Enhancing Text-to-SQL Capabilities of Large Language Models through Tailored Promptings

Anonymous submission

Abstract

Large language models (LLMs) with prompting have achieved encouraging results on many natural language processing (NLP) tasks based on task-tailored promptings. Text-to-SQL is a critical task that generates SQL queries from natural language questions. However, prompting on LLMs haven't show superior performance on Text-to-SQL task due to the absence of tailored promptings. In this work, we propose three promptings specifically designed for Text-to-SQL: SL-prompt, CC-prompt, and SL+CC prompt. SL-prompt is designed to guide LLMs to identify relevant tables; CC-prompt directs LLMs to generate SQL clause by clause; and SL+CC prompt is proposed to combine the strengths of these above promptings. The three prompting strategies makes three solutions for Text-to-SQL. Then, another prompting strategy, the RS-prompt is proposed to direct LLMs to select the best answer from the results of the solutions. We conducted extensive experiments, and experimental results show that our method achieved an execution accuracy of 86.2% and a test-suite accuracy of 76.9% on the Spider dataset, which is 1.1%, and 2.7% higher than the current state-of-the-art Text-to-SQL methods, respectively. The results confirmed that the proposed promptings enhanced the capabilities of LLMs on Text-to-SQL. Experimental results also show that the granularity of schema linking and the order of clause generation have great impact on the performance, which are considered little in previous research.

Keywords: Text-to-SQL, Prompt, Large Language Models, Schema Linking, SQL Generation

1. Introduction

Text-to-SQL is a task in natural language processing (NLP) that aims to automatically generate structured query language (SQL) queries from natural language text. This task enables users to access databases without requiring SQL knowledge or familiarity with the database schema, thus facilitating the work of data analysts and software developers who need to write complex SQL queries. Text-to-SQL has attracted significant interest from both industry and academia in recent years (Wang et al., 2020; Choi et al., 2021; Zhao et al., 2022).

With the rapid progress of Large Language Models (LLMs), the research areas of NLP are being revolutionized (Zhao et al., 2023). LLMs can now serve as a general-purpose language task solver (to some extent), and they have shown impressive performance in a series of NLP tasks, e.g., arithmetics, symbolic reasoning (Kojima et al., 2022), disambiguation QA, movie recommendation, etc. (Suzgun et al., 2022).

A new direction in Text-to-SQL tasks involves the use of LLMs. (Rajkumar et al., 2022; Liu et al., 2023) evaluated the capabilities of Text-to-SQL on LLMs using zero-shot and few-shot prompting and found that these results are still inferior to well-designed and fine-tuned models. One important reason is that Text-to-SQL is a complex task involving multiple domains such as semantic alignment, understanding of structured data, and code generation. For example, to compose a correct SQL query from a natural language question, one should first identify the relevant tables and columns based on the semantics of the question, infer the necessary components, e.g., clauses and subqueries, and decide on the proper keywords and expressions in each clause, and so on.

An effective strategy for LLMs to solve complex tasks is *decomposition*, i.e. decomposing a complex task into several simpler subtasks, and then guide the LLMs to solve the subtasks sequentially (Zhou et al., 2022; Kojima et al., 2022). Recently, DIN-SQL (Pourreza and Rafiei, 2023) is proposed for Text-to-SQL, which decomposed Text-to-SQL tasks into four subtasks: schema linking, classification, SQL generation, and self-correction, and then used the Chain of Thought prompting (CoT) (Kojima et al., 2022) to solve these subtasks. Their methods demonstrated that decomposition is effective and achieved promising results. However, DIN-SQL lacks deep analysis and investigation of subtasks, and the solutions to subtasks are sub-optimal. For example, the schema linking subtask of DIN-SQL aims to identify the related column names (e.g., *Player.player_name*), which is very challenging and it is very difficult to accurately link columns to keywords in questions. Instead, we found that a more feasible approach is to identify broader schema information, e.g. relevant tables (*Table Player, columns = [*, id, player_name, birthday, ...]*). This approach reduces the difficulty of schema linking and recalls necessary schema information, and is found to be more effective and generates better results. Moreover, the pipeline approach of DIN-SQL is prone to error propagation and lacks robustness.

In this work, we propose three tailored prompting for Text-to-SQL. Inspired by the two main challenges of Text-to-SQL (Shi et al., 2021), i.e.

LREC-COLING 2024, pages 6091–6109

schema linking and SQL generation, we designed the **SL-prompt** and **CC-prompt**. **SL-prompt** is designed for **S**chema **L**inking, which identifies the relevant schema information. **CC-prompt** is proposed to induce LLMs generating SQL **C**lasue by **C**lause. **SL+CC-prompt** is a combination of SL-prompt and CC-prompt, which utilize the relevant schema information obtained by SL-prompt to generation SQL clause by clause. We analyzed the granularity of the relevant schema information identified by SL-prompt and the order in which CC-prompt clauses are generated. The three prompting strategies makes three solutions for Text-to-SQL. We found that, the results of these solutions exhibit a surprising coverage of correct answers. Then, another prompting strategy, the **RS-prompt** (Result Selection prompt) is proposed to direct LLMs to select the best answer from the results of the solutions.

Through experiments, we demonstrate that the proposed prompting strategies are effective, resulting in an execution accuracy score of **86.2** on the Spider dataset (Yu et al., 2018), which is **1.1** points higher than the current state-of-the-art system.

In summary, we make the following contributions in this work:

- We proposed three promptings tailored for the Text-to-SQL tasks: SL-prompt, CC-prompt and SL+CC prompt. For SL-prompt, we analyze the granularity of schema information; in CC-prompt and SL+CC prompt, we explore the order of clause generation.
- We propose to combine the three prompting-based methods, and select final answers from the results obtained from the methods. The results of the three methods exhibit high diversity, and cover a large range of potentially correct answers.
- We conducted extensive performance studies. Experimental results on Spider dataset show that our proposed method outperforms state-of-the-art Text-to-SQL methods.

2. Methods

Text-to-SQL is a task that maps a natural language question to a SQL query given a database schema. As natural language question and SQL query are different in syntax and structure, Text-to-SQL involves a complex process.

LLMs can solve various NLP tasks by a simple prompt without illustrative examples (i.e., zero-shot learning) or by simply conditioning them on a few illustrative examples (i.e., few-shot learning). In few-shot learning with LLMs, the generation is often conditioned on a fixed set of m exemplars, $\{(x_i, y_i)\}_{i<m}$. Thus, the few-shot Text-to-SQL generation with LLMs can be formulated as follows:

$$P_{\text{LLMs}}(y \mid x) = P\left(y \mid \text{prompt}\left(x, \{(x_i, y_i)\}_{i<m}\right)\right), \quad (1)$$

where x is the test input, y is the test output, x_i and y_i make up a *demonstration*, and $\text{prompt}\left(x, \{(x_i, y_i)\}_{i<m}\right)$ is a string representation of overall input.

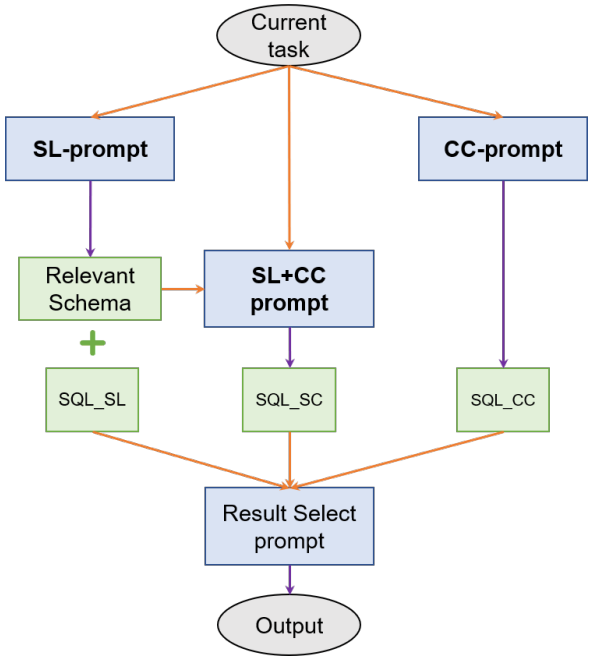

Figure 1: An overview of the method of this paper. The orange edges represent the process of constructing prompting, and the purple edges represent the process of generating the output of LLMs, the blue nodes represent our promptings, and the green nodes represent the output of LLMs.

2.1. Overview

In this work, we propose three promptings tailored for Text-to-SQL: SL-prompt, CC-prompt, and SL+CC prompt, each leading to a specific solution to Text-to-SQL. As shown in Figure 1, SL-prompt and CC-prompt generate SQL separately. SL+CC prompt combines SL-prompt and CC-prompt: it gets the relevant schema using SL-prompt and generates SQL using CC-prompt. We find that none of the methods exhibit an overwhelming advantage over others. Therefore, we propose Result Select prompt (RS-prompt) to induce LLMs to select the best answer from the candidate SQLs generated by the three methods.

2.2. Prompting for Schema Linking

Schema linking is responsible for recognizing schema information (e.g., tables, table names,

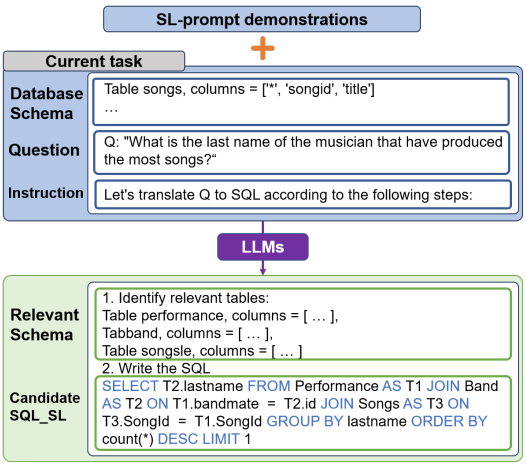

Figure 2: An example of SL-prompt.

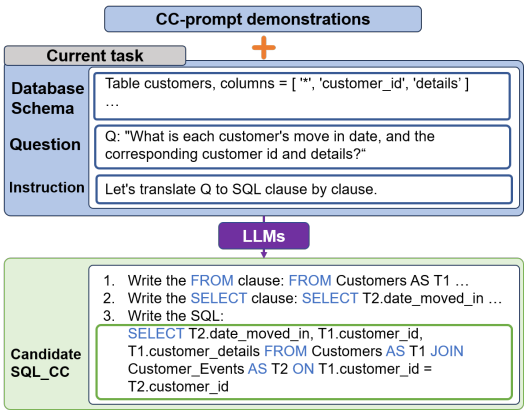

Figure 3: An example of CC-prompt.

columns) mentioned in natural language questions. It is considered a vital component for Text-to-SQL and has been studied in several works (Lei et al., 2020; Wang et al., 2020; Li et al., 2023b,a).

This paper proposes **SL-prompt**, a prompt tailored for schema linking. Figure 2 shows an example of SL-prompt. The inputs include a set of task-specific demonstrations, a task-specific instruction, the current question, and the database schema. The SL-prompt focuses on the schema linking issue. It guides LLMs to first identify the relevant schema information, then generate the SQL using the relevant tables.

Notably, schema linking can be conducted at different granularities of database schema. In this paper, we conducted experiments at three granularities, and the experimental results demonstrate that it is more favorable to identify the coarse-grained schema items, e.g. relevant tables, instead of the fine-grained schema items, e.g. relevant columns. That is because recognizing the columns mentioned in the question is very challenging, and the identified columns may recall only part of the relevant columns. As a result, generating SQL based on the false columns will inevitably lead to wrong answers. Our experiments also verified that recognizing relevant tables is more proper than recognizing relevant columns. More discussion can be found in §4.2.

2.3. Prompting for SQL Generation

Due to the mismatch between natural language questions and the corresponding SQL queries, SQL generation poses a significant challenge for Text-to-SQL tasks. In previous research, a tree-structured decoder has been used to reduce errors when generating SQL (Guo et al., 2019; Wang et al., 2020; Scholak et al., 2021). However, LLMs as general-purpose models typically do not incorporate these decoders.

This work proposes a prompting-based approach, called **CC-prompt**, to generate SQL queries. As shown in Figure 3, the input of **CC-prompt** contains a set of task-specific demonstrations, a task-specific instruction, the current question, and the database schema. As generating SQL is nontrivial, we divide the task of SQL generation into several steps, and generate a clause in each step. The idea of CoT (Kojima et al., 2022) is employed in this approach to guide LLMs generate the clauses.

There are different orders of generating SQL clauses. For example, one possible order is to generate the SELECT clause first, followed by the FROM clause, and then the others. However, we find that generating SQL in this order is suboptimal. In this work, we generate SQL in another order: the FROM clause is generated first, the SELECT clause is generated last, and other clauses are generated in between. We will compare different orders in experiments in §4.3.

2.4. Combining Promptings

We combine the SL-prompt and CC-prompt to get **SL+CC prompt**. As presented in Figure 4, the **SL+CC prompt** is slightly different in the input with CC-prompt: the relevant tables inferred by SL-prompt are incorporated. Like CC-prompt, SL+CC prompt direct the model to generate a SQL query clause by clause.

2.5. Result Selection

The output of LLMs is random in nature. To mitigate this phenomenon, one possible approach is to generate multiple answers first and then select the best one from them (Shi et al., 2022; Zhang et al., 2022; Ni et al., 2023).

Inspired by Self-Consistency (Wang et al., 2022), we designed *Result Selection* prompt (RS-prompt), which selects the best one from multiple candidate

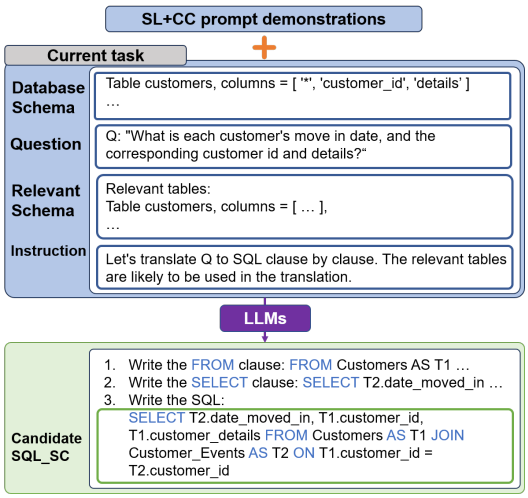

Figure 4: An example of SL+CC prompt.

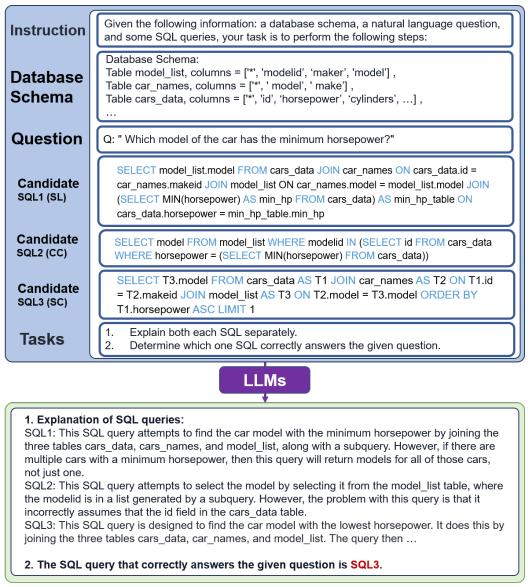

Figure 5: An example of Result Select prompt.

SQLs. As shown in Figure 5, the method is fed with a database schema, a question, the candidate SQLs generated before, and specific instructions (including task).

Figure 5 provides an example of RS-prompt, where the question is *"Which model of the car has the minimum horsepower?"*. In this case, the SL-prompt accurately identifies the relevant tables *(table model_list, table car_names, and table cars_data)* but makes an mistake in the SQL structure; the CC-prompt fails to find out the relevant tables *(missing table car_name)*; the SL+CC prompt complete this task correctly. The RS-prompt correctly explains the meaning of candidate SQLs, and select the best result accordingly.

Interestingly, we found that SL+CC prompt does not always outperform the SL-prompt and CC-prompt, which will be further discussed in §4.1.

3. Experiments

In this section, we conduct experimental studies on the proposed methods.

3.1. Experimental Setup

Models. In the experiments, we used two powerful and publicly accessible LLMs: GPT-3.5-Turbo(GPT-3.5) and GPT-4. We perform greedy decoding at temperature $\tau = 0.3$ to ensure stable output.

Datasets. We use four public Text-to-SQL benchmark datasets, they are: **(1) Spider** (Yu et al., 2018) is a large-scale cross-domain Text-to-SQL benchmark. It contains 8659 training samples across 146 databases and 1034 development samples across 20 databases. **(2) Spider-SYN** (Gan et al., 2021a) is a challenging variant of Spider, which modifies the questions from Spider by replacing the schema-related words with the corresponding synonyms. Spider-Syn is composed of 7000 training instances and 1034 development instances. **(3) Spider-DK** (Gan et al., 2021b) is another challenging variant of the Spider development set, which is constructed by adding domain knowledge that reflects real-world question paraphrases to some questions from the Spider development set. **(4) Spider-Realistic** (Deng et al., 2021) is a new evaluation set based on the Spider development set with explicit mentions of column names removed. It contains 508 samples.

Evaluation Metrics. We use Execution Accuracy(EX) as the evaluation metric for all experiments, which measures the percentage of system predictions leading to the gold execution result. We also adopt test-suite accuracy (TS) (Zhong et al., 2020) as an evaluation metric. TS could achieve high code coverage from a distilled test suite of the database, and it is also based on execution results.

Baselines. We compare the proposed methods with the following baselines:

(1) *Few-shot + CodeX* (Rajkumar et al., 2022), which has a set of demonstrations and runs on the Codex model.

(2) *Zero-shot + ChatGPT* (Liu et al., 2023), which works on ChatGPT under a zero-shot setting.

(3) *Coder-Reviewer* (Zhang et al., 2022), which generates and selects SQL queries based on their likelihood.

(4) *MBR-Exec* (Shi et al., 2022), which generate and selects SQLs with the most common execution result.

(5) *PICARD* (Scholak et al., 2021), which constrains auto-regressive coders of language models

Methods	EX	TS
Few-shot + CodeX (Rajkumar et al., 2022)	67.0	55.1
Zero-shot + ChatGPT (Liu et al., 2023)	70.1	60.1
Coder-Reviewer + CodeX(Zhang et al., 2022)	74.5	-
MBR-Exec (Shi et al., 2021)	75.2	-
T5-3B + PICARD (Scholak et al., 2021)	79.3	69.4
RASAT + PICARD (Li et al., 2023b)	80.5	70.3
LEVER + CodeX (Ni et al., 2023)	81.9	-
RESDSQL-3B + NatSQL (Li et al., 2023a)	84.1	73.5
Self-Debug + CodeX (Chen et al., 2023)	84.1	-
SPDS + CodeX (Nan et al., 2023)	84.4	-
DIN-SQL + GPT-4 (Pourreza and Rafiei, 2023)	**85.1**	**74.2**
Ours (GPT-3.5)	78.6	68.3
Ours (GPT-4)	**86.2**	**76.9**

Table 1: Execution accuracy (EX) and test-suite accuracy (TS) on the Spider development set.

through incremental parsing.

(6) *RASAT* (Qi et al., 2022), which introduces relation-aware self-attention into transformer models and utilizes constrained auto-regressive decoders.

(7) *LEVER* (Ni et al., 2023), which generates SQL queries with Codex and selects the one with highest scores.

(8) *RESDSQL* (Li et al., 2023a), a ranking-enhanced encoding and skeleton-aware decoding framework.

(9) *Self-Debug* (Chen et al., 2023), which prompts LLMs to debug SQL with explanations.

(10) *SPDS* (Nan et al., 2023), which selects few-shot demonstrations in terms of diversity and similarity.

(11) *DIN-SQL* (Pourreza and Rafiei, 2023), which decomposes text-to-SQL tasks into four subtasks and prompting LLMs to solve them sequentially.

3.2. Experimental Results

Table 1 shows the performance of our method and other baselines on the Spider development set. It can be seen that our method with GPT-4 model achieves the best performance on this benchmark. Previous researches have shown that LLMs may struggle with certain Text-to-SQL tasks (Rajkumar et al., 2022; Liu et al., 2023), but through our study, we can conclude that, by designing effective promptings, the capabilities of LLMs on Text-to-SQL can be enhanced to achieve competitive performance.

Table 2 shows the result of our method on Spider-DK, Spider-SYN, and Spider-Realistic benchmarks. We chose PICARD (Scholak et al., 2021), RASAT (Wang et al., 2020), RESDSQL (Li et al., 2023a), and ChatGPT (Liu et al., 2023) as baselines. The experimental results show that our method exhibits effectiveness across various benchmarks.

Methods	Spider-DK		Spider-Syn		Spider-Realistic	
	EX	TS	EX	TS	EX	TS
T5-3B + PICARD (Scholak et al., 2021)	62.5	-	69.8	61.8	71.4	61.7
RASAT + PICARD (Wang et al., 2020)	63.9	-	70.7	62.4	71.9	62.6
RESDSQL-3B + NatSQL (Li et al., 2023a)	**66.0**	-	**76.9**	**66.8**	**81.9**	**70.1**
Zeroshot + ChatGPT (Liu et al., 2023)	62.6	-	58.6	48.5	63.4	49.2
Ours (GPT-3.5)	63.9	-	67.1	57.6	70.7	58.3
Ours (GPT-4)	**67.2**	-	**78.1**	**68.6**	**82.8**	**70.6**

Table 2: Experimental results on Spider-DK, Spider-Syn, and Spider-Realistic.

Execution accuracy					
Methods	**Easy**	**Medium**	**Hard**	**Extra-hard**	**EX**
Few-shot + GPT-3.5	91.1	78.5	58.0	46.4	72.9
Ours (GPT-3.5)	**91.5**	**85.4**	**67.0(11.9 ↑)**	**53.6**	**78.6**
Few-shot + GPT-4	90.7	84.7	76.7	54.8	80.0
Ours (GPT-4)	**92.7**	**91.2**	**84.1**	**65.1(9.0 ↑)**	**86.2**
Test-suit accuracy					
Methods	**Easy**	**Medium**	**Hard**	**Extra-hard**	**TS**
Few-shot + GPT-3.5	90.3	67.6	42.6	26.4	62.3
Ours (GPT-3.5)	**90.7**	**77.3**	**52.8 (9.0 ↑)**	**27.1**	**68.3**
Few-shot + GPT-4	86.7	73.1	59.2	31.9	67.4
Ours (GPT-4)	**90.4**	**82.2**	**71.8**	**48.2(16.3 ↑)**	**76.9**

Table 3: Execution accuracy (EX) and Test-suit accuracy (TS) at different difficulty levels.

3.3. Results on Complex Queries

We evaluated our model on samples of different difficulty. According to the difficulty of generated SQL, The Spider dataset can be divided into 4 subsets: easy, medium, hard, and extra-hard (Yu et al., 2018). The performances on the four subsets are shown in Table 3. On GPT-3.5, our method improves the performance at all levels, with the largest improvement (11.9%) at the hard level. On GPT-4 model, our method also shows improvement at all levels, with the most remarkable improvement at extra-hard level (9.0%). From this set of experiments, we can see that our method are particularly helpful for complex Text-to-SQL tasks.

4. Analysis

4.1. Comparison of Methods

Table 4 shows the performance of the SL-prompt, CC-prompt, and SL+CC prompt at 4 difficulty levels on Spider. On the medium+ task, SL+CC prompt showed stronger performance, as we expected. However, on the easy level, SL-prompt performs the best. One possible reason is that the easy-level tasks mostly involve only two clauses (i.e., SELECT clause, FROM clause), and SL+CC prompt guides LLMs to explore as many clauses as possible through a set of demonstrations planned for all levels, which leads the LLMs to overcompli-

Methods	Easy	Medium	Hard	Extra-hard	EX
SL-prompt + GPT-3.5	**92.7**	79.3	68.8	48.2	75.7
CC-prompt + GPT-3.5	91.5	79.1	63.1	48.2	74.4
SL+CC prompt + GPT-3.5	88.7	**82.0**	**70.5**	**51.8**	**76.8**
SL-prompt + GPT-4	**96.0**	87.6	79.0	65.7	84.6
CC-prompt + GPT-4	93.1	86.5	78.4	59.0	82.3
SL+CC prompt + GPT-4	92.7	**89.4**	**80.7**	**66.9**	**85.1**

Table 4: Execution accuracy (EX) of SL-prompt, CC-prompt and SL+CC prompt at different difficulty levels.

SL-prompt	CC-prompt	SL+CC prompt	SUM
	GPT-3.5		
✓	✗	✗	21
✗	✓	✗	27
✗	✗	✓	29
✗	✗	✗	157
✓	✓	✓	669
	GPT-4		
✓	✗	✗	9
✗	✓	✗	6
✗	✗	✓	16
✗	✗	✗	111
✓	✓	✓	791

Table 5: Comparison of prompts. A ✓denotes a correctly answered instance, while a ✗indicates failure. The 'SUM' column records the total instances for each case in the 1034-instance Spider development set.

cate easy tasks and generates some unnecessary clauses. As Table 4 shows, SL+CC prompt is not always superior to SL-prompt and CC-prompt, even though SL+CC prompt combines the two prompts.

More detail can be found in Figure 6. We distribute the 1034 Spider development set instances uniformly on a plane with 1034 points. Each point represents an instance, and is color-coded: red if SL-prompt succeed on it, green for CC-prompt, blue for SL+CC-prompt, and gray if multiple prompts succeed on it. Panels (c) and (d) demonstrate that although the SL+CC prompt may outperform SL-prompt and CC-prompt, they have their own strengths.

Interestingly, the strengths and weaknesses of the SL+CC prompt are mostly consistent. For GPT-3.5 model, when comparing SL-prompt with SL+CC prompt, we find that SL-prompt correctly handled 5.4% of the samples that SL+CC prompt could not, while SL+CC prompt uniquely succeeded with 6.5% of the samples. Comparing CC-prompt and SL+CC prompt, we find that CC-prompt uniquely succeeded in 6.0% of the samples, whereas SL+CC prompt uniquely succeeded in 8.4% of the samples. This observation verified that LLMs are significantly influenced by prompting.

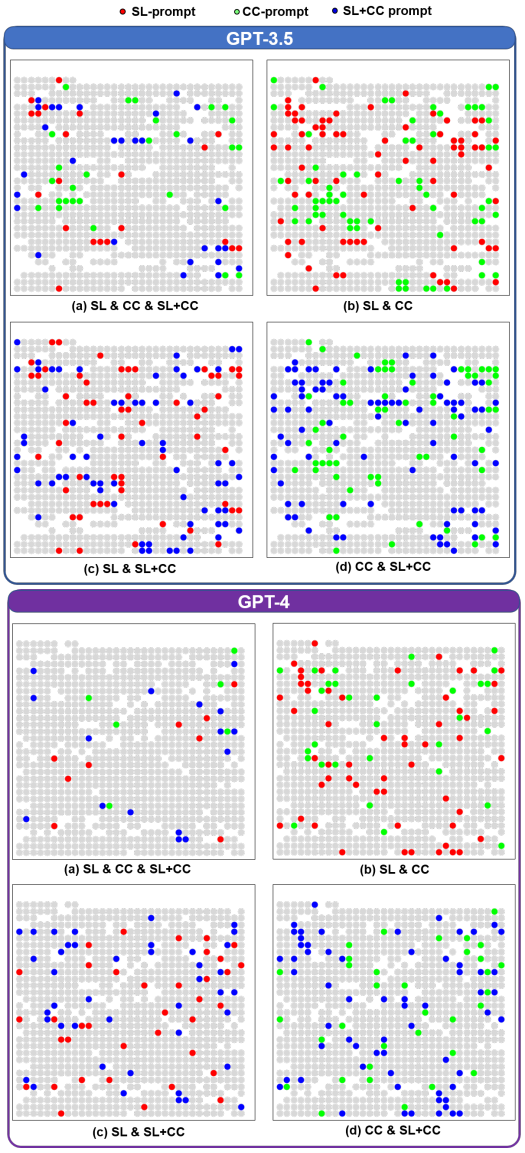

Figure 6: This figure depicts performance on the 1034-instance Spider development set with different prompts. Each dot signifies an instance, and color-coded by the prompt that correctly generated it. Grey dots represent instances correctly handled by multiple prompts.

With the diversity of results comes potential room for improvement. Specific values are given in

Table 5 shows the number of samples on which different prompts succeed. Out of 1034 instances, only 157 (15.2%) are not correctly tackled by any prompt, which means that 84.8% of the questions can be correctly answered by at least one of our prompts with GPT-3.5 model, which is very competitive. On the other hand, the EX rate of our method on GPT-3.5 is 78.6%, which means that 6.2% (84.8%-78.6%) of the samples are not predicted correctly, which means that there is still much room for improvement in the result selec-

	SL-prompt structure	EX	TS
(a)	1. Identify relevant **table names**: $t_a, t_b, \ldots$ 2. Identify relevant **column names**: $c_1^{t_a}, c_2^{t_a}, c_1^{t_b}, \ldots$ 3. Write SQL: *SELECT ...*	66.9	61.3
(b)	1. Identify relevant **tables**: $(t_a : c_1^{t_a}, \ldots, c_{\lvert\mathcal{C}\rvert}^{t_a}), (t_b : c_1^{t_b}, \ldots, c_{\lvert\mathcal{C}\rvert}^{t_b}), \ldots$ 2. Identify relevant **column names**: $c_1^{t_a}, c_2^{t_a}, c_1^{t_b}, \ldots$ 3. Write SQL: *SELECT ...*	70.9	65.0
(c)	1. Identify relevant **tables**: $(t_a : c_1^{t_a}, \ldots, c_{\lvert\mathcal{C}\rvert}^{t_a}), (t_b : c_1^{t_b}, \ldots, c_{\lvert\mathcal{C}\rvert}^{t_b}), \ldots$ 2. Write SQL: *SELECT ...*	**75.7**	**68.5**

Table 6: The performance of different structures of SL-prompt on the GPT-3.5 model.

	CC-prompt structure	EX	TS
(a)	1. Write the **SELECT** clause: *SELECT ...* 2. Write the **FROM** clause: *FROM ...* 3. Write other clauses: ... 4. Write SQL: *SELECT ...*	73.5	64.4
(b)	1. Write the **FROM** clause: *FROM ...* 2. Write the **SELECT** clause: *SELECT ...* 3. Write other clauses: ... 4. Write SQL: *SELECT ...*	73.7	64.6
(c)	1. Write the **FROM** clause: *FROM ...* 2. Write other clauses: ... 3. Write the **SELECT** clause: *SELECT ...* 4. Write SQL: *SELECT ...*	**74.4**	**65.1**

Table 7: The performance of different structures of CC-prompt the GPT-3.5 model.

tion phrase. However, for the GPT-4 model, the gap is relatively small (**3.1%**).

4.2. Structure of SL-prompt

We evaluated three different structures of SL-prompt: (1) SL-prompt(a) identifies relevant table names, and then relevant column names; (2) SL-prompt(b) recognizes related tables with their columns, and then relevant column names; (3) SL-prompt(c) detects only relevant tables with their columns.

The examples and experimental performance are presented in the Table 6.

One may think there is no difference between SL-prompt(a) and SL-prompt(b), but they actually make great difference in performance. That is probably because, in SL-prompt(a), the model didn't fully utilize the column information given in the context, although it seems obvious.

Contrary to intuition, the experimental results reveal that SL-prompt(c), which identifies only relevant tables, outperforms other structures trying to recognize relevant columns. That is because it is very challenging to infer the exact columns mentioned in the query; thus, the other two structures may miss some columns. When this erroneous set of columns is passed to the model, the model may generate false results. On the contrary, SL-prompt(c) does not filter columns, making a significantly high recall of relevant schema, and the model can reason out the correct columns when generating the results.

4.3. Structure of CC-prompt

It is observed that, in CC-prompt, the order of clause generation impacts the performance of models. Therefore, we design three CC-prompt structures with different generation orders, as shown in Table 7. CC-prompt(a) mimics the structure of a standard SQL query, while CC-prompt(b) positions the SELECT clause generation after the FROM clause, and CC-prompt(c) generates the SELECT clause at the end. The experimental results show that CC-prompt(c) surpasses other structures. This is explainable, as the SELECT clause corresponds to the projection operation, which is typically the last operation when composing a SQL. The order of SQL clause generation for SL+CC prompt is consistent with CC-prompt.

5. Related Work

Text-to-SQL aims to simplify the process of accessing data in relational databases for non-expert users. Researchers have made impressive achievements in this task by designing models (Wang et al., 2020; Cai et al., 2021; Li et al., 2023b; Qi et al., 2022; Li et al., 2023a) or fine-tuning pre-trained models (Yu et al., 2020; Shi et al., 2021; Scholak et al., 2021).

LLMs have demonstrated impressive code generation abilities without fine-tuning (Chen et al., 2021; Chowdhery et al., 2022; Zhao et al., 2022; Athiwaratkun et al., 2022). More researchers are studying the Text-to-SQL capabilities of LLMs. (Rajkumar et al., 2022; Liu et al., 2023) investigated the efficacy of Text-to-SQL on various LLMs, including GPT-3, Codex, and ChatGpt. They explored the impact of prompt structure, number of few-shot demonstrations, and other factors on the outcomes using zero-shot and few-shot prompting. Nonetheless, these studies' results are still inadequate compared to well-designed and fine-tuning models of the same period.

The rapid development of prompting-based methods has led to th e proposal of numerous effective prompting principles. For example, CoT prompting (Kojima et al., 2022) is proposed to improve LLMs' reasoning ability by producing intermediate steps before predicting a final answer; Self-Consistency (Wang et al., 2022) mitigates

the phenomenon of randomness in the output of LLMs by voting on the diversity of results and selecting the best one. For Text-to-SQL, these prompting enhancement methods are equally effective. Self-Debug(Chen et al., 2023) employing CoT prompting to obtain the question explanation and generates the initial SQL, then instruct LLMs to debug the SQL. Coder-Reviewer (Zhang et al., 2022), MBE-Exec (Shi et al., 2021) and LEVER (Ni et al., 2023) utilizing consistency principles to choose the optimal one from multiple candidate results. MBE-Exec (Shi et al., 2021) selects the SQL with the most common execution result, Coder-Reviewer (Zhang et al., 2022) selects the SQL considering both the likelihood of the predicted SQL given the question description (Coder score) and the likelihood of the question description given the predicted SQL (Reviewer score); LEVER (Ni et al., 2023) selects the SQL with the highest score, which represents the probability that the SQL is correct and is calculated based on the question, the SQL and the execution results. However, these approaches have limited improvement because they do not provide in-depth analysis of text-to-SQL characteristics and tailored solutions.

Tailored approaches to Text-to-SQL include studying prompting design strategies for Text-to-SQL (Nan et al., 2023) and the decomposition of Text-to-SQL tasks (Pourreza and Rafiei, 2023). (Nan et al., 2023) presented a study of Text-to-SQL prompt design strategies, explored the impact of demonstration selection, database schema representation, and other factors on results. Din-SQL reduces the overall difficulty of Text-to-SQL by dividing it into four subtasks; however, it lacks exploration of these key subtasks, and the pipelined approach limits the capacity of LLMs.

6. Conclusion

In this paper, we proposed three promptings tailored for Text-to-SQL tasks on LLMs: SL-prompt, CC-prompt, and SL+CC prompt. These prompts were designed to alleviate the two major concerns in Text-to-SQL tasks: schema linking and SQL generation. We also study the mechanism of our promptings interacting with LLMs. For the SL-prompt, we investigated the effect of the granularity of the relevant schema information. For the CC-prompt and SL+CC prompt, we evaluated the impact of the order of clause generation. Our experimental results demonstrated that our promptings significantly improved the performance of LLMs on Text-to-SQL tasks, achieving an EX of 86.2% and TS of 76.9%, outperforming the current state-of-the-art methods.

7. Bibliographical References

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.

Ruichu Cai, Jinjie Yuan, Boyan Xu, and Zhifeng Hao. 2021. Sadga: Structure-aware dual graph aggregation network for text-to-sql. *Advances in Neural Information Processing Systems*, 34:7664–7676.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. 2021. Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases. *Computational Linguistics*, 47(2):309–332.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

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

Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R Woodward, Jinxia Xie, and Pengsheng Huang. 2021a. Towards robustness of text-to-sql models against synonym substitution. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2505–2515.

Yujian Gan, Xinyun Chen, and Matthew Purver. 2021b. Exploring underexplored limitations of

cross-domain text-to-sql generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8926–8931.

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.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *ICML 2022 Workshop on Knowledge Retrieval and Language Models*.

Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. 2020. Re-examining the role of schema linking in text-to-sql. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6943–6954.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. *arXiv preprint arXiv:2302.05965*.

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 pre-trained transformers with graph-aware layers for text-to-sql parsing. *arXiv preprint arXiv:2301.07507*.

Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S Yu. 2023. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability. *arXiv preprint arXiv:2303.13547*.

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*.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. *arXiv preprint arXiv:2302.08468*.

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, 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*.

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

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*.

Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Cicero Nogueira dos Santos, and Bing Xiang. 2021. Learning contextual representations for semantic parsing with generation-augmented pre-training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 13806–13814.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. 2022. Challenging BIG-Bench Tasks and Whether Chain-of-Thought Can Solve Them. *arXiv preprint arXiv:2210.09261*.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Caiming Xiong, et al. 2020. Grappa: Grammar-augmented pre-training for table semantic parsing. In *International Conference on Learning Representations*.

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

Tianyi Zhang, Tao Yu, Tatsunori B Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I Wang. 2022. Coder reviewer reranking for code generation. *arXiv preprint arXiv:2211.16490*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Yiyun Zhao, Jiarong Jiang, Yiqun Hu, Wuwei Lan, Henghui Zhu, Anuj Chauhan, Alexander Hanbo Li, Lin Pan, Jun Wang, Chung-Wei Hang, et al. 2022. Importance of synthesizing high-quality data for text-to-sql parsing. In *NeurIPS 2022 Workshop on Synthetic Data for Empowering ML Research*.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-sql with distilled test suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 396–411.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

A. Prompts

A.1. SL-prompt(6-shot)

Table songs, columns = ['*', 'songid', 'title']
Table albums, columns = ['*', 'aid', 'title', 'year', 'label', 'type']
Table band, columns = ['*', 'id', 'firstname', 'lastname']
Table instruments, columns = ['*', 'songid', 'bandmateid', 'instrument']
Table performance, columns = ['*', 'songid', 'bandmate', 'stageposition']
Table track_lists, columns = ['*', 'albumid', 'position', 'songid']
Table vocals, columns = ['*', 'songid', 'bandmate', 'type']

Q: "What is the last name of the musician that have produced the most songs?"
Let's translate Q to SQL according to the following steps:
1. Identify relevant tables:
Table performance, columns = ['*', 'songid', 'bandmate', 'stageposition'],
Table band, columns = ['*', 'id', 'firstname', 'lastname'],
Table songs, columns = ['*', 'songid', 'title'];
2. Write the SQL:
SELECT band.lastname FROM performance
JOIN band ON performance.bandmate = band.id
JOIN songs ON songs.songid = performance.songid
GROUP BY lastname ORDER BY count(song.*) DESC LIMIT 1

Table addresses, columns = ['*', 'address_id', 'address_details']
Table locations, columns = ['*', 'location_id', 'other_details']
Table products, columns = ['*', 'product_id', 'product_type_code', 'product_name', 'product_price']
Table parties, columns = ['*', 'party_id', 'party_details']
Table assets, columns = ['*', 'asset_id', 'other_details']
Table channels, columns = ['*', 'channel_id', 'other_details']
Table finances, columns = ['*', 'finance_id', 'other_details']
Table events, columns = ['*', 'event_id', 'address_id', 'channel_id', 'event_type_code', 'finance_id', 'location_id']
Table products_in_events, columns = ['*', 'product_in_event_id', 'event_id', 'product_id']
Table parties_in_events, columns = ['*', 'party_id', 'event_id', 'role_code']
Table agreements, columns = ['*', 'document_id', 'event_id']
Table assets_in_events, columns = ['*', 'asset_id', 'event_id']

Q: "What are the names and type codes of products?"
Let's translate Q to SQL according to the following steps:
1. Identify relevant table:
Table products, columns = ['*', 'product_id', 'product_type_code', 'product_name', 'product_price'];
2. Write the SQL:
SELECT product_name, product_type_code FROM products

Table sqlitesequence, columns = ['*', 'id', 'name']
Table artists, columns = ['*', 'name', 'seq']
Table albums, columns = ['*', 'id', 'title', 'artist_id']
Table employees, columns = ['*', 'id', 'last_name', 'first_name', 'title', 'reports_to', 'birth_date', 'hire_date', 'address', 'city', 'state', 'country', 'postal_code', 'phone', 'fax', 'email']
Table customers, columns = ['*', 'id', 'first_name', 'last_name', 'company', 'address', 'city', 'state',

'country', 'postal_code', 'phone', 'fax', 'email', 'support_rep_id']
Table genres, columns = ['*', 'id', 'name']
Table invoices, columns = ['*', 'id', 'customer_id', 'invoice_date', 'billing_address', 'billing_city', 'billing_state', 'billing_country', 'billing_postal_code', 'total']
Table media_types, columns = ['*', 'id', 'name']
Table tracks, columns = ['*', 'id', 'name', 'album_id', 'media_type_id', 'genre_id', 'composer', 'milliseconds', 'bytes', 'unit_price']
Table invoice_lines, columns = ['*', 'id', 'invoice_id', 'track_id', 'unit_price', 'quantity']
Table playlists, columns = ['*', 'id', 'name']
Table playlist_tracks, columns = ['*', 'playlist_id', 'track_id']

Q: "What are the first and last names of the top 10 longest-serving employees?"
Let's translate Q to SQL according to the following steps:
1. Identify relevant table:
Table employees, columns = ['*', 'id', 'last_name', 'first_name', 'title', 'reports_to', 'birth_date', 'hire_date', 'address', 'city', 'state', 'country', 'postal_code', 'phone', 'fax', 'email'];
2. Write the SQL:
SELECT first_name, last_name FROM employees ORDER BY hire_date ASC LIMIT 10

Table customers, columns = ['*', 'customer_id', 'customer_first_name', 'customer_middle_initial', 'customer_last_name', 'gender', 'email_address', 'login_name', 'login_password', '
    phone_number',
'town_city', 'state_county_province', 'country']
Table orders, columns = ['*', 'order_id', 'customer_id', 'date_order_placed', 'order_details']
Table invoices, columns = ['*', 'invoice_number', 'order_id', 'invoice_date']
Table accounts, columns = ['*', 'account_id', 'customer_id', 'date_account_opened', 'account_name', 'other_account_details']
Table product_categories, columns = ['*', 'production_type_code', 'product_type_description', 'vat_rating']
Table products, columns = ['*', 'product_id', 'parent_product_id', 'production_type_code', 'unit_price', 'product_name', 'product_color', 'product_size']
Table financial_transactions, columns = ['*', 'transaction_id', 'account_id', 'invoice_number', 'transaction_type', 'transaction_date', 'transaction_amount', 'transaction_comment', 'other_transaction_details']
Table order_items, columns = ['*', 'order_item_id', 'order_id', 'product_id', 'product_quantity', 'other_order_item_details']
Table invoice_lineitems, columns = ['*', 'order_item_id', 'invoice_number', 'product_id', 'product_title', 'product_quantity', 'product_price', 'derived_product_cost', 'derived_vat_payable', 'derived_total_cost']

Q: "What is the customer last name, id and phone number with most number of orders?"
Let's translate Q to SQL according to the following steps:
1. Identify relevant tables:
Table orders, columns = ['*', 'order_id', 'customer_id', 'date_order_placed', 'order_details'],
Table customers, columns = ['*', 'customer_id', 'customer_first_name', 'customer_middle_initial', 'customer_last_name', 'gender', 'email_address', 'login_name', 'login_password', '
    phone_number',
'town_city', 'state_county_province', 'country'];
2. Write the SQL:
SELECT customers.customer_last_name, orders.customer_id, customers.phone_number
FROM orders JOIN customers ON orders.customer_id = customers.customer_id
GROUP BY orders.customer_id ORDER BY count(*) DESC LIMIT 1

Table routes, columns = ['*', 'rid', 'dst_apid', 'dst_ap', 'src_apid', 'src_ap', 'alid', 'airline', 'codeshare']
Table airports, columns = ['*', 'apid', 'name', 'city', 'country', 'x', 'y', 'elevation', 'iata', 'icao']
Table airlines, columns = ['*', 'alid', 'name', 'iata', 'icao', 'callsign', 'country', 'active']

Q: "What is the count of cities with more than 3 airports?"
Let's translate Q to SQL according to the following steps:
1. Identify relevant table:
Table airports, columns = ['*', 'apid', 'name', 'city', 'country', 'x', 'y', 'elevation', 'iata', 'icao'];
2. Write the SQL:
SELECT count (airports.*) FROM airports WHERE count (airports.*) > 3

Table class, columns = ['*', 'class_code', 'crs_code', 'class_section', 'class_time', 'class_room', 'prof_num']
Table course, columns = ['*', 'crs_code', 'dept_code', 'crs_description', 'crs_credit']
Table department, columns = ['*', 'dept_code', 'dept_name', 'school_code', 'emp_num', 'dept_address', 'dept_extension']
Table employee, columns = ['*', 'emp_num', 'emp_lname', 'emp_fname', 'emp_initial', 'emp_jobcode', 'emp_hiredate', 'emp_dob']
Table enroll, columns = ['*', 'class_code', 'stu_num', 'enroll_grade']
Table professor, columns = ['*', 'emp_num', 'dept_code', 'prof_office', 'prof_extension', 'prof_high_degree']
Table student, columns = ['*', 'stu_num', 'stu_lname', 'stu_fname', 'stu_init', 'stu_dob', 'stu_hrs', 'stu_class', 'stu_gpa', 'stu_transfer', 'dept_code', 'stu_phone', 'prof_num']

Q: "What is the name, address, and number of students in the departments that have the 3 most students?"
Let's translate Q to SQL according to the following steps:
1. Identify relevant tables:
Table student, columns = ['*', 'stu_num', 'stu_lname', 'stu_fname', 'stu_init', 'stu_dob', 'stu_hrs', 'stu_class', 'stu_gpa', 'stu_transfer', 'dept_code', 'stu_phone', 'prof_num'],
Table department, columns = ['*', 'dept_code', 'dept_name', 'school_code', 'emp_num', 'dept_address', 'dept_extension'];
2. Write the SQL:
SELECT department.dept_name, department.dept_address, count(student.*) FROM student GROUP BY student.dept_code ORDER BY count (student.*) DESC LIMIT 3

A.2. CC-prompt(6-shot)

Table customers, columns = ['*', 'customer_id', 'customer_details']
Table customer_events, columns = ['*', 'customer_event_id', 'customer_id', 'date_moved_in', 'property_id', 'resident_id', 'thing_id']
Table customer_event_notes, columns = ['*', 'customer_event_note_id', 'customer_event_id', 'service_type_code', 'resident_id', 'property_id', 'date_moved_in']

Q: "What is each customer's move in date, and the corresponding customer id and details?"
Let's translate Q to SQL clause by clause.
1. Write the FROM clause:
FROM Customers AS T1 JOIN Customer_Events AS T2 ON T1.customer_id = T2.customer_id
2. Write the SELECT clause:
SELECT T2.date_moved_in , T1.customer_id , T1.customer_details

3. Write the SQL:
SELECT T2.date_moved_in , T1.customer_id , T1.customer_details FROM Customers AS T1 JOIN Customer_Events AS T2 ON T1.customer_id = T2.customer_id

Table instructor, columns = ['*', 'id', 'name', 'dept_name', 'salary']
Table section, columns = ['*', 'course_id', 'sec_id', 'semester', 'year', 'building', 'room_number', 'time_slot_id']
Table teaches, columns = ['*', 'id', 'course_id', 'sec_id', 'semester', 'year']

Q: "Which year had the greatest number of courses?"
Let's translate Q to SQL clause by clause.
1. Write the FROM clause:
FROM SECTION
2. Write the GROUP BY clause:
GROUP BY YEAR
3. Write the ORDER BY clause:
ORDER BY count(*) DESC LIMIT 1
4. Write the SELECT clause:
SELECT YEAR
5. Write the SQL:
SELECT YEAR FROM SECTION GROUP BY YEAR ORDER BY count(*) DESC LIMIT 1

Table follows, columns = ['*', 'f1', 'f2']
Table tweets, columns = ['*', 'id', 'uid', 'text', 'createdate']
Table user_profiles, columns = ['*', 'uid', 'name', 'email', 'partitionid', 'followers']

Q: "Find the average number of followers for the users who do not have any tweet."
Let's translate Q to SQL clause by clause.
1. Write the FROM clause:
FROM user_profiles
2. Write the subquery:
SELECT UID FROM tweets
3. Write the WHERE clause:
WHERE UID NOT IN (SELECT UID FROM tweets)
4. Write the SELECT clause:
SELECT avg(followers)
5. Write the SQL:
SELECT avg(followers) FROM user_profiles
WHERE UID NOT IN (SELECT UID FROM tweets)

Table classroom, columns = ['*', 'building', 'room_number', 'capacity']
Table department, columns = ['*', 'dept_name', 'building', 'budget']
Table course, columns = ['*', 'course_id', 'title', 'dept_name', 'credits']

Let's translate Q to SQL clause by clause.
1. Write the FROM clause:
FROM classroom
2. Write the WHERE clause:
WHERE capacity > 50
3. Write the GROUP BY clause:

GROUP BY building

4. Write the SELECT clause:

SELECT count(*) , building

5. Write the SQL:

SELECT count(*) , building FROM classroom WHERE capacity > 50 GROUP BY building

Tables locations, columns = ['*', 'location_id', 'other_details']
Tables products, columns = ['*', 'product_id', 'product_type_code', 'product_name', 'product_price']
Tables events, columns = ['*', 'event_id', 'address_id', 'channel_id', 'event_type_code' , 'finance_id', 'location_id']
Tables products_in_events, columns = ['*', 'product_in_event_id', 'event_id', 'product_id']

Q: "Show the names of products that are in at least two events."
Let's translate Q to SQL clause by clause.

1. Write the FROM clause:

FROM Products AS T1 JOIN Products_in_Events AS T2 ON T1.Product_ID = T2.Product_ID

2. Write the GROUP BY clause:

GROUP BY T1.Product_Name HAVING COUNT(*) >= 2

3. Write the SELECT clause:

SELECT T1.Product_Name

4. Write the SQL:

SELECT T1.Product_Name
FROM Products AS T1 JOIN Products_in_Events AS T2 ON T1.Product_ID = T2.Product_ID
GROUP BY T1.Product_Name HAVING COUNT(*) >= 2

Tables products, columns =['*', 'product_id', 'parent_product_id', 'production_type_code', 'unit_price', 'product_name','product_color', 'product_size']
Tables financial_transactions, columns = ['*', 'transaction_id', 'account_id', 'invoice_number', 'transaction_type', 'transaction_date', 'transaction_amount', 'transaction_comment', 'other_transaction_details']
Tables order_items, columns = ['*', 'order_item_id', 'order_id', 'product_id', 'product_quantity', 'other_order_item_details']

Q: "What are the ids for transactions that have an amount greater than the average amount of a transaction?"
Let's translate Q to SQL clause by clause.

1. Write the FROM clause:

FROM Financial_transactions

2. Write the subquery:

SELECT avg(transaction_amount) FROM Financial_transactions

3. Write the WHERE clause:

WHERE transaction_amount >
(SELECT avg(transaction_amount) FROM Financial_transactions)

4. Write the SELECT:

SELECT transaction_id

5. Write the SQL:

SELECT transaction_id FROM Financial_transactions WHERE transaction_amount >
(SELECT avg(transaction_amount) FROM Financial_transactions)

A.3. SL+CC prompt(6-shot)

Table customers, columns = ['*', 'customer_id', 'customer_details']
Table customer_events, columns = ['*', 'customer_event_id', 'customer_id', 'date_moved_in', 'property_id', 'resident_id', 'thing_id']
Table customer_event_notes, columns = ['*', 'customer_event_note_id', 'customer_event_id', 'service_type_code', 'resident_id', 'property_id', 'date_moved_in']

Q: "What is each customer's move in date, and the corresponding customer id and details?"
Relevant tables:
Table customers, columns = ['*', 'customer_id', 'customer_details'],
Table customer_events, columns = ['*', 'customer_event_id', 'customer_id','date_moved_in', 'property_id', 'resident_id', 'thing_id']"
Let's translate Q to SQL clause by clause. The relevant tables are likely to be used in the translation.
1. Write the FROM clause:
FROM Customers AS T1 JOIN Customer_Events AS T2 ON T1.customer_id = T2.customer_id
2. Write the SELECT clause:
SELECT T2.date_moved_in , T1.customer_id , T1.customer_details
3. Write the SQL:
SELECT T2.date_moved_in , T1.customer_id , T1.customer_details FROM Customers AS T1 JOIN Customer_Events AS T2 ON T1.customer_id = T2.customer_id

Table instructor, columns = ['*', 'id', 'name', 'dept_name', 'salary']
Table section, columns = ['*', 'course_id', 'sec_id', 'semester', 'year', 'building', 'room_number', 'time_slot_id']
Table teaches, columns = ['*', 'id', 'course_id', 'sec_id', 'semester', 'year']

Q: "Which year had the greatest number of courses?"
Relevant table:
Table section, columns = ['*', 'course_id', 'sec_id', 'semester', 'year', 'building', 'room_number', 'time_slot_id']
Let's translate Q to SQL clause by clause. The relevant tables are likely to be used in the translation.
1. Write the FROM clause:
FROM SECTION
2. Write the GROUP BY clause:
GROUP BY YEAR
3. Write the ORDER BY clause:
ORDER BY count(*) DESC LIMIT 1
4. Write the SELECT clause:
SELECT YEAR
5. Write the SQL:
SELECT YEAR FROM SECTION GROUP BY YEAR ORDER BY count(*) DESC LIMIT 1

Table follows, columns = ['*', 'f1', 'f2']
Table tweets, columns = ['*', 'id', 'uid', 'text', 'createdate']
Table user_profiles, columns = ['*', 'uid', 'name', 'email', 'partitionid', 'followers']

Q: "Find the average number of followers for the users who do not have any tweet."
Relevant tables:
Table user_profiles, columns = ['*', 'uid', 'name', 'email', 'partitionid', 'followers'],

Table tweets, columns = ['*', 'id', 'uid', 'text', 'createdate']
Let's translate Q to SQL clause by clause. The relevant tables are likely to be used in the translation.

1. Write the FROM clause:
FROM user_profiles

2. Write the subquery:
SELECT UID FROM tweets

3. Write the WHERE clause:
WHERE UID NOT IN (SELECT UID FROM tweets)

4. Write the SELECT clause:
SELECT avg(followers)

5. Write the SQL:
SELECT avg(followers) FROM user_profiles
WHERE UID NOT IN (SELECT UID FROM tweets)

Table classroom, columns = ['*', 'building', 'room_number', 'capacity']
Table department, columns = ['*', 'dept_name', 'building', 'budget']
Table course, columns = ['*', 'course_id', 'title', 'dept_name', 'credits']

Q: "How many rooms in each building have a capacity of over 50?"

Relevant tables:
Table classroom, columns = ['*', 'building', 'room_number', 'capacity']
Let's translate Q to SQL clause by clause. The relevant tables are likely to be used in the translation.

1. Write the FROM clause:
FROM classroom

2. Write the WHERE clause:
WHERE capacity > 50

3. Write the GROUP BY clause:
GROUP BY building

4. Write the SELECT clause:
SELECT count(*) , building

5. Write the SQL:
SELECT count(*) , building FROM classroom WHERE capacity > 50 GROUP BY building

Tables locations, columns = ['*', 'location_id', 'other_details']
Tables products, columns = ['*', 'product_id', 'product_type_code', 'product_name', 'product_price']
Tables events, columns = ['*', 'event_id', 'address_id', 'channel_id', 'event_type_code' , 'finance_id', 'location_id']
Tables products_in_events, columns = ['*', 'product_in_event_id', 'event_id', 'product_id']

Q: "Show the names of products that are in at least two events."

Relevant tables:
Tables products, columns = ['*', 'product_id', 'product_type_code', 'product_name', 'product_price'],
Tables products_in_events, columns = ['*', 'product_in_event_id', 'event_id', 'product_id']
Let's translate Q to SQL clause by clause. The relevant tables are likely to be used in the translation.

1. Write the FROM clause:
FROM Products AS T1 JOIN Products_in_Events AS T2 ON T1.Product_ID = T2.Product_ID

2. Write the GROUP BY clause:
GROUP BY T1.Product_Name HAVING COUNT(*) >= 2
3. Write the SELECT clause:
SELECT T1.Product_Name
4. Write the SQL:
SELECT T1.Product_Name
FROM Products AS T1 JOIN Products_in_Events AS T2 ON T1.Product_ID = T2.Product_ID
GROUP BY T1.Product_Name HAVING COUNT(*) >= 2

Tables products, columns =['*', 'product_id', 'parent_product_id', 'production_type_code', 'unit_price', 'product_name','product_color', 'product_size']
Tables financial_transactions, columns = ['*', 'transaction_id', 'account_id', 'invoice_number', 'transaction_type', 'transaction_date', 'transaction_amount', 'transaction_comment', 'other_transaction_details']
Tables order_items, columns = ['*', 'order_item_id', 'order_id', 'product_id', 'product_quantity', 'other_order_item_details']

Q: "What are the ids for transactions that have an amount greater than the average amount of a transaction?"
Relevant table:
Tables financial_transactions, columns = ['*', 'transaction_id', 'account_id', 'invoice_number', 'transaction_type', 'transaction_date', 'transaction_amount', 'transaction_comment', 'other_transaction_details'] "
Let's translate Q to SQL clause by clause. The relevant tables are likely to be used in the translation.
1. Write the FROM clause:
FROM Financial_transactions
2. Write the subquery:
SELECT avg(transaction_amount) FROM Financial_transactions
3. Write the WHERE clause:
WHERE transaction_amount >
(SELECT avg(transaction_amount) FROM Financial_transactions)
4. Write the SELECT:
SELECT transaction_id
5. Write the SQL:
SELECT transaction_id FROM Financial_transactions WHERE transaction_amount >
(SELECT avg(transaction_amount) FROM Financial_transactions)

A.4. Result Select prompt

The Result Select prompt was implemented in a zero-shot setting. An example of this prompt is illustrated in Figure 7.

Model Input

Given the following information: a database schema, a natural language question, and some SQL queries, your task is to perform the following steps:

Database Schema:
Table model_list, columns = ['*', 'modelid', 'maker', 'model'] ,
Table car_names, columns = ['*', 'model', 'make'] ,
Table cars_data, columns = ['*', 'id', 'horsepower', 'cylinders', 'edispl', 'horsepower', 'weight', 'accelerate', year] .

Q: " Which model of the car has the minimum horsepower?"

SQL1:
SELECT model_list.model FROM cars_data JOIN car_names ON cars_data.id = car_names.makeid JOIN model_list ON car_names.model = model_list.model JOIN (SELECT MIN(horsepower) AS min_hp FROM cars_data) AS min_hp_table ON cars_data.horsepower = min_hp_table.min_hp

SQL2:
SELECT model FROM model_list WHERE modelid IN (SELECT id FROM cars_data WHERE horsepower = (SELECT MIN(horsepower) FROM cars_data))

SQL3:
SELECT T3.model FROM cars_data AS T1 JOIN car_names AS T2 ON T1.id = T2.makeid JOIN model_list AS T3 ON T2.model = T3.model ORDER BY T1.horsepower ASC LIMIT 1

Please complete the following task:
1. Explain both each SQL separately.
2. Determine which one SQL correctly answers the given question.

Model Output

1. Explanation of SQL queries:
SQL1: This SQL query attempts to find the car model with the minimum horsepower by joining the three tables cars_data, cars_names, and model_list, along with a subquery. However, if there are multiple cars with a minimum horsepower, then this query will return models for all of those cars, not just one.
SQL2: This SQL query attempts to select the model by selecting it from the model_list table, where the modelid is in a list generated by a subquery. However, the problem with this query is that it incorrectly assumes that the id field in the cars_data table.
SQL3: This SQL query is designed to find the car model with the lowest horsepower. It does this by joining the three tables cars_data, car_names, and model_list. The query then orders the results by the horsepower field in ascending order, meaning the car with the lowest horsepower will be first.
2. The SQL query that correctly answers the given question is SQL3.

Figure 7: An example of input and output of Result Select prompt.