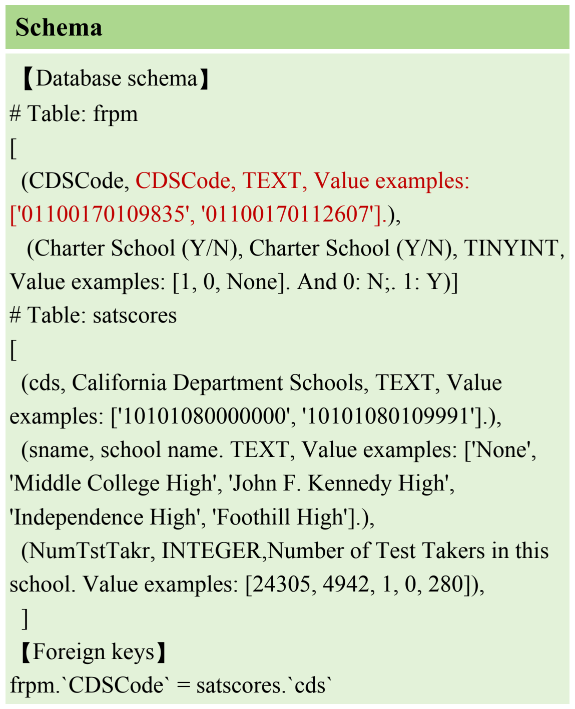
As shown in Figure 2, the MCTS-SQL framework consists of three key components: the Selector, Direct Generator, and MCTS-Refiner. The Selector identifies relevant tables and schema elements based on the user query, while the Direct Generator quickly produces an initial SQL query. Queries that fail or yield errors are refined by the MCTS-Refiner through iterative tree search. A detailed explanation of each component is provided in the subsequent section.

The collaboration process of our MCTS-SQL is presented in Algorithm 1:

Algorithm 1 The algorithm of MCTS-SQL

Input: query q, database schema db, knowledge kg
Output: SQL statement

```
1: db' = $LLM_{Selector}$(q, db, kg)
2: sql,err = $LLM_{DirectGenerator}$(q, db, kg)
3: ver = $LLM_{Verifier}$(sql,q,db,kg,)
4: if err is NULL and ver is ok then
5:     return sql
6: else
7:     count = 0
8:     while count $<$ maxRollout do
9:         select a node
10:        cri = $LLM_{Critiquer}$(sql, err, q, db, kg)
11:        ref = $LLM_{Refiner}$(sql, err, q, db, kg, cri)
12:        score = $LLM_{Evaluater}$(ref, err, q, db, kg)
13:        back-propagation
14:        update the UCT value
15:    end while
16:    sql = ref with best score
17:    return sql
18: end if
```


Schema

【Database schema】
# Table: frpm
[
(CDSCode, CDSCode, TEXT, Value examples: ['01100170109835', '01100170112607'].),
(Charter School (Y/N), Charter School (Y/N), TINYINT, Value examples: [1, 0, None]. And 0: N;. 1: Y)]
# Table: satscores
[
(cds, California Department Schools, TEXT, Value examples: ['10101080000000', '10101080109991'].),
(sname, school name. TEXT, Value examples: ['None', 'Middle College High', 'John F. Kennedy High', 'Independence High', 'Foothill High'].),
(NumTstTakr, INTEGER,Number of Test Takers in this school. Value examples: [24305, 4942, 1, 0, 280]),
]
【Foreign keys】
frpm.`CDSCode` = satscores.`cds`

Figure 3: An example of proposed semi-structed database schema format.The format consists of table names, descriptions and column level details(name, data type, description, and examples) to represent the hierarchical information of databeses.

4.1 Schema

Combining the database schema information in the prompt is essential for enabling the LLM to comprehend the database structure accurately and generate precise queries. This paper presents a novel method that illustrates the hierarchical relationships between databases, tables, and columns using a semi-structured format.

To be specific, we provide the table name and corresponding description for each table(which can be omitted if not necessary). The table information is converted into a list where each entry is a tuple containing a column of details. Each column includes the name, data type, description, and example values, thus providing a comprehensive view of its contents. In addition, foreign keys must be included to represent the relationships between tables accurately. Understanding hierarchical relationships is critical for query generation. An example of proposed schema format is shown in Figure 3.

4.2 Selector

The role of the Selector can be formally described as follows. Given an input triplet $\mathcal{X} = (Q, S, \mathcal{K})$, where Q is the query, $S = T, C$ is the database schema consisting of tables (T) and columns (C), and $\mathcal{K}$ denotes the knowledge provided. The Selector aims to identify a minimal subset of tables and columns, denoted as $S' = T', C'$, which are necessary to answer the query Q. The behavior of the Selector is formally defined as follows:

$$S' = f_{\text{Selector}}(Q, S, \mathcal{K} \mid \mathcal{M}) \tag{1}$$

Where $f_{\text{Selector}}(\cdot \mid \mathcal{M})$ represents the Selector's function, implemented via prompt engineering powered by a large language model $\mathcal{M}$.

The design of the Selector is motivated by two key considerations: (i) Including unnecessary schema elements in the prompt increases the risk of irrelevant or extraneous items being incorporated into the generated SQL, which can degrade output quality; (ii) Directly utilizing the entire database schema may result in excessively long prompts, which could lead to higher computational costs and potentially exceed the input length limitations of the language model.

4.3 Generator

To balance accuracy and efficiency in Text-to-SQL, we introduce a fast-slow thinking mechanism consisting of a Direct Generator and an MCTS-Refiner. Initially, the user's input is processed by the Direct Generator, which leverages COT in one prompt to generate SQL queries rapidly. If the executor successfully executes the SQL and the LLM confirms it meets the user's intent, the result is immediately returned. If an error occurs or the LLM thinks the output is insufficient, the MCTS-Refiner iterative optimization is based on the error feedback and self-criticism. Once reaching the maximum of iterations, the SQL with the highest evaluation value is selected and provided.

A. Direct Generator

The purpose of the Direct Generator is to generate SQL queries directly within a single step through an end-to-end process. It can be described as follows, where R represents the generated SQL query.

$$R = f_{\text{Direct Generator}}(Q, S', \mathcal{K} \mid \mathcal{M}) \tag{2}$$

After the SQL is generated, it follows two steps of evaluation. First, an executor checks its syntactic correctness and successful execution. Then, an LLM verifies if the SQL meets the user's requirements. The LLM-based verifier can be formalized as:

$$V = f_{\text{Verifier}}(R, Q, S', K \mid \mathcal{M}) \tag{3}$$

Specifically, the Direct Generator employs chain-of-thought (COT) prompting, which is proven effective during SQL generation. The Selector provides a selected subset of tables and columns to minimize confusing information during the generation. And the user's query is added to the same prompt. The LLM processes this input to generate SQL queries, accompanied by a detailed rationale.

Additionally, we employ a few-shot learning strategy, using several in-context examples to improve the LLM's understanding of task-specific instructions and enhance its generalization capabilities.

B. MCTS-Refiner

The MCTS-Refiner aims to refine SQL queries using execution errors, and the self-critique mechanism is adopted to optimize the query iteratively. The main workflow of the proposed method consists of several stages, detailed as follows:

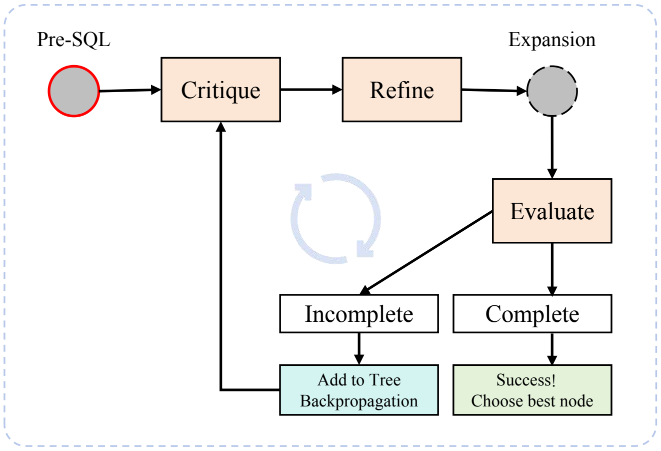


Figure 4: The main workflow of our proposed MCTS-refiner. The SQL generated in the last step is firstly get a critique. Then, based on the critique, a refinement is provided. The search tree is now expanded.If the iteration is complete, the node with best score is selected as the final output, otherwise, the node will be added to the search tree and backpropagation.

Initialization: The root node is initialized with the suboptimal SQL generated by the Direct Generator as a reference for step-by-step optimization to reduce the complexity of the search process.

Selection: Following the existing practices, we define a function P to rank all generated SQL queries that are not fully expanded. The node with the highest value is selected for further refinement. The function P of a node a can be defined as follows, where r_a represents the set of results associated with node a.

$$P(a) = \frac{1}{2}\left(\min r_a + \frac{1}{|r_a|}\sum_{i=1}^{|r_a|} r_a^i\right) \tag{4}$$

Self-Refine: The SQL query a is initially executed by the executor to get the error information E_a, which is then used to refine the query through the self-refine framework. In this process, the LLM generates a critique c, serving as the guidance for refining the query and producing an improved SQL query a'. Specifically, E_a represents the error details related to the initial SQL query a, and I denotes the prompt used in the Direct Generator, which includes the input query Q, the database schema S', and relevant knowledge K. The details can be formally described as follows:

$$c = f_{\text{Critiquer}}(a, E_a, I \mid \mathcal{M}) \tag{5}$$

$$a' = f_{\text{Refiner}}(a, c, E_a, I \mid \mathcal{M}) \tag{6}$$

The Self-refine module designed a refinement mechanism using error feedback and critique generation to enhance the accuracy and robustness of SQL queries.

Self-Evaluation: The refined SQL query is evaluated to obtain a reward value, denoted as r, and its corresponding P-value is computed. To be specific, we proposed a model-based self-reward feedback mechanism, with the reward value constrained within the range of -95 to 95. To ensure the reliability and fairness, the highest scores are deliberately suppressed. The reward r is formally defined as:

$$r_a = f_{\text{Evaluater}}(a', E_{a'}, I \mid \mathcal{M}) \tag{7}$$

Backpropagation: The value r of the refined SQL query is backpropagated through the search tree, updating the value information of the parent node and other relevant nodes. If the P-value of any child node is changed, the corresponding P-value of its parent node is recalculated accordingly. The process can be described as follows:

$$P'(a) = \frac{1}{2}\left(P(a) + \max_{i \in \text{Children}(a)} P(i)\right) \tag{8}$$

UCT update: After updating the P values for all nodes, we choose the *UCT* function to measure the combined value of each node, which is used as an important basis for expansion in the next selection stage. The *UCT* value of a node a is formally defined as:

$$UCT_a = P(a) + c\sqrt{\frac{\ln N(\text{Father}(a)) + 1}{N(a) + \epsilon}} \tag{9}$$

In this formulation, $N(.)$ denotes the total number of visits to a given node, and c is a constant that balances the trade-off between P-value and visit times. The term ϵ is a small constant to prevent division by zero.

The algorithm proceeds through all these steps iteratively until the maximum rollout numbers are reached. And the SQL querie with the highest score r is chosen as the final output.

5 Experiments

To evaluate the performance of our MCTS-SQL, we present the implementation details, explain the experiments performed, and offer a thorough analysis of the results.

5.1 Datasets

We evaluate our MCTS-SQL framework using two NL2SQL benchmarks: Spider and BIRD. The Spider contains 7000 training question-query pairs and 1038 development pairs, which are from 200 different databases across 138 domains. In this study, we only concentrate on the development set. In contrast, BIRD includes 95 large-scale databases with high-quality Text-to-SQL pairs, covering 37 specialized domains. Unlike Spider, BIRD focuses on real-world database content and provides external knowledge reasoning to bridge natural language queries and database contextual information.

5.2 Evaluation Metrics

To evaluate our proposed method's performance, we use two metrics: Execution Accuracy(EX) and Valid Efficiency Score(VES). The Execution Accuracy(EX) calculates the percentage of queries where the predicted SQL queries match the correct SQL queries when executed. Valid Efficiency Score(VES) measures the percentage of predicted SQL queries that output sets consisting of the results from the ground-truth SQL queries.

5.3 Base Models

In this paper, we utilized the latest models, GPT-4o-mini and GPt-4o, as the base models for our conducted experiments. Most of our ablation studies experiments are carried out with GPT-4o-mini, which offers a nearly 30% cost reduction compared to GPT-4o. Despite this, GPT-4o-mini proved to be more effective, offering a powerful balance between performance and cost. In future works, we will explore other models and fine-tune an open-source LLM.

5.4 Hyper-parameters

In order to ensure the stability of our experiment results, we standardized the hyper-parameters as follows. The temperature is fixed at 0.0, the top-p parameter is set to 1.0, and the max-token length is 32168. As for the hyper-parameters in the MCTS-Refiners, the child nodes of a node are set to 2, and the max-rollout numbers are 5.

5.5 Results

A. BIRD Results

In Table 1, we present the comparison of our method's performance on the BIRD against existing approaches. Our proposed MCTS-SQL achieves the highest performance with **69.40%** for dev EX and **66.24%** for dev VES, demonstrating its superiority over other methods. Even when using the more cost-effective model GPT-4o-mini, our method still performs well, with scores of 63.15% and 60.78%. This highlights the effectiveness of our approach, which can achieve competitive results even with a lighter model.

The detailed performance of different complexity levels is shown in Table 2. The results show a notable improvement. Based on the GPT-4o, the EX of the simple subset is **74.32%**, the moderate subset is **65.17%**, and a challenging subset is **51.48%**. Analyzing the results, we can find a great outperforming of challenging sets, mainly due to the MCTS-based refiner, which is an important aspect of handling real-world applications.

Method	**dev EX**	**text EX**	**dev VES**
Palm-2	27.38	33.04	-
ChatGPT+COT	36.64	40.08	42.30
DIN-SQL+GPT-4	50.72	55.09	58.79
DAIL-SQL+GPT-4	54.76	57.41	56.08
MAC-SQL+GPT-4	59.39	59.59	66.39
PB-SQL,v1	60.50	64.84	60.36
MCS-SQL+GPT-4	63.36	65.45	61.23
CHESS	65.00	66.69	62.77
ByteBrain	65.45	68.87	-
ASKData+GPT-4o	65.19	65.62	60.25
E-SQL+GPT-4o	65.58	66.29	62.43
Ours+GPT-4o-mini	63.15	61.39	60.78
Ours+GPT-4o	**69.40**	**68.91**	**66.24**

Table 1: Comparison with the results of existing methods on BIRD of the Execution accuracy and Valid Efficiency Score.

Method	Simp.	Mod.	Chall.	All
E-SQL+GPT-4o-mini	67.44	56.94	40.00	59.81
E-SQL+GPT-4o	73.02	64.14	48.07	66.29
Ours+GPT-4o-mini	68.56	57.76	45.83	**63.15**
Ours+GPT-4o	**74.32**	**65.17**	**51.48**	**69.40**

Table 2: Execution accuracy in BIRD development set.

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
MAC-SQL+GPT-4	86.75	82.80
CHESS	87.2	-
MCS-SQL+GPT-4	89.5	89.6
Ours+GPT-4o-mini	86.16	83.74
Ours+GPT-4o	**88.71**	**86.63**

Table 3: Execution accuracy on both dev and test set of spider

B. Spider Results

Table 3 shows the performance comparison on the Spider dataset. Our MCTS-SQL achieves excellent results with 88.71% on the development set and 86.63% on the test set. We can see that existing work has achieved outstanding performance; even in this situation, our work still performs remarkably well.

5.6 Ablation study

We conducted an ablation study to analyze the impact of three of our proposed components-Schema, Selector, and MCTS-Refiner. The results, shown in Table 4, highlight the importance of each module in the MCTS-SQL. To reduce costs, we use GPT-4o-mini as the base model. Specifically, removing the carefully designed Schema and using a normal DDL slightly reduces performance. The Selector is more important, especially on both moderate and challenging subsets. The most significant decrease occurs when the MCTS-Refiner is removed, demonstrating its critical role in solving challenging conditions. All these tricks boost the system's effectiveness in the NL2SQL task and can be transferred to other relevant problems.

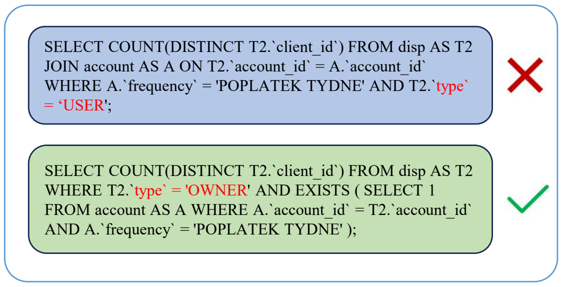
In this paper, we use the Monte Carlo Tree Search to enhance the model's performance, and the number of child nodes and max-rollouts are two hyper-parameters. Table 5

Pipeline	Simp.	Mod.	Chall.	All
Ours + GPT-4o-mini	**68.56**	**57.76**	**45.83**	**63.15**
w/o schema	66.98	56.27	43.82	61.56(↓)
w/o Selector	66.71	55.40	40.18	60.79(↓)
w/o MCTS-Refiner	64.46	52.00	35.44	57.96(↓)

Table 4: Ablation study using GPT-4o-mini with EX on the development set.

Pipeline	Simp.	Mod.	Chall.	All
Ours + GPT-4o-mini	**68.56**	**57.76**	**45.83**	**63.15**
max-rollouts-5	68.56	57.76	45.83	63.15
max-rollouts-6	68.14	57.81	44.91	62.83
max-rollouts-7	68.42	57.12	45.93	62.88
child nodes-2	68.56	57.76	45.83	63.15
chile nodes-3	68.62	57.57	45.83	63.13

Table 5: Ablation study of hyper-parameters in MCTS with EX on the development set.



```
SELECT COUNT(DISTINCT T2.`client_id`) FROM disp AS T2
JOIN account AS A ON T2.`account_id` = A.`account_id`
WHERE A.`frequency` = 'POPLATEK TYDNE' AND T2.`type`
= ‘USER';
```

```
SELECT COUNT(DISTINCT T2.`client_id`) FROM disp AS T2
WHERE T2.`type` = 'OWNER' AND EXISTS ( SELECT 1
FROM account AS A WHERE A.`account_id` = T2.`account_id`
AND A.`frequency` = 'POPLATEK TYDNE' );
```

Figure 5: The comparison of normal COT and our MCTS-SQL in solving the same query. The analysis shows that the COT struggle with detailed information and complex constrains due to LLM's hallucinations

shows the specific results of the ablation study. We find that variations of these settings have minimal effect on the overall performance. For instance, the performance across different max-rollouts and child nodes shows a few changes. Therefore, to lower token consumption, we choose the most efficient settings.

6 Discussion

Figure 5 shows an example of normal COT and MCTS-SQL to solve the same user's query. Through the error analysis, we can find that the syntax and structure generated by the COT are generally correct, but due to the hallucination of LLM, it struggles with detailed information and complex constraints. The feedback and iterative optimization mechanism of MCTS helps reduce errors caused by these details, thus improving the performance of SQL generation.

7 Conclusion

In conclusion, this paper introduces MCTS-SQL, a novel framework that utilizes the Monte Carlo Tree Search to enhance the LLMs to address challenges in Text-to-SQL tasks. Experimental results from the Spider and BIRD demonstrate the significant advantages of our method, achieving state-of-the-art performance, especially in challenging queries. Specifically,we achieve an EX **69.40%** on the BIRD with **51.48%** on the challenging set, outperforming existing method by **3.41%**. These results show the potential of MCTS-SQL for real-world database applications. However, the MCTS uses multiple iterations which inevitable increases token consumption and time. In the future work, we will explore some methods to solve the constrain.