Text-to-SQL Systems in the Era of Advanced Large Language Models

by

Mohammadreza Pourreza

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Text-to-SQL conversion, the process of transforming natural language queries into executable SQL commands, stands at the forefront of bridging human linguistic capabilities with the structured logic of databases. This dissertation embarks on a journey to elevate text-to-SQL systems to new heights, aiming to narrow the performance gap between human expertise and automated systems within the landscape of large language models (LLMs). Our endeavor unfolds in three pivotal stages. Initially, we harness the power of cutting-edge proprietary LLMs such as GPT-4, enhancing their prowess through an in-context learning methodology tailored explicitly for text-to-SQL tasks. Our proposed method is the state-of-the-art Text-to-SQL method which improved upon the previous works by %5 execution accuracy. Recognizing the critical importance of privacy and the economic considerations tied to proprietary LLMs, we then introduce a decomposed, two-stage supervised fine-tuning approach. This method not only optimizes the efficiency of smaller LLMs but also achieves performance metrics on par with their larger counterparts. Using our proposed two-step method, a small LLMs with 7B parameter can achieve comparable results to GPT-4. Finally, our thorough examination and critique of existing text-to-SQL benchmarks, using human annotation and Standard SQL validation, illuminates the path for future research, highlighting the necessity for more comprehensive and accurate evaluation frameworks. Our analysis of the current Text-to-SQL benchmarks reveals critical limitations, which can hinder further advancement in this domain. By proposing methodologies in

the realm of LLMs and shedding light on areas ripe for further advancement, this thesis aspires to inch closer to the elusive goal of achieving human-level proficiency in Text-to-SQL translation.

Preface

The core chapters of this thesis draw upon papers that are either published or currently under review. Specifically, Chapters 2 and 4 originate from papers presented at the NeurIPS 2023 and EMNLP 2023 conferences, respectively [1], [2]. Chapter 3 elaborates on research encapsulated in a paper that is presently undergoing the review process [3].

[1] M. Pourreza and D. Rafiei, "Evaluating cross-domain text-to-sql models and benchmarks," in *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[2] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed in-context learning of text-to-sql with self-correction," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[3] M. Pourreza and D. Rafiei, "DTS-SQL: Decomposed text-to-sql with small large language models," *arXiv preprint arXiv:2402.01117*, 2024.

To my family for their unwavering support.

Acknowledgements

Embarking on this journey, I was unaware of the challenges and opportunities that awaited me in the realm of fast-paced research, enriched by contributions from both industry professionals and the academic community. This journey, however, would have been insurmountable without the support and guidance of numerous individuals to whom I owe my deepest gratitude.

Foremost, I extend my heartfelt appreciation to my advisor, Professor Davood Rafiei. His unwavering support and invaluable mentorship were the beacons that guided me through unfamiliar territories. Professor Rafiei's profound expertise and patience played a pivotal role in navigating the complexities of this research. His readiness to invest his time to teach me have been instrumental in my journey. The completion of this thesis would have been unthinkable without his generous assistance and enduring support.

I am also indebted to my Master's committee members, Dr. Jorg Sander and Dr. Lili Mou, for their participation in my thesis defense and their insightful feedback. Their engagement and valuable contributions have significantly enhanced the quality of my work. Additionally, my sincere thanks go to Dr. Schuurmans and Dr. Szepesvari for their insightful and constructive feedback on the second chapter of this thesis, which have been immensely beneficial in refining my research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"Data is the oil of the 21st century, and analytics is its combustion engine." This insightful observation, often credited to Peter Sondergaard from Gartner Research, highlights the critical importance and transformative influence of data in today's world. In an era defined by rapid advancements and the need for swift decision-making, the ability to derive meaningful insights from data stands as a cornerstone of success. Since the inception of the Structured Query Language (SQL) in 1986, businesses of all sizes have embraced relational databases for data storage and have relied on SQL for querying and extracting valuable insights. SQL revolutionized data handling by offering two pivotal benefits over previous read-write APIs such as ISAM or VSAM: first, it enabled the retrieval of multiple records through a single command; second, it obviated the necessity to specify the path to a record, whether indexed or not, thereby streamlining data access [70].

Crafting SQL queries demands a deep understanding of the SQL language, leading many companies to hire data scientists specifically for this purpose. The role of a data scientist often entails interpreting a natural language question from a business user, pinpointing the relevant tables, columns, and database entities related to the query, and then iteratively developing and refining a SQL query to extract the information that answers the business question. Text to SQL technology streamlines this process by converting natural language queries into SQL commands, thus enabling users to interact with databases without the need to master SQL syntax. Situated at the crossroads

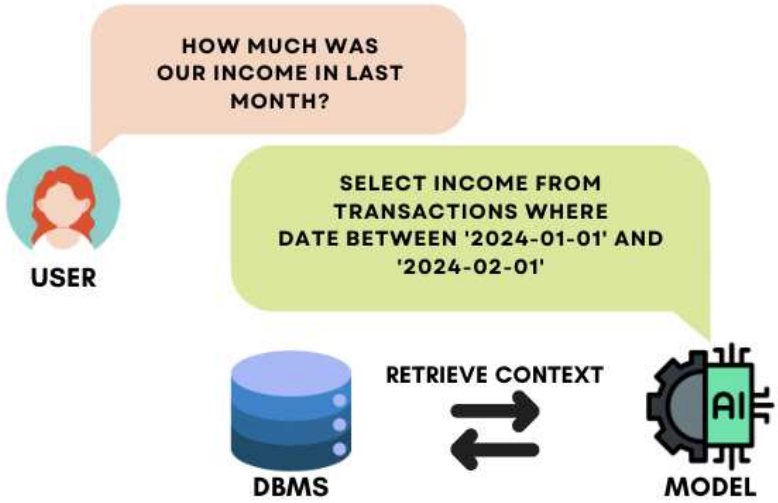

Figure 1.1: An example of text-to-SQL task.

of natural language processing (NLP) and database management, text to SQL aims to facilitate data retrieval by allowing questions or requests to be made in everyday language. Utilizing sophisticated algorithms and machine learning models, text to SQL systems decipher the semantics of natural language inputs, deduce the query's intent, and translate it into the corresponding SQL commands for database execution. This innovation finds broad application, from improving database management system interfaces to making data analysis more intuitive in business intelligence tools. By making database querying more accessible to non-technical users, text to SQL is instrumental in narrowing the divide between intricate database systems and users, thereby enhancing the accessibility of data-driven insights across different sectors. Figure 1.1 illustrates an example of the text-to-SQL task.

There are several challenges associated with the text to SQL task, marking it as a promising area for further research and development. As highlighted by [20], despite significant advancements in the NLP field through the introduction of Large Language Models (LLMs), there remains a substantial disparity between human performance and the capabilities of top-performing LLMs. These challenges can be broadly categorized into two main areas. The first pertains to the issues linked with formulating questions, akin to the difficulties encountered in question-answering domains. Non-technical users, when posing

questions, often lack a detailed understanding of how data is organized within relational databases. Consequently, their questions are prone to ambiguities, complicating the text to SQL generation process and the evaluation of text to SQL systems—sometimes introducing subjectivity. In Chapter 4, we will delve deeper into these issues through human annotation and standard SQL validation techniques. In addition to question-related challenges, there are obstacles related to how database information is presented to models and instructing models on interpreting data stored in databases. Data in relational databases often spans millions of rows, making the linearization of tables and feeding this information into models impractical. Furthermore, industrial databases frequently contain noisy, disorganized data, including columns with ambiguous names that poorly represent the contained data. Such challenges largely arise because, in many organizations, the individual responsible for database creation also undertakes SQL query writing, leading to numerous implicit assumptions that are difficult to identify simply by examining the data.

Initial endeavors in text-to-SQL parsing by the database community [78] achieved notable progress, albeit requiring extensive human engineering and frequent user system interactions. Designing SQL templates beforehand for a wide array of scenarios or domains proves to be challenging. Recent years have witnessed significant advancements in deep learning and the availability of vast training datasets, which have considerably improved text-to-SQL parsing through neural generation models. A notable example of such methods is the sequence-to-sequence (Seq2Seq) model [60], which autonomously learns the mapping function from an input natural language (NL) question to the corresponding SQL output via encoder-decoder frameworks. The fundamental concept involves developing an encoder to comprehend the input NL questions and associated table schema, and employing a grammar-based neural decoder to predict the desired SQL. Seq2Seq-based approaches have risen to prominence in text-to-SQL parsing largely because they can be trained end-to-end, thus reducing the reliance on specialized domain knowledge [52].

Various neural generation models have evolved to enhance both the encoder and decoder for more effective text-to-SQL translation. On the encoding side,

multiple general neural networks are applied for comprehensive reasoning over natural language queries and database schemas. For instance, IRNet [23] uses a bi-directional LSTM [22] combined with a self-attention [64] mechanism to encode the question and table schema separately. RYANSQL [7] employs a convolutional neural network with dense connections for encoding. The advent of pre-trained language models (PLMs) marked a significant evolution; SQLova [28] was among the first to utilize PLMs like BERT for encoding. Further advancements saw the use of graph neural networks in RAT-SQL [66], SADGA [3], and LGESQL [4] to capture the relational structure between database schema and queries.

On the decoding side, approaches to SQL generation fall into two main categories: sketch-based and generation-based methods [72]. Sketch-based methods, such as those employed by SQLova [28], decompose the SQL generation task into sub-modules, each responsible for a specific type of prediction slot. These sub-modules are later combined to generate the final SQL query.

However, the advent of large language models, with their impressive ability for few-shot or zero-shot in-context learning [2], and their exceptional performance across a wide array of tasks, has led researchers to pivot towards leveraging these models for text to SQL conversion. Initial investigations in this area primarily focused on the zero-shot in-context learning capabilities of these models [53]. Our research aims to delve deeper into their potential by employing more sophisticated prompting techniques and supervised fine-tuning strategies to closely mirror human performance, thereby narrowing the existing gap [40].

1.1 Key Contributions

In this thesis, we concentrate on enhancing the performance of LLMs for the text-to-SQL task. Our approach involves decomposing the task into simpler components, thereby making it more manageable and aiming to achieve state-of-the-art results. We place a particular emphasis on both the scalability and efficiency of our proposed methods to ensure they are practical and effective.

Overall, the key contributions are as follows:

1. Introducing a multi-stage few-shot in-context learning approach, DIN-SQL, which sets a new benchmark in performance for Text-to-SQL tasks. Our method enhances performance by customizing prompts to match the complexity of the queries and by developing Text-to-SQL specific reasoning prompts that incorporate intermediate steps.

2. Proposing Text-to-SQL specific self-correction prompts, enabling large language models to autonomously refine and adjust their own generated SQL queries.

3. Bridging the gap between smaller, open-source large language models with 7B parameters and their larger, costly proprietary counterparts, such as GPT-4, by introducing a decomposed text-to-SQL fine-tuning methodology.

4. Conducting a thorough evaluation and systematic analysis of text-to-SQL benchmarks to identify the critical limitations within these benchmarks that obstruct a fair comparison among different text-to-SQL methods.

1.2 Dissertation Layout

The dissertation layout is structured into chapters that delve into different aspects of improving text-to-SQL translation using LLMs and evaluating benchmarks. Here's a brief explanation of what each chapter does:

1. **Decomposed In-context Learning With Self-correction**: This chapter addresses the challenge of in-context learning for text-to-SQL. It introduces a decomposition approach that enhances the efficacy of in-context learning, particularly focusing on learning from few-shot samples provided in the prompt and task decomposition.

2. **Decomposed Text-to-SQL with Small Large Language Models**: This chapter proposes a method leveraging smaller open-source LLMs

with fine-tuning to achieve results comparable to those of larger proprietary models like GPT-4. It explains the methodology behind supervised fine-tuning for text-to-SQL task and introduces a novel two-step decomposed fine-tuning approach.

3. **Evaluating Models and Benchmarks**: This chapter focuses on evaluating text-to-SQL models and benchmarks. It covers text-to-SQL benchmarks, discusses evaluation metrics, and identifies common execution accuracy failures. The chapter details experiments conducted to evaluate models and benchmarks, including query rewriting, human evaluation, and standard SQL validation.

4. **Conclusion**: The concluding chapter summarizes the key findings and contributions of this thesis. It reflects on the implications of this work for the field of text-to-SQL translation and points out potential avenues for future research.

Each chapter of this dissertation is self-contained, covering the relevant works to that chapter.

Chapter 2

Decomposed In-context Learning With Self-correction

This chapter explore a decomposition approach to enhance the efficacy of in-context learning, specifically focusing on learning from few-shot samples provided in the prompt, for the text-to-SQL task, under the assumption that only a limited number of question and SQL pair examples are available. In many real-world scenarios, especially in niche domains or newly developed databases, the abundance of labeled data (question and SQL pairs) is rare. This assumption mirrors the reality of many applications, making research under this assumption applicable to practical situations.

2.1 Introduction

Natural language interfaces to databases aim at making it easier for end users to access data in a relational database. For example, given the utterance "find employees who make more than their managers" and the schema of tables *employees* and *manages*, one may want to generate a query in SQL that retrieves those employees from a database. Over the past two decades, research in this field has progressed through several phases, with early systems being domain-specific, supporting controlled natural language [36], [42], [48], [49] or relying on rule-based approaches [58] while more recent systems offering greater domain-independence using supervised models trained on diverse domains and datasets [77], [87] and more recently deep neural models trained on

large text and code repositories [14], [15].

The latest development in this progression is the use of LLMs under zero-shot and few-shot prompting [44], [53]. It has been shown that LLMs provide strong baselines using only a few demonstrations and no fine-tuning [2], [5], [45]. However, these models fall behind on commonly used benchmarks (e.g., Spider) compared to well-designed and fine-tuned models. Table 2.1 shows the performance of two latest LLMs, CodeX and GPT-4, on the development set of the Spider dataset. Despite a strong performance, LLMs fall behind, compared to existing methods [37], [55], especially on medium and complex queries. The question investigated in this paper is where these LLMs fail and if some of the problems that they are facing can be mitigated to push the performance to reach or surpass fine-tuned SOTA models.

Approach	**Execution Accuracy**
Fine-tuning Approaches	
RED-SQL 3B + NatSQL [37]	84.5
T5-3B + PICARD [55]	79.3
Inference-only Approaches	
Zero-shot GPT-4 (Ours)	64.9
Few-shot GPT-4 (Ours)	67.4
Zero-shot CodeX [53]	55.1
Few-shot CodeX (Ours)	61.5

Table 2.1: Zero-shot and few-shot prompting compared to fine-tuned approaches on the dev set of Spider.

It has been recently shown that the performance of LLMs can be improved on more complex tasks (e.g., math word problems, compositional navigation steps) using approaches such as chain-of-thought [69], least-to-most [88], and decomposed [31] prompting techniques where a task is broken down into multiple steps and the intermediate results are used to generate a final answer. Unlike algebraic expressions, which consist of clear steps or operations, breaking a complex SQL query can be a more daunting task because of the declarative structure of the language and the complex relationships between query clauses.

In this chapter, we propose a novel method based on few-shot prompt-

ing that decomposes the task of natural language text to SQL (referred to as text-to-SQL) into multiple sub-tasks.Previous works on text-to-SQL prompting using LLMs are only evaluated in a zero-shot setting [44], [53]. However, zero-shot prompting only provides a lower bound on the potential power of LLMs for most tasks [2], [32], [67], [69], [80]. We show that our proposed method outperforms the few-shot prompting method by a large margin. We also compare our method with previous approaches on two cross-domain challenging benchmarks, Spider and BIRD. For Spider dataset, we use the two official evaluation metrics of *execution accuracy* and *exact set match accuracy* [83]. We utilize two variants of the CodeX family, namely Davinci and Cushman [5], and the GPT-4 model for prompting. On the holdout test set of Spider, our method achieves an execution accuracy of 85.3% and 78.2% respectively using GPT-4 and CodeX Davinci models and an exact set match accuracy of 60% and 57% respectively using the same models. The large gap between the exact match and execution accuracies is due to the few-shot in-context nature of our method. Pretrained and fine-tuned approaches are more likely to generate SQL queries with a higher exact set match accuracy simply because these models have seen many examples during training that follow the composition style of the queries in the test set (queries in both sets are often written by the same people). Before our work, the SOTA on the test set had an execution accuracy of 79.9% [37] and an exact set match accuracy of 74% [39], and our method sets a new ground in terms of the execution accuracy. On the BIRD benchmark, our approach achieves a new SOTA result, attaining an execution accuracy of 55.9% on the holdout test set and 50.72% on the development set when employing GPT-4. Moreover, using the *valid efficiency score* introduced in this benchmark, our approach outperformed a GPT-4 baseline, demonstrating a 9% improvement on the development set. This highlights the effectiveness of our method.

Our contributions can be summarized as follows: (1) improving the performance of LLM-based text-to-SQL models through task decomposition, (2) introducing adaptive prompting strategies tailored to task complexity, (3) addressing schema linking challenges in the context of prompting, and (4) using

LLMs for self correction.

To replicate the reported results, visit our GitHub repository [1] for access to the prompts, results, and the code.

2.2 Related Work

Sequence-to-sequence models [60] have shown great potential in code generation tasks including text-to-SQL. The key idea is to jointly encode a given natural language question and the database schema and leverage a decoder to predict the target SQL.

On the encoder side, learning a representation for the question and the database schema is carried out using bidirectional LSTM in IRNet [22], convolutional neural networks in RYANSQL [7], pretrained language models such as BERT in SQLova [28] and graph neural networks in RATSQL [66], SADGA [3], and LGESQL [4].

[20] propose an intermediate representation to bridge the gap between the natural language question and SQL statements. There has been also work on tabular language models that encode both tables and text such as TaBERT [74], TaPas [25], and Grappa [75].

The methods on the decoder side can be categorized into sketch-based slot-filling and generation-based methods [52]. Sketch-based methods break the problem into several slot prediction sub-problems and aggregate the predictions for the slots of the SQL query to be generated [27], [28], [72]. A drawback of these methods is that they cannot generalize to queries that do not follow the predefined templates. The generation-based methods [4], [23], [26], [66] decode the SQL query as an abstract syntax tree.

In contrast to pretrained and fine-tuned models, [53] and [44] conduct an evaluation of the zero-shot prompting capability of LLMs on text-to-SQL using different prompts on the Spider dataset. Prompting techniques have been also used for tasks such as table understanding, table reasoning, and table-to-text generation [6], [24], and some remarkable results have been reported using

[1] `https://github.com/MohammadrezaPourreza/Few-shot-NL2SQL-with-prompting`

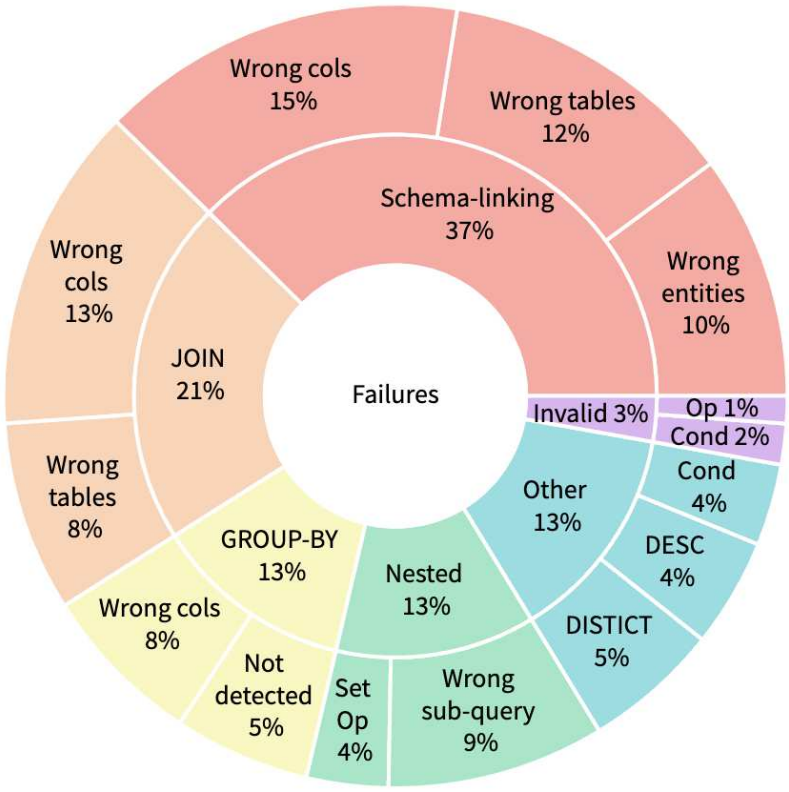

Figure 2.1: Statistics of simple few-shot failures using CodeX Davinci (Op refers to operators, Cond refers to conditions, and cols refers to columns)

LLMs with just a small number of examples given in the prompt.

2.3 Few-shot Error Analysis

To better understand where LLMs fail under a few-shot setting, we randomly sampled 500 queries from different databases in the training set of the Spider dataset, excluding all databases used in our prompts. We searched for the queries that produced results different than those of gold queries, hence failing the execution accuracy. We manually examined these failures and classified them into six categories as shown in Figure 2.1 and discussed next.

Schema linking This category contained the largest number of failed queries and included instances where the model failed to identify column names, table names, or entities mentioned in questions. In some cases, the query required an aggregation function, but a matching column name was chosen instead.

For instance, the database schema for question "What are the average and maximum capacities for all stadiums?" included a column named "average", which was selected by the model instead of taking the average of the capacity column.

JOIN This was the second largest category and included queries that needed a JOIN but the model was unable to identify all the tables required or the correct foreign keys to join the tables.

GROUP BY This category included cases where the SQL statement required a GROUP BY clause, but the model either did not recognize the need for grouping or wrong columns were used for grouping the results.

Queries with nesting and set operations

For this category, the gold query used nesting or set operations but the model did not recognize the nested structure or was unable to detect the correct nesting or set operation.

Invalid SQL A small set of the generated SQL statements had syntax errors and could not be executed.

Miscellaneous This category included cases that did not fit under any of the previously mentioned categories. Examples included SQL queries that contained extra predicates, missed a predicate, or had missing or redundant DISTINCT or DESC keywords. This category also included cases where the WHERE clause was missing or the query had redundant aggregation functions.

2.4 Methodology

Despite improvements over zero-shot, few-shot models struggle on more complex queries including those where schema linking is less trivial and the queries that use multiple joins or have a nested structure, as discussed in Section 2.3.

Our approach to address these challenges is to break down the problem into smaller sub-problems, solve each sub-problem, and use those solutions to construct a solution for the original problem. Similar approaches (e.g., chain-of-thought prompting [69] and least-to-most prompting [88]) have been taken to improve the performance of LLMs on tasks that can be broken down

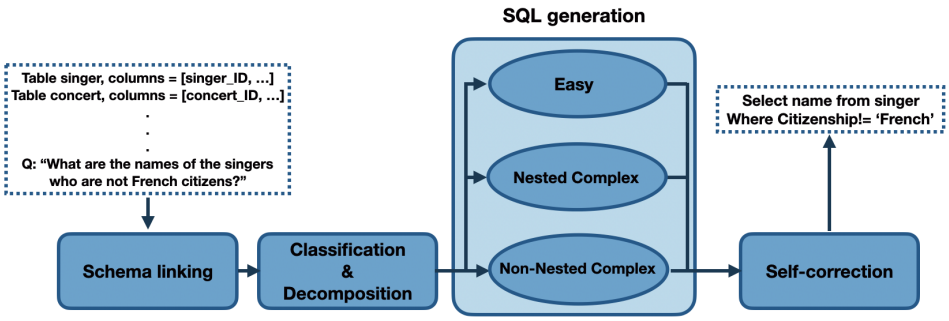

Figure 2.2: An overview of the proposed methodology including all four modules

into multiple steps such as math word problems and compositional generalization [9], [33]. Unlike these domains where the tasks have a procedural structure with one step directly feeding into the next step, SQL queries in most parts are declarative and the possible steps and their boundaries are less clear. However, the thought process for writing SQL queries may be broken down to (1) detecting database tables and columns that are relevant to the query, (2) identifying the general query structure for more complex queries (e.g., group by, nesting, multiple joins, set operations, etc.), (3) formulating any procedural sub-components if they can be identified, and (4) writing the final query based on the solutions of the sub-problems.

Based on this thought process, our proposed method for decomposing a text-to-SQL task consists of four modules (as depicted in Figure 2.2): (1) schema linking, (2) query classification and decomposition, (3) SQL generation, and (4) self-correction, which are explained in detail in the following subsections. While these modules may be implemented using techniques from the literature, we implement them all using prompting techniques to show that LLMs are capable of solving them all if the problems are simply broken down to the right level of granularity. The few-shot examples used in the prompts are obtained from the training set of the respective benchmarks.

2.4.1 Schema Linking Module

Schema linking is responsible for identifying references to database schema and condition values in natural language queries. It is shown to help with

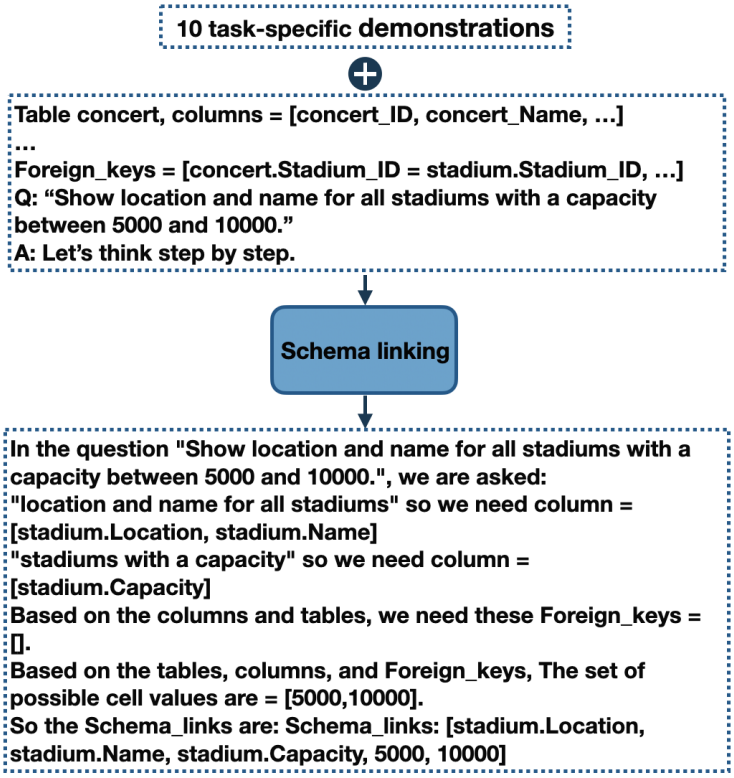

Figure 2.3: schema linking module

the generalizability across domains and the synthesis of complex queries [35], making it a critical preliminary step in almost all existing text-to-SQL methods [4], [23], [66], [73]. This was also a single category with the largest number of failures made by the LLM in our case (Figure 2.1).

We designed a prompt-based module for schema linking. The prompt includes ten randomly selected samples from the training set of the Spider dataset. Following the chain-of-thought template [69], the prompt begins with "Let's think step by step," as suggested by [32]. For each mention of a column name in the question, the corresponding columns and their tables are selected from the given database schema. Possible entities and cell values are also extracted from the question. Figure 2.3 illustrates an example with the prompt used for this module.

2.4.2 Classification & Decomposition Module

For each join, there is some chance that a correct table or join condition is not detected. As the number of joins in a query increases, the chance that at least one join fails to generate correctly increases. One way to alleviate the problem is introduce a module that detects the tables to be joined. Also some queries have procedural components such as uncorrelated sub-queries, which may be generated independently and be merged with the main query.

To address these issues, we introduce a query classification and decomposition module. The module classifies each query into one of the three classes: easy, non-nested complex and nested complex. The easy class includes single-table queries that can be answered without join or nesting. The non-nested class includes queries that require join but no sub-queries, and the queries in the nested class can contain joins, sub-queries and set operations. The class labels are important for our query generation module, which uses different prompts for each query class. In addition to class labels, query classification and decomposition also detects the set of tables to be joined for both non-nested and nested queries as well as any sub-queries that may be detected for nested queries. Figure 2.4 shows an example input given to the model and the output that the model generates.

2.4.3 SQL Generation Module

As the queries become more complex, additional intermediate steps must be incorporated to bridge the gap between the natural language question and the SQL statement. This gap, known as the *mismatch problem* in the literature [23], poses a significant challenge to SQL generation, which stems from the fact that SQL is primarily designed for querying relational databases and not representing the meaning in natural language [30]. While more complex queries can benefit from listing the intermediate steps in a chain-of-thought style prompting, such listings can degrade the performance for simpler tasks [69]. On the same basis, our query generation comprises of three modules, each geared toward different classes.

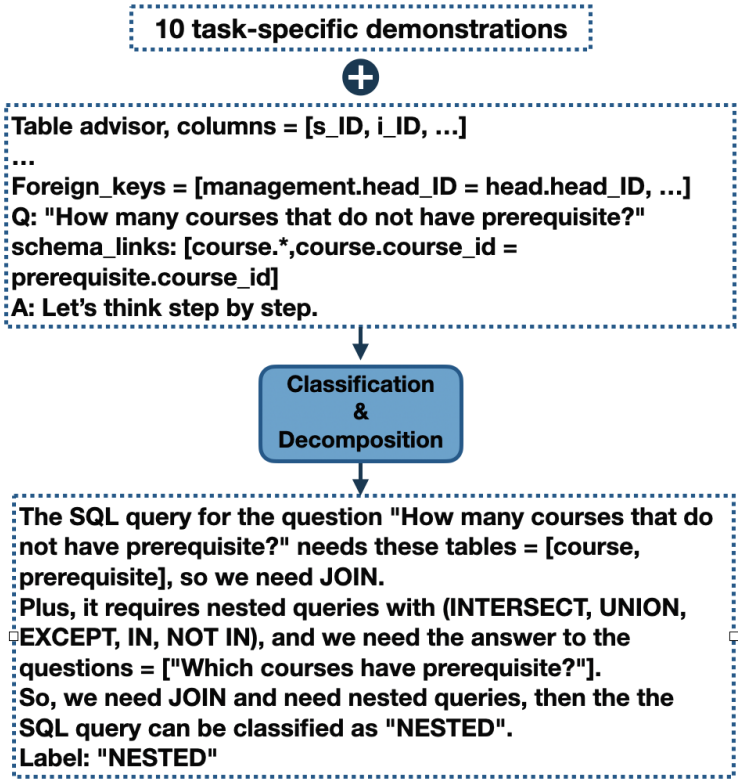

Figure 2.4: classification and decomposition module

For questions in our *easy class*, a simple few-shot prompting with no intermediate steps is adequate. The demonstration for an example E_j of this class follows the format (Q_j, S_j, A_j), where Q_j and A_j give the query text in English and SQL respectively and S_j indicates the schema links.

Our *non-nested complex* class includes queries that require join. Our error analysis (§ 2.3) revealed that finding the right columns and foreign keys to join two tables can be challenging for LLMs under simple few-shot prompting, especially when the query requires joining multiple tables. To address this issue, we resort to an intermediate representation to bridge the gap between queries and SQL statements. Various intermediate representations have been introduced in the literature. In particular, SemQL [23] removes operators JOIN ON, FROM, and GROUP BY, which have no clear counterparts in natural language queries, and merges the HAVING and WHERE clauses. NatSQL [20] builds upon SemQL and removes the set operators. Expressions in natural language queries may not clearly map to a unique SQL clause or

they may map to multiple clauses, so removing operators makes the transition from natural language to SQL easier. As our intermediate representation, we use NatSQL, which is shown to have a state-of-the-art performance when combined with other models [37]. The demonstration for an example E_j of the non-nested complex class follows the format (Q_j, S_j, I_j, A_j), where S_j and I_j respectively denote the schema links and the intermediate representation for the jth example.

Lastly, the nested complex class is the most sophisticated type and requires several intermediate steps before generating the final answer. This class can contain queries that not only require sub-queries using nesting and set operations such as EXCEPT, UNION, and INTERSECT but also multiple table joins, same as the previous class. To break down the problem further into multiple steps, our prompt for this class is designed in a way that the LLM should first solve the sub-queries, generated from the previous module, and then use them to generate the final answer. The prompt for this class follows the format $(Q_j, S_j\ , (Q_{j_1}, A_{j_1}, ..., Q_{j_k}, A_{j_k})\ , I_j, A_j)$, where k denotes the number of sub-questions, and Q_{j_i} and A_{j_i} respectively denote the i-th sub-question and the i-th sub-query. As before, Q_j and A_j denote the query in English and SQL respectively, S_j gives the schema links and I_j is a NatSQL intermediate representation.

2.4.4 Self-correction Module

The generated SQL queries can sometimes have missing or redundant keywords such as DESC, DISTINCT and aggregation functions. Our experience with multiple LLMs indicates that these issues are less common in larger LLMs (e.g., queries generated by GPT-4 have less bugs than those from CodeX) but are still present. To address this, we propose a self-correction module where the model is instructed to correct those minor mistakes.This is achieved in a zero-shot setting, where only the buggy code is provided to the model and it is asked to fix the bugs. We propose two different prompts for the self-correction module: *generic* and *gentle*. With a generic prompt, we request the model to identify and correct the errors in the "BUGGY SQL". The gentle prompt, on

the other hand, does not assume the SQL query is buggy, and instead asks the model to check for any potential issues and provides some hints on the clauses to be checked. Our evaluation indicates that a generic prompt can yield a better result with the CodeX model, while a gentle prompt is more effective for the GPT-4 model. Unless explicitly stated otherwise, the default self-correction prompt in DIN-SQL is set to gentle for GPT-4 and generic for CodeX.

2.5 Experiments

2.5.1 Models

We evaluated the proposed method using two variants of the CodeX family (Davinci and Cushman variants) and the GPT-4 model. These are the largest open-access LLMs at the time of writing this paper. Smaller models are less applicable since prompting is believed to be an emergent ability of the LLMs with the number of parameters in the scale of billions [68].

2.5.2 Hyperparameter

All models were accessed via the OpenAI API. Greedy decoding was used to generate the output by setting the temperature at zero. The max tokens was set to 350 for the self-correction module and 600 for all other modules. The stopping token sequence was set to "#;\n \n" for the self-correction module and "Q:" for all other modules.

2.5.3 Dataset

Our evaluation was conducted on two cross-domain challenging datasets, Spider and BIRD. Spider consists of 10,181 questions and 5,693 unique complex SQL queries across 200 databases, covering 138 domains, each containing multiple tables. The standard protocol for this dataset divides it into 8,659 training examples across 146 databases, 1,034 development examples across 20 databases, and a holdout of 2,147 test examples across 34 databases. The databases used in each of these sets are non-overlapping. SQL queries

are categorized into four difficulty levels, based on the number of SQL keywords used, the presence of nested subqueries, and the usage of column selections and aggregations. BIRD comprises an extensive dataset with 12,751 unique question-SQL pairs, encompassing 95 large databases totaling 33.4 GB in size. It spans a wide array of more than 37 professional domains, including blockchain, hockey, healthcare, and education. BIRD also introduces external knowledge as an additional resource to assist models in generating accurate SQL queries. Specifically four sources of external knowledge were introduced: numeric reasoning knowledge, domain knowledge, synonym knowledge, and value illustration. Notably, the SQL queries in the BIRD dataset tend to be more intricate than those in the Spider dataset. Language models without access to database content often encounter challenges with schema linking. Therefore, our prompts for the BIRD dataset include sample rows from each table to aid the model in schema linking. Furthermore, we have concatenated the provided external knowledge for each question as a hint, placed immediately after each question. However, due to constraints such as limited context window size, the presence of external knowledge, and the inclusion of sample rows, we have had to reduce the number of demonstrations within the prompts for the BIRD dataset.

2.5.4 Metrics

The performance of our models are evaluated using the official metrics of each dataset: exact-set-match accuracy (EM) and execution accuracy (EX) for Spider and valid efficiency score (VES) and execution accuracy (EX) for BIRD.

The exact-set-match accuracy (EM) treats each clause as a set and compares the prediction for each clause to its corresponding clause in the reference query. A predicted SQL query is considered correct only if all of its components match the ground truth. This metric does not take values into account. The execution accuracy (EX) compares the execution output of the predicted SQL query with that of the ground truth SQL query on some database instances. Execution accuracy provides a more precise estimate of the model's

performance since there may be multiple valid SQL queries for a given question, and exact set match accuracy only evaluates the predicted SQL against one of them. The Valid Efficiency Score (VES) is a metric designed to measure the efficiency of running the generated SQL queries. This metric is meaningful if the generated queries are correct, meaning their result matches that of the reference query. Therefore, the VES metric takes into account both the accuracy of the generated queries and their efficiency in terms of the execution time.

2.5.5 Results

Test set results

As shown in Table 2.2 for the holdout test set of Spider, our method achieves the highest execution accuracy using GPT-4 and the third-highest execution accuracy using CodeX Davinci among all officially published results at the time of this writing. This is achieved without even utilizing the database content. In terms of exact set match accuracy, our method achieves comparable results to previous works that do not utilize database content. As demonstrated in Table 2.3, in the case of the BIRD dataset, our method using GPT-4 achieved a test set execution accuracy of 55.9%, setting a new SOTA.

Model	EX	EM
DIN-SQL + GPT-4 (Ours)	**85.3**	60
RESDSQL-3B + NatSQL (DB content used) [37]	79.9	72
DIN-SQL + CodeX davinci (Ours)	78.2	57
Graphix-3B+PICARD (DB content used) [39]	77.6	**74**
SHiP+PICARD (DB content used) [81]	76.6	73.1
N-best Rerankers + PICARD (DB content used) [79]	75.9	72.2
RASAT+PICARD (DB content used) [51]	75.5	70.9
T5-3B+PICARD (DB content used) [55]	75.1	71.9
RATSQL+GAP+NatSQL (DB content used) [20]	73.3	68.7
RYANSQL v2 + BERT [7]	-	60.6
SmBoP + BART [54]	-	60.5

Table 2.2: Execution accuracy (EX) and exact set match accuracy (EM) on the holdout test set of Spider.

Model	VES	EX
DIN-SQL + GPT-4 (Ours)	59.44	**55.9**
GPT-4	**60.77**	54.89
Claude-2	-	49.02
ChatGPT + CoT [40]	56.56	40.08
ChatGPT	51.40	39.30
Codex	41.60	36.47
Palm-2	-	33.04
T5-3B	27.80	24.05
T5-Large	25	20.94
T5-Base	14.7	12.89

Table 2.3: Execution accuracy (EX) and Valid Efficiency Score (VES) on the holdout test set of BIRD.

Development set results

Most of our evaluation during development was conducted on the development set of Spider which was easily accessible unlike the test set that was only accessible through an evaluation server provided by [77]. Table 2.4 shows the performance of our method using different LLMs, compared to zero-shot prompting of [53] and [44] and our own few-shot prompting. To ensure a fair comparison for the few-shot prompting, we incorporate all the examples utilized for our three classes (easy, non-nested complex, and nested complex) inside the prompt. Given that the CodeX Cushman model has a smaller input context size than the CodeX Davinci and the GPT-4 models, we only use 2 examples from each class (for a total of 6 examples).

Our method significantly outperforms both simple few-shot prompting and zero-shot prompting, in terms of both exact set match and execution accuracies, and the improvement is consistent across all models despite their sizes. For example, compared to few-shot prompting, our method improves the execution accuracy for all models by at least 10%.

On the development set of BIRD, our approach demonstrates a substantial improvement, achieving a 4% gain in execution accuracy and a remarkable 9% improvement in valid efficiency score over a GPT-4 baseline [40], establishing a new SOTA. These and other results are reported in Table 2.5.

The performance of our method on the test set (as reported in Tables 2.2

Prompting	Model	EX	EM
DIN-SQL (Ours)	GPT-4	**74.2**	**60.1**
DIN-SQL (Ours)	CodeX Davinci	69.9	57.2
DIN-SQL (Ours)	CodeX Cushman	47.6	35.7
Few-shot (Ours)	GPT-4	67.4	54.3
Few-shot (Ours)	CodeX Davinci	61.5	50.2
Few-shot (Ours)	CodeX Cushman	43.1	30.9
Zero-shot (Ours)	GPT-4	64.9	40.4
Zero-shot [44]	ChatGPT	60.1	-
Zero-shot [53]	CodeX Davinci	47.5	
Zero-shot (DB content used) [53]	CodeX Davinci	55.1	
Zero-shot (DB content used) [53]	CodeX Cushman	53	
Zero-shot (DB content used) [53]	GPT3	21.7	

Table 2.4: Performance compared to zero-shot and few-shot prompting using different LLMs on the dev set of Spider.

Model	VES	EX
DIN-SQL + GPT-4 (Ours)	**58.79**	**50.72**
GPT-4	49.77	46.35
Claude-2	-	42.70
ChatGPT + CoT [40]	42.30	36.64
ChatGPT	43.81	37.22
Codex	43.41	34.35
Palm-2	-	27.38
T5-3B	25.57	23.34
T5-Large	22.74	19.75
T5-Base	12.90	11.54

Table 2.5: Execution accuracy (EX) and Valid Efficiency Score (VES) on the development set of BIRD.

and 2.3) is higher than that on the development set for both Spider and BIRD. It is hard to pinpoint the exact reason when the test set is hidden, but we speculate that fewer questions in the test set may require the knowledge of the database content, making it easier for our method to predict a correct SQL query. Furthermore, the development set has schema ambiguity (e.g., a query entity can be mapped to multiple database entities but only one is considered correct), and it is possible that the test set has less ambiguity.

We further analyzed the performance of our proposed method on queries with different levels of difficulty. Table 2.6 presents the performance of our

proposed method compared to a basic few-shot prompting on the development set of Spider.

Execution accuracy (EX)						
Prompting	**Model**	**Easy**	**Medium**	**Hard**	**Extra**	**All**
DIN-SQL	GPT-4 (Ours)	**91.1**	**79.8**	**64.9**	**43.4**	**74.2**
DIN-SQL	CodeX Davinci (Ours)	89.1	75.6	58	38.6	69.9
Few-shot	GPT-4 (Ours)	86.7	73.1	59.2	31.9	67.4
Few-shot	CodeX Davinci (Ours)	84.7	67.3	47.1	26.5	61.5
Exact set match accuracy (EM)						
Prompting	**Model**	**Easy**	**Medium**	**Hard**	**Extra**	**All**
DIN-SQL	GPT-4 (Ours)	82.7	65.5	42	**30.7**	**60.1**
DIN-SQL	CodeX Davinci (Ours)	78.6	**67.3**	38.5	17.5	57.2
Few-shot	GPT-4 (Ours)	**87.9**	54	**47.1**	12	54.3
Few-shot	CodeX Davinci (Ours)	77	53.8	38.5	12.7	50.2

Table 2.6: Performance compared to our basic few-shot prompting across different query difficulty levels.

Our proposed method outperforms the basic few-shot prompting across all difficulty levels, with the greatest improvement in performance observed for the extra hard and hard classes where the few-shot prompting performed poorly. Our improvement on the easy class (compared to basic few-shot) is due to incorporating schema links in the prompt, highlighting the importance of our schema-linking module.

Error improvements

In Section 2.3, we did an error analysis of basic few-shot prompting on 500 queries randomly chosen from the training set. To understand the degree those errors are resolved, we ran DIN-SQL on the same 500 queries. As shown in Figure 2.5, our proposed approach improves the performance for all categories with the largest improvement seen for the JOIN and Nested categories. Despite having an explicit module for schema-linking, the largest portion of failure cases still belong to this category.

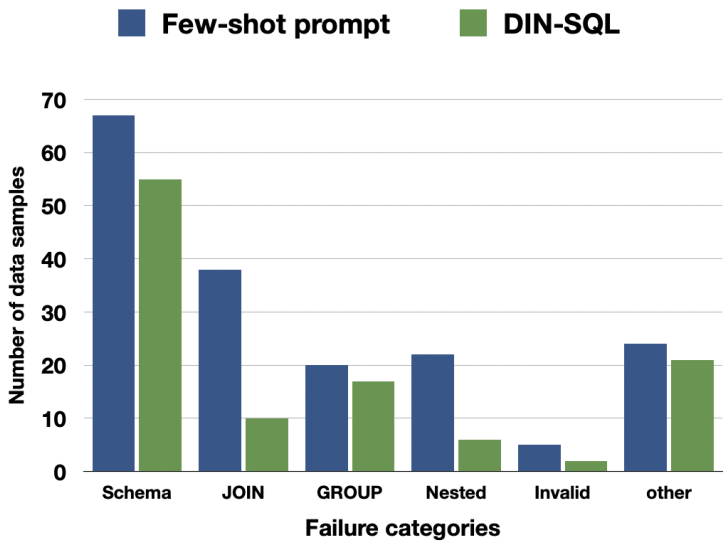

Figure 2.5: The break-down of failure cases for DINSQL (green) and the basic few-shot prompting (blue) across different categories

2.5.6 Ablation study

In an ablation study, we evaluated our approach with and without each of the four modules. As shown in Table 2.7 for the CodeX Davinci model, excluding any of the modules leads to an overall decrease in performance, in terms of the execution accuracy.

More details emerge as we study the effectiveness of each module across different query classes. Schema linking helps all query classes with the least improvement for the hard class. Our inspection of a sample of the failed cases reveals that schema linking sometimes finds redundant links due to an ambiguity in the question or schema, and this can introduce redundant joins or output columns.

Without a classification, we had to use either a simple few-shot prompting or a decomposed chain-of-thought (COT) prompting for all queries. The reported performance without a classification module in Table 2.7 is for our comprehensive framework that includes all our components except classification. This means that the approach contains not only COT prompting but also Schema Linking, Self-Correction, and NatSQL Intermediate Representation, all of which are significant contributions of our work. The decomposed

Prompting	Model	Easy	Medium	Hard	Extra	All
DIN-SQL (generic self-corr)	CodeX	**89.1**	75.6	**58**	**38.6**	**69.9**
DIN-SQL (gentle self-corr)	CodeX	87.5	**76.9**	51.7	36.1	68.7
DIN-SQL w/o self-corr	CodeX	83.9	75.4	52.3	36.1	67.3
DIN-SQL w/o schema-linking	CodeX	87.3	70.6	57.6	27.1	65.9
DIN-SQL w/o classification (simple few-shot prompting)	CodeX	87.9	68.2	51.7	27.1	63.1
DIN-SQL w/o classification (decomposed COT prompting)	CodeX	84.2	71.2	54.3	38.6	68.2
DIN-SQL (gentle self-corr)	GPT-4	**91.1**	**79.8**	**64.9**	**43.4**	**74.2**
DIN-SQL (generic self-corr)	GPT-4	89.9	76.5	59.2	34.3	70.0
DIN-SQL w/o self-correc	GPT-4	**91.1**	79.1	63.2	41.6	73.3

Table 2.7: Performance of our method, in terms of execution accuracy, on the dev set with and without each module

chain-of-thought result presented in this table refers to employing the most complex prompt, developed for the nested complex class, for all questions instead of adopting a classification-based approach to determine prompt complexity based on the question's level of difficulty. In contrast, the result for the DIN-SQL with simple few-shot prompting refers to using the simplest prompting class, easy class, for all questions across different level's of difficulty. As expected, a decomposed chain-of-thought prompting works better for hard and extra hard queries whereas a simple few-shot works better for the easy class.

For self-correction, we ran our study using both CodeX Davinci and GPT-4. For CodeX Davinci, a generic self-correction prompt helps the model across all query classes. A gentle self-correction prompt is also helpful but the gain is smaller than generic one for CodeX Davinci. However, there is less chance that GPT-4 generates a buggy code, and giving a generic prompt of "Buggy SQL:... Fixed SQL:..." can hurt the performance. A gentle prompt work better for GPT-4 and improves the performance across all of the classes except the easy class.

2.6 Discussion

In this chapter, we have witnessed a notable performance improvement in the text-to-SQL domain through task decomposition, employing large proprietary

language models like GPT-4 and Codex. Prompting with task decomposition is a strategy that warrants careful consideration for each specific task. While breaking down the problem into smaller, more manageable parts can lead to higher accuracy levels from the model due to the simplification of tasks, it's also essential to weigh the costs involved. These costs include both the potential increase in system latency and the expenses associated with using or deploying the models. In the text-to-SQL domain, given the complex nature of understanding data formatting in tables and constructing accurate SQL queries, we believe the performance gains achieved through decomposition justify these costs.

Chapter 3

Decomposed Text-to-SQL with Small Large Language Models

Although employing few-shot in-context learning with proprietary language models like GPT-4 and GPT-3.5-Turbo shows promising results, two major concerns arise with this approach. Firstly, since these models are accessed through APIs, there are valid concerns regarding data privacy and security. Secondly, the expense associated with utilizing these models can be significant, making them unaffordable for many small businesses. Therefore, in this chapter, we propose a method that leverages smaller, open-source large language models with fine-tuning to achieve results comparable to those of large proprietary models like GPT-4.

3.1 Introduction

Natural language interfaces for databases allow users to derive insights from structured databases using natural language instead of complex SQL queries. Leading open-source methods [21], [50] for this task heavily depend on proprietary LLMs like GPT-4 and GPT-3.5-turbo, which have demonstrated superior performance in text-to-SQL benchmarks [17], [41], [77]. However, this reliance on large proprietary models has privacy and cost implications. For instance, many large enterprises cannot share their customer data with the model-providing companies due to privacy considerations. Additionally, cost is a factor, especially for small businesses, in adopting these models.

Recent attempts to utilize open-source LLMs [21] and fine-tune them using question-SQL query pairs have fallen short of the zero-shot performance of GPT-3.5-turbo. Table 3.1 presents a performance comparison of the fine-tuned open-source LLMs on the Spider development set, contrasting with methods that employ GPT-4's prompting techniques. This paper aims to address this disparity by introducing a novel two-step decomposed fine-tuning method, employing two smaller LLMs, each with a parameter size of 7 billion. This approach achieves a performance comparable to methods that are using GPT-4 with few-shot learning and well-designed prompts.

Model	**EX**	**EM**
Fine-tuning methods		
Llama2 7B [21]	66.7	63.9
Llama2 13B [21]	67.0	62.7
Prompting methods		
DAIL-SQL + GPT4 [21]	84.4	74.4
DIN-SQL + GPT4 [50]	74.2	60.1

Table 3.1: Performance comparison of the prompting methods and fine-tuning method on the Spider validation dataset

We evaluate the performance of our proposed method using two text-to-SQL benchmarks: Spider [77] and Spider-SYN [17] and two 7B LLMs: DeepSeek [13] and Mistral [29]. Our approach demonstrates a performance improvement of approximately 3 to 7 percent in execution accuracy compared to the conventional single-step fine-tuning method employed in previous studies [21]. This consistent performance gain across both datasets highlights the generalizability of our method. Moreover, our fine-tuning strategy, utilizing a 7 billion parameter LLM, surpasses all previous open-source methods on the Spider development set and achieves comparable results to the state-of-the-art open-source methods using GPT-4 [21], [50] on the Spider test set. We have provided all the necessary code to replicate the results, along with the models' predicted SQL queries, in our GitHub repository [1].

[1] `https://anonymous.4open.science/r/DTS-SQL-2A42`

3.2 Methodology

A notable development in LLMs is their post-pretraining refinement, which enhances their alignment with preferred behaviors, as documented by [46], [61], [65]. Common methods of alignment include Supervised Fine-Tuning (SFT) using human demonstrations, as reported by [47], [63] and Reinforcement Learning from Human Feedback (RLHF), as detailed by [1], [8], [57], [89].

The absence of extensive datasets containing either human or AI feedback [34] has led to a predominant focus on supervised fine-tuning in the text-to-SQL field. This approach necessitates a collection of specific instructions or prompts along with their corresponding outputs or responses. In the following section, we will delve into the established methods of supervised fine-tuning for LLMs within the text-to-SQL context. Subsequently, we introduce our novel two-step fine-tuning approach, designed to enhance the performance of models in the text-to-SQL domain.

3.2.1 Supervised fine-tuning for Text-to-SQL

In this section, we explore the supervised fine-tuning process for text-to-SQL tasks, as practiced in the open-source community [21]. Given a set of databases D_i comprising pairs of questions q_i and corresponding SQL queries s_i, the goal is to fine-tune a large language model M using a set of training data $T = \{(q_i, s_i, D_i)\}$, where q_i and s_i represent the natural language question and its associated SQL query on database D_i. The objective of supervised fine-tuning is to minimize the empirical loss defined as:

$$\min_{\sigma, M^*} \frac{1}{|T|} \sum_{i=1}^{|T|} \mathcal{L}(M^*(\sigma_f(q_i, D_i, s_i)), \tag{3.1}$$

where $\mathcal{L}$ is the loss function used to measure the difference between the SQL queries generated by the model and the actual, correct (ground truth) queries. The function σ_f determines the formatting of the question, the database schema, and the SQL queries. A key challenge during inference is that we

do not know in advance among all of the tables inside the database which tables are relevant to a given question for generating accurate SQL queries. Therefore, a common approach in fine-tuning involves including the schema of all tables within the prompts together with the question and SQL pairs. This method serves a dual purpose: teaching the model to generate the correct SQL query and to identify the relevant tables from among all the provided tables. This approach of training for two objectives simultaneously complicates the SQL generation task for LLMs, particularly for smaller models with only a few billion parameters. Each task – generating SQL queries and correctly linking to the relevant schema – demands its own reasoning process. A significant proportion of errors in LLMs can be attributed to incorrect schema linking, highlighting this as a major challenge in the field [16], [50].

3.2.2 Decomposed Supervised Fine-tuning

We propose a two-stage fine-tuning process, which separates schema linking and SQL generation, aiming to enhance the performance of NL-to-SQL systems.

Schema-linking Fine-tuning

Schema linking involves identifying the pertinent columns and tables in a database in response to natural language queries. It has been demonstrated to enhance cross-domain generalizability and facilitate the creation of intricate queries [35]. In prior studies, schema linking has primarily been accomplished through in-context learning methods or implicitly during the fine-tuning process for SQL generation [4], [23], [50], [71]. In this work, we treat schema linking as a distinct task and explicitly fine-tune LLMs to identify relevant tables and columns when presented with a natural language query. Given the training dataset $T = \{(q_i, s_i, D_i)\}$, we extract all of the columns and tables used in the SQL queries and create a new dataset of $T = \{(q_i, T_i, C_i, D_i)\}$ where T_i and C_i represent lists of tables and columns used in the SQL query s_i. The primary objective during supervised fine-tuning for schema linking is to minimize the empirical loss, as defined by the following equation:

$$\min_{\sigma, M^*} \frac{1}{|T|} \sum_{i=1}^{|T|} \mathcal{L}(M^*(\sigma_s(q_i, T_i, C_i, D_i)), \tag{3.2}$$

Here, $\mathcal{L}$ represents the loss related to the model's next token prediction, comparing the predicted column and table names with the actual ground truth names.

SQL Generation Fine-tuning

After identifying the appropriate tables for SQL generation, the next step is to utilize a model that constructs the SQL query based on the question and the schema of the correct tables. Since we have already identified the potentially correct tables using the schema-linking module, there is no need to include all tables in the input for the SQL generation model. In contrast to previous approaches for fine-tuning LLMs, we extract the relevant tables from the training dataset $T = \{(q_i, s_i, D_i)\}$ corresponding to the ground truth SQL queries. We then fine-tune the LLM while minimizing the following loss function:

$$\min_{\sigma, M^*} \frac{1}{|T|} \sum_{i=1}^{|T|} \mathcal{L}(M^*(\sigma_g(q_i, T_i, s_i)), \tag{3.3}$$

The loss function is same as the loss function defined in Section 3.2.1. This decomposition of the text-to-SQL training process allows LLMs to be trained with a singular objective. By segregating the schema-linking and SQL query generation tasks, we improve the training process, enabling more focused and effective fine-tuning.

3.3 Experiments

3.3.1 Models

Our methodology's performance was assessed using two recent LLMs from distinct architectures. These models are Mistral 7B [29] and DeepSeek 7B [13]. The DeepSeek model, sharing similar architecture with the LLama model family [62], has been pretrained on an extensive dataset comprising 2 trillion

tokens and supports a sequence length of 4096. Mistral 7B, although not specifically pretrained for code generation, surpasses many counterparts in its scale category [29].

3.3.2 Hyperparameters

The two LLMs were trained on Nvidia Tesla A100 GPUs, employing a batch sizes of 64 and 32 with a learning rate of 1*e-5 and 5*e-5 respectively. To enhance the training efficiency, we incorporated Flash Attention techniques as detailed in [10], [11].

3.3.3 Datasets

We conducted our evaluation using cross-domain, challenging text-to-SQL datasets. Spider, was introduced by [77] and includes 200 database schemas. Of these, 160 schemas are allocated for training and development, while the remaining 40 are set aside for testing purposes. Our second dataset was Spider-Syn [17], which modifies the original Spider dataset by replacing schema-related words with synonyms and removing explicit mentions that link natural language queries (NLQs) to the database schema.

3.3.4 Metrics

In our evaluation of text-to-SQL models, we utilized exact set match accuracy and execution accuracy. The former involves comparing the components of SQL queries, such as select, where, having, group by, and order by clauses, focusing on the matching of columns and predicates without considering the order. The latter determines equivalence between a model-generated query and a reference query if they produce identical results across various database instances.

3.3.5 Results

Spider test set

As depicted in Table 3.2, our method employing DeepSeek 7B, when tested on the Spider test dataset, achieves results comparable to state-of-the-art open-source methods in terms of execution accuracy and exact set match accuracy.

Model	EX	EM
DAIL-SQL + GPT-4 [21]	86.6	-
DIN-SQL + GPT-4 [50]	85.3	60
DTS-SQL + DeepSeek 7B (Ours)	84.4	73.7
C3 + ChatGPT + Zero-Shot [16]	82.3	-
RESDSQL-3B + NatSQL [38]	79.9	72
DIN-SQL + CodeX [50]	78.2	57
DTS-SQL + Mistral (Ours)	77.1	69.3
Graphix-3B + PICARD [39]	-	74

Table 3.2: The comparison of different methods on the test set of Spider.

Spider dev set

In Table 3.3, we showcase the results of our two-stage fine-tuning method on the dev set of Spider. The performance is compared against two distinct scenarios: firstly, a one-stage scenario where the model is fine-tuned on all tables without employing our two-stage approach, and secondly, a perfect schema linking scenario where we provide the ground truth tables to our fine-tuned SQL generators. This latter scenario is denoted as the 'Upper Bound' in the table. Our two-stage model's performance is measured by initially using our fine-tuned schema linker model to identify potentially relevant tables, which are then provided as context to the SQL generator model.

In Table 3.4, we offer a detailed comparison between our method and various other baseline approaches. For the baselines, we selected diverse methods from different families of approaches that are using LLMs and are available as open source. Our two-stage decomposed approach with DeepSeek 7B attained state-of-the-art performance on the Spider development set, surpassing all previous methods that utilized prompting techniques and fine-tuning.

Model	Tuning	EX	EM
Mistral 7B	FT Tuning	71.9	70.9
Mistral 7B	DTS-SQL (Ours)	78.6	73.3
Mistral 7B	Upper bound	86.6	80.7
DeepSeek 7B	FT Tuning	82.1	69.0
DeepSeek 7B	DTS-SQL (Ours)	85.5	79.1
DeepSeek 7B	Upper bound	90.3	84.2

Table 3.3: Performance of the LLMs with different tuning methods on the Spider dev set. FT stands for Full Table Finetuning; Upper bound performance is the performance achievable with perfect schema linking.

Additionally, the results of our two-stage method on Spider-SYN dataset is provided in the Table 3.5 section.

Model	EX	EM
Instruction tuning methods		
DTS-SQL + Mistral 7B (Ours)	78.6	73.3
DTS-SQL + DeepSeek 7B (Ours)	**85.5**	**79.1**
Llama2 7B [21]	66.7	63.9
Llama2 13B [21]	67.0	62.7
Prompting methods		
DIN-SQL + GPT4 [50]	74.2	60.1
DIN-SQL + CodeX [50]	69.9	57.2
DAIL-SQL + GPT4 [21]	84.4	74.4
C3 + GPT-3.5 [16]	81.8	-

Table 3.4: Performance of different methods with LLMs on the dev set of Spider.

Schema-linking Performance

As discussed in Section 3.2, our approach employs two LLMs: one for schema linking and another for SQL query generation. The schema-linking model plays a pivotal role in our pipeline, as inaccuracies in table detection could hinder the SQL generator's ability to formulate the correct SQL queries. We fine-tuned two models, based on the Deepseek and Mistral models, for schema linking. Evaluation metrics, including exact set match, precision, and recall, were used to assess their performance. Detailed information about these models on two distinct datasets can be found in Table 3.6.

Model	Tuning	EX	EM
Mistral 7B	FT Tuning	67.0	63.9
Mistral 7B	DTS-SQL (Ours)	71.1	64.6
Mistral 7B	Upper bound	81.9	74.5
DeepSeek 7B	FT Tuning	70.4	56.6
DeepSeek 7B	DTS-SQL (Ours)	76.2	68.9
DeepSeek 7B	Upper bound	85.5	78.1

Table 3.5: Performance of the LLMs with different tuning methods on the Spider-SYN dev set. FT stands for Full Table Finetuning; Upper bound performance is the performance achievable with perfect schema linking.

Model	Dataset	EX	PR	RE
DeepSeek	Spider	93.1	98.4	97.7
Mistral	Spider	91.1	97.5	97.8
DeepSeek	Spider-SYN	87.6	94.6	94.7
Mistral	Spider-SYN	85.3	91.2	90.5

Table 3.6: Performance of the schema-linker model on Spider and Spider-SYN dev sets. PR stands for Precision, RE is Recall, and EX is Exact Set Match Accuracy.

3.4 Discussion

While our two-step approach has achieved state-of-the-art results on the development set of Spider and demonstrated comparable performance to larger models like GPT-4 on the test set, there is still significant room for improvement, particularly for the schema-linking models. Currently, our schema-linking models achieve roughly 90% exact set match accuracy. However, as noted in Table 3.3, the substantial gap between the upper bound performance of the SQL generator and that of DTS-SQL calls for further research into the schema-linking.

Chapter 4

Evaluating Models and Benchmarks

In our experiments with various text-to-SQL models, we observed instances where, despite achieving an execution accuracy of zero, the generated SQL queries accurately addressed the posed questions. This chapter aims to critically assess and analyze text-to-SQL benchmarks to uncover their limitations.

4.1 Introduction

Significant progress has been made in translating natural language text to SQL statements over the past few years. The execution accuracy on the hold-out test of Spider [77]–a large-scale cross-domain text-to-SQL benchmark– has improved from 53.5 in May, 2020 [85] to 85.3 in March, 2023 [50]. The exact set match accuracy, without considering database cell values, on the same benchmark and over the same period has improved from 65.6 [66] to 74.0 [39]. Measuring such progress is hinged on reliable benchmarks and evaluation metrics.

Two standard metrics for evaluating the performance in this domain have been *exact set match accuracy* and *execution accuracy*. The former measures if a model-generated SQL query lexically matches a reference SQL query, whereas the latter measures if a model-generated SQL query produces the same output as a reference query (§ 4.4).

Consider the example in Figure 4.1, which consists of a model-generated

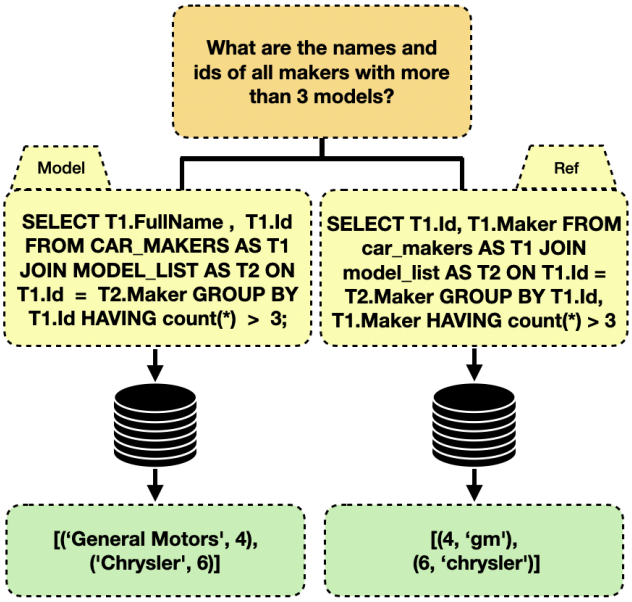

Figure 4.1: An example question with two correct SQL queries, each corresponding to a different interpretation. There is an ambiguity in schema mapping, with two different database columns describing the name.

query (shown on the left) and a reference query (shown on the right). Both SQL queries return the id and name of makers that have more than 3 models. However, the model-generated query returns the column FullName, which gives the full name of a maker (e.g., "Ford Motor Company"), whereas the reference query given in the benchmark returns the column Maker, which gives the short common name of a maker (e.g., "Ford"). The model-generated query fails an exact set match since the column names in the select clause are different. The query outputs are also different and the model-generated query fails the execution accuracy as well. The natural language utterance is not specific about the type of name to be returned, and a human evaluator tags both queries correct.

As the models improve, these types of failures make up most of the errors, and the performance metrics become less relevant, as shown in our evaluation. In particular, we re-evaluated all development set queries of Spider on which two top-performing models, one using a fine-tuned model [55] and another using a large language model [50], failed. We found out that 25% of the queries generated by one model and 87% of the queries generated by the other model

were indeed correct but were wrongly evaluated by the benchmark. For the same set of queries, our re-evaluation of the ground truth queries found 33% of the SQL queries incorrect, which was more than the number of incorrect queries generated by one of the models. This evaluation places one of the models above the ground truth queries in this re-evaluation.

We further re-evaluated two well-known benchmarks, Spider [77] and Spider-DK [19], and a newly released benchmark, BIRD [41], and found similar problems in all three benchmarks that affect the evaluation. Our evaluation reveals that 18% of the queries in the train sets and 20%-23% of the queries in the dev sets of these benchmarks are subject to ties in the dataset and which one of the tied rows are returned. This means a model-generated query will be deemed incorrect if it does not return the same row, among tied rows, as the ground truth query. This can severely impact the evaluation, especially when there is a tight race among models. Considering these observations, it is crucial to emphasize the significance of additional independent evaluations when utilizing these benchmarks. To enhance the evaluation process further, a potential solution is to incorporate multiple SQL queries as the ground truth, each representing a different interpretation that may be valid.

Our objective in this paper is to provide a comprehensive evaluation of existing Text-to-SQL benchmarks, underscoring the inherent issues they possess. We refrain from introducing a new dataset due to several considerations. First, addressing the identified issues by updating these benchmarks requires considerable human effort. Additionally, benchmarks in the Text-to-SQL domain, like Spider and BIRD, have holdout test sets used for official leaderboards and comparisons of text-to-SQL methodologies. We only have access to the development and training sets of these benchmarks, which limits our capability to alter the test sets. As a result, making changes only to the development and training sets would not completely address the benchmark's inherent problems, given that final performance is gauged using the problematic test sets.

4.2 Related Work

Limited research has been dedicated to assessing the reliability and effectiveness of Text-to-SQL benchmarks. The authors of SQL-PaLM [59] note in their qualitative analysis of their model that some queries, labelled as incorrect by execution accuracy, were considered correct by human annotators.

Similarly, [35] conduct an analysis highlighting the discrepancy between automatic evaluations and human annotations. They emphasize that certain queries produced by the models were labeled as incorrect SQL queries but human annotators labelled them as correct queries. Generally, a query that is equivalent (but not identical) to ground truth may be mistakenly classified as incorrect by automated evaluation metrics. Another study by [82] identifies limitations within the Spider benchmark, such as issues with ties and certain syntactic problems. Their analysis is primarily focused on a subset of Spider, without quantifying the extent or impact of these limitations or conducting an assessment of other benchmarks.

4.3 Text-to-SQL Benchmarks

Benchmarks have played a crucial role in advancing the field and providing a platform for evaluation. WikiSQL [86] consists of over 24,000 tables from Wikipedia with SQL queries generated based on some predefined rules and templates. The queries in this dataset are considered easy since they are all single-table queries. Spider, introduced by [77], consists of 200 database schemas of which 160 schemas are published as train and dev sets and 40 schemas are kept hidden for testing. The queries are written on those schemas by Computer Science students without using templates. This is considered a challenging dataset. Some other benchmarks are developed based on Spider, including Spider-Syn [18], which replaces schema-related words with synonyms and eliminates explicit mentions between NLQ and schema, and Spider-DK [19], which introduces rarely observed domain knowledge into the Spider development set. Other benchmarks include FIBEN [56], created for the financial

domain and BIRD [41], which comprises 12,751 queries over 95 databases spanning 37 professional domains.

Our study in this paper focuses on cross-domain large-scale benchmark Spider, its variants Spider-DK and Spider-SYN, and a more recent cross-domain large-scale benchmark BIRD. The selection of these benchmarks stems from their resemblance to real-world datasets, which is a crucial factor in conducting comprehensive research and analysis. One notable advantage of these benchmarks is the availability of a large training set, which plays a pivotal role in training and fine-tuning large-scale models. The inclusion of a substantial amount of training data enables the development of more robust and powerful models that can better handle the complexities and nuances present in real-world databases.

4.4 Evaluation Metrics

The performance evaluation of text-to-SQL systems involves comparing them to a reference system, typically a gold standard set of known correct SQL queries. Generating a reference can be challenging due to multiple interpretations of natural language questions, while SQL queries are based on logic and tend to cover only one interpretation. Even if an interpretation is fixed, detecting if a model-generated query is equivalent to a reference query is challenging, due to the halting problem which is undecidable [12]. Nonetheless, to assess progress, proxy measures of performance have been developed in the literature. As two such metrics, we review exact set match accuracy and execution accuracy in this paper.

Under *exact set match accuracy*, SQL queries are evaluated by matching the query clauses and components independently, such as the *select*, *where*, *having*, *group by*, and *order by* clauses. The matching is based on comparing columns and predicates, disregarding the ordering of columns and predicates. An exact matching of literals can be challenging since predicates such as `nationality=‘‘Canada’’` and `nationality=‘‘Canadian’’` will not match. However, accurately generating those literals without accessing database con-

tent may not be possible. Under *exact set matching without values*, which is used in Spider [77], a matching of literals is not required.

Two equivalent SQL queries can have different expressions and may not match under an exact set match. An alternative metric that can reduce the number of false negatives is the *execution accuracy*. Under execution accuracy, the equivalence between a model-generated query and a reference query is established if they both produce the same results on all possible databases instances [76]. While testing all instances is impractical, running queries on a subset of instances can help identify candidates that are not equivalent to the reference query. Although execution accuracy can detect queries that are equivalent but not identical, it may mistakenly identify queries as equivalent if they produce the same result on tested instances. Therefore, an effective execution-based evaluation requires finding instances that cover various edge cases and can detect queries that are not equivalent to the reference. Test suite accuracy [84], which is simply referred to as execution accuracy in Spider benchmark and in our work, aims to minimize false positives by evaluating queries on a carefully selected collection of database instances, known as a test suite. Nevertheless, an execution-based accuracy cannot capture all correct SQL queries, highlighting the limitations and the continued importance of human evaluation for reliable assessment.

4.5 Execution Accuracy Failures

A model-generated query can be correct but still fail the execution accuracy. We classify these failures into three categories: (1) failures due to ties in output, (2) ambiguity in schema matching, (3) wrong assumptions made about database content.

4.5.1 Failures Due to Ties in Output

SQL queries can lead to ties and a subset of the tied rows may be returned. The selection of tied rows can vary between queries and this can affect the execution accuracy. We identify a few sources for such ties, as discussed next,

Benchmark	LIMIT 1	LIMIT N	GROUP BY	ORDER BY	Total
Dev set					
BIRD	255(16%)	42(2%)	20(1%)	4(0.2%)	321(20.86%)
Spider	171(16%)	10(0.9%)	51(4.5%)	2(0.2%)	234(22.63%)
Spider-DK	94(17%)	2(0.3%)	30(4.5%)	2(0.3%)	128(23.85%)
Train set					
BIRD	1558(16%)	211 (2%)	23 (0.2%)	4(0.04%)	1796 (18.22%)
Spider	989(14%)	106(1%)	254(3%)	10(0.1%)	1359(18.1%)

Table 4.1: The number of SQL queries having a specific type of limitation together with the percentage on both development set and train set. The Spider-DK dataset does not have any training set.

and study their impact on benchmark evaluations in Section 4.6. Table 4.1 provides a detailed breakdown of the number of queries that can potentially yield tied rows in both train and development set of Spider, Spider-DK, and BIRD benchmarks.

Top with Ties

Sometimes the query asks for top rows that satisfy some conditions (e.g., the student with the highest GPA, or the youngest student). When there is a tie for the top position, and the query in natural language is not specific on how the ties should be handled, the corresponding SQL query may return all ties or only one. This becomes a problem in evaluation if a model-generated query and the reference query treat the ties differently. Figure 4.2 provides a concrete example from the Spider dataset, illustrating this issue, where the reference SQL query in the benchmark fails to account for ties and returns only one of them using the LIMIT keyword.

LIMIT N

The problems associated with using the *LIMIT n* clause in SQL queries is not limited to the top position, as discussed above. The use of this clause is problematic for evaluation in general. Firstly, without an explicit ordering, the result of a SQL query is expected to be a set. Two equivalent (but not identical) queries can return the same set of results, each listed in different orders, but selecting *the first n* rows from one ordering will not necessarily match the same selection from a different ordering. Secondly, with query

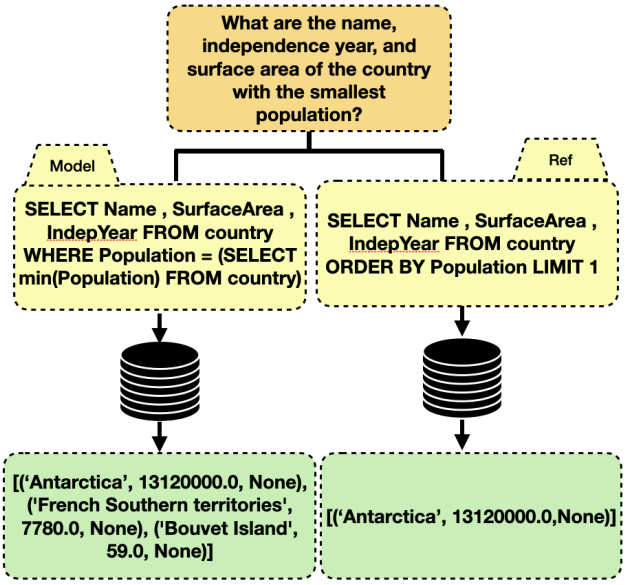

Figure 4.2: An example question that can have two correct SQL queries, each corresponding to a different interpretation. The SQL query on the left returns all tied values, while the SQL query on the right returns only one of the tied values.

results sorted, there can be a tie on row n with multiple rows having the same values. The ordering among tied rows can vary between two queries, and so is the first n rows that are returned. All benchmarks studied in this paper (Spider, Spider-DK, Spider-SYN, BIRD) use the limit keyword and suffer from the aforementioned problems associated with ties.

GROUP BY

Many text-to-SQL benchmarks encounter a different type of issue associated with ties, particularly arising due to incorrect usage of non-aggregated columns in both the SELECT clause and the GROUP BY clause. Within the benchmarks, these ties manifest in two scenarios: 1) a column appears in the SELECT clause without being inside an aggregation function and without being included in the GROUP BY clause; 2) the SELECT clause contains a mix of aggregated and non-aggregated columns without utilizing a GROUP BY clause. In both cases, multiple records can be associated with the same grouping column or aggregation value, whereas each group can only return one record. Some database systems including Oracle and DB2 prevent these cases

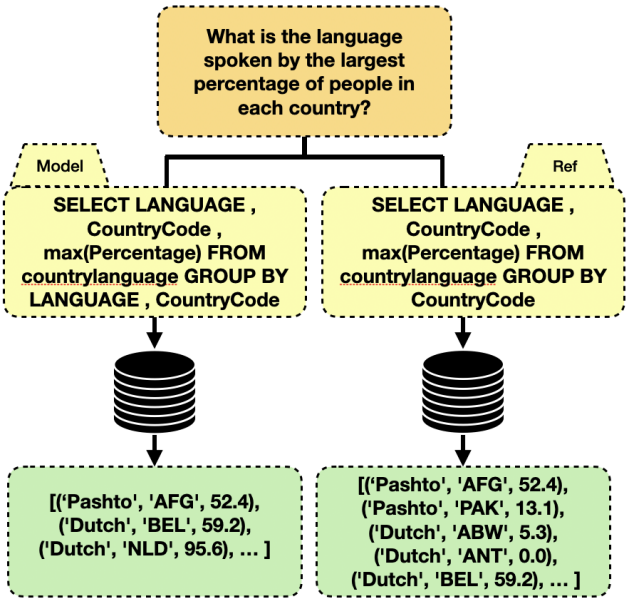

Figure 4.3: An example question that can have two correct SQL queries, each corresponding to a different interpretation. The SQL query on the left returns all languages of each country, each pair of country and language in a separate row, whereas the SQL query on the right returns one of tied values for the column LANGUAGE.

by treating them as syntax errors. However, other database systems such as SQLite and MySQL take a more lazy approach (sometimes for efficiency reasons) and allow these cases to happen. Many text-to-SQL benchmarks follow SQLite syntax and suffer from this issue. The affected queries in our benchmarks were identified after migrating from SQLite to PostgreSQL, as detailed in Section 4.6.4, and checking for queries that failed during PostgreSQL execution. Figure 4.3, illustrates one example of such a problem from the Spider dataset.

ORDER BY

Another subtle ambiguity with tied values arises in queries where the SELECT clause incorporates the "distinct" keyword, paired with an ORDER BY clause referencing a column absent in the SELECT clause. Consider the exemplary query from Spider train set: `SELECT DISTINCT district_name FROM district ORDER BY city_area DESC`. The ordering of the output, as well as the result of a comparison with a reference query, becomes uncertain if a single

'district_name' value maps to multiple 'city_area' values. Similar to GROUP BY, the affected queries in the benchmarks were identified through a SQLite to PostgreSQL migration(§ 4.6.4).

4.5.2 Ambiguity in Schema Matching

Schema matching refers to the task of establishing the correspondence between a natural language question and the tables, columns, and cell values in the database ([4], [41], [50], [66]. Ambiguities arise when there are multiple columns in the database that can represent the same semantic meaning, and the information needs of a query may be satisfied using any of those columns. As a result, there exist multiple SQL queries that can produce the correct answer, yet most benchmarks only provide one query among the many possible correct answers. Figure 4.1 illustrates an example question that can be satisfied by two different SQL queries, both of which are valid responses to the question at hand.

4.5.3 Wrong Assumptions on DB Content

Lastly, one type of limitation in text-to-SQL benchmarks stems from incorrect assumptions regarding cell values. It is common to make assumptions about database content and constraints when writing SQL queries, but those assumptions may not be supported by the database schema or content. This issue arises when the database content is created under assumptions that do not align with those in queries, leading to potential failures in the evaluation process. Text-to-SQL models often lack access to full database content due to limitations such as the context window problem and the inability to pass all cell values to the models for reasons such as privacy and cost. These models typically rely on the provided database schema and a selected sample of database rows to represent potential values [39], [43], [44], [50], [53], [59]. Consequently, the assumptions made by these models may not align with the actual ground truth, resulting in SQL queries that are correct under the assumption made but do not match the reference query in the benchmark.

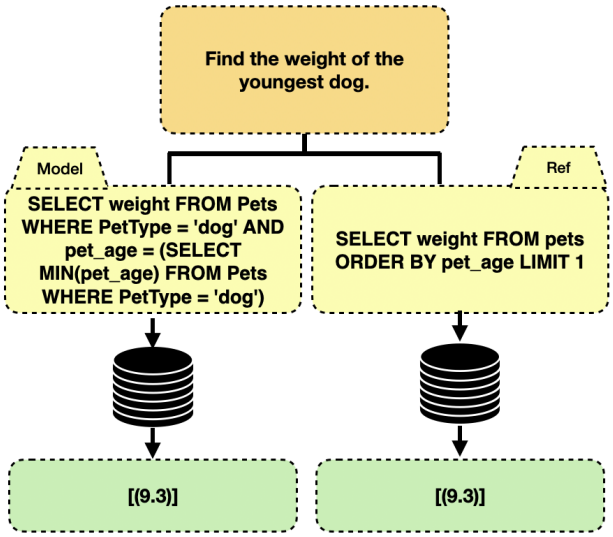

Figure 4.4: An example of a question and SQL pair with a wrong assumption on the cell values. The SQL query on the left does not make the same assumption.

One observed case is when certain conditions (e.g., PetType=‘dog’) are omitted from SQL queries due to the erroneous assumption that the condition holds for all rows in the database. Figure 4.4 exemplifies this issue using an example from the Spider dataset, where both queries yield the same answer on a specific database instance. However, changing the database values could result in failure, especially when evaluating performance using test-suite accuracy, which involves querying different database instances. Another case observed in the benchmarks is when the ground truth SQL queries assume a specific column has unique values, but in reality, that column does not possess that unique constraint. Figure 4.5 depicts an example of this problem from the Spider dataset.

4.6 Experiments

To understand the extent at which the aforementioned problems affect the benchmarks, our evaluation and the ranking of the models, we conducted three types of evaluations on three benchmarks: Spider, Spider-DK, BIRD. Our findings here apply to the Spider-SYN dataset as well, which employs the

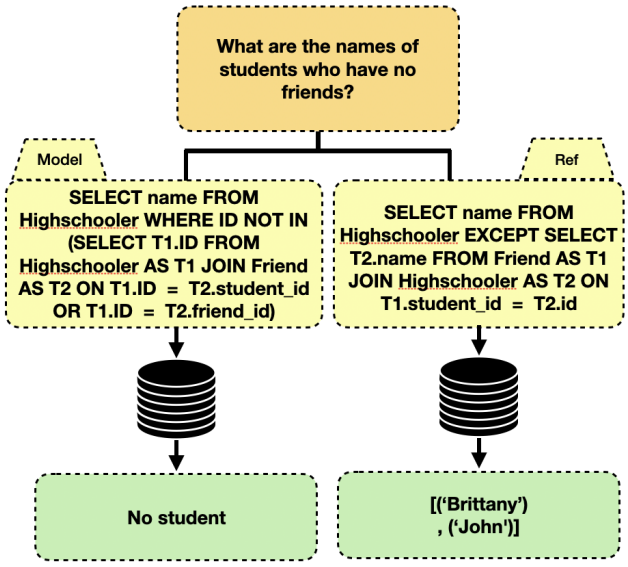

Figure 4.5: An example of a question and SQL pair with a uniqueness assumption on the "name" column, which is not supported by the schema. The SQL query on the left does not make the same assumption.

same SQL queries as in the Spider dataset. For the same reason, we did not conduct a separate analysis of that benchmark.

4.6.1 Evaluation Through Query Rewriting

In this experiment, our focus is on ties and how a tie breaking strategy affects the benchmarks and our evaluation. This is done through query rewriting. Automating query rewriting faces inherent challenges, particularly when dealing with failures stemming from schema ambiguity, erroneous assumptions about the database content, and the ambiguity of natural language utterances. These challenges arise because there is no specific structure to address the failures systematically. Successful query rewriting in these cases necessitates a deeper understanding of table and column semantics to identify ambiguities and erroneous assumptions. In cases of ambiguity, human expertise is essential to disambiguate the context, as these situations often lack clear guidelines. Detecting erroneous assumptions often involves introducing new data to the database and meticulously reviewing and correcting failed queries on a case-by-case basis. Therefore, our efforts have been channeled towards rewriting

queries concerning tied values, which adhere to a specific syntax structure, and the problems associated with the ambiguity in schema matching and wrong assumptions on database content are studied in the next section.

Many benchmark queries use "LIMIT 1" to find top rows that satisfy some conditions. If there are ties on top, one arbitrary row among ties is returned. An alternative is to return all ties. We rewrote all queries that used "LIMIT 1" to return all ties. This was done by introducing min() and max() aggregation functions within nested queries to accurately identify extreme values. An example of such rewriting is shown in Figure 4.1. Breaking ties for queries that used "LIMIT n" for $n > 1$ was not straightforward, and those queries were left unchanged.

For resolving ties introduced by an incorrect usage of GROUP BY in benchmark queries, we included all non-aggregated columns from the SELECT clause in the GROUP BY clause. For example, if the SELECT clauses included id and name, but the GROUP BY clause only included name, we added id to the GROUP BY clause. This change will not affect queries where there is a one-to-one mapping between id and name, but it will resolve the ambiguity when such mapping does not hold.

With these two changes, 16% to 20% of the reference queries in our benchmarks were affected. Under a perfect evaluation scheme, the accuracy should not be affected with these changes that simply resolve the uncertainty. Table 4.2 displays both the execution accuracy and the exact set match accuracy for the reference queries from the BIRD, Spider, and Spider-DK benchmarks after our modifications. It's important to highlight that the performance metrics provided in this table encompass the entire development set of these benchmarks, combining both modified and unaltered queries. For clarity, in the Spider dataset, out of 1034 queries, 206 were modified. The performance assessment took into account a mixed set of predicted queries: 206 that were adjusted and 828 that remained as originally presented. This culminated in an execution accuracy of 92.3 percent.

It can be noted that the execution accuracy is not as adversely affected as the exact set match accuracy. We hypothesize that this could be attributed

Benchmark	Affected Queries	Exec Acc	Set Match Acc
Spider	206 (19%)	92.3	81.6
Spider-DK	112 (20%)	95	83.9
BIRD	252 (16%)	96.87	-

Table 4.2: Performance of the revised SQL queries on the development set of the benchmarks.

to the absence of ties in the test data used for these benchmarks. An evidence of this is the following two queries, `(Q1) SELECT name, capacity FROM stadium WHERE average = (SELECT max(average) FROM stadium)`, and `(Q2) SELECT name, capacity FROM stadium ORDER BY average DESC LIMIT 1`, labelled as a correct match by the test scripts of Spider.

4.6.2 Human Evaluation

To gain a deeper understanding of the limitations within the benchmarks, we conducted an experiment focused on the widely-used text-to-SQL benchmark, the Spider dataset. Specifically, we evaluated two top-performing methods from the Spider leaderboard: DIN-SQL [50] and T5-large + PICARD [55]. This experiment involved running these methods on the development set of Spider, which comprised 1034 question-query pairs. From the results obtained, we extracted the questions for which both methods failed to produce a correct answer, based on the execution accuracy, resulting in 102 pairs. We then presented these questions, along with the SQL queries generated by the methods as well as the ground truth SQL queries (treating them the same as model-generated queries), to two annotators [1] for labelling. The annotators had access to the database schemas and were tasked with identifying the queries they deemed correct for each question, without knowing which model generated which query or if the query was from the ground truth queries. Annotators could also create databases and validate queries, ensuring a thorough evaluation.

Following our initial labelling process, we wanted to minimize the potential impact of human errors in our evaluation. For this, we identified queries with

[1]The human annotators are the authors of this paper.

Method	Acc	Incon
DIN-SQL [50]	**81.6**	4
T5-large + Picard [55]	25.5	4
Ground Truth	67.3	4

Table 4.3: Accuracy of the SQL queries generated by two methods and the ground truth SQL queries based on human evaluation. In four cases, the two annotators did not agree on a label even after a second round.

inconsistent labels among the annotators and presented them to the annotators. Each annotator was asked to provide an explanation for their assigned labels. In the final stage of evaluation, each annotator was presented the inconsistent queries and the explanations provided by the other annotator. They were then asked if they would revise their labels based on this additional information.

The results of this experiment are presented in Table 4.3. This table presents the outcome of human evaluation on a sample of 102 queries that both DIN-SQL and T5+PICARD methods were deemed incorrect in terms of execution accuracy. SQL experts conducted this evaluation, with 81.6% of these queries judged as correct for DIN-SQL, and only 25.5% for T5+PICARD. Notably, among the reference queries, only 67.3% were deemed correct. Even after the second round of annotation, a few queries (more specifically, four question-query pairs) still exhibit inconsistent labeling by the annotators. The main challenge with these particular pairs is the inherent ambiguity in the questions or the subjectivity of interpretations, which leads to a lack of a definitive answer. Figure 4.6 demonstrates one example of such a question with two possible SQL query as answers.

An intriguing observation emerged from this experiment: the DIN-SQL method, powered by GPT-4, produced the highest number of correct answers, surpassing even the ground truth SQL queries. This finding sheds light on the limitations of the current benchmarks and raises doubts about the reliability of current leaderboards and performance metrics.

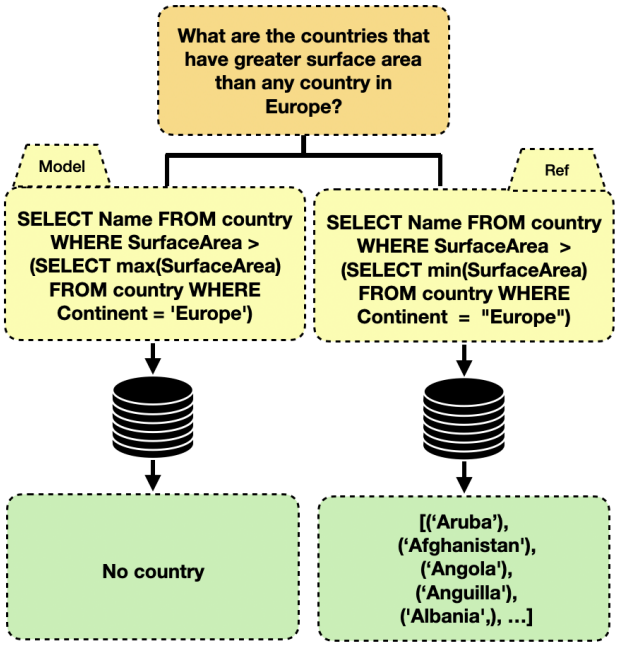

Figure 4.6: An example of a question with two possible SQL queries as the answers. Both of these SQL queries are correct under different interpretations.

4.6.3 Error Analysis of Human Evaluation

We performed an error analysis of the SQL queries that were labelled as incorrect in our human evaluation to better understand the error types and causes and to provide insights into areas for improving the ground truth SQL queries. Additionally, we compared the errors in ground truth queries with those of fine-tuning and prompting approaches. The identified errors, categorized into five groups, are briefly discussed next. The distribution of SQL queries across these groups is depicted in Figure 4.7.

Schema The primary issue responsible for the majority of errors, affecting both the reference SQL queries and the two methods, is the incorrect usage of schemas, which arises when the SQL query utilizes incorrect tables or columns to answer the given question. These errors indicate ambiguities in the database schema and/or questions, as discussed in Section 4.5. Notably, the reference set shows the least number of errors, which is closely followed by DIN-SQL.

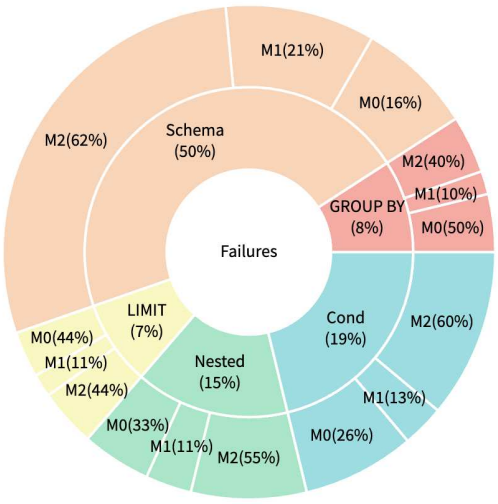

Figure 4.7: Distribution of SQL queries across error groups for the two models being evaluated and the ground truth. M0 refers to SQL queries in the reference (ground truth) set, M1 refers to the DIN-SQL method, and M2 refers to T5+PICARD.

Condition The second-largest group of errors observed pertains to the usage of incorrect conditions within the SQL queries. Unlike the schema group, where the tables and columns were incorrect, in this group, the correct tables and columns are used, but the conditions in the WHERE clause are erroneous. This error primarily manifested in the queries generated by the T5-PICARD method, but was also present in the reference set. The T5 model's tendency to introduce additional columns or omit necessary conditions could be attributed to its smaller size relative to larger models like GPT-4, limiting its grasp of intricate SQL syntax.

Nested The source of this problem is using a non-unique column for the nested SQL query, as also discussed in Section 4.5. Figure 4.5 shows an example of such an error in a SQL query. This error was more common in the SQL queries provided in the reference set as well as those of T5-PICARD.

GROUP BY This category includes queries that incorrectly used GROUP BY, resulting in ambiguity or uncertainty in the result as discussed in Section 4.5. Notably, the reference set showed the largest number of errors, closely

followed by the fine-tuned T5-PICARD. DIN-SQL exhibited the least number of errors.

LIMIT As highlighted in Section 4.5, one of the error scenarios involves not properly handling ties when using the LIMIT keyword. The DIN-SQL method demonstrates a lower incidence of this type of error, attributed to its prompting nature. Conversely, T5-PICARD exhibits identical performance to the ground truth in this particular case.

4.6.4 Standard SQL validation

We undertook an extensive review of the development set of Spider, BIRD, and Spider-DK benchmarks through the lens of standard SQL validation. The objective was to identify some of the problematic queries discussed in Section 4.5 and assess the portability of the benchmarks. As part of this analysis, we migrated the databases and queries of these three benchmarks from Sqlite to PostgreSQL. Our decision to use PostgreSQL, a widely recognized RDBMS, stemmed from its rigorous adherence to SQL standards. Following the migration, we executed every query from the development set on these PostgreSQL databases, with a keen focus on identifying queries that failed during PostgreSQL execution.

Table 4.4 provides a breakdown of queries by error type across all three benchmarks. Notably, errors such as UndefinedColumn, SyntaxError, and UndefinedFunction emerge due to the different SQL formats supported by Sqlite and PostgreSQL. These variances necessitate adjustments to make the queries compatible with PostgreSQL standards. For instance, the Spider dataset frequently showcases errors stemming from PostgreSQL's strict typing conventions. While SQLite allows for comparisons of int with text, PostgreSQL does not. Also, some queries run into problems because of SQLite-exclusive functions, such as strftime and iff, or because PostgreSQL interprets literals in double quotations as column names.

The two other types of failures, group by and Order by, included queries that introduced ambiguities to the benchmarks, as discussed in Section 4.5.

Benchmark	SyntaxErr	UndFunc	UndCol	Order By	Group By
Spider	4	69	211	2	51
Spider-DK	134	62	80	2	30
BIRD	5	103	1	4	20

Table 4.4: Breakdown of SQL errors observed in Spider, BIRD, and Spider-DK, following migration to PostgreSQL.

It should be noted that these benchmarks present a range of issues that are not solely confined to syntax. Challenges related to wrong assumptions on DB content and ambiguities in schema matching are notably pervasive.

4.7 Discussion

Our analysis (§ 4.6.1) reveals the limitations of major text-to-SQL benchmarks, highlighting the fact that even with a perfect model, achieving a perfect accuracy on these benchmarks is not possible. The accuracies presented in Table 4.2 serve as a lose upper bound for the achievable accuracy by models. It is lose because our rewritings were unable to address cases that required manual intervention to reconstruct a correct query. Thus, the upper bound is expected to be lower considering other issues such as wrong assumptions on the database content and ambiguity in schema matching.

Our human evaluation (§ 4.6.2) further supports our claim and provides more insight into the limitations within one of the benchmarks studied. The results in Table 4.3 demonstrate that prompting methods, such as DIN-SQL, are less affected by the inherent limitations of the training set in the benchmarks. However, they are not fully immune because of the few-shot input-output demonstrations that are taken from the train set. On the other hand, fine-tuned approaches, such as T5+PICARD, perfectly mirror the distribution of errors seen in the ground truth queries for types nested, LIMIT, and GROUP BY. The largest number of wrong queries in schema and condition classes belong to our fine-tuned model, due to inability of the model to generate correct SQL queries.

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

This thesis aims to advance the field of text-to-SQL translation by addressing the challenges associated with creating more accurate and efficient Natural Language Interface to Databases (NLIDBs). Through the research presented in the core chapters, we have made significant contributions toward understanding and improving the performance of LLMs in the text-to-SQL domain. The key contributions of this thesis can be summarized as follows:

1. **Decomposed In-context Learning with Self-correction**: Prompting has enabled large language models to achieve impressive performance on numerous NLP tasks across different domains, without requiring a large training set. Prior to our research, the effectiveness of prompting methods utilizing LLMs for the text-to-SQL task was inferior to that of models fine-tuned for the task. To bridge this gap, we have devised a decomposition technique to tackle some of the challenges that caused this disparity. Our extensive experiments on two challenging datasets of Spider and BIRD show that our method significantly improves the performance of prompting across all query classes, producing comparable or even superior results to state-of-the-art fine-tuned approaches.

2. **Decomposed Text-to-SQL with Small Large Language Models**: Before our research, small open-source models lagged behind large proprietary models in performance on the text-to-SQL task. Our two-stage

fine-tuning approach breaks down the task into two simpler components, enabling small open-source models to rival larger ones. Subsequent efforts could focus on enhancing the performance of these stages and exploring improved methods for transferring the output of one stage to the next.

3. **Evaluating Models and Benchmarks**: The reliance on standard text-to-SQL evaluation metrics, namely exact set match accuracy and execution accuracy, has become less reliable as the model performance approaches human-level performance. Our work is the first to systematically study the limitations of these metrics and benchmarks through both human evaluation and query rewriting. Our re-evaluation of well-known benchmarks (Spider, Spider-DK, and BIRD) uncovers common systematic issues that affect the evaluation process and performance estimates, revealing that a significant portion of queries in the train and development sets are impacted by these issues. Incorporating multiple SQL queries as the ground truth and representing different interpretations of queries offer a promising solution to enhance the evaluation process and achieve a more comprehensive and accurate assessment of text-to-SQL models.

The findings of this thesis underscore the importance of task decomposition and fine-tuning strategies in enhancing the performance of LLMs for the text-to-SQL task. The proposed methodologies demonstrate significant improvements over existing approaches, thereby contributing to the development of more robust and scalable NLIDBs suitable for real-world applications.

5.2 Future Work

Given the complexities posed by ambiguous questions, intricate table schemas, the vast quantity of tables and columns, and the disorderly nature of database contents, achieving truly human-like SQL generation remains a considerable challenge. Even the most advanced LLMs, including GPT-4, fall short of

human performance when confronted with these hurdles. This underscores the pressing necessity for more robust and effective strategies to bridge the gap. In this chapter, we discuss the open problems based on the insights we gained doing this thesis.

5.2.1 Schema linking

Schema linking remains a formidable challenge in the field of text-to-SQL due to the inherent complexities of accurately mapping natural language queries to specific database schema elements. This process is crucial for generating correct SQL queries, as it involves identifying references to database schema components and condition values within the queries. However, ambiguities in natural language, such as synonyms, vague references, or the presence of homonyms within the schema, compound the difficulty of schema linking. Moreover, the diverse structures and naming conventions across different databases exacerbate the challenge, as a single term in a query might correspond to multiple columns or tables, or conversely, similar database elements might be referred to by different terms in queries. Additionally, the schema linking performance is directly influenced by the ability of the underlying model to understand and interpret the query in the context of the given database schema, which varies significantly based on the model's training and the specificity of the data. As text-to-SQL models strive for generalizability across domains, the need to adapt schema linking strategies to handle a wide range of databases and query types further highlights the complexity of this task.

5.2.2 Self-reflection

When data scientists craft SQL queries, the process typically involves not just writing a SQL query but executing it, analyzing its results, and iterating over this cycle to refine the query until the desired outcome is achieved. This iterative refinement process, crucial for navigating complex databases and extracting precise information, presents a significant challenge for enhancing the performance of LLMs in SQL generation tasks. Currently, even the most advanced LLMs lack the capability to interact with databases directly or to

introspect and refine their own output based on the execution results of previously generated queries. They operate in a largely one-shot manner, generating SQL queries based on the input prompt without the ability to evaluate the effectiveness of these queries or their results. To bridge this gap, a promising avenue for future research involves the integration of agents and tools that enable LLMs to simulate or enact this iterative process. By equipping LLMs with the ability to "execute" queries in a controlled environment, analyze hypothetical outcomes, and adjust their strategies accordingly, we can move closer to replicating the nuanced and adaptive approach of human data scientists. This could involve developing sophisticated simulation environments or interfaces that allow LLMs to interact with mock databases, incorporating feedback loops that inform the model's ongoing learning and query refinement. Pursuing these innovations could significantly enhance the applicability and accuracy of LLM-generated SQL queries, opening new possibilities for automated data analysis and insight generation.

References

[1] Y. Bai, A. Jones, K. Ndousse, *et al.*, "Training a helpful and harmless assistant with reinforcement learning from human feedback," *arXiv preprint arXiv:2204.05862*, 2022.

[2] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] R. Cai, J. Yuan, B. Xu, and Z. Hao, "Sadga: Structure-aware dual graph aggregation network for text-to-sql," *Advances in Neural Information Processing Systems*, vol. 34, pp. 7664–7676, 2021.

[4] R. Cao, L. Chen, Z. Chen, Y. Zhao, S. Zhu, and K. Yu, "Lgesql: Line graph enhanced text-to-sql model with mixed local and non-local relations," *arXiv preprint arXiv:2106.01093*, 2021.

[5] M. Chen, J. Tworek, H. Jun, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[6] W. Chen, "Large language models are few (1)-shot table reasoners," *arXiv preprint arXiv:2210.06710*, 2022.

[7] D. Choi, M. C. Shin, E. Kim, and D. R. Shin, "Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases," *Computational Linguistics*, vol. 47, no. 2, pp. 309–332, 2021.

[8] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, vol. 30, 2017.

[9] K. Cobbe, V. Kosaraju, M. Bavarian, *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[10] T. Dao, "FlashAttention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[11] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems*, 2022.

[12] M. Davis, *The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions*. Courier Corporation, 2004.

[13] DeepSeek-AI, "Deepseek llm: Scaling open-source language models with longtermism," *arXiv preprint arXiv:2401.02954*, 2024. [Online]. Available: https://github.com/deepseek-ai/DeepSeek-LLM.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[15] L. Dong and M. Lapata, "Language to logical form with neural attention," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 33–43.

[16] X. Dong, C. Zhang, Y. Ge, *et al.*, "C3: Zero-shot text-to-sql with chatgpt," *arXiv preprint arXiv:2307.07306*, 2023.

[17] Y. Gan, X. Chen, Q. Huang, *et al.*, "Towards robustness of text-to-SQL models against synonym substitution," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Online: Association for Computational Linguistics, Aug. 2021, pp. 2505–2515. DOI: 10.18653/v1/2021.acl-long.195. [Online]. Available: https://aclanthology.org/2021.acl-long.195.

[18] Y. Gan, X. Chen, Q. Huang, *et al.*, "Towards robustness of text-to-sql models against synonym substitution," *arXiv preprint arXiv:2106.01065*, 2021.

[19] Y. Gan, X. Chen, and M. Purver, "Exploring underexplored limitations of cross-domain text-to-sql generalization," *arXiv preprint arXiv:2109.05157*, 2021.

[20] Y. Gan, X. Chen, J. Xie, *et al.*, "Natural sql: Making sql easier to infer from natural language specifications," *arXiv preprint arXiv:2109.05153*, 2021.

[21] D. Gao, H. Wang, Y. Li, *et al.*, "Text-to-sql empowered by large language models: A benchmark evaluation," *arXiv preprint arXiv:2308.15363*, 2023.

[22] A. Graves and A. Graves, "Long short-term memory," *Supervised sequence labelling with recurrent neural networks*, pp. 37–45, 2012.

[23] J. Guo, Z. Zhan, Y. Gao, *et al.*, "Towards complex text-to-sql in cross-domain database with intermediate representation," *arXiv preprint arXiv:1905.08205*, 2019.

[24] Z. Guo, M. Yan, J. Qi, *et al.*, "Few-shot table-to-text generation with prompt planning and knowledge memorization," *arXiv preprint arXiv:2302.04415*, 2023.

[25] J. Herzig, P. K. Nowak, T. Müller, F. Piccinno, and J. M. Eisenschlos, "Tapas: Weakly supervised table parsing via pre-training," *arXiv preprint arXiv:2004.02349*, 2020.

[26] J. Huang, Y. Wang, Y. Wang, Y. Dong, and Y. Xiao, "Relation aware semi-autoregressive semantic parsing for nl2sql," *arXiv preprint arXiv:2108.00804*, 2021.

[27] B. Hui, X. Shi, R. Geng, *et al.*, "Improving text-to-sql with schema dependency learning," *arXiv preprint arXiv:2103.04399*, 2021.

[28] W. Hwang, J. Yim, S. Park, and M. Seo, "A comprehensive exploration on wikisql with table-aware word contextualization," *arXiv preprint arXiv:1902.01069*, 2019.

[29] A. Q. Jiang, A. Sablayrolles, A. Mensch, *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[30] R. Kate, "Transforming meaning representation grammars to improve semantic parsing," in *CoNLL 2008: Proceedings of the Twelfth Conference on Computational Natural Language Learning*, 2008, pp. 33–40.

[31] T. Khot, H. Trivedi, M. Finlayson, *et al.*, "Decomposed prompting: A modular approach for solving complex tasks," *arXiv preprint arXiv:2210.02406*, 2022.

[32] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *arXiv preprint arXiv:2205.11916*, 2022.

[33] B. Lake and M. Baroni, "Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks," in *International conference on machine learning*, PMLR, 2018, pp. 2873–2882.

[34] H. Lee, S. Phatale, H. Mansoor, *et al.*, "Rlaif: Scaling reinforcement learning from human feedback with ai feedback," *arXiv preprint arXiv:2309.00267*, 2023.

[35] W. Lei, W. Wang, Z. Ma, *et al.*, "Re-examining the role of schema linking in text-to-sql," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 6943–6954.

[36] F. Li and H. V. Jagadish, "Constructing an interactive natural language interface for relational databases," *Proceedings of the VLDB Endowment*, vol. 8, no. 1, pp. 73–84, 2014.

[37] H. Li, J. Zhang, C. Li, and H. Chen, "Decoupling the skeleton parsing and schema linking for text-to-sql," *arXiv preprint arXiv:2302.05965*, 2023.

[38] H. Li, J. Zhang, C. Li, and H. Chen, "Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, 2023, pp. 13 067–13 075.

[39] J. Li, B. Hui, R. Cheng, *et al.*, "Graphix-t5: Mixing pre-trained transformers with graph-aware layers for text-to-sql parsing," *arXiv preprint arXiv:2301.07507*, 2023.

[40] J. Li, B. Hui, G. Qu, *et al.*, *Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls*, 2023. arXiv: 2305.03111 [cs.CL].

[41] J. Li, B. Hui, G. Qu, *et al.*, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," *arXiv preprint arXiv:2305.03111*, 2023.

[42] Y. Li, H. Yang, and H. Jagadish, "Nalix: A generic natural language search environment for xml data," *ACM Transactions on database systems (TODS)*, vol. 32, no. 4, 30–es, 2007.

[43] X. V. Lin, R. Socher, and C. Xiong, "Bridging textual and tabular data for cross-domain text-to-sql semantic parsing," *arXiv preprint arXiv:2012.12627*, 2020.

[44] A. Liu, X. Hu, L. Wen, and P. S. Yu, "A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability," *arXiv preprint arXiv:2303.13547*, 2023.

[45] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[46] S. Mishra, D. Khashabi, C. Baral, and H. Hajishirzi, "Cross-task generalization via natural language crowdsourcing instructions," *arXiv preprint arXiv:2104.08773*, 2021.

[47] L. Ouyang, J. Wu, X. Jiang, *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[48] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates, "Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability," in *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, 2004, pp. 141–147.

[49] A.-M. Popescu, O. Etzioni, and H. Kautz, "Towards a theory of natural language interfaces to databases," in *Proceedings of the 8th international conference on Intelligent user interfaces*, 2003, pp. 149–157.

[50] M. Pourreza and D. Rafiei, "Din-sql: Decomposed in-context learning of text-to-sql with self-correction," *arXiv preprint arXiv:2304.11015*, 2023.

[51] J. Qi, J. Tang, Z. He, *et al.*, "Rasat: Integrating relational structures into pretrained seq2seq model for text-to-sql," *arXiv preprint arXiv:2205.06983*, 2022.

[52] B. Qin, B. Hui, L. Wang, *et al.*, "A survey on text-to-sql parsing: Concepts, methods, and future directions," *arXiv preprint arXiv:2208.13629*, 2022.

[53] N. Rajkumar, R. Li, and D. Bahdanau, "Evaluating the text-to-sql capabilities of large language models," *arXiv preprint arXiv:2204.00498*, 2022.

[54] O. Rubin and J. Berant, "Smbop: Semi-autoregressive bottom-up semantic parsing," *arXiv preprint arXiv:2010.12412*, 2020.

[55] T. Scholak, N. Schucher, and D. Bahdanau, "Picard: Parsing incrementally for constrained auto-regressive decoding from language models," *arXiv preprint arXiv:2109.05093*, 2021.

[56] J. Sen, C. Lei, A. Quamar, *et al.*, "Athena++ natural language querying for complex nested sql queries," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2747–2759, 2020.

[57] N. Stiennon, L. Ouyang, J. Wu, *et al.*, "Learning to summarize with human feedback," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3008–3021, 2020.

[58] N. Stratica, L. Kosseim, and B. C. Desai, "Using semantic templates for a natural language interface to the cindi virtual library," *Data & Knowledge Engineering*, vol. 55, no. 1, pp. 4–19, 2005.

[59] R. Sun, S. O. Arik, H. Nakhost, *et al.*, "Sql-palm: Improved large language modeladaptation for text-to-sql," *arXiv preprint arXiv:2306.00739*, 2023.

[60] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[61] R. Thoppilan, D. De Freitas, J. Hall, *et al.*, "Lamda: Language models for dialog applications," *arXiv preprint arXiv:2201.08239*, 2022.

[62] H. Touvron, L. Martin, K. Stone, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[63] L. Tunstall, E. Beeching, N. Lambert, *et al.*, "Zephyr: Direct distillation of lm alignment," *arXiv preprint arXiv:2310.16944*, 2023.

[64] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[65] S. Victor, W. Albert, R. Colin, *et al.*, "Multitask prompted training enables zero-shot task generalization," in *International Conference on Learning Representations*, 2022.

[66] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, "Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers," *arXiv preprint arXiv:1911.04942*, 2019.

[67] J. Wei, M. Bosma, V. Y. Zhao, *et al.*, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652*, 2021.

[68] J. Wei, Y. Tay, R. Bommasani, *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[69] J. Wei, X. Wang, D. Schuurmans, *et al.*, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[70] Wikipedia contributors, *Sql — Wikipedia, the free encyclopedia*, [Online; accessed 9-March-2024], 2024. [Online]. Available: `https://en.wikipedia.org/wiki/SQL`.

[71] K. Xu, Y. Wang, Y. Wang, Z. Wen, and Y. Dong, "Sead: End-to-end text-to-sql generation with schema-aware denoising," *arXiv preprint arXiv:2105.07911*, 2021.

[72] X. Xu, C. Liu, and D. Song, "Sqlnet: Generating structured queries from natural language without reinforcement learning," *arXiv preprint arXiv:1711.04436*, 2017.

[73] K. Xuan, Y. Wang, Y. Wang, Z. Wen, and Y. Dong, "Sead: End-to-end text-to-sql generation with schema-aware denoising," *arXiv preprint arXiv:2105.07911*, 2021.

[74] P. Yin, G. Neubig, W.-t. Yih, and S. Riedel, "Tabert: Pretraining for joint understanding of textual and tabular data," *arXiv preprint arXiv:2005.08314*, 2020.

[75] T. Yu, C.-S. Wu, X. V. Lin, *et al.*, "Grappa: Grammar-augmented pre-training for table semantic parsing," *arXiv preprint arXiv:2009.13845*, 2020.

[76] T. Yu, M. Yasunaga, K. Yang, *et al.*, "Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task," *arXiv preprint arXiv:1810.05237*, 2018.

[77] T. Yu, R. Zhang, K. Yang, *et al.*, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," *arXiv preprint arXiv:1809.08887*, 2018.

[78] J. M. Zelle and R. J. Mooney, "Learning to parse database queries using inductive logic programming," in *Proceedings of the national conference on artificial intelligence*, 1996, pp. 1050–1055.

[79] L. Zeng, S. H. K. Parthasarathi, and D. Hakkani-Tur, "N-best hypotheses reranking for text-to-sql systems," *arXiv preprint arXiv:2210.10668*, 2022.

[80] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," *arXiv preprint arXiv:2210.03493*, 2022.

[81] Y. Zhao, J. Jiang, Y. Hu, *et al.*, "Importance of synthesizing high-quality data for text-to-sql parsing," *arXiv preprint arXiv:2212.08785*, 2022.

[82] R. Zhong, C. Snell, D. Klein, and J. Eisner, "Active programming by example with a natural language prior," *arXiv preprint arXiv:2205.12422*, 2022.

[83] R. Zhong, T. Yu, and D. Klein, "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, 2020.

[84] R. Zhong, T. Yu, and D. Klein, "Semantic evaluation for text-to-sql with distilled test suites," *arXiv preprint arXiv:2010.02840*, 2020.

[85] V. Zhong, M. Lewis, S. I. Wang, and L. Zettlemoyer, "Grounded adaptation for zero-shot executable semantic parsing," *arXiv preprint arXiv:2009.07396*, 2020.

[86] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *CoRR*, vol. abs/1709.00103, 2017.

[87] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *arXiv preprint arXiv:1709.00103*, 2017.

[88] D. Zhou, N. Schärli, L. Hou, *et al.*, "Least-to-most prompting enables complex reasoning in large language models," *arXiv preprint arXiv:2205.10625*, 2022.

[89] D. M. Ziegler, N. Stiennon, J. Wu, *et al.*, "Fine-tuning language models from human preferences," *arXiv preprint arXiv:1909.08593*, 2019.

Appendix A

Prompts

This section presents a comprehensive list of all the prompts utilized in the four modules of our proposed methodology in Chapter 2 for both the GPT-4 and CodeX models. The prompts used for each module are provided in detail to allow for easy replication and understanding of the approach. Additionally, we have also included the prompt we used for the few-shot and zero-shot implementations of our method.

A.1 Few-shot prompting

Create SQL queries for the given questions.

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year,building,room_number, time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]

Q: "Find the buildings which have rooms with capacity more than 50."
SQL: SELECT DISTINCT building FROM classroom WHERE capacity ¿ 50

Q: "Find the room number of the rooms which can sit 50 to 100 students and their buildings."
SQL: SELECT building , room_number FROM classroom WHERE capacity BETWEEN 50 AND 100

Q: "Give the name of the student in the History department with the most credits."
SQL: SELECT name FROM student WHERE dept_name = 'History' ORDER BY tot_cred DESC LIMIT 1

Q: "Find the total budgets of the Marketing or Finance department."
SQL: SELECT sum(budget) FROM department WHERE dept_name = 'Mar-

keting' OR dept_name = 'Finance'

Q: "Find the department name of the instructor whose name contains 'Soisalon'."
SQL: SELECT dept_name FROM instructor WHERE name LIKE '%Soisalon%'

Q: "What is the name of the department with the most credits?"
SQL: SELECT dept_name FROM course GROUP BY dept_name ORDER BY sum(credits) DESC LIMIT 1

Q: "How many instructors teach a course in the Spring of 2010?"
SQL: SELECT COUNT (DISTINCT ID) FROM teaches WHERE semester = 'Spring' AND YEAR = 2010

Q: "Find the name of the students and their department names sorted by their total credits in ascending order."
SQL: SELECT name , dept_name FROM student ORDER BY tot_cred

Q: "Find the year which offers the largest number of courses."
SQL: SELECT YEAR FROM SECTION GROUP BY YEAR ORDER BY count(*) DESC LIMIT 1

Q: "What are the names and average salaries for departments with average salary higher than 42000?"
SQL: SELECT dept_name , AVG (salary) FROM instructor GROUP BY dept_name HAVING AVG (salary) ¿ 42000

Q: "How many rooms in each building have a capacity of over 50?"
SQL: SELECT count(*) , building FROM classroom WHERE capacity ¿ 50 GROUP BY building

Q: "Find the names of the top 3 departments that provide the largest amount of courses?"
SQL: SELECT dept_name FROM course GROUP BY dept_name ORDER BY count(*) DESC LIMIT 3

Q: "Find the maximum and average capacity among rooms in each building."
SQL: SELECT max(capacity) , avg(capacity) , building FROM classroom GROUP BY building

Q: "Find the title of the course that is offered by more than one department."
SQL: SELECT title FROM course GROUP BY title HAVING count(*) ¿ 1

Q: "Find the total budgets of the Marketing or Finance department."
SQL: SELECT sum(budget) FROM department WHERE dept_name = 'Marketing' OR dept_name = 'Finance'

Q: "Find the name and building of the department with the highest budget."
SQL: SELECT dept_name , building FROM department ORDER BY budget DESC LIMIT 1

Q: "What is the name and building of the departments whose budget is more than the average budget?"
SQL: SELECT dept_name , building FROM department WHERE budget ¿ (SELECT avg(budget) FROM department)

Q: "Find the total number of students and total number of instructors for each department."
SQL: SELECT count(DISTINCT T2.id) , count(DISTINCT T3.id) , T3.dept_name FROM department AS T1 JOIN student AS T2 ON T1.dept_name = T2.dept_name JOIN instructor AS T3 ON T1.dept_name = T3.dept_name GROUP BY

T3.dept_name

Q: "Find the title of courses that have two prerequisites?"
SQL: SELECT T1.title FROM course AS T1 JOIN prereq AS T2 ON T1.course_id = T2.course_id GROUP BY T2.course_id HAVING count(*) = 2

Q: "Find the name of students who took any class in the years of 2009 and 2010."
SQL: SELECT DISTINCT T1.name FROM student AS T1 JOIN takes AS T2 ON T1.id = T2.id WHERE T2.YEAR = 2009 OR T2.YEAR = 2010

Q: "list in alphabetic order all course names and their instructors' names in year 2008."
SQL: SELECT T1.title , T3.name FROM course AS T1 JOIN teaches AS T2 ON T1.course_id = T2.course_id JOIN instructor AS T3 ON T2.id = T3.id WHERE T2.YEAR = 2008 ORDER BY T1.title

Q: "Find the title of courses that have two prerequisites?"
SQL: SELECT T1.title FROM course AS T1 JOIN prereq AS T2 ON T1.course_id = T2.course_id GROUP BY T2.course_id HAVING count(*) = 2

Q: "Find the name and building of the department with the highest budget."
SQL: SELECT dept_name , building FROM department ORDER BY budget DESC LIMIT 1

Q: "Find the title, credit, and department name of courses that have more than one prerequisites?"
SQL: SELECT T1.title , T1.credits , T1.dept_name FROM course AS T1 JOIN prereq AS T2 ON T1.course_id = T2.course_id GROUP BY T2.course_id HAVING count(*) ¿ 1

Q: "Give the name and building of the departments with greater than average budget."
SQL: SELECT dept_name , building FROM department WHERE budget ¿ (SELECT avg(budget) FROM department)

Q: "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010."
SQL: SELECT id FROM teaches WHERE semester = 'Fall' AND YEAR = 2009 EXCEPT SELECT id FROM teaches WHERE semester = 'Spring' AND YEAR = 2010

Q: "Find the name of the courses that do not have any prerequisite?"
SQL: SELECT title FROM course WHERE course_id NOT IN (SELECT course_id FROM prereq)

Q: "Find the salaries of all distinct instructors that are less than the largest salary."
SQL: SELECT DISTINCT salary FROM instructor WHERE salary ¡ (SELECT max(salary) FROM instructor)

Q: "Find the names of students who have taken any course in the fall semester of year 2003."
SQL: SELECT name FROM student WHERE id IN (SELECT id FROM takes WHERE semester = 'Fall' AND YEAR = 2003)

Q: "Find the minimum salary for the departments whose average salary is above the average payment of all instructors."
SQL: SELECT min(salary) , dept_name FROM instructor GROUP BY dept_name HAVING avg(salary) ¿ (SELECT avg(salary) FROM instructor)

Q: "What is the course title of the prerequisite of course Mobile Computing?"

SQL: SELECT title FROM course WHERE course_id IN (SELECT T1.prereq_id FROM prereq AS T1 JOIN course AS T2 ON T1.course_id = T2.course_id WHERE T2.title = 'Mobile Computing')

Q: "Give the title and credits for the course that is taught in the classroom with the greatest capacity."
SQL: SELECT T3.title , T3.credits FROM classroom AS T1 JOIN SECTION AS T2 ON T1.building = T2.building AND T1.room_number = T2.room_number JOIN course AS T3 ON T2.course_id = T3.course_id WHERE T1.capacity = (SELECT max(capacity) FROM classroom)

A.1.1 Schema linking prompt

Find the schema_links for generating SQL queries for each question based on the database schema and Foreign keys.

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]
Q: "Show the status of the city that has hosted the greatest number of competitions."
A: Let's think step by step. In the question "Show the status of the city that has hosted the greatest number of competitions.", we are asked:
"the status of the city" so we need column = [city.Status]
"greatest number of competitions" so we need column = [farm_competition.*]
Based on the columns and tables, we need these Foreign_keys = [farm_competition.Host_city_ID

= city.City_ID].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = []. So the Schema_links are:
Schema_links: [city.Status,farm_competition.Host_city_ID = city.City_ID,farm_competition.*]

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]
Q: "How many heads of the departments are older than 56 ?"
A: Let's think step by step. In the question "How many heads of the departments are older than 56 ?", we are asked:
"How many heads of the departments" so we need column = [head.*]
"older" so we need column = [head.age]
Based on the columns and tables, we need these Foreign_keys = [].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = [56]. So the Schema_links are:
Schema_links: [head.*,head.age,56]

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]
Q: "what are the distinct creation years of the departments managed by a secretary born in state 'Alabama'?"
A: Let's think step by step. In the question "what are the distinct creation years of the departments managed by a secretary born in state 'Alabama'?", we are asked:
"distinct creation years of the departments" so we need column = [depart-

ment.Creation]
"departments managed by" so we need column = [management.department_ID]
"born in" so we need column = [head.born_state]
Based on the columns and tables, we need these Foreign_keys = [department.Department_ID = management.department_ID,management.head_ID = head.head_ID].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = ['Alabama']. So the Schema_links are:
Schema_links: [department.Creation,department.Department_ID = management.department_ID, head.head_ID = management.head_ID,head.born_state,'Alabama']

Table Addresses, columns = [*,address_id,line_1,line_2,city,zip_postcode
,state_province_county,country]
Table Candidate_Assessments, columns = [*,candidate_id,qualification,assessment_date
,asessment_outcome_code]
Table Candidates, columns = [*,candidate_id,candidate_details]
Table Courses, columns = [*,course_id,course_name,course_description,other_details]
Table People, columns = [*,person_id,first_name,middle_name,
last_name,cell_mobile_number,email_address,login_name,password]
Table People_Addresses, columns = [*,person_address_id,
person_id,address_id,date_from,date_to]
Table Student_Course_Attendance, columns = [*,student_id,course_id,date_of_attendance]
Table Student_Course_Registrations, columns = [*,student_id,course_id,registration_date]
Table Students, columns = [*,student_id,student_details]
Foreign_keys = [Students.student_id = People.person_id,People_Addresses.address_id
= Addresses.address_id,People_Addresses.person_id =
People.person_id,Student_Course_Registrations.course_id =
Courses.course_id,Student_Course_Registrations.student_id =
Students.student_id,Student_Course_Attendance.student_id =
Student_Course_Registrations.student_id,Student_Course_Attendance.course_id
= Student_Course_Registrations.course_id,Candidates.candidate_id =
People.person_id,Candidate_Assessments.candidate_id = Candidates.candidate_id]

Q: "List the id of students who never attends courses?"
A: Let's think step by step. In the question "List the id of students who never attends courses?", we are asked:
"id of students" so we need column = [Students.student_id]
"never attends courses" so we need column = [Student_Course_Attendance.student_id]
Based on the columns and tables, we need these Foreign_keys = [Students.student_id = Student_Course_Attendance.student_id].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = []. So the Schema_links are:
Schema_links: [Students.student_id = Student_Course_Attendance.student_id]

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year,building,room_number,time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]
Foreign_keys = [course.dept_name = department.dept_name,instructor.dept_name = department.dept_name,section.building = classroom.building
,section.room_number = classroom.room_number
,section.course_id = course.course_id,teaches.ID = instructor.ID,teaches.course_id = section.course_id,teaches.sec_id = section.sec_id,
teaches.semester = section.semester,teaches.year = section.year,student.dept_name = department.dept_name,
takes.ID = student.ID,takes.course_id = section.course_id,
takes.sec_id = section.sec_id,takes.semester = section.semester,
takes.year = section.year,advisor.s_ID = student.ID,

advisor.i_ID = instructor.ID,prereq.prereq_id = course.course_id,prereq.course_id = course.course_id]

Q: "Give the title of the course offered in Chandler during the Fall of 2010."

A: Let's think step by step. In the question "Give the title of the course offered in Chandler during the Fall of 2010.", we are asked:

"title of the course" so we need column = [course.title]

"course offered in Chandler" so we need column = [SECTION.building]

"during the Fall" so we need column = [SECTION.semester]

"of 2010" so we need column = [SECTION.year]

Based on the columns and tables, we need these Foreign_keys = [course.course_id = SECTION.course_id].

Based on the tables, columns, and Foreign_keys, The set of possible cell values are = [Chandler,Fall,2010]. So the Schema_links are:

Schema_links: [course.title,course.course_id = SECTION.course_id,SECTION.building,SECTION.year

,SECTION.semester,Chandler,Fall,2010]

Table advisor, columns = [*,s_ID,i_ID]

Table classroom, columns = [*,building,room_number,capacity]

Table course, columns = [*,course_id,title,dept_name,credits]

Table department, columns = [*,dept_name,building,budget]

Table instructor, columns = [*,ID,name,dept_name,salary]

Table prereq, columns = [*,course_id,prereq_id]

Table section, columns = [*,course_id,sec_id,semester,year,

building,room_number,time_slot_id]

Table student, columns = [*,ID,name,dept_name,tot_cred]

Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]

Table teaches, columns = [*,ID,course_id,sec_id,semester,year]

Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]

Foreign_keys = [course.dept_name = department.dept_name,instructor.dept_name = department.dept_name,

section.building = classroom.building,section.room_number = classroom.room_number,

section.course_id = course.course_id,teaches.ID = instructor.ID,teaches.course_id = section.course_id,
teaches.sec_id = section.sec_id,teaches.semester = section.semester,teaches.year = section.year,
student.dept_name = department.dept_name,takes.ID = student.ID,takes.course_id = section.course_id,
takes.sec_id = section.sec_id,takes.semester = section.semester,
takes.year = section.year,advisor.s_ID = student.ID,advisor.i_ID = instructor.ID,
prereq.prereq_id = course.course_id,prereq.course_id = course.course_id]
Q: "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010."
A: Let's think step by step. In the question "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010.", we are asked:
"id of instructors who taught " so we need column = [teaches.id]
"taught a class in" so we need column = [teaches.semester,teaches.year]
Based on the columns and tables, we need these Foreign_keys = [].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = [Fall,2009,Spring,2010]. So the Schema_links are:
Schema_links: [teaches.id,teaches.semester,teaches.year,Fall,2009,Spring,2010]

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year
,building,room_number,time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]

Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]
Foreign_keys = [course.dept_name = department.dept_name,instructor.dept_name
= department.dept_name,
section.building = classroom.building,section.room_number = classroom.room_number,
section.course_id = course.course_id,teaches.ID = instructor.ID,teaches.course_id
= section.course_id,teaches.sec_id = section.sec_id,
teaches.semester = section.semester,teaches.year = section.year,student.dept_name
= department.dept_name,takes.ID = student.ID,takes.course_id = section.course_id,
takes.sec_id = section.sec_id,takes.semester = section.semester,takes.year =
section.year,advisor.s_ID = student.ID,
advisor.i_ID = instructor.ID,prereq.prereq_id = course.course_id,prereq.course_id
= course.course_id]
Q: "Find the buildings which have rooms with capacity more than 50."
A: Let's think step by step. In the question "Find the buildings which have
rooms with capacity more than 50.", we are asked:
"the buildings which have rooms" so we need column = [classroom.capacity]
"rooms with capacity" so we need column = [classroom.building]
Based on the columns and tables, we need these Foreign_keys = [].
Based on the tables, columns, and Foreign_keys, The set of possible cell values
are = [50]. So the Schema_links are:
Schema_links: [classroom.building,classroom.capacity,50]

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]
Q: "Show the status shared by cities with population bigger than 1500 and
smaller than 500."

A: Let's think step by step. In the question "Show the status shared by cities with population bigger than 1500 and smaller than 500.", we are asked:
"the status shared by cities" so we need column = [city.Status]
"cities with population" so we need column = [city.Population]
Based on the columns and tables, we need these Foreign_keys = [].
Based on the tables, columns, and Foreign_keys, The set of possible cell values are = [1500,500]. So the Schema_links are:
Schema_links: [city.Status,city.Population,1500,500]

A.1.2 Classification & decomposition prompt

For the given question, classify it as EASY, NON-NESTED, or NESTED based on nested queries and JOIN.

if need nested queries: predict NESTED
elif need JOIN and don't need nested queries: predict NON-NESTED
elif don't need JOIN and don't need nested queries: predict EASY

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year
,building,room_number,time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]
Foreign_keys = [course.dept_name = department.dept_name,instructor.dept_name
= department.dept_name,section.building = classroom.building,section.room_number

= classroom.room_number,
section.course_id = course.course_id,teaches.ID = instructor.ID,teaches.course_id
= section.course_id,teaches.sec_id = section.sec_id,teaches.semester = section.semester,
teaches.year = section.year,student.dept_name = department.dept_name,takes.ID
= student.ID,takes.course_id = section.course_id,takes.sec_id = section.sec_id,takes.semester
= section.semester,takes.year = section.year,advisor.s_ID = student.ID,
advisor.i_ID = instructor.ID,prereq.prereq_id = course.course_id,prereq.course_id
= course.course_id]

Q: "Find the buildings which have rooms with capacity more than 50."
schema_links: [classroom.building,classroom.capacity,50]
A: Let's think step by step. The SQL query for the question "Find the buildings which have rooms with capacity more than 50." needs these tables = [classroom], so we don't need JOIN.
Plus, it doesn't require nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we don't need JOIN and don't need nested queries, then the the SQL query can be classified as "EASY".
Label: "EASY"

Q: "What are the names of all instructors who advise students in the math depart sorted by total credits of the student."
schema_links: [advisor.i_id = instructor.id,advisor.s_id = student.id,instructor.name, student.dept_name,student.tot_cred,math]
A: Let's think step by step. The SQL query for the question "What are the names of all instructors who advise students in the math depart sorted by total credits of the student." needs these tables = [advisor,instructor,student], so we need JOIN.
Plus, it doesn't need nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we need JOIN and don't need nested queries, then the the SQL query can be classified as "NON-NESTED".

Label: "NON-NESTED"

Q: "How many courses that do not have prerequisite?"
schema_links: [course.*,course.course_id = prereq.course_id]
A: Let's think step by step. The SQL query for the question "How many courses that do not have prerequisite?" needs these tables = [course,prereq], so we need JOIN.
Plus, it requires nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = ["Which courses have prerequisite?"].
So, we need JOIN and need nested queries, then the the SQL query can be classified as "NESTED".
Label: "NESTED"

Q: "Find the room number of the rooms which can sit 50 to 100 students and their buildings."
schema_links: [classroom.building,classroom.room_number,classroom.capacity,50,100]
A: Let's think step by step. The SQL query for the question "Find the room number of the rooms which can sit 50 to 100 students and their buildings." needs these tables = [classroom], so we don't need JOIN.
Plus, it doesn't require nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we don't need JOIN and don't need nested queries, then the the SQL query can be classified as "EASY".
Label: "EASY"

Q: "Find the title of course that is provided by both Statistics and Psychology departments."
schema_links: [course.title,course.dept_name,Statistics,Psychology]
A: Let's think step by step. The SQL query for the question "Find the title of course that is provided by both Statistics and Psychology departments." needs these tables = [course], so we don't need JOIN.

Plus, it requires nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = ["Find the titles of courses that is provided by Psychology departments"].
So, we don't need JOIN and need nested queries, then the the SQL query can be classified as "NESTED".
Label: "NESTED"

Q: "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010."
schema_links: [teaches.id,teaches.semester,teaches.year,Fall,2009,Spring,2010]
A: Let's think step by step. The SQL query for the question "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010." needs these tables = [teaches], so we don't need JOIN.
Plus, it requires nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = ["Find the id of instructors who taught a class in Spring 2010"].
So, we don't need JOIN and need nested queries, then the the SQL query can be classified as "NESTED".
Label: "NESTED"

Q: "Find the name of the department that offers the highest total credits?"
schema_links: [course.dept_name,course.credits]
A: Let's think step by step. The SQL query for the question "Find the name of the department that offers the highest total credits?." needs these tables = [course], so we don't need JOIN.
Plus, it doesn't require nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we don't need JOIN and don't need nested queries, then the the SQL query can be classified as "EASY".
Label: "EASY"

Q: "Find the total number of students and total number of instructors for

each department."
schema_links: [department.dept_name = instructor.dept_name,student.id
,student.dept_name = department.dept_name,instructor.id]
A: Let's think step by step. The SQL query for the question "Find the total number of students and total number of instructors for each department." needs these tables = [department,instructor,student], so we need JOIN.
Plus, it doesn't need nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = [""].
So, we need JOIN and don't need nested queries, then the the SQL query can be classified as "NON-NESTED".
Label: "NON-NESTED"

Q: "Give the name and building of the departments with greater than average budget."
schema_links: [department.budget,department.dept_name,department.building]
A: Let's think step by step. The SQL query for the question "Give the name and building of the departments with greater than average budget." needs these tables = [department], so we don't need JOIN.
Plus, it requires nested queries with (INTERSECT, UNION, EXCEPT, IN, NOT IN), and we need the answer to the questions = ["What is the average budget of the departments"].
So, we don't need JOIN and need nested queries, then the the SQL query can be classified as "NESTED".
Label: "NESTED"

A.2 SQL generation

A.2.1 Easy Class

Use the the schema links to generate the SQL queries for each of the questions.

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year,
building,room_number,time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]

Q: "Find the buildings which have rooms with capacity more than 50."
Schema_links: [classroom.building,classroom.capacity,50]
SQL: SELECT DISTINCT building FROM classroom WHERE capacity ¿ 50

Q: "Find the room number of the rooms which can sit 50 to 100 students and their buildings."
Schema_links: [classroom.building,classroom.room_number,classroom.capacity,50,100]
SQL: SELECT building , room_number FROM classroom WHERE capacity BETWEEN 50 AND 100

Q: "Give the name of the student in the History department with the most credits."
Schema_links: [student.name,student.dept_name,student.tot_cred,History]
SQL: SELECT name FROM student WHERE dept_name = 'History' ORDER BY tot_cred DESC LIMIT 1

Q: "Find the total budgets of the Marketing or Finance department."
Schema_links: [department.budget,department.dept_name,Marketing,Finance]
SQL: SELECT sum(budget) FROM department WHERE dept_name = 'Mar-

keting' OR dept_name = 'Finance'

Q: "Find the department name of the instructor whose name contains 'Soisalon'."
Schema_links: [instructor.dept_name,instructor.name,Soisalon]
SQL: SELECT dept_name FROM instructor WHERE name LIKE '%Soisalon%'

Q: "What is the name of the department with the most credits?"
Schema_links: [course.dept_name,course.credits]
SQL: SELECT dept_name FROM course GROUP BY dept_name ORDER BY sum(credits) DESC LIMIT 1

Q: "How many instructors teach a course in the Spring of 2010?"
Schema_links: [teaches.ID,teaches.semester,teaches.YEAR,Spring,2010]
SQL: SELECT COUNT (DISTINCT ID) FROM teaches WHERE semester = 'Spring' AND YEAR = 2010

Q: "Find the name of the students and their department names sorted by their total credits in ascending order."
Schema_links: [student.name,student.dept_name,student.tot_cred]
SQL: SELECT name , dept_name FROM student ORDER BY tot_cred

Q: "Find the year which offers the largest number of courses."
Schema_links: [SECTION.YEAR,SECTION.*]
SQL: SELECT YEAR FROM SECTION GROUP BY YEAR ORDER BY count(*) DESC LIMIT 1

Q: "What are the names and average salaries for departments with average salary higher than 42000?"
Schema_links: [instructor.dept_name,instructor.salary,42000]
SQL: SELECT dept_name , AVG (salary) FROM instructor GROUP BY

dept_name HAVING AVG (salary) ¿ 42000

Q: "How many rooms in each building have a capacity of over 50?"
Schema_links: [classroom.*,classroom.building,classroom.capacity,50]
SQL: SELECT count(*) , building FROM classroom WHERE capacity ¿ 50 GROUP BY building

Q: "Find the names of the top 3 departments that provide the largest amount of courses?"
Schema_links: [course.dept_name,course.*]
SQL: SELECT dept_name FROM course GROUP BY dept_name ORDER BY count(*) DESC LIMIT 3

Q: "Find the maximum and average capacity among rooms in each building."
Schema_links: [classroom.building,classroom.capacity]
SQL: SELECT max(capacity) , avg(capacity) , building FROM classroom GROUP BY building

Q: "Find the title of the course that is offered by more than one department."
Schema_links: [course.title]
SQL: SELECT title FROM course GROUP BY title HAVING count(*) ¿ 1

A.2.2 Non-Nested Complex

Use the the schema links and Intermediate_representation to generate the SQL queries for each of the questions.

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

Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year,
building,room_number,time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,
start_hr,start_min,end_hr,end_min]
Foreign_keys = [course.dept_name = department.dept_name,
instructor.dept_name = department.dept_name,section.building = classroom.building,
section.room_number = classroom.room_number,section.course_id = course.course_id,teaches.ID
= instructor.ID,teaches.course_id = section.course_id,
teaches.sec_id = section.sec_id,teaches.semester = section.semester,teaches.year
= section.year,
student.dept_name = department.dept_name,takes.ID = student.ID,takes.course_id
= section.course_id,takes.sec_id = section.sec_id,takes.semester = section.semester,
takes.year = section.year,advisor.s_ID = student.ID,advisor.i_ID = instructor.ID,prereq.prereq_id = course.course_id,
prereq.course_id = course.course_id]

Q: "Find the total budgets of the Marketing or Finance department."
Schema_links: [department.budget,department.dept_name,Marketing,Finance]
A: Let's think step by step. For creating the SQL for the given question, we need to join these tables = [].
First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select sum(department.budget) from department where department.dept_name = "Marketing" or department.dept_name = "Finance"
SQL: SELECT sum(budget) FROM department WHERE dept_name = 'Marketing' OR dept_name = 'Finance'

Q: "Find the name and building of the department with the highest budget."
Schema_links: [department.budget,department.dept_name,department.building]
A: Let's think step by step. For creating the SQL for the given question, we need to join these tables = [].
First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select department.dept_name , department.building from department order by department.budget desc limit 1
SQL: SELECT dept_name , building FROM department ORDER BY budget DESC LIMIT 1

Q: "Find the total number of students and total number of instructors for each department."
Schema_links: [department.dept_name = student.dept_name,student.id, department.dept_name = instructor.dept_name,instructor.id]
A: Let's think step by step. For creating the SQL for the given question, we need to join these tables = [department,student,instructor].
First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: "select count(distinct student.ID) , count(distinct instructor.ID) , department.dept_name from department group by instructor.dept_name
SQL: SELECT count(DISTINCT T2.id) , count(DISTINCT T3.id) , T3.dept_name FROM department AS T1 JOIN student AS T2 ON T1.dept_name = T2.dept_name JOIN instructor AS T3 ON T1.dept_name = T3.dept_name GROUP BY T3.dept_name

Q: "Find the title of courses that have two prerequisites?"
Schema_links: [course.title,course.course_id = prereq.course_id]
A: Let's think step by step. For creating the SQL for the given question, we

need to join these tables = [course,prereq].
First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select course.title from course where count (prereq.*) = 2 group by prereq.course_id
SQL: SELECT T1.title FROM course AS T1 JOIN prereq AS T2 ON T1.course_id = T2.course_id GROUP BY T2.course_id HAVING count(*) = 2

Q: "Find the name of students who took any class in the years of 2009 and 2010."
Schema_links: [student.name,student.id = takes.id,takes.YEAR,2009,2010]
A: Let's think step by step. For creating the SQL for the given question, we need to join these tables = [student,takes].
First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select distinct student.name from student where takes.year = 2009 or takes.year = 2010
SQL: SELECT DISTINCT T1.name FROM student AS T1 JOIN takes AS T2 ON T1.id = T2.id WHERE T2.YEAR = 2009 OR T2.YEAR = 2010

Q: "list in alphabetic order all course names and their instructors' names in year 2008."
Schema_links: [course.title,course.course_id = teaches.course_id,teaches.id = instructor.id,instructor.name,teaches.year,2008]
A: Let's think step by step. For creating the SQL for the given question, we need to join these tables = [course,teaches,instructor].
First, create an intermediate representation, then use it to construct the SQL query.
Intermediate_representation: select course.title , instructor.name from course where teaches.year = 2008 order by course.title asc
SQL: SELECT T1.title , T3.name FROM course AS T1 JOIN teaches AS T2 ON T1.course_id = T2.course_id JOIN instructor AS T3 ON T2.id = T3.id

WHERE T2.YEAR = 2008 ORDER BY T1.title

A.2.3 Nested Complex

Use the intermediate representation and the schema links to generate the SQL queries for each of the questions.

Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]
Table department, columns = [*,dept_name,building,budget]
Table instructor, columns = [*,ID,name,dept_name,salary]
Table prereq, columns = [*,course_id,prereq_id]
Table section, columns = [*,course_id,sec_id,semester,year,
building,room_number,time_slot_id]
Table student, columns = [*,ID,name,dept_name,tot_cred]
Table takes, columns = [*,ID,course_id,sec_id,semester,year,grade]
Table teaches, columns = [*,ID,course_id,sec_id,semester,year]
Table time_slot, columns = [*,time_slot_id,day,start_hr,start_min,end_hr,end_min]
Foreign_keys = [course.dept_name = department.dept_name,
instructor.dept_name = department.dept_name,
section.building = classroom.building,
section.room_number = classroom.room_number,
section.course_id = course.course_id,teaches.ID = instructor.ID,teaches.course_id
= section.course_id,teaches.sec_id = section.sec_id,
teaches.semester = section.semester,teaches.year = section.year,student.dept_name
= department.dept_name,takes.ID = student.ID,takes.course_id = section.course_id,
takes.sec_id = section.sec_id,takes.semester = section.semester,takes.year =
section.year,advisor.s_ID = student.ID,advisor.i_ID = instructor.ID,prereq.prereq_id
= course.course_id,prereq.course_id = course.course_id]

Q: "Find the title of courses that have two prerequisites?"

Schema_links: [course.title,course.course_id = prereq.course_id]
A: Let's think step by step. "Find the title of courses that have two prerequisites?" can be solved by knowing the answer to the following sub-question "What are the titles for courses with two prerequisites?".
The SQL query for the sub-question "What are the titles for courses with two prerequisites?" is SELECT T1.title FROM course AS T1 JOIN prereq AS T2 ON T1.course_id = T2.course_id GROUP BY T2.course_id HAVING count(*) = 2
So, the answer to the question "Find the title of courses that have two prerequisites?" is =
Intermediate_representation: select course.title from course where count (prereq.*) = 2 group by prereq.course_id
SQL: SELECT T1.title FROM course AS T1 JOIN prereq AS T2 ON T1.course_id = T2.course_id GROUP BY T2.course_id HAVING count(*) = 2

Q: "Find the name and building of the department with the highest budget."
Schema_links: [department.dept_name,department.building,department.budget]
A: Let's think step by step. "Find the name and building of the department with the highest budget." can be solved by knowing the answer to the following sub-question "What is the department name and corresponding building for the department with the greatest budget?".
The SQL query for the sub-question "What is the department name and corresponding building for the department with the greatest budget?" is SELECT dept_name , building FROM department ORDER BY budget DESC LIMIT 1
So, the answer to the question "Find the name and building of the department with the highest budget." is =
Intermediate_representation: select department.dept_name , department.building from department order by department.budget desc limit 1
SQL: SELECT dept_name , building FROM department ORDER BY budget DESC LIMIT 1

Q: "Find the title, credit, and department name of courses that have more than one prerequisites?"
Schema_links: [course.title,course.credits,course.dept_name,course.course_id = prereq.course_id]
A: Let's think step by step. "Find the title, credit, and department name of courses that have more than one prerequisites?" can be solved by knowing the answer to the following sub-question "What is the title, credit value, and department name for courses with more than one prerequisite?".
The SQL query for the sub-question "What is the title, credit value, and department name for courses with more than one prerequisite?" is SELECT T1.title , T1.credits , T1.dept_name FROM course AS T1 JOIN prereq AS T2 ON T1.course_id = T2.course_id GROUP BY T2.course_id HAVING count(*) ¿ 1
So, the answer to the question "Find the name and building of the department with the highest budget." is =
Intermediate_representation: select course.title , course.credits , course.dept_name from course where count (prereq.*) ¿ 1 group by prereq.course_id
SQL: SELECT T1.title , T1.credits , T1.dept_name FROM course AS T1 JOIN prereq AS T2 ON T1.course_id = T2.course_id GROUP BY T2.course_id HAVING count(*) ¿ 1

Q: "Give the name and building of the departments with greater than average budget."
Schema_links: [department.dept_name,department.building,department.budget]
A: Let's think step by step. "Give the name and building of the departments with greater than average budget." can be solved by knowing the answer to the following sub-question "What is the average budget of departments?".
The SQL query for the sub-question "What is the average budget of departments?" is SELECT avg(budget) FROM department
So, the answer to the question "Give the name and building of the departments with greater than average budget." is =

Intermediate_representation: select department.dept_name , department.building from department where @.@ ¿ avg (department.budget)
SQL: SELECT dept_name , building FROM department WHERE budget ¿ (SELECT avg(budget) FROM department)

Q: "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010."
Schema_links: [teaches.id,teaches.semester,teaches.YEAR,Fall,2009,Spring,2010]
A: Let's think step by step. "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010." can be solved by knowing the answer to the following sub-question "Find the id of instructors who taught a class in Spring 2010".
The SQL query for the sub-question "Find the id of instructors who taught a class in Spring 2010" is SELECT id FROM teaches WHERE semester = 'Spring' AND YEAR = 2010
So, the answer to the question "Find the id of instructors who taught a class in Fall 2009 but not in Spring 2010." is =
Intermediate_representation: select teaches.ID from teaches where teaches.semester = "Fall" and teaches.year = 2009 and teaches.semester != "Spring" and teaches.year = 2010
SQL: SELECT id FROM teaches WHERE semester = 'Fall' AND YEAR = 2009 EXCEPT SELECT id FROM teaches WHERE semester = 'Spring' AND YEAR = 2010

Q: "Find the name of the courses that do not have any prerequisite?"
Schema_links: [course.title,course.course_id]
A: Let's think step by step. "Find the name of the courses that do not have any prerequisite?" can be solved by knowing the answer to the following sub-question "What are the courses that have any prerequisite?".
The SQL query for the sub-question "What are the courses that have any prerequisite?" is SELECT course_id FROM prereq
So, the answer to the question "Find the name of the courses that do not have

any prerequisite?" is =
Intermediate_representation: select course.title from course where @.@ not in prereq.course_id
SQL: SELECT title FROM course WHERE course_id NOT IN (SELECT course_id FROM prereq)

Q: "Find the salaries of all distinct instructors that are less than the largest salary."
Schema_links: [instructor.salary]
A: Let's think step by step. "Find the salaries of all distinct instructors that are less than the largest salary." can be solved by knowing the answer to the following sub-question "What is the largest salary of instructors".
The SQL query for the sub-question "What is the largest salary of instructors" is SELECT max(salary) FROM instructor
So, the answer to the question "Find the salaries of all distinct instructors that are less than the largest salary." is =
Intermediate_representation: select distinct instructor.salary from instructor where @.@ ¡ max (instructor.salary)
SQL: SELECT DISTINCT salary FROM instructor WHERE salary ¡ (SELECT max(salary) FROM instructor)

Q: "Find the names of students who have taken any course in the fall semester of year 2003."
Schema_links: [student.id,student.name,takes.id,takes.semester,fall,2003]
A: Let's think step by step. "Find the names of students who have taken any course in the fall semester of year 2003." can be solved by knowing the answer to the following sub-question "Find the students who have taken any course in the fall semester of year 2003.".
The SQL query for the sub-question "Find the students who have taken any course in the fall semester of year 2003." is SELECT id FROM takes WHERE semester = 'Fall' AND YEAR = 2003
So, the answer to the question "Find the names of students who have taken

any course in the fall semester of year 2003." is =
Intermediate_representation: select student.name from student where takes.semester = "Fall" and takes.year = 2003
SQL: SELECT name FROM student WHERE id IN (SELECT id FROM takes WHERE semester = 'Fall' AND YEAR = 2003)

Q: "Find the minimum salary for the departments whose average salary is above the average payment of all instructors."
Schema_links: [instructor.salary,instructor.dept_name]
A: Let's think step by step. "Find the minimum salary for the departments whose average salary is above the average payment of all instructors." can be solved by knowing the answer to the following sub-question "What is the average payment of all instructors.".
The SQL query for the sub-question "What is the average payment of all instructors." is SELECT avg(salary) FROM instructor
So, the answer to the question "Find the minimum salary for the departments whose average salary is above the average payment of all instructors." is =
Intermediate_representation: select min(instructor.salary) , instructor.dept_name from instructor where avg (instructor.salary) ¿ avg (instructor.salary) group by instructor.dept_name
SQL: SELECT min(salary) , dept_name FROM instructor GROUP BY dept_name HAVING avg(salary) ¿ (SELECT avg(salary) FROM instructor)

Q: "What is the course title of the prerequisite of course Mobile Computing?"
Schema_links: [course.title,course.course_id = prereq.course_id,prereq.prereq_id, course.title,Mobile Computing]
A: Let's think step by step. "What is the course title of the prerequisite of course Mobile Computing?" can be solved by knowing the answer to the following sub-question "What are the ids of the prerequisite of course Mobile Computing?".
The SQL query for the sub-question "What are the ids of the prerequisite

of course Mobile Computing?" is SSELECT T1.prereq_id FROM prereq AS T1 JOIN course AS T2 ON T1.course_id = T2.course_id WHERE T2.title = 'Mobile Computing'

So, the answer to the question "What is the course title of the prerequisite of course Mobile Computing?" is =

Intermediate_representation: select course.title from course where @.@ in prereq.* and course.title = "Mobile Computing"

SQL: SELECT title FROM course WHERE course_id IN (SELECT T1.prereq_id FROM prereq AS T1 JOIN course AS T2 ON T1.course_id = T2.course_id WHERE T2.title = 'Mobile Computing')