PTD-SQL: Partitioning and Targeted Drilling with LLMs in Text-to-SQL

Anonymous ACL submission

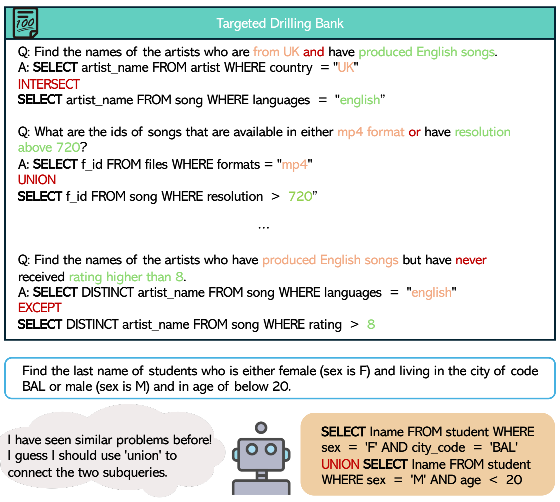
Abstract

Large Language Models (LLMs) have emerged as powerful tools for Text-to-SQL tasks, exhibiting remarkable reasoning capabilities. Different from tasks such as math word problem and commonsense reasoning, SQL solutions have a relatively fixed pattern. This facilitates the investigation of whether LLMs can benefit from categorical thinking, mirroring how humans acquire knowledge through inductive reasoning based on comparable examples. In this study, we propose that employing query group partitioning allows LLMs to focus on learning the thought processes specific to a single problem type, consequently enhancing their reasoning abilities across diverse difficulty levels and problem categories. Our experiments reveal that multiple advanced LLMs, when equipped with PTD-SQL, can either surpass or match previous state-of-the-art (SOTA) methods on the Spider and BIRD datasets. Intriguingly, models with varying initial performances have exhibited significant improvements mainly at the boundary of their capabilities after targeted drilling, suggesting a parallel with human progress.

1 Introduction

The Text-to-SQL task involves the automatic generation of SQL statements from natural language (Qin et al., 2022). Prior research primarily focused on training encoder-decoder models on text corpora and database schemas to capture generation patterns (Xu et al., 2021). Given the impressive capabilities of Large Language Models (LLMs) in various Natural Language Processing (NLP) tasks, numerous studies have endeavored to apply LLMs to this task (Li et al., 2024a; Zhang et al., 2024).

Recent investigations have proposed enhancing the reasoning capabilities of LLMs in the Text-to-SQL task, yielding substantial progress. Diverse methods such as the few-shot Chain-of-Thought (CoT) (Wei et al., 2022), self-consistency (Wang et al., 2022), and the decomposition prompt that emphasizes dissecting complex problems and solving them sequentially (Khot et al., 2022) have been introduced. A leading method, DIN-SQL (Pourreza and Rafiei, 2024), breaks down the task into several subtasks, classifies the complexity based on the nested logic of the problem, and applies different prompt strategies accordingly. However, like other studies, it overlooks the unique characteristics of SQL statements, which differ from math word problems and other code tasks. For calculations involving multiple sets, keywords like 'INTERSECT' or 'UNION' are often used to combine statements of several subproblems, making these queries naturally suitable for decomposition. Counting and sorting problems typically rely on 'GROUP BY' operations to identify objects to be aggregated and use 'ORDER BY' to sort other objects. Just like during a test with various question types, the knowledge points and problem-solving experiences that emerge in our minds are different.



```
Q: Find the names of the artists who are from UK and have produced English songs.
A: SELECT artist_name FROM artist WHERE country = "UK"
INTERSECT
SELECT artist_name FROM song WHERE languages = "english"

Q: What are the ids of songs that are available in either mp4 format or have resolution above 720?
A: SELECT f_id FROM files WHERE formats = "mp4"
UNION
SELECT f_id FROM song WHERE resolution > 720"

...

Q: Find the names of the artists who have produced English songs but have never received rating higher than 8.
A: SELECT DISTINCT artist_name FROM song WHERE languages = "english"
EXCEPT
SELECT DISTINCT artist_name FROM song WHERE rating > 8
```

Find the last name of students who is either female (sex is F) and living in the city of code BAL or male (sex is M) and in age of below 20.

I have seen similar problems before! I guess I should use 'union' to connect the two subqueries.

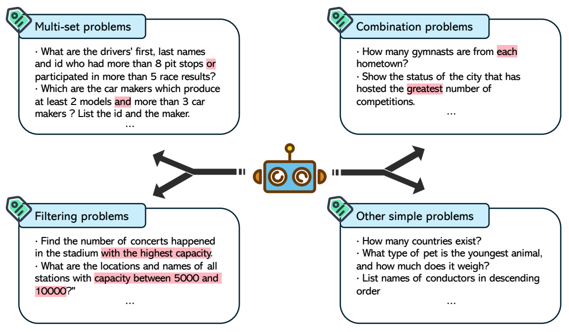
```
SELECT lname FROM student WHERE
sex = 'F' AND city_code = 'BAL'
UNION SELECT lname FROM student
WHERE sex = 'M' AND age < 20
```

Figure 1: Demonstration of targeted drilling prompt on multi-set problems.

Motivated by the brief overview of SQL question types above, we consider whether it is feasible to guide LLMs, akin to training human students for specific question types to master key concepts, by focusing on type-related examples during reasoning (Zhou et al., 2024). Accordingly, we randomly select 100 multi-set operation questions from the training set, which require the use of keywords like ’INTERSECT’ or ’EXCEPT’. We adopt two different prompt strategies: one from DIN-SQL, where these questions are classified as nested-level questions, providing samples of various question types under this complexity level; and another, as depicted in Figure 1, where we only provide LLM multi-set question examples with the same number. With these strategies, we achieve execution accuracy rates of 39.0 and 55.0 using ChatGPT, respectively. The former exhibits more sub-query errors and logical confusion.

Drawing on the above observation, we propose the **P**artitioning and **T**argeted **D**rilling (PTD-SQL) framework to enhance LLMs’ reasoning capabilities in Text-to-SQL tasks. This strategy mirrors the human learning process, where students typically first identify the group of question and then search for the most relevant knowledge points to answer it. Initially, we categorize the types of textual queries in the training set based on the keywords in the ground-truth SQL statements. Informed by previous studies, we opt not to rely solely on the LLM’s few-shot discrimination ability, but instead delegate a small LLM with fine-tuning for this task (Juneja et al., 2023; Zhuang et al., 2023). In the second step, we design prompts with different emphases for various categories of problems in the training set, and automatically generate problem sets and reference answers – the areas that the LLM needs to learn. Both of these operations are performed offline and avoid invoking GPT during testing, thus achieving cost-efficiency. Finally, during the inference stage, we classify the original textual query and design an automatic selection module to compose a few-shot prompt in the corresponding group of the problem set (An et al., 2023a).

We extensively validate the effectiveness of PTD-SQL on the Spider-dev, Spider-realistic, and BIRD-dev datasets using three powerful LLMs, where it outperforms state-of-the-art frameworks such as DIN-SQL and DAIL-SQL. We also find that the model becomes more capable of achieving breakthroughs at the capability boundaries when equipped with PTD-SQL, which may potentially



· What are the drivers' first, last names and id who had more than 8 pit stops or participated in more than 5 race results?
· Which are the car makers which produce at least 2 models and more than 3 car makers ? List the id and the maker.
...

· How many gymnasts are from each hometown?
· Show the status of the city that has hosted the greatest number of competitions.
...

· Find the number of concerts happened in the stadium with the highest capacity.
· What are the locations and names of all stations with capacity between 5000 and 10000?"
...

· How many countries exist?
· What type of pet is the youngest animal, and how much does it weigh?
· List names of conductors in descending order
...

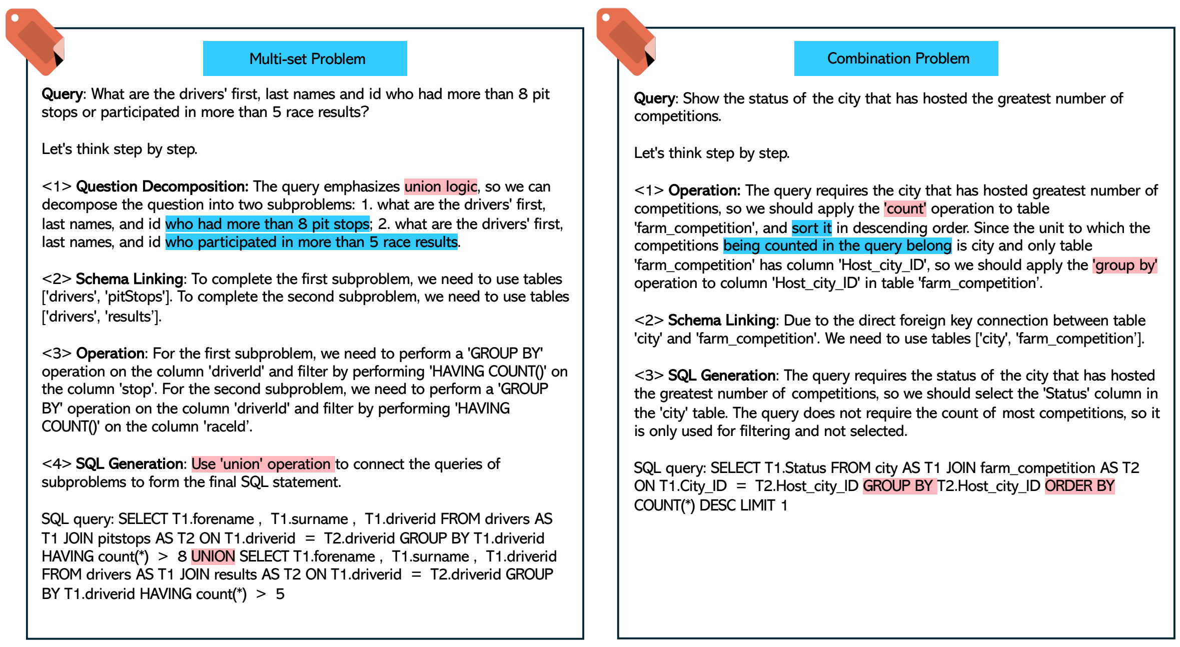
Figure 2: Some samples of proposed partition.

extend to other reasoning tasks. Furthermore, our approach adheres to a one-time query paradigm, showing advantages in terms of token consumption and inference time, also allowing many methods targeting schema linking or database content alignment to be seamlessly integrated, thereby anticipating even higher performance.

2 Related Work

LLM Reasoning Nowadays, the development of reasoning models based on LLM has become a popular and critical field. Many efficient prompting methods have been proposed, such as Chain-of-Thought (Wei et al., 2022), which guides LLM in step-by-step thinking; Least-to-Most (Zhou et al., 2022), which makes the model adapt to the difficulty gradient; and Decomposition-based prompting (Khot et al., 2022; Ye et al., 2023), which breaks down difficult problems to solve them separately. In addition, Self-Consistency (Wang et al., 2022) demonstrates the overall tendency of LLM towards the correct answer through voting, Self-discover (Zhou et al., 2024) allows the model to make different problem-solving plans according to different types of questions, and Self-refine (Madaan et al., 2024) enables LLM to learn from the feedback of its problem-solving process. Besides, many works also strengthen the weaker aspects of LLM at the code level, such as PAL (Gao et al., 2023b) and PoT (Chen et al., 2022).

LLM-based Text-to-SQL Nowadays, many studies are focusing on utilizing LLMs to complete Text-to-SQL tasks, primarily involving more efficient prompt design and advanced process deployment. Strategies that have proven effective in common sense reasoning and mathematical reasoning, such as CoT and self-consistency, have also been applied to enhance Text-to-SQL reasoning. C3 (Dong et al., 2023) and StructGPT (Jiang


Multi-set Problem

Query: What are the drivers' first, last names and id who had more than 8 pit stops or participated in more than 5 race results?

Let's think step by step.

<1> **Question Decomposition:** The query emphasizes union logic, so we can decompose the question into two subproblems: 1. what are the drivers' first, last names, and id who had more than 8 pit stops; 2. what are the drivers' first, last names, and id who participated in more than 5 race results.

<2> **Schema Linking**: To complete the first subproblem, we need to use tables ['drivers', 'pitStops']. To complete the second subproblem, we need to use tables ['drivers', 'results'].

<3> **Operation**: For the first subproblem, we need to perform a 'GROUP BY' operation on the column 'driverId' and filter by performing 'HAVING COUNT()' on the column 'stop'. For the second subproblem, we need to perform a 'GROUP BY' operation on the column 'driverId' and filter by performing 'HAVING COUNT()' on the column 'raceId'.

<4> **SQL Generation**: Use 'union' operation to connect the queries of subproblems to form the final SQL statement.

SQL query: SELECT T1.forename , T1.surname , T1.driverid FROM drivers AS T1 JOIN pitstops AS T2 ON T1.driverid = T2.driverid GROUP BY T1.driverid HAVING count(*) > 8 UNION SELECT T1.forename , T1.surname , T1.driverid FROM drivers AS T1 JOIN results AS T2 ON T1.driverid = T2.driverid GROUP BY T1.driverid HAVING count(*) > 5

Combination Problem

Query: Show the status of the city that has hosted the greatest number of competitions.

Let's think step by step.

<1> **Operation:** The query requires the city that has hosted greatest number of competitions, so we should apply the 'count' operation to table 'farm_competition', and sort it in descending order. Since the unit to which the competitions being counted in the query belong is city and only table 'farm_competition' has column 'Host_city_ID', so we should apply the 'group by' operation to column 'Host_city_ID' in table 'farm_competition'.

<2> **Schema Linking**: Due to the direct foreign key connection between table 'city' and 'farm_competition'. We need to use tables ['city', 'farm_competition'].

<3> **SQL Generation**: The query requires the status of the city that has hosted the greatest number of competitions, so we should select the 'Status' column in the 'city' table. The query does not require the count of most competitions, so it is only used for filtering and not selected.

SQL query: SELECT T1.Status FROM city AS T1 JOIN farm_competition AS T2 ON T1.City_ID = T2.Host_city_ID GROUP BY T2.Host_city_ID ORDER BY COUNT(*) DESC LIMIT 1

Figure 3: Prompt demonstrations for Multi-set and Combination problem.

et al., 2023) have introduced effective zero-shot strategies based on GPT, along with meticulous interface settings. DIN-SQL (Pourreza and Rafiei, 2024) divides the Text-to-SQL task into phased subtasks and assigns different LLMs to specialize in completing each stage, as well as categorizes the difficulty of questions to provide varying prompt strategies. DAIL-SQL (Gao et al., 2023a) has conducted a comprehensive evaluation of many prompt-based methods and proposed a more precise samples matching approach to improve results. Recent approaches have also concentrated on addressing issues not yet considered in the data itself. For instance, PET-SQL (Li et al., 2024b) focuses on leveraging prior knowledge within databases to enhance the accuracy of responses at the token level. SQL-CRAFT (Xia et al., 2024) suggests allowing models to engage in interactive refinement to improve reasoning accuracy. DEA-SQL (Xie et al., 2024) integrates multiple optimization techniques to propose workflow agents.

3 Pipeline of PTD-SQL

In this section, we present the process of the PTD-SQL framework as illustrated in Figure 4, which includes: i. The design and implementation of the proposed Query Group Partition (QGP) sub-task; ii. The automatic construction of distinct query group question banks, each containing its unique reasoning process; iii. The inference process.

3.1 Query Group Partition

In this section, we first provide the definition of the QGP sub-task and then describe the process of fine-tuning the small LLM using PEFT to accomplish the QGP task.

Problem Formulation SQL queries differ from math word problems and other code problems, such as Python, as their textual labels often contain highly characteristic expressions, making problem group identification convenient. We cluster them based on label keywords: multi-set, combination, filter, and other simple problems. Multi-set problems frequently involve two or more layers of logic and require keywords like 'INTERSECT', 'UNION', 'EXCEPT' for connection. Combination problems necessitate the use of 'GROUP BY' operation to group data, followed by sorting, taking extreme values, and other purposeful operations. Filter problems involve constructing conditional statements and using them for target screening. The remaining problems are classified as other simple problems, as depicted in Figure 2. Considering that some queries may have implicit labels of other types, we provide prioritized classification criteria in the prompt to alleviate the impact of model bias. Specific examples are shown in the Appendix E.3. The task objective is explicitly defined as: given a text query q, we need to output its problem group $\hat{g}$. It is formulated as:

Method	Exact Match
Llama-2-7b + LoRA	85.0%
ChatGPT + 10-shot	68.0%

Table 1: Performance on validation set of QGP sub-task.

$$\hat{g} = f(q \mid \theta) \tag{1}$$

where $f(\cdot \mid \theta)$ can present a model with parameters θ. We randomly select the training set $\mathcal{S}_T$ for the QGP task on the original training set and separate the validation set $\mathcal{S}_V$.

Fine-tuned LLM Classifier Inspired by previous works (Juneja et al., 2023; Zhuang et al., 2023), we consider delegating the ability to determine categories to the fine-tuning process of the small LLM rather than directly trusting the discrimination capability of LLM. With the rapid advancement of PEFT technology, we choose Low-Rank Adaptation (LoRA) (Hu et al., 2021) to fine-tune the Llama-2-7b model to solve the QGP problem. For a pre-trained weight matrix $W_0 \in R^{d\times k}$, LoRA adds a bypass using two decomposition matrices $A \in R^{d\times r}$ and $B \in R^{r\times k}$, where $r \ll min(d, k)$. The forward process of single weight matrix is modified to:

$$h = W_0 x + BAx \tag{2}$$

During finetuning with LoRA, we freeze the original weights of LLM and only update low-rank matrices A and B.

For annotated labels G and outputs of LLM, the objective loss is defined as :

$$\mathcal{L} = CrossEntropy(G \mid f(q \mid \theta + \delta\theta)) \tag{3}$$

Finetuned Small Model vs. Few-shot GPT The performance of the fine-tuned Llama-2-7b model and the few-shot prompting ChatGPT on the QGP task is presented in Table 1. This highlights the superiority of PEFT in downstream tasks, and prompts us to use the former on test set.

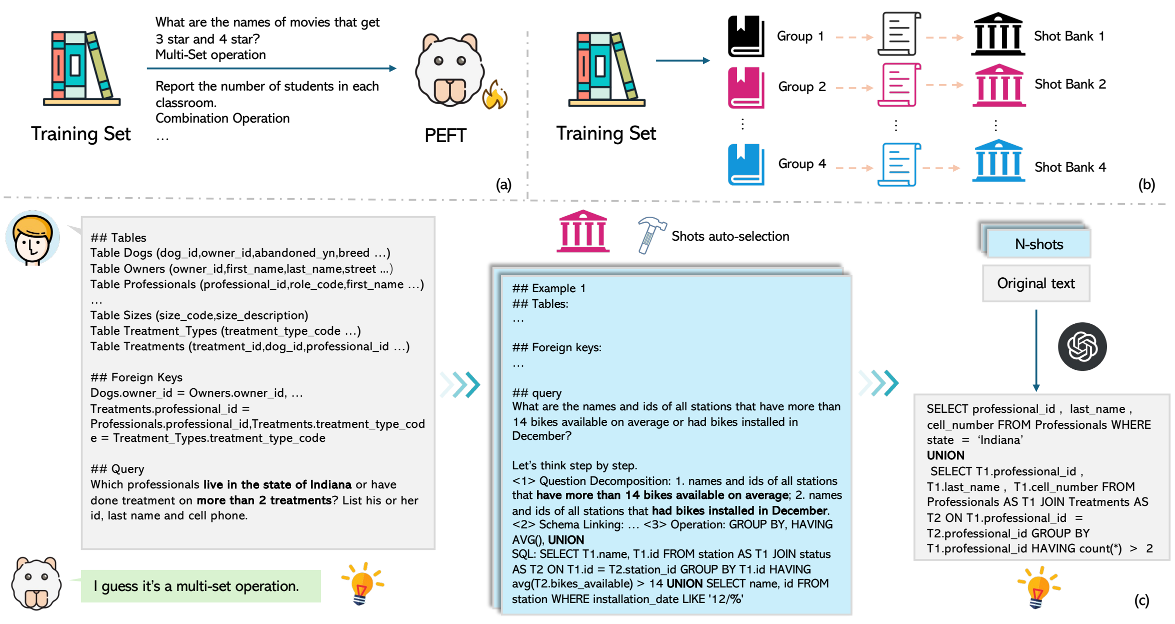
3.2 Targeted Drilling Bank Auto-construction

In this section, we explain how to construct targeted drilling banks for different question groups in PTD-SQL, which can be compared to the specialized training and reference ideas and answers designed by teachers for students before examinations. Previous works grade the difficulty based on whether the problem requires nesting and design corresponding prompt templates. However, this approach only focuses on the surface logic of SQL queries and does not consider the distinct thinking paths required by the essence of different question groups for LLM. Given that selecting irrelevant examples may also be detrimental to LLM's thinking, in PTD-SQL, we can benefit from the proposed QGP. That is, for test queries of specific question groups, we can directly and accurately locate the problem banks with similar thinking paths.

Multi-set problems often require breaking down a complex problem into multiple subqueries, integrating the different results through connecting keywords. For filtering problems, we can often prompt LLM to first propose the organization of filtering conditions and then process the selection target. Therefore, these two types of problems are naturally suitable for the design inspiration of decomposed prompting. We show an example of prompt construction for a multi-set problem, as depicted in Figure 3. For filtering problems, our decomposition focuses on the division of conditional statements and the extraction of target columns, and the specific prompts are shown in the Appendix E.1. It is worth mentioning that we treat schema linking as a byproduct of LLM's thinking process, thereby achieving the purpose of one-time generation, which reduces the query cost.

For combination problems and other simple problems, we construct concise CoT templates. For the former, the model is required to distinguish the objects that need to be counted (sorted or taking extreme values) and the groups they belong to, thus improving the ability to organize answers under this question type. An example is shown in Figure 3. For the remaining simple problems, we choose to use the ground truth SQL query directly as the composition of the few-shot prompt, without introducing other thinking processes.

After creating four different types of few-shot prompts, we apply them separately to their respective problem groups in the training set to generate the thinking process and the final SQL query. We select the samples with correct execution results of the SQL query to form four targeted drilling banks because we believe that the thinking paths in the examples with correct final answers are highly likely to be reasonable and enlightening. These are the sources of the examples that LLM refers to during the inference phase. The specific statistics of different targeted drilling banks are shown in Appendix A.1.



```
## Tables
Table Dogs (dog_id,owner_id,abandoned_yn,breed …)
Table Owners (owner_id,first_name,last_name,street …)
Table Professionals (professional_id,role_code,first_name …)
…
Table Sizes (size_code,size_description)
Table Treatment_Types (treatment_type_code …)
Table Treatments (treatment_id,dog_id,professional_id …)

## Foreign Keys
Dogs.owner_id = Owners.owner_id, …
Treatments.professional_id =
Professionals.professional_id,Treatments.treatment_type_cod
e = Treatment_Types.treatment_type_code

## Query
Which professionals live in the state of Indiana or have
done treatment on more than 2 treatments? List his or her
id, last name and cell phone.
```

I guess it's a multi-set operation.

```
## Example 1
## Tables:
…

## Foreign keys:
…

## query
What are the names and ids of all stations that have more than
14 bikes available on average or had bikes installed in
December?

Let's think step by step.
<1> Question Decomposition: 1. names and ids of all stations
that have more than 14 bikes available on average; 2. names
and ids of all stations that had bikes installed in December.
<2> Schema Linking: … <3> Operation: GROUP BY, HAVING
AVG(), UNION
SQL: SELECT T1.name, T1.id FROM station AS T1 JOIN status
AS T2 ON T1.id = T2.station_id GROUP BY T1.id HAVING
avg(T2.bikes_available) > 14 UNION SELECT name, id FROM
station WHERE installation_date LIKE '12/%'
```

```
SELECT professional_id , last_name ,
cell_number FROM Professionals WHERE
state = 'Indiana'
UNION
 SELECT T1.professional_id ,
T1.last_name , T1.cell_number FROM
Professionals AS T1 JOIN Treatments AS
T2 ON T1.professional_id =
T2.professional_id GROUP BY
T1.professional_id HAVING count(*) > 2
```

Figure 4: Overflow of PTD-SQL. (a) QGP sub-task. (b) Targeted drilling bank auto-construction. (c) Reasoning step.

3.3 Few-shot Selection

Few-shot example construction is a crucial step in prompt engineering because LLMs are sensitive to few-shot samples. In PTD-SQL, we perform QGP on each textual query and then automatically select shots in the corresponding targeted drilling bank.

Semantic matching Previous work has verified the effectiveness of methods based on semantic vector matching (An et al., 2023b). We calculate and store sentence embeddings for all textual queries in the targeted drilling bank using OpenAI text-embedding-ada-002[1], resulting in an offline bank matrix $\mathcal{M}$. For test queries, we encode them with text-embedding-ada-002 and calculate the cosine similarity with $\mathcal{M}$ to measure the degree of semantic matching as some previous works do.

$$sim_1(s, s_i) = \frac{Emb(s)Emb(s_i)^T}{|Emb(s)||Emb(s_i)|} \quad (4)$$

Syntactic matching Considering that textual SQL queries have strong syntactic features, such as counting problems often having phrases like "how many", and extreme value demands often accompanied by comparative adjectives like "largest" or "lowest". Therefore, we use token overlap counts to rank the syntactic relevance of samples in the corresponding targeted drilling bank.

[1]https://platform.openai.com/docs/guides/embeddings

$$sim_2(s, s_i) = \frac{set(tokenize(s))\ \&\ set(tokenize(s_i))}{set(tokenize(s))} \quad (5)$$

Mix-of-matching Similar to the idea of multi-way recall, we mix an equal amount of examples selected by the two strategies above, for instance, choosing the top 2 most relevant examples from each in a 4-shot scenario, in order to provide as rich and relevant samples as possible within the same problem group, thus guiding effective thinking.

4 Experiments

4.1 Experimental Setup

Datasets Spider (Yu et al., 2018) is the most widely used cross-domain dataset. This dataset has 7,000 training data in the training set and 1,034 data in the development set, covering 200 different databases and spanning 138 domains. Spider-realistic (Deng et al., 2020) is a more challenging dataset containing 508 test data points which manually mask the specific column selections in the text query. BIRD (Li et al., 2024a) dataset contains 95 large-scale real databases, convering 37 professional domains. More details and usage of the data can be found in Appendix A.2.

Evaluation Most previous work adheres to two common evaluation metrics: 1) Exact Match Accuracy (EM): It requires that each subcomponent of

Methods	Type	EX
T5-3B + PICARD† (Scholak et al., 2021)	Fine-tuning	79.3
RESDSQL + NatSQL† (Li et al., 2023)	Fine-tuning	$\underline{84.1}$
C3 + ChatGPT† (Dong et al., 2023)	Zero-shot	81.2
ChatGPT (Liu et al., 2023)	Zero-shot	70.1
GPT-4 (Achiam et al., 2023; Gao et al., 2023a)	Zero-shot	72.3
DIN-SQL + ChatGPT§ (Pourreza and Rafiei, 2024)	Few-shot	76.8
DIN-SQL + GPT-4§	Few-shot	80.6
DIN-SQL + Deepseek-coder-6.7b-instruct‡	Few-shot	73.6
DAIL-SQL + ChatGPT† (Gao et al., 2023a)	Few-shot	79.1
DAIL-SQL + GPT-4†	Few-shot	83.1
DAIL-SQL + GPT-4 + Self-Consistency†	Few-shot	83.6
DAIL-SQL + Deepseek-coder-6.7b-instruct‡	Few-shot	75.7
PTD-SQL + ChatGPT$_{ours}$	Few-shot	80.3
PTD-SQL + GPT-4$_{ours}$	Few-shot	**85.7**
PTD-SQL + Deepseek-coder-6.7b-instruct$_{ours}$	Few-shot	76.7

Table 2: EX on Spider-dev set. Results of methods with † are taken from original paper or open-source code repository. Results with label ‡ are implemented by us. Results with § are obtained from the running results files provided by (Pourreza and Rafiei, 2024) and evaluation program (Zhong et al., 2020).

Methods	Type	EX
ChatGPT	Zero-shot	67.3
GPT-4	Zero-shot	66.5
DIN-SQL+ChatGPT	Few-shot	$\underline{70.3}$
DIN-SQL+Deepseek-coder-6.7b-instruct	Few-shot	68.3
DAIL-SQL+ChatGPT	Few-shot	69.3
DAIL-SQL+ Deepseek-coder-6.7b-instruct	Few-shot	68.9
PTD-SQL+ChatGPT$_{Ours}$	Few-shot	**72.2**
PTD-SQL+Deepseek-coder-6.7b-instruct$_{ours}$	Few-shot	69.9

Table 3: EX on Spider-realistic dataset.

Methods	EX	VES
CodeX	34.4	41.6
ChatGPT+CoT	36.6	42.3
GPT-4	46.4	49.8
DIN-SQL + ChatGPT	41.0	51.4
DIN-SQL + GPT-4	50.2	**58.1**
DIN-SQL + Deepseek-coder-6.7b-instruct	40.7	49.0
DAIL-SQL + ChatGPT	41.2	49.2
DAIL-SQL + GPT-4	$\underline{53.6}$	56.5
DAIL-SQL + Deepseek-coder-6.7b-instruct	42.4	50.2
PTD-SQL + ChatGPT$_{ours}$	44.2	53.3
PTD-SQL + GPT-4$_{ours}$	**57.0**	$\underline{57.7}$
PTD-SQL + Deepseek-coder-6.7b-instruct$_{ours}$	45.4	55.0

Table 4: EX and VES comparison on BIRD dataset.

the SQL query generated by the model matches the gold SQL query provided in the dataset. 2) Execution Accuracy (EX): EX judges correctness based on whether the answer returned by executing the predicted SQL query in the database is consistent with the gold query. Since a textual query may correspond to several correct but stylistically different SQL query formulations, it is a more accurate measure of Text-to-SQL methods. Besides, Valid Efficiency Score (VES) is used to demonstrate the efficiency of valid SQLs provided by models.

Baselines We compare three different path Text-to-SQL methods, including fine-tuning, zero-shot, and few-shot prompting methods. Among them, the fine-tuning method includes PICARD (Scholak et al., 2021) and the current SOTA RESDSQL+NatSQL (Li et al., 2023). The zero-shot method C3 (Dong et al., 2023) focuses on schema linking filtering and removing GPT's inherent bias for SQL generation. DIN-SQL (Pourreza and Rafiei, 2024), which breaks down the textual query into multiple staged questions. DAIL-SQL (Gao et al., 2023a) considers optimizing sample selection and organization to further enhance LLM's reasoning ability in Text-to-SQL.

Implementation Details In order to comprehensively evaluate the performance of the framework on closed-source and open-source models and demonstrate its effectiveness, we employ three LLMs for comparison purposes: OpenAI GPT-3.5-turbo-0613 for ChatGPT, GPT-4-0613, and Deepseek-coder-6.7b-instruct[2] (Guo et al., 2024). The latter is pretrained on high-quality code corpora and has attained the current state-of-the-art performance among open-source code models in the realm of code generation. Maximum context length is limit to 4096 for OpenAI LLMs and 2048 for open-source LLMs.

4.2 Main Results

As shown in Table 2, PTD-SQL + GPT4 achieves the best EX metric on the Spider-dev dataset. Additionally, PTD-SQL surpasses DIN-SQL and DAIL-SQL when using ChatGPT and Deepseek-coder-6.7b-instruct as base models. Compared to the more advanced DAIL-SQL framework, PTD-SQL achieves relative increases of 1.5%, 3.1%, and 1.3% on ChatGPT, GPT-4 and Deepseek-coder-6.7b-instruct respectively. When compared with previous fine-tuning and prompting methods, PTD-SQL also attains a comparative performance. Besides, as shown in Table 3, ChatGPT equipped PTD-SQL also outperforms previous methods and GPT-4 using zero-shot. Furthermore, the results shown in Table 4 indicate that all three powerful models equipped with PTD-SQL demonstrate stronger EX. In terms of VES indicators, PTD-SQL

[2]https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct

Base Model	Method	Easy	Medium	Hard	Extra	All
Deepseek-coder -6.7b-instruct	DIN-SQL	86.3	81.2	59.8	48.8	73.6
	DAIL-SQL	86.7	**81.6**	59.2	50.0	75.7
	PTD-SQL	**87.1**	78.9	**74.9**	**57.2**	**76.7**
ChatGPT	DIN-SQL	90.7	82.3	62.1	**56.6**	76.8
	DAIL-SQL	**91.5**	**83.8**	71.2	56.0	79.1
	PTD-SQL	90.7	83.1	**80.6**	**56.6**	**80.3**
GPT-4	DIN-SQL	89.9	84.3	78.2	57.8	80.4
	DAIL-SQL	90.7	**89.7**	75.3	62.0	83.1
	PTD-SQL	**94.8**	88.8	**85.1**	**64.5**	**85.7**

Table 5: Performance comparison on three LLMs across difficulty levels on Spider-Dev dataset.

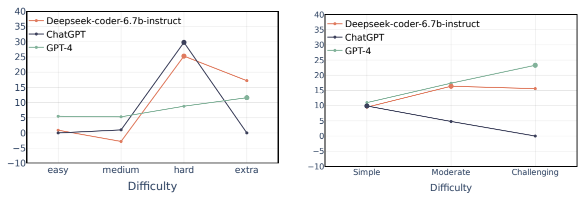


Figure 5: Under different difficulty levels, the percentage gain (%) in EX metric on Spider (left) and BIRD (right) obtained by the three models using PTD-SQL compared to DIN-SQL.

also has a certain competitiveness. A case study on Spider is given in Appendix B.6. Furthermore, we discuss the advantages of PTD-SQL in terms of token consumption and inference time in Appendix D.

5 More Discussion

In this section, we investigate the efficacy of PTD-SQL, taking into account both the challenges posed by the database itself (**RQ1**) and the performance across various problem groups (**RQ2**). Concurrently, we delve into the insights that PTD-SQL contributes to the LLM-based Text-to-SQL domain. Furthermore, we perform ablation studies on the employed modules, primarily focusing on the effectiveness of introduced QGP task (**RQ3**), and the influence of shot selection strategies within the same targeted drilling bank (**RQ4**).

5.1 RQ1: Performance from a Difficulty-level

In this subsection, we evaluate the superiority of PTD-SQL over existing state-of-the-art frameworks based on the difficulty levels defined by the database, respectively. As depicted in Table 5, PTD-SQL outperforms DIN-SQL and DAIL-SQL across different base LLMs, particularly at hard and extra difficulty levels, indicating that LLM can specialize in a problem group and demonstrate enhanced targeted reasoning ability after imitating and delving into problems within the same group.

Moreover, we illustrate the performance variations of PTD-SQL in comparison to DIN-SQL across different problem types, thereby discerning the disparities between problem group partitioning strategies and difficulty grading strategies. As inferred from Figure 5, LLMs have made great progress at their respective capacity limits under PTD-SQL. For instance, ChatGPT, akin to a diligent student, achieves a 29.8% improvement in hard difficulty by focusing on similar problems, but fails to progress in extra difficulty, possibly due to inherent model limitations. The deepseek-coder-6.7b-instruct model, with capabilities comparable to ChatGPT, also shows the most significant improvement in hard difficulty (25.3% vs 17.2% on extra). However, GPT-4, resembling an elite student, achieves the most substantial breakthrough in extra difficulty and refines its responses across other difficulty levels through referencing and absorption. The results on the BIRD dataset show that GPT-4 achieves the largest increase in performance on the challenging group, while the other two models focus on simple and moderate difficulties. This suggests that LLMs with different levels of reasoning capability can guarantee their upper limit by practicing questions. Detailed results on BIRD are depicted in Appendix B.1.

Model	QGP Method	Easy	Medium	Hard	Extra	All
ChatGPT	w/o QGP	84.7	76.5	71.8	52.4	73.8
	ChatGPT + 10-shot	86.7	78.9	74.1	56.0	76.3
	Llama-2-7b + LoRA	**90.7**	**83.1**	**80.6**	**56.6**	**80.3**
Deepseek-coder -6.7b-instruct	w/o QGP	84.7	76.5	71.8	52.4	73.8
	ChatGPT + 10-shot	84.3	**79.1**	69.5	54.8	74.9
	Llama-2-7b + LoRA	**87.1**	78.9	**74.9**	**57.2**	**76.7**

Table 6: EX performance based on partition with different accuracy levels on the Spider-dev dataset.

5.2 RQ2: Performance under Problem Groups

As depicted in Figure 6, PTD-SQL demonstrates a more pronounced advantage in multi-set problems and combination problems when employing three different baseline models. These problem types entail more intricate reasoning and perplexing conditions. Apart from when using GPT-4, the other two models yield very similar results in the filtering problem across the three methods. This suggests that this category of problem relies more on the inherent ability of the model to effectively organize the filtering conditions, rather than emphasizing the logical level. Besides, we consider the detailed performance of queries with multiple question type

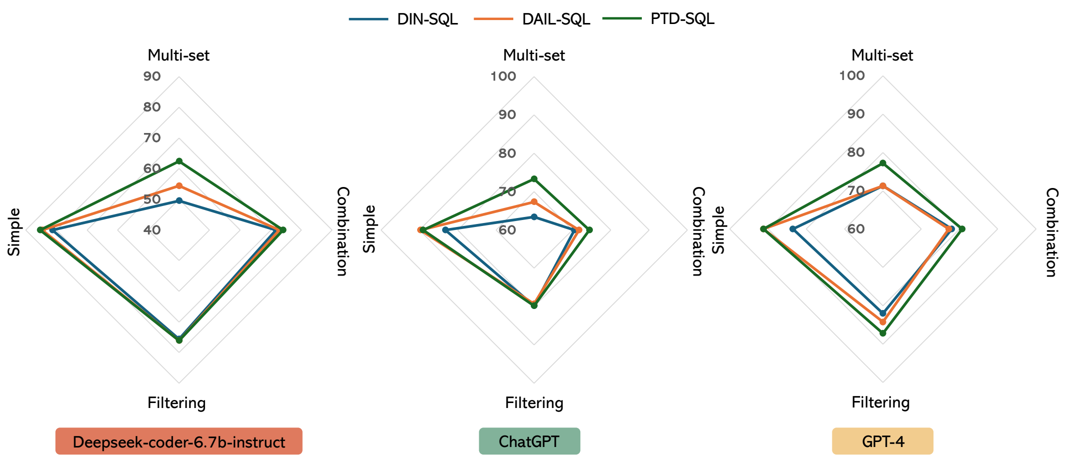


Figure 6: EX of three LLMs on Spider-dev dataset when equipped with DIN-SQL, DAIL-SQL and PTD-SQL.

features in Appendix B.5, and propose findings and directions for further improvement.

5.3 RQ3: Effectiveness of QGP

In this section, we examine the impact of the QGP subtask. As shown in Table 1, the Few-shot method does not align well within a specific context, resulting in weaker performance compared to the fine-tuned model. To further investigate this, we conduct additional experiments involving problem groups classified by ChatGPT, as well as experiments that eliminate the QGP stage and directly recall shots from all targeted drilling banks. The findings presented in Table 6 indicate that a decline in QGP accuracy adversely affects the final outcomes, with a relative decrease of 5.0% when testing on ChatGPT. Besides, ChatGPT exhibits a slight reduction in extra difficulty, while Deepseek demonstrates tolerance for classification accuracy at medium to easy difficulty levels. However, upon removing the QGP, the model surpasses the zero-shot performance, but there is a substantial decline in the results. This observation implies that incorporating various types of questions during similarity retrieval might introduce confusion and burden to the model, and also validates the relevance of the QGP stage.

5.4 RQ4: Ablation on Few-shot Selection

In this section, ablation experiments are conducted for three distinct shot selection strategies within the same problem group. As illustrated in Figure 7, the hybrid strategy demonstrates a favorable integration effect beyond the ’easy’ category, resulting in an overall improvement. This finding suggests that taking into account both query keywords and semantic similarity can yield a more comprehensive prompting effect. Additionally, an ablation study on the few-shot number is demonstrated in Appendix B.4, revealing its effect.

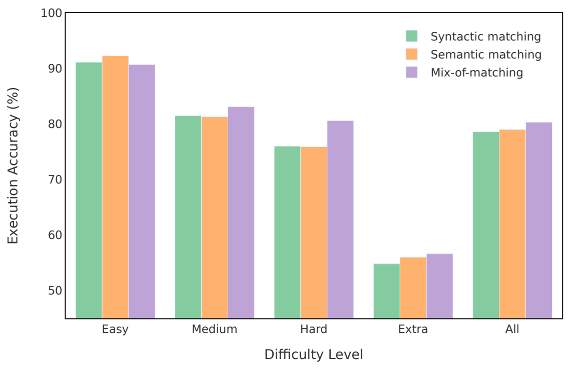


Figure 7: Ablations on few-shot selection strategy on the Spider-dev dataset. (Detailed data in Table 14)

6 Conclusion

In this article, a novel method called PTD-SQL is proposed for LLMs to conduct targeted drilling on specific groups of questions after partitioning. This approach addresses the category tendency of SQL queries, which has been overlooked in previous work. By focusing on the thinking logic of specific types, LLM can effectively enhance its reasoning capabilities. Empirical observations from our comprehensive ablation studies reveal that PTD-SQL significantly reduces the likelihood of LLM making errors within its distinct capability range while demonstrating substantial gains across various question groups. Furthermore, it is posited that this approach can be extended to other domains, such as math word problems and different types of code problems, paving the way for future research.

7 Limitations

The limitations of this article lie in the exploration of its effectiveness on larger-scale databases with a broader domain span. Moreover, even SQL statements with strong structural characteristics may have different types of divisions. Therefore, a more detailed investigation of performance under these different divisions can be further improved and optimized. Besides, as stated in Appendix B.5, for queries with multiple question types, we can also recall example questions from multiple shot banks to comprehensively consider the model and improve the fault tolerance of QGP subtasks. This may be an interesting topic that can be improved in the future. In addition, due to space constraints, this article doesn't optimize for more detailed issues such as schema linking and database content alignment. However, the optimization methods for these issues can be relatively easily integrated into PTD-SQL as a downstream optimization method. Due to our greater focus on the improvement of LLM's reasoning ability for the question answering itself in this article, we are confident that we can achieve better results by adding the aforementioned sub-optimization methods.

References

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

Shengnan An, Bo Zhou, Zeqi Lin, Qiang Fu, Bei Chen, Nanning Zheng, Weizhu Chen, and Jian-Guang Lou. 2023a. Skill-based few-shot selection for in-context learning. *arXiv preprint arXiv:2305.14210*.

Shengnan An, Bo Zhou, Zeqi Lin, Qiang Fu, Bei Chen, Nanning Zheng, Weizhu Chen, and Jian-Guang Lou. 2023b. Skill-based few-shot selection for in-context learning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13472–13492, Singapore. Association for Computational Linguistics.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2020. Structure-grounded pretraining for text-to-sql. *arXiv preprint arXiv:2010.12773*.

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, et al. 2023. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023a. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023b. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

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

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. 2023. Structgpt: A general framework for large language model to reason over structured data. *arXiv preprint arXiv:2305.09645*.

Gurusha Juneja, Subhabrata Dutta, Soumen Chakrabarti, Sunny Manchanda, and Tanmoy Chakraborty. 2023. Small language models fine-tuned to coordinate larger language models improve complex reasoning. *arXiv preprint arXiv:2310.18338*.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 13067–13075.

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

Zhishuai Li, Xiang Wang, Jingjing Zhao, Sun Yang, Guoqing Du, Xiaoru Hu, Bin Zhang, Yuxiao Ye, Ziyue Li, Rui Zhao, et al. 2024b. Pet-sql: A prompt-enhanced two-stage text-to-sql framework with cross-consistency. *arXiv preprint arXiv:2403.09732*.

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*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

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

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

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Hanchen Xia, Feng Jiang, Naihao Deng, Cunxiang Wang, Guojiang Zhao, Rada Mihalcea, and Yue Zhang. 2024. Sql-craft: Text-to-sql through interactive refinement and enhanced reasoning. *arXiv preprint arXiv:2402.14851*.

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

Kuan Xu, Yongbo Wang, Yongliang Wang, Zujie Wen, and Yang Dong. 2021. Sead: End-to-end text-to-sql generation with schema-aware denoising. *arXiv preprint arXiv:2105.07911*.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large language models are versatile decomposers: Decompose evidence and questions for table-based reasoning. *arXiv preprint arXiv:2301.13808*.

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. *arXiv preprint arXiv:1809.08887*.

Bin Zhang, Yuxiao Ye, Guoqing Du, Xiaoru Hu, Zhishuai Li, Sun Yang, Chi Harold Liu, Rui Zhao, Ziyue Li, and Hangyu Mao. 2024. Benchmarking the text-to-sql capability of large language models: A comprehensive evaluation. *arXiv preprint arXiv:2403.02951*.

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

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

Pei Zhou, Jay Pujara, Xiang Ren, Xinyun Chen, Heng-Tze Cheng, Quoc V Le, Ed H Chi, Denny Zhou, Swaroop Mishra, and Huaixiu Steven Zheng. 2024. Self-discover: Large language models self-compose reasoning structures. *arXiv preprint arXiv:2402.03620*.

Yan Zhuang, Qi Liu, Yuting Ning, Weizhe Huang, Rui Lv, Zhenya Huang, Guanhao Zhao, Zheng Zhang, Qingyang Mao, Shijin Wang, et al. 2023. Efficiently measuring the cognitive ability of llms: An adaptive testing perspective. *arXiv preprint arXiv:2306.10512*.

Appendices Content

A Supplementary Statistics 12
A.1 Statistics of Targeted Drilling Banks 12
A.2 Statistics of Employed Benchmark 12

B Supplementary Results 13
B.1 Percentage Gain under Different Difficulty-level 13
B.2 Performance under Different Problem Groups 13
B.3 Detailed Performance of Shots Selection Ablation 13
B.4 Ablation on Few-shot Evaluation 13
B.5 Fine grained analysis of multiple-type queries 14
B.6 Case Study 16

C Use of Evaluation Program 16

D Time and Token cost 16

E Prompt Design 17
E.1 Targeted Drilling Bank Auto-construction on Spider 17
E.2 Targeted Drilling Bank Auto-construction on BIRD 25
E.3 QGP Prompt 29

A Supplementary Statistics

A.1 Statistics of Targeted Drilling Banks

On Spider-dev and Spider-realistic datasets, the samples from the four different targeted drilling banks all come from random selections within their respective categories after automated classification in the training set, as shown in Table 7.

However, the BIRD dataset does not provide a training set with regular attributes for generating candidate question banks. Our testing criterion is to randomly divide the Spider-dev dataset into 20% for training and the remaining 80% as a testing benchmark at three different difficulty levels. The training set is used for fine-tuning the classifier and building the targeted drilling bank. The statistical data of the targeted drilling bank on the BIRD-dev dataset is shown in Table A. Additionally, due to the lack of clearly defined multi-set operation queries in the BIRD-dev dataset, we only need to investigate the remaining three question types.

Bank Group	Multi-set Problem	Combination Problem	Filtering Problem	Other Simple Problem
Number	200	518	377	500

Table 7: Statistics of targeted drilling banks on Spider-dev and Spider-realistic datasets.

Bank Group	Combination Problem	Filtering Problem	Other Simple Problem
Number	61	234	11

Table 8: Statistics of targeted drilling banks on BIRD-dev datasets.

A.2 Statistics of Employed Benchmark

The two datasets we use, Spider-dev and Spider-realistic, both have native difficulty levels defined by the database itself. The specific data is shown in Table 9.

Dataset	Easy	Medium	Hard	Extra	All
Spider-dev (Yu et al., 2018)	248	446	174	166	1034
Spider-realistic (Deng et al., 2020)	109	203	99	97	508
Dataset	Simple	Moderate	Challenging	All	
BIRD-dev	925	465	144	1534	

Table 9: Statistics of employed three benchmarks.

B Supplementary Results

B.1 Percentage Gain under Different Difficulty-level

The specific data of Figure 5 (left) is shown in Table 11. Correspondingly, data on BIRD-dev (right) is demonstrated in Table 12.

And we also provide the detailed results on BIRD-dev dataset split by difficulty in Table 10.

Base Model	Method	Simple	Moderate	Challenging	All
Deepseek-coder -6.7b-instruct	DIN-SQL	47.0	32.6	27.1	40.7
	DAIL-SQL	46.3	37.7	32.2	42.4
	PTD-SQL	51.5	38.0	31.4	45.4
ChatGPT	DIN-SQL	46.6	33.4	29.7	41.0
	DAIL-SQL	45.7	36.4	28.8	41.2
	PTD-SQL	51.2	35.0	29.7	44.2
GPT-4	DIN-SQL	56.9	41.4	36.4	50.2
	DAIL-SQL	<u>60.7</u>	<u>43.1</u>	<u>42.4</u>	<u>53.6</u>
	PTD-SQL	**63.2**	**48.7**	**44.9**	**57.0**

Table 10: Performance comparison on three LLMs across difficulty levels on BIRD-dev dataset. The best results under each difficulty-level when using different LLMs are addressed by **bold**.

Base Model	Easy	Medium	Hard	Extra
Deepseek-coder-6.7b-instruct	0.9	2.8	25.3	17.2
ChatGPT	0.0	1.0	29.8	0.0
GPT-4	5.5	5.3	8.8	11.6

Table 11: EX percentage gain on Spider-dev when compared to DIN-SQL. Number with green means an increase, while red means a decrease (%) or no change.

Base Model	Simple	Moderate	Challenging
Deepseek-coder-6.7b-instruct	9.5	16.4	15.6
ChatGPT	9.9	4.8	0.0
GPT-4	11.0	17.4	23.3

Table 12: EX percentage gain on BIRD-dev when compared to DIN-SQL. Number with green means an increase, while red means a decrease (%) or no change.

B.2 Performance under Different Problem Groups

The detailed data of Figure 6 is demonstrated in Table 13.

B.3 Detailed Performance of Shots Selection Ablation

We implement detailed EX performance under different shots auto-selection strategy in Table 14.

B.4 Ablation on Few-shot Evaluation

As a few-shot prompting method, we believe that the number of examples is also an important factor affecting the results. Due to the context limitations we mentioned, we conduct ablation experiments with 4 shots or fewer. For the 1-shot scenario, we selected the single most similar example based on semantic

Base Model	Method	Multi-set	Combination	Filtering	Other Simple Problem
Deepseek-coder -6.7b-instruct	DIN-SQL	49.5	71.7	75.6	81.3
	DAIL-SQL	54.4	72.8	76.1	84.6
	PTD-SQL	62.4	74.0	76.1	85.3
ChatGPT	DIN-SQL	63.4	70.5	79.8	83.2
	DAIL-SQL	67.3	71.7	79.3	<u>89.7</u>
	PTD-SQL	<u>73.3</u>	74.4	79.8	89.0
GPT-4	DIN-SQL	71.3	<u>78.0</u>	82.0	83.5
	DAIL-SQL	71.3	77.2	<u>84.2</u>	**91.2**
	PTD-SQL	**77.2**	**80.7**	**87.2**	**91.2**

Table 13: Detailed EX accuracy of three methods on Spider-dev dataset split by problem groups.

Model	Easy	Medium	Hard	Extra	All
PTD-SQL + Syntactic Matching	91.1	81.5	76.0	54.8	78.6
PTD-SQL + Semantic Matching	**92.3**	81.3	75.9	56.0	79.0
PTD-SQL + Mix-of-Matching	90.7	**83.1**	**80.6**	**56.6**	**80.3**

Table 14: Ablation study on different few-shot auto-selection strategies on Spider-dev dataset. We employ ChatGPT as reasoning LLM.

Few-shot	Easy	Medium	Hard	Extra	All
1-shot	89.1	76.6	64.9	<u>56.0</u>	74.4
2-shot	<u>89.9</u>	<u>80.2</u>	<u>72.0</u>	55.4	<u>77.2</u>
4-shot	**90.7**	**83.1**	**80.6**	**56.6**	**80.3**

Table 15: EX on different numbers of few-shot samples w.r.t difficulty-level.

Few-shot	Multi-set	Combination	Filtering	Other Simple Problem
1-shot	64.4	65.0	74.6	<u>86.4</u>
2-shot	<u>66.3</u>	<u>72.8</u>	<u>76.6</u>	86.1
4-shot	**73.3**	**74.4**	**79.8**	**89.0**

Table 16: EX on different numbers of few-shot samples w.r.t problem groups.

similarity. The performance of PTD-SQL under different numbers of examples, different difficulty levels, and different question types is shown in Table 15 and Table 16, respectively.

Our results show that when the number of examples is small, it has a greater impact on the final results, and the EX indicator generally shows a growing trend with the increase of examples. This suggests that more examples can stimulate more diverse thinking abilities under relatively limited context constraints. In our framework, more examples mean that the model has done more research on the same type of questions, thus achieving better results.

B.5 Fine grained analysis of multiple-type queries

In this section, we explore the potential constraints arising from the fact that certain questions may fall into multiple question groups. We posit that, based on the keyword classification method delineated in Section 3.1 applied to the training set, we can directly apply this to the test set to derive a potential set of question groups. For instance, the ground-truth SQL query 'SELECT Country FROM singer WHERE Age > 40 INTERSECT SELECT Country FROM singer WHERE Age < 30' is categorized as (Multi-set, Filtering).

We define the set of multiple categories to which each query should belong as X, and the single group label Y obtained after the fine-tuned Llama-2-7b model completes the QGP. In Table 17 and Table 18, we present the EX for all possible partition sets when using ChatGPT and GPT-4, respectively.

Initially, a generally accurate classification can yield relatively satisfactory results. For instance, queries featuring combination problem and filtering problem characteristics exhibit a commendable EX when they are divided into these two subclasses, given a sufficiently large number of samples. Similarly, queries with multi-set and filtering problem features can also attain comparable and favorable EX indicators when they are divided into their respective groups. This suggests that in most instances, a question with multiple types of tendencies can draw insights from a single question bank and make reasonably accurate inferences.

Nonetheless, certain observations also highlight specific limitations of PTD-SQL. For example, in the case of combination-type questions, superior overall results were achieved when they were directly classified as simple problems. This is because these questions, in contrast to those classified as combination problems, contain a greater number of easy and medium difficulty problems, thus directly benefiting from the simplicity of CoT. Consequently, for future optimization of PTD-SQL, it could be considered, as suggested in the DIN-SQL method, to incorporate the difficulty of the query, thereby preventing some simple questions from being disrupted by complex thought processes.

When comparing the data between Table 1 and Table 2, we can find that GPT-4's stronger fundamental reasoning ability allows for a greater tolerance for misclassification risks. At the same time, the gap in performance between Combination-type questions classified as simple questions and those classified as Combination itself is also reduced.

	Combination Problem		Multi-set Problem		Filtering Problem		Simple Problem	
	Num	EX	Num	EX	Num	EX	Num	EX
(Combination,)	236	60.6	-	-	18	44.4	100	86.0
(Combination,Filtering,)	12	83.3	2	0.0	27	44.4	6	50.0
(Combination,Multi-set,)	-	-	2	0.0	-	-	-	-
(Combination,Multi-set,Filtering,)	-	-	4	50.0	-	-	-	-
(Multi-set,)	-	-	18	61.1	2	50.0	-	-
(Multi-set,Filtering,)	-	-	50	72.0	4	100.0	-	-
(Filtering,)	3	100.0	25	52.0	349	79.4	12	83.3
(Simple,)	3	33.3	-	-	6	33.3	155	87.7

Table 17: Fine grained EX results of ambiguity in question types when using ChatGPT.

	Combination Problem		Multi-set Problem		Filtering Problem		Simple Problem	
	Num	EX	Num	EX	Num	EX	Num	EX
(Combination,)	236	79.7	-	-	18	66.7	100	91.0
(Combination,Filtering,)	12	75.0	2	50.0	27	66.7	6	100.0
(Combination,Multi-set,)	-	-	2	50.0	-	-	-	-
(Combination,Multi-set,Filtering,)	-	-	4	50.0	-	-	-	-
(Multi-set,)	-	-	18	72.2	2	100.0	-	-
(Multi-set,Filtering,)	-	-	50	78.0	4	75.0	-	-
(Filtering,)	3	100.0	25	60.0	349	90.0	12	83.3
(Simple,)	3	100.0	-	-	6	83.3	155	91.0

Table 18: Fine grained EX results of ambiguity in question types when using GPT-4.

B.6 Case Study

In this section, we give some detailed cases to show how PTD-SQL assists LLMs in Text-to-SQL. As shown in Table 19, the output of DIN-SQL shows that DIN-SQL doesn't properly handle multi-set queries. Simply using 'OR' for connection may cause conflict issues in multi-table queries. However, PTD-SQL successfully solves the problem by properly learning the decomposition of the problem and using corresponding keywords to connect subqueries correctly. In addition, as shown in Table 20, the sentences generated by DIN-SQL may have the risk of confusing filtering phrases, while PTD-SQL teaches LLM to think hierarchically about filtering logic and answer organization, thus leading to the correct results.

Query	Find all airlines that have flights from both airports 'APG' and 'CVO'.
DIN-SQL output	SELECT Airline FROM airlines JOIN flights ON airlines.uid = flights.Airline WHERE flights.SourceAirport = 'APG' OR flights.SourceAirport = 'CVO'
PTD-SQL output	SELECT Airline FROM airlines WHERE EXISTS (SELECT * FROM flights JOIN airports ON flights.SourceAirport = airports.AirportCode WHERE airports.AirportCode = 'APG' AND airlines.uid = flights.Airline) INTERSECT SELECT Airline FROM airlines WHERE EXISTS (SELECT * FROM flights JOIN airports ON flights.SourceAirport = airports.AirportCode WHERE airports.AirportCode = 'CVO' AND airlines.uid = flights.Airline)
Reasoning	<1> Question Decomposition: In this step, we contemplate how to decompose the query. The query emphasizes the intersection set logic, so we can decompose the question into two subproblems: 1. find all airlines that have flights from airport 'APG'; 2. find all airlines that have flights from airport 'CVO'. <2> Schema Linking: In this step, we identify the tables and columns that should be used based on the requirements of the query and the foreign key relationships. To complete both subproblems, we need to use tables ['airlines', 'flights', 'airports']. <3> Operation: Use 'where' to filter using column 'SourceAirport' in table 'flights' and column 'AirportCode' in table 'airports'. <4> SQL Generation: Use 'intersect' operation to connect the queries of subproblems to form the final SQL statement.

Table 19: A detailed case of complex problem on Spider-dev dataset when using ChatGPT.

C Use of Evaluation Program

Following previous work, we employ a widely-used program to attain EX, which is released by (Zhong et al., 2020). The script used on Spider-dev and Spider-realistic datasets is shown below.

```
python3 evaluation.py --gold gold_path  --pred prediction_path  --db spider/database/
--table spider/tables.json --etype all --plug_value --keep_distinct
```

D Time and Token cost

In this section, we highlight the comparative benefits of PTD-SQL over alternative frameworks concerning time and token usage. Owing to our approach necessitating only a single query, we gain a considerable edge in token efficiency, while simultaneously ensuring effective time management and exceptional outcomes. Although certain optimizations aimed at addressing difficulty granularity and schema linking could potentially enhance PTD-SQL's performance, they would unavoidably result in increased time and token expenditures. The detailed comparison is demonstrated in Table. 21. Data of previous methods are from (Xie et al., 2024).

Query	Which countries in europe have at least 3 car manufacturers?
DIN-SQL output	SELECT CountryName FROM countries WHERE Continent = 'Europe' AND CountryId IN (SELECT Country FROM car_makers GROUP BY Country HAVING COUNT(*) >= 3)
PTD-SQL output	SELECT CountryName FROM car_makers JOIN countries ON car_makers.Country = countries.CountryId JOIN continents ON countries.Continent = continents.ContId GROUP BY Country HAVING COUNT(*) >= 3 AND continents.Continent = 'Europe'
Reasoning	<1> Operation: The query requires the countries in Europe that have at least 3 car manufacturers. So we should apply the 'count' operation to the 'Id' column in the 'car_makers' table, and find the countries with a count of at least 3. Since the unit to which the car manufacturers being counted in the query belong is the country, we should apply the 'group by' operation to the 'Country' column in the 'car_makers' table.<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. In this question, we need to use tables ['car_makers', 'countries', 'continents'].<3> SQL Generation: The query requires the countries in Europe that have at least 3 car manufacturers, so we should select the 'CountryName' column from the 'countries' table. We also need to join the 'car_makers' table with the 'countries' table and the 'continents' table to ensure that we are only considering countries in Europe.

Table 20: A detailed case of combination problem on Spider-dev dataset when using ChatGPT.

Method	Tokens per Query	Inference Time per Query	EX
C3	2803	19.34s	**81.2**
DIN-SQL	9126	<u>4.37s</u>	76.8
DAIL-SQL	**700**	-	79.1
PTD-SQL	<u>1855</u>	**3.34s**	<u>80.3</u>

Table 21: Tokens and time cost comparison using ChatGPT.

E Prompt Design

In this section, we elaborate on the prompt design employed in our study, which is crucial for the effective application of Large Language Models (LLMs) in Text-to-SQL tasks. The prompts serve as guiding questions or statements that help the LLMs focus on specific aspects of the problem and facilitate their learning process.

E.1 Targeted Drilling Bank Auto-construction on Spider

This section can be seen as supplementary materials for section 3.2. We provide all four types of shots generation prompts on Spider dataset, which are leveraged on Spirder-dev and Spider-realistic datasets.

Shots Generation Prompt of Multi-set Problem
You are a powerful text-to-SQL reasoner. Currently, I am seeking to transform intricate text queries into analytical statements that simplify the creation of SQL statements, leading to the generation of the final SQL query. Our current focus lies in the category of multi-set operations. Please learn from the provided examples, design a detailed plan for the text query, and present the resulting SQL query.

Example 1:
## Tables:
Table aircraft, columns = [*,aid,name,distance]
Table certificate, columns = [*,eid,aid]
Table employee, columns = [*,eid,name,salary]
Table flight, columns = [*,flno,origin,destination,distance,departure_date,arrival_date,price,aid]
## Foreign_keys:
[flight.aid = aircraft.aid,certificate.aid = aircraft.aid,certificate.eid = employee.eid]
## Query:
Show names for all employees who have certificates on both Boeing 737-800 and Airbus A340-300.
Let's think step by step.
<1> Question Decomposition: In this step, we contemplate how to decompose the query. The query emphasizes intersection logic, so we can decompose the question into two subproblems: 1. what are the names of employees who have certificates on Boeing 737-800; 2. what are the names of employees who have certificates on Airbus A340-300.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the requirements of the query and the foreign key relationships. To complete the first subproblem, we need to use table 'employee' and 'aircraft'. since table 'employee' and table 'aircraft' do not have a direct foreign key connection, we need to use tables ['employee', 'certificate', 'aircraft']. To complete the second subproblem, we need to use tables ['employee', 'certificate', 'aircraft'] in the same reason.
<3> Operation: Use 'where' to filter using column 'name' in table 'aircraft'.
<4> SQL Generation: Use 'intersect' operation to connect the queries of subproblems to form the final SQL statement.
SQL query: SELECT T1.name FROM Employee AS T1 JOIN Certificate AS T2 ON T1.eid = T2.eid JOIN Aircraft AS T3 ON T3.aid = T2.aid WHERE T3.name = "Boeing 737-800" INTERSECT SELECT T1.name FROM Employee AS T1 JOIN Certificate AS T2 ON T1.eid = T2.eid JOIN Aircraft AS T3 ON T3.aid = T2.aid WHERE T3.name = "Airbus A340-300"

Example 2:
## Tables:
Table station, columns = [*,id,name,lat,long,dock_count,city,installation_date]
Table status, columns = [*,station_id,bikes_available,docks_available,time]
Table trip, columns = [*,id,duration,start_date,start_station_name,start_station_id,end_date, end_station_name,end_station_id,bike_id,subscription_type,zip_code]
Table weather, columns = [*,date,max_temperature_f,mean_temperature_f,min_temperature_f, max_dew_point_f,mean_dew_point_f,min_dew_point_f,max_humidity,mean_humidity,min_humidity, max_sea_level_pressure_inches,mean_sea_level_pressure_inches,min_sea_level_pressure_inches, max_visibility_miles,mean_visibility_miles,min_visibility_miles,max_wind_Speed_mph, mean_wind_speed_mph,max_gust_speed_mph,precipitation_inches,cloud_cover,events,wind_dir_degrees, zip_code]
## Foreign_keys:
[status.station_id = station.id]
## Query:
What are names of stations that have average bike availability above 10 and are not located in San Jose city?
Let's think step by step.
<1> Question Decomposition: In this step, we contemplate how to decompose the query. The query emphasizes difference set logic, so we can decompose decompose the question into two subproblems: 1. what are names of stations that have average bike availability above 10; 2. what are names of stations that are located in San Jose city.

<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the requirements of the query and the foreign key relationships. To complete the first subproblem, we need to use tables ['station', 'status']. To complete the second subproblem, we need to use table ['station'].
<3> Operation: Due to the need for calculating the average bike availability for different stations, we need to perform a 'GROUP BY' operation on the column 'station_id', filter by perform 'HAVING AVG()' on the column 'bikes_available'.
<4> SQL Generation: Use 'except' operation to connect the queries of subproblems to form the final SQL statement.
SQL query: SELECT T1.name FROM station AS T1 JOIN status AS T2 ON T1.id = T2.station_id GROUP BY T2.station_id HAVING avg(bikes_available) > 10 EXCEPT SELECT name FROM station WHERE city = "San Jose"

Example 3:
## Tables:
Table aircraft, columns = [*,aid,name,distance]
Table certificate, columns = [*,eid,aid]
Table employee, columns = [*,eid,name,salary]
Table flight, columns = [*,flno,origin,destination,distance,departure_date,arrival_date,price,aid]
## Foreign_keys:
[flight.aid = aircraft.aid,certificate.aid = aircraft.aid,certificate.eid = employee.eid]
## Query:
Show ids for all employees who don't have a certificate.
Let's think step by step.
<1> Question Decomposition: In this step, we contemplate how to decompose the query. The query emphasizes difference set logic, so we can decompose the question into two subproblems: 1. what are the ids of employees who have certificates; 2. what are the ids of all employees.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the requirements of the query and the foreign key relationships. To complete the first subproblem, we need to use tables ['employee', 'certificate']. To complete the second subproblem, we only need to use the table 'employee'.
<3> Operation: Use 'where' to filter out the employees who have certificates.
<4> SQL Generation: Use 'except' operation to connect the queries of subproblems to form the final SQL statement.
SQL query: SELECT eid FROM employee EXCEPT SELECT eid FROM certificate

Example 4:
## Tables:
Table Campuses, columns = [*,Id,Campus,Location,County,Year]
Table csu_fees, columns = [*,Campus,Year,CampusFee]
Table degrees, columns = [*,Year,Campus,Degrees]
Table discipline_enrollments, columns = [*,Campus,Discipline,Year,Undergraduate,Graduate]
Table enrollments, columns = [*,Campus,Year,TotalEnrollment_AY,FTE_AY]
Table faculty, columns = [*,Campus,Year,Faculty]
## Foreign_keys:
[csu_fees.Campus = Campuses.Id,degrees.Campus = Campuses.Id,discipline_enrollments.Campus = Campuses.Id,enrollments.Campus = Campuses.Id,faculty.Campus = Campuses.Id]
## Query:
Find the name of the campuses that is in Northridge, Los Angeles or in San Francisco, San Francisco.
Let's think step by step.

<1> Question Decomposition: In this step, we contemplate how to decompose the query. The query emphasizes union logic, so we can decompose the question into two subproblems: 1. what are the names of the campuses that are in Northridge, Los Angeles; 2. what are the names of the campuses that are in San Francisco, San Francisco.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the requirements of the query and the foreign key relationships. To complete both subproblems, we need to use table 'Campuses'.
<3> Operation: Use 'where' to filter using column 'Location' in table 'Campuses'.
<4> SQL Generation: Use 'union' operation to connect the queries of subproblems to form the final SQL statement.
SQL query: SELECT Campus FROM Campuses WHERE Location = "Northridge, Los Angeles" UNION SELECT Campus FROM Campuses WHERE Location = "San Francisco, San Francisco"

Table 22: The prompt used for generating targeted drilling bank shots under **multi-set problem** group.

Shots Generation Prompt of Combination Problem

You are a powerful text-to-SQL reasoner. Currently, I am seeking to transform intricate text queries into analytical statements that simplify the creation of SQL statements, leading to the generation of the final SQL query. Our current focus lies in the category of combination operations. Please learn from the provided examples, design a detailed plan for the text query, and present the resulting SQL query.

Example 1:
## Tables:
Table gymnast, columns = [*,Gymnast_ID,Floor_Exercise_Points,Pommel_Horse_Points,Rings_Points, Vault_Points,Parallel_Bars_Points,Horizontal_Bar_Points,Total_Points]
Table people, columns = [*,People_ID,Name,Age,Height,Hometown]
## Foreign_keys:
[gymnast.Gymnast_ID = people.People_ID]
## Query:
How many gymnasts are from each hometown?
Let's think step by step.
<1> Operation: the query requires the number of gymnasts from each hometown, so we should apply the 'count' operation to table 'gymnast', and it does not need sort. Since the unit to which the gymnasts being counted in the query belong is hometown and only table 'people' has column 'Hometown', so we should apply the 'group by' operation to column 'Hometown' in table 'people'.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. Due to the direct foreign key connection between table 'gymnast' and 'people', we need to use tables ['gymnast', 'people'].
<3> SQL Generation: The query requires number of gymnasts from each hometown, so we should select the count and 'hometown' in people.
SQL query: SELECT T2.Hometown, COUNT(*) FROM gymnast AS T1 JOIN people AS T2 ON T1.Gymnast_ID = T2.People_ID GROUP BY T2.Hometown

Example 2:
## Tables:
Table county, columns = [*,County_Id,County_name,Population,Zip_code]
Table election, columns = [*,Election_ID,Counties_Represented,District,Delegate,Party,First_Elected, Committee]
Table party, columns = [*,Party_ID,Year,Party,Governor,Lieutenant_Governor,Comptroller, Attorney_General,US_Senate]
## Foreign_keys:

[election.District = county.County_Id,election.Party = party.Party_ID]
## Query:
Show the name of each party and the corresponding number of delegates from that party.
Let's think step by step.
<1> Operation: The query requires the name of each party and the corresponding number of delegates from that party, so we should apply the 'count' operation to table 'election' for the 'Delegate' column, and it does not need sorting. Since the unit to which the delegates being counted in the query belong is the party and only table 'party' has the column 'Party', so we should apply the 'group by' operation to column 'Party' in table 'party'.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. Due to the direct foreign key connection between table 'election' and 'party'. We need to use tables ['election', 'party'].
<3> SQL Generation: The query requires the name of each party and the corresponding number of delegates from that party, so we should select the 'Party' column in the 'party' table and count the 'Delegate' column in the 'election' table.
SQL query: SELECT T1.Party , COUNT(*) FROM party AS T1 JOIN election AS T2 ON T1.Party_ID = T2.Party GROUP BY T1.Party

Example 3:
## Tables:
Table city, columns = [*,City_ID,Official_Name,Status,Area_km_2,Population,Census_Ranking]
Table competition_record, columns = [*,Competition_ID,Farm_ID,Rank]
Table farm, columns = [*,Farm_ID,Year,Total_Horses,Working_Horses,Total_Cattle,Oxen,
Bulls,Cows,Pigs,Sheep_and_Goats]
Table farm_competition, columns = [*,Competition_ID,Year,Theme,Host_city_ID,Hosts]
## Foreign_keys:
[farm_competition.Host_city_ID = city.City_ID,competition_record.Farm_ID
= farm.Farm_ID,competition_record.Competition_ID = farm_competition.Competition_ID]
## Query:
Show the status of the city that has hosted the greatest number of competitions.
Let's think step by step.
<1> Operation: The query requires the name of each party and the corresponding number of delegates from that party, so we should apply the 'count' operation to table 'election' for the 'Delegate' column, and it does not need sorting. Since the unit to which the delegates being counted in the query belong is the party and only table 'party' has the column 'Party', so we should apply the 'group by' operation to column 'Party' in table 'party'.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. Due to the direct foreign key connection between table 'election' and 'party'. We need to use tables ['election', 'party'].
<3> SQL Generation: The query requires the name of each party and the corresponding number of delegates from that party, so we should select the 'Party' column in the 'party' table and count the 'Delegate' column in the 'election' table.
SQL query: SELECT T1.Party , COUNT(*) FROM party AS T1 JOIN election AS T2 ON T1.Party_ID = T2.Party GROUP BY T1.Party

Example 4:
## Tables:
Table city, columns = [*,City_ID,Official_Name,Status,Area_km_2,Population,Census_Ranking]
Table competition_record, columns = [*,Competition_ID,Farm_ID,Rank]
Table farm, columns = [*,Farm_ID,Year,Total_Horses,Working_Horses,Total_Cattle,Oxen,Bulls,Cows,
Pigs,Sheep_and_Goats]
Table farm_competition, columns = [*,Competition_ID,Year,Theme,Host_city_ID,Hosts]

## Foreign_keys:
[farm_competition.Host_city_ID = city.City_ID,competition_record.Farm_ID= farm.Farm_ID,
competition_record.Competition_ID = farm_competition.Competition_ID]
## Query:
Please show the different statuses, ordered by the number of cities that have each.
Let's think step by step.
<1> Operation: The query requires the different statuses ordered by the number of cities that have each status, so we should apply the 'count' operation to the 'city' table for the 'Status' column, and sort it in ascending order. Since the unit to which the statuses being counted in the query belong is the city, we should apply the 'group by' operation to the 'Status' column in the 'city' table.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. In this question, we only need to use table ['city'].
<3> SQL Generation: The query requires the different statuses ordered by the number of cities that have each status, so we should select the 'Status' column in the 'city' table. The query does not require the count of cities so it is only used for filtering and not selected.
SQL query: SELECT Status FROM city GROUP BY Status ORDER BY COUNT(*) ASC

Table 23: The prompt used for generating targeted drilling bank shots under **combination problem** group.

Shots Generation Prompt of Filtering Problem

You are a powerful text-to-SQL reasoner. Currently, I am seeking to transform intricate text queries into analytical statements that simplify the creation of SQL statements, leading to the generation of the final SQL query. Our current focus lies in the category of filtering problem. Please learn from the provided examples, design a detailed plan for the text query, and present the resulting SQL query.

Example 1:
## Tables:
Table city, columns = [*,City_ID,Official_Name,Status,Area_km_2,Population,Census_Ranking]
Table competition_record, columns = [*,Competition_ID,Farm_ID,Rank]
Table farm, columns = [*,Farm_ID,Year,Total_Horses,Working_Horses,Total_Cattle,Oxen,Bulls,Cows,Pigs,Sheep_and_Goats]
Table farm_competition, columns = [*,Competition_ID,Year,Theme,Host_city_ID,Hosts]
## Foreign_keys:
[farm_competition.Host_city_ID = city.City_ID,competition_record.Farm_ID = farm.Farm_ID,
competition_record.Competition_ID = farm_competition.Competition_ID]
## Query:
Return the hosts of competitions for which the theme is not Aliens?
Let's think step by step.
<1> Decomposition: The query requires filtering on column 'theme', so we should apply the 'where' to column 'theme' and then return the hosts of selected competition.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. Since table 'farm_competition' has columns 'Theme' and 'Hosts', we only need table 'farm_competition'.
<3> SQL Generation: Directly write the sql using 'where'.
SQL query: SELECT Hosts FROM farm_competition WHERE Theme != 'Aliens'

Example 2:
## Tables:
Table Allergy_Type, columns = [*,Allergy,AllergyType]
Table Has_Allergy, columns = [*,StuID,Allergy]
Table Student, columns = [*,StuID,LName,Fname,Age,Sex,Major,Advisor,city_code]

## Foreign_keys:
[Has_Allergy.Allergy = Allergy_Type.Allergy,Has_Allergy.StuID = Student.StuID]
## Query:
How many female students have milk or egg allergies?
Let's think step by step.
<1> Decomposition: Firstly, we filter candidates using column 'Sex' in table 'Student' and column 'Allergy' in table 'Has_Allergy'. Secondly, we use 'count' to calculate the number of selected female students.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. Since table 'Student' and table 'Has_Allergy' have direct foreign keys, so we need tables ['Student', 'Has_Allergy'].
<3> SQL Generation: We need to join the 'Student' and 'Has_Allergy' tables on the 'StuID' column. Then, we filter the rows where 'Sex' is 'F' and 'Allergy' is either 'Milk' or 'Eggs'. Finally, we count the number of rows that meet these conditions.
SQL query: SELECT count(*) FROM has_allergy AS T1 JOIN Student AS T2 ON T1.StuID = T2.StuID WHERE T2.sex = 'F' AND T1.allergy = 'Milk' or T1.allergy = 'Eggs'

Example 3:
## Tables:
Table station, columns = [*,id,name,lat,long,dock_count,city,installation_date]
Table status, columns = [*,station_id,bikes_available,docks_available,time]
Table trip, columns = [*,id,duration,start_date,start_station_name,start_station_id,end_date, end_station_name,end_station_id,bike_id,subscription_type,zip_code]
Table weather, columns = [*,date,max_temperature_f,mean_temperature_f,min_temperature_f, max_dew_point_f,mean_dew_point_f,min_dew_point_f,max_humidity,mean_humidity,min_humidity, max_sea_level_pressure_inches,mean_sea_level_pressure_inches,min_sea_level_pressure_inches, max_visibility_miles,mean_visibility_miles,min_visibility_miles,max_wind_Speed_mph, mean_wind_speed_mph,max_gust_speed_mph,precipitation_inches,cloud_cover,events, wind_dir_degrees,zip_code]
## Foreign_keys:
[status.station_id = station.id]
## Query:
How many trips did not end in San Francisco?
Let's think step by step.
<1> Decomposition: The query requires filtering on trips that did not end in San Francisco. Firstly, we need to identify the stations located in San Francisco. Secondly, we need to filter trips based on their end_station_id.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. In the first step, we need to select id from table 'station' where city = 'San Francisco'. In the second step, we need to select id from table 'trip' and filter by end_station_id.
<3> SQL Generation: Use 'where' to filter stations in San Francisco, and then use 'not in' to filter trips that did not end in San Francisco.
SQL query: SELECT COUNT(*) FROM trip WHERE end_station_id NOT IN (SELECT id FROM station WHERE city = 'San Francisco')

Example 4:
## Tables:
Table concert, columns = [*,concert_ID,concert_Name,Theme,Stadium_ID,Year]
Table singer, columns = [*,Singer_ID,Name,Country,Song_Name,Song_release_year,Age,Is_male]
Table singer_in_concert, columns = [*,concert_ID,Singer_ID]
Table stadium, columns = [*,Stadium_ID,Location,Name,Capacity,Highest,Lowest,Average]

## Foreign_keys:
[concert.Stadium_ID = stadium.Stadium_ID,singer_in_concert.Singer_ID = singer.Singer_ID, singer_in_concert.concert_ID = concert.concert_ID]
## Query:
Find the number of concerts happened in the stadium with the highest capacity .
Let's think step by step.
<1> Decomposition: Firstly, we need to find the stadium with the highest capacity. Secondly, we need to filter concerts based on their happened stadium and count them.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. In the first step, we need to select stadium_id with highest capacity from table 'stadium'. In the second step, we need to filter stadium_id from table 'concert' and count them.
<3> SQL Generation: Use 'order by' and 'desc' to select stadium with highest capacity, and then use 'where' to filter concert and count it.
SQL query: select count(*) from concert where stadium_id = (select stadium_id from stadium order by capacity desc limit 1)

Table 24: The prompt used for generating targeted drilling bank shots under **filtering problem** group.

Shots Generation Prompt of Other Simple Problem

You are a powerful text-to-SQL reasoner. Currently, I am seeking to transform intricate text queries into analytical statements that simplify the creation of SQL statements, leading to the generation of the final SQL query.

Example 1:
## Tables:
Table department, columns = [*,Department_ID,Name,Creation,Ranking,Budget_in_Billions,Num_Employees]
Table head, columns = [*,head_ID,name,born_state,age]
Table management, columns = [*,department_ID,head_ID,temporary_acting]
## Foreign_keys:
[management.head_ID = head.head_ID,management.department_ID = department.Department_ID]
## Query:
List the name, born state and age of the heads of departments ordered by age.
SQL query: SELECT name , born_state , age FROM head ORDER BY age

Example 2:
## Tables:
Table department, columns = [*,Department_ID,Name,Creation,Ranking,Budget_in_Billions,Num_Employees]
Table head, columns = [*,head_ID,name,born_state,age]
Table management, columns = [*,department_ID,head_ID,temporary_acting]
## Foreign_keys:
[management.head_ID = head.head_ID,management.department_ID = department.Department_ID]
## Query:
List the creation year, name and budget of each department.
SQL query: SELECT creation , name , budget_in_billions FROM department

Example 3:
## Tables:
Table race, columns = [*,Race_ID,Name,Class,Date,Track_ID]
Table track, columns = [*,Track_ID,Name,Location,Seating,Year_Opened]
## Foreign_keys:

[race.Track_ID = track.Track_ID]
## Query:
Show year where a track with a seating at least 5000 opened and a track with seating no more than 4000 opened.
SQL query: SELECT year_opened FROM track WHERE seating BETWEEN 4000 AND 5000

Example 4:
## Tables:
Table Available_Policies, columns = [*,Policy_ID,policy_type_code,Customer_Phone]
Table Claims, columns = [*,Claim_ID,FNOL_ID,Effective_Date]
Table Customers, columns = [*,Customer_ID,Customer_name]
Table Customers_Policies, columns = [*,Customer_ID,Policy_ID,Date_Opened,Date_Closed]
Table First_Notification_of_Loss, columns = [*,FNOL_ID,Customer_ID,Policy_ID,Service_ID]
Table Services, columns = [*,Service_ID,Service_name]
Table Settlements, columns = [*,Settlement_ID,Claim_ID,Effective_Date,Settlement_Amount]
## Foreign_keys:
[Customers_Policies.Policy_ID = Available_Policies.Policy_ID,Customers_Policies.Customer_ID = Customers.Customer_ID,First_Notification_of_Loss.Customer_ID = Customers_Policies.Customer_ID,
First_Notification_of_Loss.Policy_ID = Customers_Policies.Policy_ID,
First_Notification_of_Loss.Service_ID = Services.Service_ID,
Claims.FNOL_ID = First_Notification_of_Loss.FNOL_ID,Settlements.Claim_ID = Claims.Claim_ID]
## Query:
Which policy type has the most records in the database?
SQL query: SELECT policy_type_code FROM available_policies GROUP BY policy_type_code ORDER BY count(*) DESC LIMIT 1

Table 25: The prompt used for generating targeted drilling bank shots under **other simple problem**.

E.2 Targeted Drilling Bank Auto-construction on BIRD

In this section, we provide the specific shots generation prompt for three types of problems on BIRD dataset.

Shots Generation Prompt of Filtering Problem

You are a powerful text-to-SQL reasoner. Currently, I am seeking to transform intricate text queries into analytical statements that simplify the creation of SQL statements, leading to the generation of the final SQL query. Our current focus lies in the category of filtering problems. Please learn from the provided examples, design a detailed plan for the text query, and present the resulting SQL query.

Example 1:
## Tables:
Table frpm, columns = [*,CDSCode,Academic Year,County Code,District Code,School Code,County Name,District Name,School Name,District Type,School Type,Educational Option Type,NSLP Provision Status,Charter School (Y/N),Charter School Number,Charter Funding Type,IRC,Low Grade,High Grade,Enrollment (K-12),Free Meal Count (K-12),Percent (%) Eligible Free (K-12),FRPM Count (K-12),Percent (%) Eligible FRPM (K-12),Enrollment (Ages 5-17),Free Meal Count (Ages 5-17),Percent (%) Eligible Free (Ages 5-17),FRPM Count (Ages 5-17),Percent (%) Eligible FRPM (Ages 5-17),2013-14 CALPADS Fall 1 Certification Status]
Table satscores, columns = [*,cds,rtype,sname,dname,cname,enroll12,NumTstTakr,AvgScrRead, AvgScrMath,AvgScrWrite,NumGE1500]
Table schools, columns = [*,CDSCode,NCESDist,NCESSchool,StatusType,County,District,School,Street, StreetAbr,City,Zip,State,MailStreet,MailStrAbr,MailCity,MailZip,MailState,Phone,Ext,Website,OpenDate,

ClosedDate,Charter,CharterNum,FundingType,DOC,DOCType,SOC,SOCType,EdOpsCode,EdOpsName, EILCode,EILName,GSoffered,GSserved,Virtual,Magnet,Latitude,Longitude,AdmFName1,AdmLName1, AdmEmail1,AdmFName2,AdmLName2,AdmEmail2,AdmFName3,AdmLName3,AdmEmail3,LastUpdate]
## Foreign_keys:
[frpm.CDSCode = schools.CDSCode,satscores.cds = schools.CDSCode]
## Query:
Which schools served a grade span of Kindergarten to 9th grade in the county of Los Angeles and what is its Percent (%) Eligible FRPM (Ages 5-17)?
Let's think step by step.
<1> Decomposition: The query requires finding schools in Los Angeles County that serve a grade span of Kindergarten to 9th grade and calculating their Percent (%) Eligible FRPM (Ages 5-17).
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. We need to select School from table 'schools' and filter by County and GSserved columns. Then, we need to calculate the Percent (%) Eligible FRPM (Ages 5-17) using the FRPM Count (Ages 5-17) and Enrollment (Ages 5-17) columns from the 'frpm' table.
<3> SQL Generation: We can use an INNER JOIN to combine the 'frpm' and 'schools' tables on the 'CDSCode' column. Then, we can filter the rows where the County is 'Los Angeles' and GSserved is 'K-9'. Finally, we can calculate the Percent (%) Eligible FRPM (Ages 5-17) by dividing the FRPM Count (Ages 5-17) by the Enrollment (Ages 5-17) and multiplying by 100.
SQL query: SELECT T2.School, T1.‘FRPM Count (Ages 5-17)‘ * 100 / T1.‘Enrollment (Ages 5-17)‘ FROM frpm AS T1 INNER JOIN schools AS T2 ON T1.CDSCode = T2.CDSCode WHERE T2.County = 'Los Angeles' AND T2.GSserved = 'K-9'

Example 2:
## Tables:
Table account, columns = [*,account_id,district_id,frequency,date]
Table card, columns = [*,card_id,disp_id,type,issued]
Table client, columns = [*,client_id,gender,birth_date,district_id]
Table disp, columns = [*,disp_id,client_id,account_id,type]
Table district, columns = [*,district_id,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16]
Table loan, columns = [*,loan_id,account_id,date,amount,duration,payments,status]
Table order, columns = [*,order_id,account_id,bank_to,account_to,amount,k_symbol]
Table trans, columns = [*,trans_id,account_id,date,type,operation,amount,balance,k_symbol,bank,account]
## Foreign_keys:
[account.district_id = district.district_id,card.disp_id = disp.disp_id,client.district_id = district.district_id,disp.client_id = client.client_id,disp.account_id = account.account_id,loan.account_id = account.account_id,order.account_id = account.account_id,trans.account_id = account.account_id]
## Query:
For loan amount less than USD100,000, what is the percentage of accounts that is still running with no issue.
Let's think step by step.
<1> Decomposition: The query requires finding the percentage of accounts with loans less than USD 100,000 that are still running with no issue.
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. We need to filter the 'loan' table by the 'amount' column and then calculate the percentage of accounts with a 'status' of 'C', which means running with no issue.
<3> SQL Generation: We can use the SELECT statement to filter the 'loan' table by the 'amount' column, where the amount is less than 100,000. Then, we can calculate the percentage of accounts with a 'status' of 'C' by dividing the number of accounts with status 'C' by the total number of accounts and multiplying by 100.
SQL query: SELECT CAST(SUM(status = 'C') AS REAL) * 100 / COUNT(amount) FROM loan WHERE amount < 100000

Example 3:
## Tables:
Table atom, columns = [*,atom_id,molecule_id,element]
Table bond, columns = [*,bond_id,molecule_id,bond_type]
Table connected, columns = [*,atom_id,atom_id2,bond_id]
Table molecule, columns = [*,molecule_id,label]
## Foreign_keys:
[atom.molecule_id = molecule.molecule_id,bond.molecule_id = molecule.molecule_id,connected.bond_id = bond.bond_id,connected.atom_id2 = atom.atom_id,connected.atom_id = atom.atom_id]
## Query:
What is the percentage of carcinogenic molecules in triple type bonds?
Let's think step by step.
<1> Decomposition: The query requires finding the percentage of carcinogenic molecules (indicated by '+') in triple type bonds (indicated by '#').
<2> Schema Linking: In this step, we identify the tables and columns that should be used based on the first step and the foreign key relationships. We need to select the 'label' column from the 'molecule' table and the 'bond_type' column from the 'bond' table. We also need to use the 'molecule_id' column from the 'atom', 'molecule', and 'bond' tables to join these tables together.
<3> SQL Generation: We can use an INNER JOIN to combine the 'atom', 'molecule', and 'bond' tables on the 'molecule_id' column. Then, we can filter the rows where the bond_type is '#'. Finally, we can calculate the percentage of carcinogenic molecules by dividing the number of distinct carcinogenic molecules by the total number of distinct molecules and multiplying by 100.
SQL query: SELECT CAST(COUNT(DISTINCT CASE WHEN T2.label = '+' THEN T2.molecule_id ELSE NULL END) AS REAL) * 100 / COUNT(DISTINCT T2.molecule_id) FROM atom AS T1 INNER JOIN molecule AS T2 ON T1.molecule_id = T2.molecule_id INNER JOIN bond AS T3 ON T2.molecule_id = T3.molecule_id WHERE T3.bond_type = '#'

Table 26: The prompt used for generating targeted drilling bank shots under **filtering problem** on BIRD dataset.

Shots Generation Prompt of Combination Problem

You are a powerful text-to-SQL reasoner. Currently, I am seeking to transform intricate text queries into analytical statements that simplify the creation of SQL statements, leading to the generation of the final SQL query. Our current focus lies in the category of combination problems. Please learn from the provided examples, design a detailed plan for the text query, and present the resulting SQL query.

Example 1:
## Tables:
Table badges, columns = [*,Id,UserId,Name,Date]
Table comments, columns = [*,Id,PostId,Score,Text,CreationDate,UserId,UserDisplayName]
Table postHistory, columns = [*,Id,PostHistoryTypeId,PostId,RevisionGUID,CreationDate,UserId,Text, Comment,UserDisplayName]
Table postLinks, columns = [*,Id,CreationDate,PostId,RelatedPostId,LinkTypeId]
Table posts, columns = [*,Id,PostTypeId,AcceptedAnswerId,CreaionDate,Score,ViewCount,Body, OwnerUserId,LasActivityDate,Title,Tags,AnswerCount,CommentCount,FavoriteCount,LastEditorUserId, LastEditDate,CommunityOwnedDate,ParentId,ClosedDate,OwnerDisplayName,LastEditorDisplayName]
Table tags, columns = [*,Id,TagName,Count,ExcerptPostId,WikiPostId]
Table users, columns = [*,Id,Reputation,CreationDate,DisplayName,LastAccessDate,WebsiteUrl,Location, AboutMe,Views,UpVotes,DownVotes,AccountId,Age,ProfileImageUrl]
Table votes, columns = [*,Id,PostId,VoteTypeId,CreationDate,UserId,BountyAmount]

## Foreign_keys:
[badges.UserId = users.Id,comments.UserId = users.Id,comments.PostId = posts.Id,postHistory.UserId = users.Id,postHistory.PostId = posts.Id,postLinks.RelatedPostId = posts.Id,postLinks.PostId = posts.Id,posts.ParentId = posts.Id,posts.OwnerUserId = users.Id,posts.LastEditorUserId = users.Id,tags.ExcerptPostId = posts.Id,votes.UserId = users.Id,votes.PostId = posts.Id]
## Query:
Which is the most valuable post in 2010? Please give its id and the owner's display name.
Let's think step by step.
Firstly, the query requires the most valuable post, and the value is related to FavoriteCount column of table 'posts', so we should apply order by to it.
Secondly, we need to retrieve the ids and owner's display name of posts selected from first step.
Finally, based on the above analysis and requirements in user query, we only need to use tables 'users' and 'posts'.
SQL query: SELECT T2.OwnerUserId, T1.DisplayName FROM users AS T1 INNER JOIN posts AS T2 ON T1.Id = T2.OwnerUserId WHERE STRFTIME('%Y', T1.CreationDate) = '2010' ORDER BY T2.FavoriteCount DESC LIMIT 1

Example 2:
## Tables:
Table customers, columns = [*,CustomerID,Segment,Currency]
Table gasstations, columns = [*,GasStationID,ChainID,Country,Segment]
Table products, columns = [*,ProductID,Description]
Table transactions_1k, columns = [*,TransactionID,Date,Time,CustomerID,CardID,GasStationID,ProductID,Amount,Price]
Table yearmonth, columns = [*,CustomerID,Date,Consumption]
## Foreign_keys:
[yearmonth.CustomerID = customers.CustomerID]
## Query:
Which year recorded the most consumption of gas paid in CZK?
Let's think step by step.
Firstly, the query requires the most consumption of gas paid in CZK, and the consumption is related to the Consumption column of table 'yearmonth'. Moreover, we need to consider the currency, which is in the table 'customers'. So, we should join these two tables based on the CustomerID.
Secondly, we need to filter the records where the currency is CZK. We can do this using a WHERE clause to filter records from the 'customers' table.
Thirdly, we need to group the results by year, which can be extracted from the Date column of the 'yearmonth' table. We can use the SUBSTRING function to get the year from the Date, and then use GROUP BY to group the records by year.
Finally, we need to order the results by the sum of consumption in descending order and select the top record to get the year with the most consumption of gas paid in CZK.
SQL query: SELECT SUBSTRING(T2.Date, 1, 4) as Year FROM customers AS T1 INNER JOIN yearmonth AS T2 ON T1.CustomerID = T2.CustomerID WHERE T1.Currency = 'CZK' GROUP BY Year ORDER BY SUM(T2.Consumption) DESC LIMIT 1

Example 3:
## Tables:
Table circuits, columns = [*,circuitId,circuitRef,name,location,country,lat,lng,alt,url]
Table constructorResults, columns = [*,constructorResultsId,raceId,constructorId,points,status]
Table constructorStandings, columns = [*,constructorStandingsId,raceId,constructorId,points,position,positionText,wins]
Table constructors, columns = [*,constructorId,constructorRef,name,nationality,url]
Table driverStandings, columns = [*,driverStandingsId,raceId,driverId,points,position,positionText,wins]

Table drivers, columns = [*,driverId,driverRef,number,code,forename,surname,dob,nationality,url]
Table lapTimes, columns = [*,raceId,driverId,lap,position,time,milliseconds]
Table pitStops, columns = [*,raceId,driverId,stop,lap,time,duration,milliseconds]
Table qualifying, columns = [*,qualifyId,raceId,driverId,constructorId,number,position,q1,q2,q3]
Table races, columns = [*,raceId,year,round,circuitId,name,date,time,url]
Table results, columns = [*,resultId,raceId,driverId,constructorId,number,grid,position,positionText,
positionOrder,points,laps,time,milliseconds,fastestLap,rank,fastestLapTime,
fastestLapSpeed,statusId]
Table seasons, columns = [*,year,url]
Table status, columns = [*,statusId,status]
## Foreign_keys:
[races.circuitId = circuits.circuitId,races.year = seasons.year,constructorResults.constructorId = constructors.constructorId,constructorResults.raceId = races.raceId,constructorStandings.constructorId = constructors.constructorId,constructorStandings.raceId = races.raceId,driverStandings.driverId = drivers.driverId,driverStandings.raceId = races.raceId,lapTimes.driverId = drivers.driverId,lapTimes.raceId = races.raceId,pitStops.driverId = drivers.driverId,pitStops.raceId = races.raceId,qualifying.constructorId = constructors.constructorId,qualifying.driverId = drivers.driverId,qualifying.raceId = races.raceId,results.statusId = status.statusId,results.constructorId = constructors.constructorId,results.driverId = drivers.driverId,results.raceId = races.raceId]
## Query:
List out top 10 Spanish drivers who were born before 1982 and have the latest lap time.
Let's think step by step.
Firstly, the query requires the top 10 Spanish drivers born before 1982. We need to consider the drivers' nationality and date of birth, which are in the table 'drivers'. We can use a WHERE clause to filter records based on these conditions.
Secondly, we need to find the latest lap time for these filtered drivers. The lap time information is in the table 'pitStops'. So, we should join the 'drivers' and 'pitStops' tables based on the driverId.
Thirdly, we need to order the results by the lap time in descending order to get the latest lap time. We can use the ORDER BY clause for this purpose.
Finally, we need to select the top 10 records to get the required result.
SQL query: SELECT T2.driverId FROM pitStops AS T1 INNER JOIN drivers AS T2 on T1.driverId = T2.driverId WHERE T2.nationality = 'Spanish' AND STRFTIME('%Y', T2.dob) < '1982' ORDER BY T1.time DESC LIMIT 10

Table 27: The prompt used for generating targeted drilling bank shots under **combination problem** on BIRD dataset.

E.3 QGP Prompt

In this section, we demonstrate our few-shot instruction prompt using in QGP sub-task (Table 28).

Few-shot Prompt used in QGP sub-task
You are a text-to-sql expert. Your task is to classify text-based queries. The types are defined as follows: 1. Set operations, which require complex logical connections between multiple conditions and often involve the use of intersect, except, union, and other operations; 2. Combination operations, which require grouping of specific objects and finding the maximum value or sorting, often achieved using GROUP BY; 3. Filtering problems, which select targets based on specific judgment conditions, generally completed using where statements; 4. Other simple problems, including simple retrieval and sorting.

Your task is to judge the query step by step to see if it belongs to a certain category. For example, if you think the query has the characteristics of the first type, then classify it as the first type without considering the subsequent types. If you think the query does not have the characteristics of the first type but has the second type, then classify it as the second type without considering the subsequent types.

## Example 1:
What are the ids of the students who either registered or attended a course?
Reason: We firstly consider Set operations. The query can be considered union logic which finds students that registered or attended a course, so it is classified as Set operations.
Type: Multi-set operations

## Example 2:
List the states where both the secretary of 'Treasury' department and the secretary of 'Homeland Security' were born.
Reason: We firstly consider Set operations. The query can be considered intersection logic which requires the intersection of states that 'Treasury' and 'Homeland Security' were born, so it is classified as Set operations.
Type: Multi-set operations

## Example 3:
Find all the zip codes in which the max dew point have never reached 70.
Reason: We firstly consider Set operations. The query can be seen as a difference logic, which removes zip codes that have reached a dew point of 70 from all zip codes, so it is classified as Set operations.
Type: Multi-set operations

## Example 4:
Find the name of customers who do not have an saving account.
Reason: We firstly consider Set operations. The query can be consiederd difference logic, which removes customers having an saving account from all customers, so it is classified as Set operations.
Type: Multi-set operations

## Example 5:
Which origin has most number of flights?
Reason: We firstly consider Set operations. This query does not involve logical connection relationships. We secondly consider Combination operations. This query requires statistical counting of flights within different origins, so it is classified as Combination operations.
Type: Combination operations

## Example 6:
Which course is enrolled in by the most students? Give me the course name.
Reason: We firstly consider Set operations. This query does not involve logical connection relationships. We secondly consider Combination operations. This query requires statistical counting of students within different courses, so it is classified as Combination operations.
Type: Combination operations

## Example 7:
Find the name of the train whose route runs through greatest number of stations.
Reason: We firstly consider Set operations. This query does not involve logical connection relationships. We secondly consider Combination operations. This query requires statistical counting of running stations of different trains, so it is classified as Combination operations.
Type: Combination operations

## Example 8:
What are the names of musicals with nominee "Bob Fosse"?
Reason: We firstly consider Set operations. This query does not involve logical connection relationships. We secondly consider Combination operations. This query does not involve group count. We thirdly consider Filtering problems. This query needs to filter musicals based on the name of the nomenee, so it is classified as Filtering problems.
Type: Filtering problems

## Example 9:
How many distinct kinds of camera lenses are used to take photos of mountains in the country 'Ethiopia'?
Reason: We firstly consider Set operations. This query does not involve logical connection relationships. We secondly consider Combination operations. This query does not involve group count. We thirdly consider Filtering problems. This query needs to filter camera lenses based on the utilization on mountains in country 'Ethiopia', so it is classified as Filtering problems.
Type: Filtering problems

## Example 10:
How many products are there?
Reason: We firstly consider Set operations. This query does not involve logical connection relationships. We secondly consider Combination operations. This query does not involve group count. We thirdly consider Filtering problems. This query does not involve filter criteria. So it is classified as Other simple problems.
Type: Other simple problems

Table 28: The few-shot prompt for CoT and fine-tuning used in QGP sub-task.