

A Survey of NL2SQL with Large Language Models: Where are we, and where are we going?

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan,
Guoliang Li, *Fellow, IEEE*, Nan Tang, and Yuyu Luo

NL2SQL Handbook: https://github.com/HKUSTDial/NL2SQL_Handbook

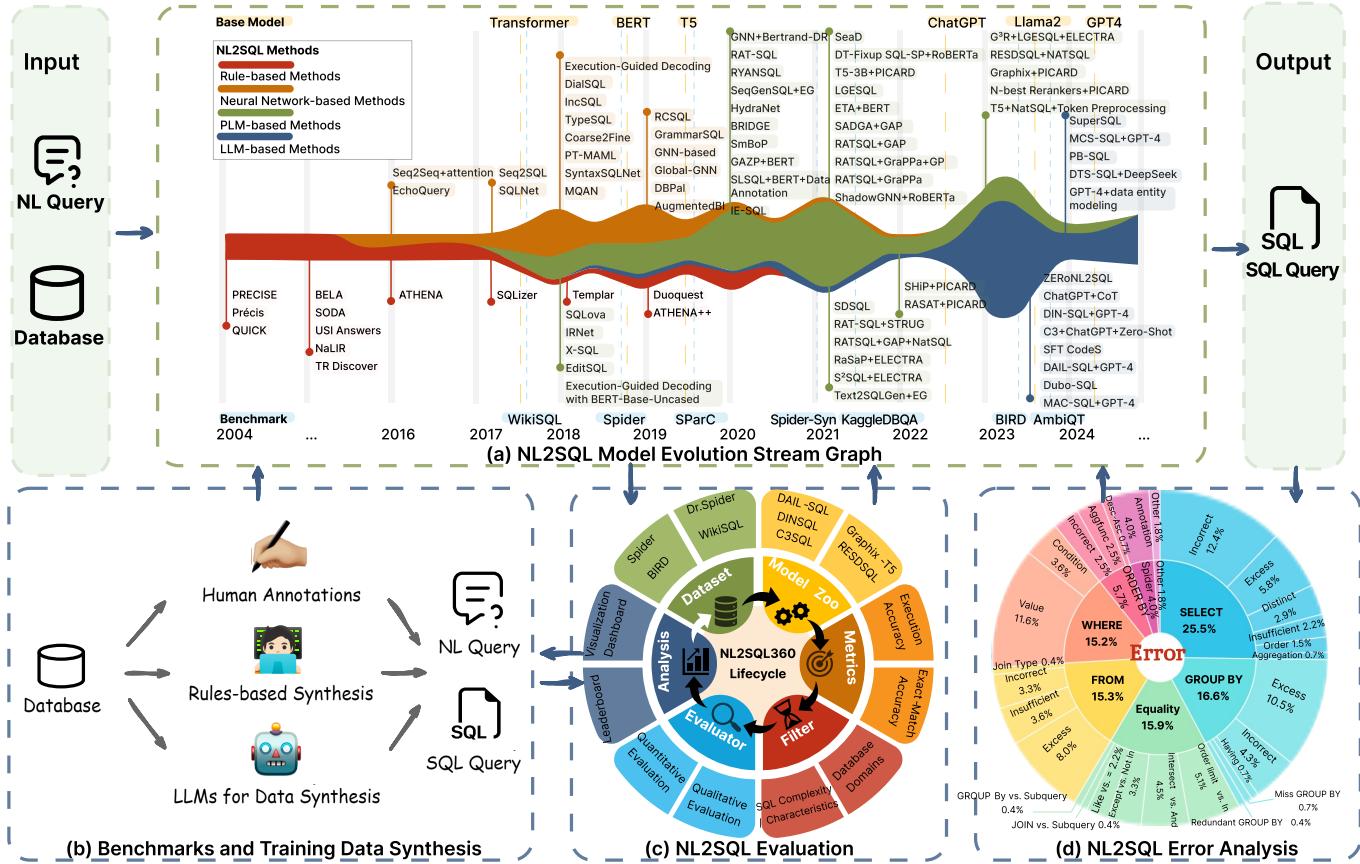


Fig. 1: An Overview of the Survey: The Lifecycle of the NL2SQL Task.

Abstract—Translating users’ natural language queries (NL) into SQL queries (*i.e.*, NL2SQL, *a.k.a.*, Text-to-SQL) can significantly reduce barriers to accessing relational databases and support various commercial applications. The performance of NL2SQL has been greatly enhanced with the emergence of Large Language Models (LLMs). In this survey, we provide a comprehensive review of NL2SQL techniques powered by LLMs, covering its entire lifecycle from the following four aspects: (1) *Model*: NL2SQL translation techniques that tackle not only NL ambiguity and under-specification, but also properly map NL with database schema and instances; (2) *Data*: From the collection of training data, data synthesis due to training data

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Nan Tang and Yuyu Luo are with The Hong Kong University of Science and Technology (Guangzhou), China. E-mail: {xliu371, sshen190, bli303, rjiang073, pma929}@connect.hkust-gz.edu.cn, {yuyuluo, nantang}@hkust-gz.edu.cn.

Guoliang Li is with Tsinghua University, Beijing, China. E-mail: liguo-liang@tsinghua.edu.cn. Yuxin Zhang and Ju Fan are with Renmin University of China, Beijing, China. E-mail: {zhangyuxin159, fanj}@ruc.edu.cn.

*Corresponding author: Yuyu Luo (yuyuluo@hkust-gz.edu.cn).

scarcity, to NL2SQL benchmarks; (3) *Evaluation*: Evaluating NL2SQL methods from multiple angles using different metrics and granularities; and (4) *Error Analysis*: analyzing NL2SQL errors to find the root cause and guiding NL2SQL models to evolve. Moreover, we offer a rule of thumb for developing NL2SQL solutions. Finally, we discuss the research challenges and open problems of NL2SQL in the LLMs era.

Index Terms—Natural Language to SQL, Text-to-SQL, Database Interface, Large Language Models.

I. INTRODUCTION

NATURAL Language to SQL (*i.e.*, NL2SQL, *a.k.a.*, Text-to-SQL), which converts a natural language query (NL) into an SQL query, is a key technique toward lowering the barrier to accessing relational databases [1]–[7]. This technique supports various important applications such as business intelligence and natural language interfaces for databases,

making it a key step toward democratizing data science [8]–[19]. Recent advancements in language models have significantly extended the frontiers of research and application in NL2SQL. Concurrently, the trend among database vendors to offer NL2SQL solutions has evolved from a mere notion to a necessary strategy [20], [21]. Therefore, we need to understand the fundamentals, techniques, and challenges regarding NL2SQL.

In this survey, we will systematically review recent NL2SQL techniques through a new framework, as shown in Figure 1.

- **NL2SQL with Language Models.** We will first review existing NL2SQL solutions from the perspective of language models, categorizing them into four major categories (see Figure 1(a)). We will then focus on the recent advances in Pre-trained Language Models (PLMs) and Large Language Models (LLMs) for NL2SQL.
- **Benchmarks and Training Data Synthesis.** Undoubtedly, the performance of PLM- and LLM-based NL2SQL models is highly dependent on the amount and quality of the training data. Therefore, we will first summarize the characteristics of existing benchmarks and analyze their statistical information (*e.g.*, database and query complexity) in detail. We will then discuss methods for collecting and synthesizing high-quality training data, emphasizing this as a research opportunity (see Figure 1(b)).
- **Evaluation.** Comprehensively evaluating NL2SQL models is crucial for optimizing and selecting models for different usage scenarios. We will discuss the multi-angle evaluation and scenario-based evaluation for the NL2SQL task (see Figure 1(c)). For example, we can assess the NL2SQL model in specific contexts by filtering benchmarks based on SQL characteristics, NL variants, database domains, and so on.
- **NL2SQL Error Analysis.** Error analysis is essential in NL2SQL research for identifying limitations and improving the model robustness. We review existing error taxonomies, analyze their limitations, and propose principles for designing comprehensive taxonomies for NL2SQL output errors. Using these principles, we create a two-level error taxonomy and utilize it to summarize and analyze NL2SQL output errors (see Figure 1(d)).

In addition to the above, we will provide practical guidance for developing NL2SQL solutions, including a roadmap for optimizing LLMs for NL2SQL tasks and a decision flow for selecting NL2SQL modules tailored to various NL2SQL scenarios. Finally, we will discuss key open problems in the field, such as open NL2SQL tasks, cost-effective NL2SQL with LLMs, and trustworthy NL2SQL solutions.

Differences from Existing Surveys. Our survey distinguishes itself from existing NL2SQL surveys [22]–[29] and tutorials [30]–[32] in five aspects.

- We systematically review the entire lifecycle of NL2SQL problem, as shown in Figure 1. This lifecycle includes various NL2SQL translation methodologies powered by language models (Figure 1(a)), training data collection and synthesis methods (Figure 1(b)), multi-angle and

scenarios-based evaluations (Figure 1(c)), and NL2SQL output error analysis techniques (Figure 1(d)).

- We provide a more detailed and comprehensive summary of the inherent challenges in NL2SQL. Additionally, we analyze the technical challenges when developing a robust NL2SQL solution for real-world scenarios, which are often overlooked in other surveys.
- We particularly focus on recent advances in *LLM-based* NL2SQL methods, summarizing key modules and comparing different strategies within this scope. We are the first survey to provide a modular summary of methods and provide detailed analyses for each key module (*e.g.*, database content retrieval).
- We highlight the importance of *evaluating NL2SQL methods in a multi-angle way*, analyze the key NL2SQL error patterns, and provide a two-level error taxonomy.
- We provide practitioners with a roadmap for optimizing LLMs to NL2SQL task and a decision flow for selecting the suitable NL2SQL modules for various usage scenarios.

Contributions. We make the following contributions.

- *NL2SQL with Language Models.* We comprehensively review existing NL2SQL techniques from a lifecycle perspective (Figure 1). We introduce the NL2SQL task definition, discuss challenges (Figure 2), provide a taxonomy of NL2SQL solutions based on language models (Figure 3), and summarize the key modules of language model-powered NL2SQL solutions (Figure 5 and Table I). Next, we elaborate on each module of language model-powered NL2SQL methods, including the pre-processing strategies (Section IV), NL2SQL translation methods (Section V), and post-processing techniques (Section VI).
- *Benchmarks and Training Data Synthesis.* We summarize existing NL2SQL benchmarks based on their characteristics (Figure 10). We analyze each benchmark in depth and discuss its pros and cons (Table II). (Section VII)
- *Evaluation and Errors Analysis.* We highlight the importance of evaluation in developing practical NL2SQL solutions. We review widely used evaluation metrics and toolkits for assessing NL2SQL solutions. We provide a taxonomy to summarize typical errors produced by NL2SQL methods. (Section VIII)
- *Practical Guidance for Developing NL2SQL Solutions.* We provide a roadmap for optimizing existing LLMs to NL2SQL tasks (Figure 11(a)). In addition, we design a decision flow to guide the selection of appropriate NL2SQL modules for different scenarios (Figure 11(b)).
- *Open Problems in NL2SQL.* Finally, we discuss new research opportunities, including the open-world NL2SQL problem and cost-effective NL2SQL solutions (Section X).
- *NL2SQL Handbook.* We maintain an online handbook (https://github.com/HKUSTDial/NL2SQL_Handbook) to help readers stay current with advancements in NL2SQL.

II. NL2SQL PROBLEM AND BACKGROUND

In this section, we first formalize the definition of the NL2SQL task (Section II-A). We then introduce the workflow of how humans perform the NL2SQL task (Section II-B) and

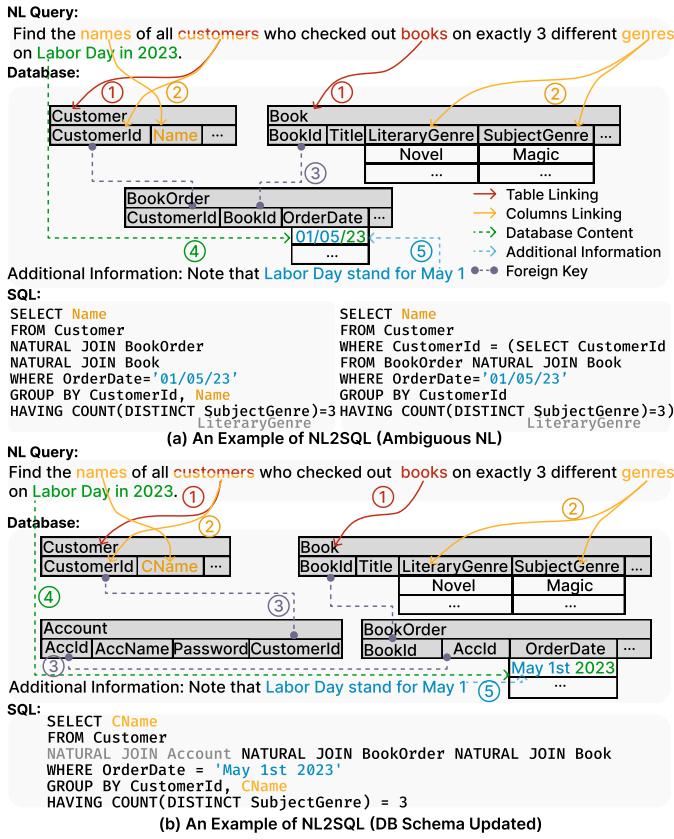


Fig. 2: Examples of the NL2SQL Task and Its Challenges.

discuss the key challenges (Section II-C). Finally, we describe the evolution of NL2SQL solutions based on the development of language models (Section II-D).

A. Problem Formulation

Definition 1 (Natural Language to SQL (NL2SQL)). Natural Language to SQL (NL2SQL), also known as Text-to-SQL, is the task of converting natural language queries (NL) into corresponding SQL queries (SQL) that can be executed on a relational database (DB). Specifically, given an NL and a DB, the goal of NL2SQL is to generate an SQL that accurately reflects the user’s intent and returns the appropriate results when executed on the database.

Discussion. In some cases, the corresponding SQL query to an NL may be multiple due to the ambiguity or underspecification of the NL, or ambiguity in the database schema. In addition, even when the NL, database schema and database content are clear and specific, there may still be multiple equivalent SQL queries that can satisfy the given NL question.

B. NL2SQL Human Workflow

When professional users (*e.g.*, DBAs) perform the NL2SQL task, they first interpret the NL question, examine the database schema and contents, and then construct the corresponding SQL based on their SQL expertise. Below, we outline this process in detail, as illustrated in Figure 2(a).

Step-1: Understanding Natural Language Query: Given the NL query “Find the names of all customers who checked out books on exactly 3 different genres on Labor Day in 2023”, the DBA’s first task is to grasp the user’s intent and identify key components. Key elements include: 1) *Entities or Attributes*: “names”, “customers”, “books”, and “genres”; 2) *Temporal Context*: “Labor Day in 2023”; and 3) *Specific Conditions*: “exactly 3 different genres”. Then, the DBA may further understand the overall purpose of the NL query. In this case, the DBA should retrieve a list of customer names based on specific borrowing behavior on a particular date.

Step-2: Finding Relevant Tables, Columns, and Cell Values: Next, the DBA examines the database schema and contents to identify the relevant tables, columns, and cell values for constructing the SQL. For example, the DBA may determine that the “Customer” and “Book” tables are relevant based on their understanding of the NL (see Figure 2(a)-①). The DBA then decides which columns should be mentioned. For example, the keyword “genres” can refer to either “LiteraryGenre” or “SubjectGenre” (see Figure 2(a)-②). Furthermore, the DBA should interpret “Labor Day in 2023” based on the context. In the US, “Labor Day in 2023” refers to “September 4th, 2023”, while in China, it refers to “May 1st, 2023”. This judgment relies on domain knowledge or available additional information (see Figure 2(a)-⑤).

Note that Step-2 aligns with the concepts of *schema linking*, *database content retrieval*, and *additional information acquisition* in recent NL2SQL solutions powered by language models (please refer to Figure 5 for more details).

Step-3: Writing SQL based on NL and DB Understanding:

Finally, the DBA writes the corresponding SQL based on the insights gained in Steps-1 and -2. This process, known as “NL2SQL Translation”, relies heavily on the DBA’s SQL expertise. However, this process can be very challenging due to the ambiguity of the NL or the complexity of the database. For example, as shown in Figure 2(a), despite understanding the need to link the *Customer* and *Book* tables, one must be familiar with the usage and norms of employing either a natural join or a subquery. In addition, there may be multiple possible SQL queries because “genres” can refer to either “LiteraryGenre” or “SubjectGenre”.

Takeways. From the above steps, we intuitively identify three *inherent* challenges in the NL2SQL task: the uncertainty of the natural language, the complexity of the database, and the translation from the “free-form” natural language queries to the “constrained and formal” SQL queries.

C. NL2SQL Task Challenges

In this section, we will first discuss the fundamental challenges of the NL2SQL task. We will then analyze the *technical* challenges, *i.e.*, the challenges we face when developing a strong NL2SQL solution in real-world scenarios.

C1: Uncertain Natural Language Query. Natural language queries often contain uncertainties due to ambiguity and underspecification [33]. In NL2SQL tasks, the challenges related to NL can be summarized as follows:

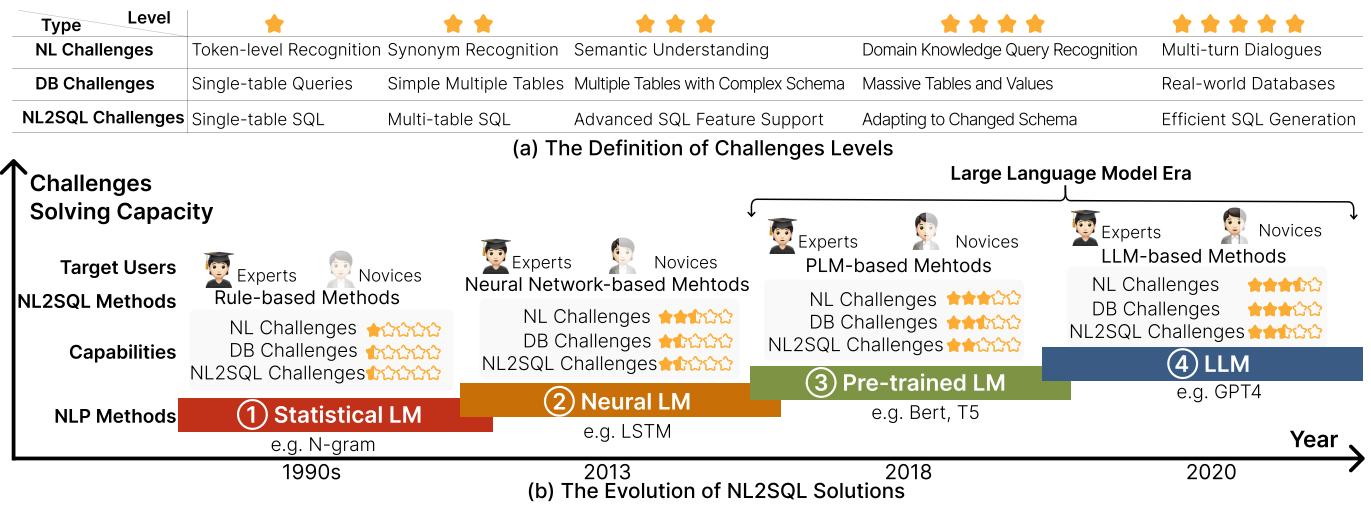


Fig. 3: The Evolution of NL2SQL Solutions from the Perspective of Language Models.

- Lexical Ambiguity: This occurs when a single word has multiple meanings. For example, the word “bat” can refer to an animal or a *baseball bat* (noun) or the action of *swinging* (verb).
- Syntactic Ambiguity: This occurs when a sentence can be parsed in multiple ways. For example, in the sentence “Mary saw the man with the telescope”, the phrase “with the telescope” can mean either that Mary used a telescope to see the man or that the man had a telescope.
- Under-specification: This occurs when linguistic expressions lack sufficient detail to convey specific intentions or meanings clearly. For example, “Labor Day in 2023” refers to September 4th in the US but May 1st in China.

C2: Complex Database and Dirty Content. The NL2SQL task requires a deep understanding of the database schema, including table names, columns, relationships, and data attributes. The complexity of modern schemas and large data volumes make this task especially challenging.

- Complex Relationships Among Tables: Databases often contain hundreds of tables with complex interrelationships. NL2SQL solutions must accurately comprehend and leverage these relationships when generating SQL queries.
- Ambiguity in Attributes and Values: Ambiguous values and attributes in a database can complicate NL2SQL systems’ ability to identify the correct context.
- Domain-Specific Schema Designs: Different domains often have unique database designs and schema patterns. The variations in schema design across domains make it difficult to develop a *one-size-fits-all* NL2SQL solution.
- Large and Dirty Database Values: Efficiently handling vast data volumes in large databases is critical, as processing all data as input is impractical. Additionally, dirty data, such as missing values, duplicates, or inconsistencies, can lead to erroneous query results (e.g., affecting WHERE clauses) if not properly managed.

C3: NL2SQL Translation. The NL2SQL task differs from the compilation of a high-level programming language to a low-level machine language, as it usually has a *one-to-*

many mapping between the input NL and output SQL queries. Specifically, the NL2SQL task faces several unique challenges:

- Free-form NL vs. Constrained and Formal SQL: Natural language is flexible, while SQL queries must adhere to strict syntax. Translating NL into SQL requires precision to ensure the generated queries are executable.
- Multiple Possible SQL Queries: A single NL query can correspond to multiple SQL queries that fulfill the query intent, leading to ambiguity in determining appropriate SQL translation (see the example in Figure 2(a)).
- Database Schema Dependency: The NL2SQL translation is highly dependent on the underlying database schema. As shown in Figure 2 (a) and (b), the same NL may produce different SQL queries based on schema variations. This requires NL2SQL models to bridge gaps between training data and real-world schema differences.

Beyond the intrinsic challenges of the NL2SQL task, developers must also overcome several technical obstacles to build reliable and efficient NL2SQL systems, as discussed below.

C4: Technical Challenges in Developing NL2SQL Solutions. Developing robust NL2SQL solutions requires addressing several key technical challenges, including:

- Cost-effective Solution: Deploying NL2SQL models, particularly those using large language models, demands significant resources, such as hardware and/or API costs. Achieving an optimal balance between model performance and cost efficiency remains a crucial challenge.
- Model Efficiency: A trade-off often exists between model size and performance, with larger models generally yielding better results. Optimizing efficiency without compromising accuracy is essential, especially in interactive querying scenarios requiring low latency.
- SQL Efficiency: The SQL generated by NL2SQL models must be both correct and optimized for performance. This includes optimizing join operations, index usage, and query structures. Efficient queries reduce database load, improving system responsiveness and throughput.

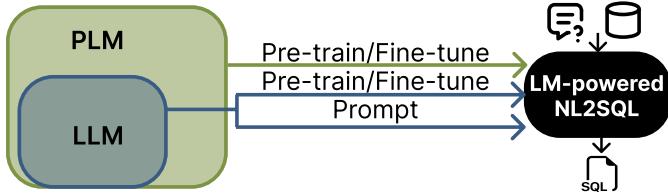


Fig. 4: The Categorization of PLM and LLM in NL2SQL.

- **Insufficient and Noisy Training Data:** High-quality NL2SQL training data is challenging to obtain. Public datasets are often limited and may include noisy annotations, affecting model performance. Annotation requires database expertise, increasing costs, and the complexity of NL2SQL tasks often leads to errors.
- **Trustworthiness and Reliability:** NL2SQL models must be trustworthy and reliable, consistently producing accurate results across diverse datasets and scenarios. Trustworthiness requires transparency, allowing users to understand and verify the generated SQL.

D. Challenges Solving with Large Language Models

Difficulty Levels. We categorize the difficulty of NL2SQL into five levels, each addressing specific hurdles, as shown in Figure 3(a). The first three levels cover challenges that have been or are currently being addressed, highlighting the gradual progress in NL2SQL capabilities. The fourth level includes challenges that are the focus of current LLM-based solutions, while the fifth level represents future challenges, showing our vision for NL2SQL advancements over the next five years.

The Evolution of NL2SQL Solutions. The development of NL2SQL solutions, illustrated in Figure 3(b), progresses through four distinct stages: the rule-based stage, the neural network-based stage, the PLM-based stage, and the LLM-based stage. At each stage, we analyze shifts in target users, *i.e.*, from experts to broader user groups, and the extent to which various NL2SQL challenges are addressed.

1) **Rule-based Stage:** In the early stages, statistical language models (*e.g.*, semantic parsers) were used to interpret NL queries and convert them into SQL queries using predefined rules [33]–[36]. However, rule-based NL2SQL methods face challenges in adaptability, scalability, and generalization. At this stage, natural language understanding was limited to the token level, with research primarily focused on single-table SQL queries (see Figure 3(b)-①).

2) **Neural Network-based Stage:** To alleviate the limitations of rule-based methods, researchers explored neural networks for the NL2SQL task. This led to the development of models based on sequence-to-sequence architectures and graph neural networks [37]–[39], which enhanced the handling of synonyms and intent understanding. Thus, research advanced from single-table scenarios to more complex multi-table scenarios (see Figure 3(b)-②). However, the generalization ability of these methods is still limited by model size and the availability of sufficient training data.

3) **PLM-based Stage:** The introduction of PLMs like BERT [40] and T5 [41] in 2018 led to significant advancements in NL2SQL methods based on PLMs [7], [42], [43], achieving competitive performance on various benchmarks (see Figure 3(b)-③). At this stage, PLMs-based NL2SQL models trained on large corpora have greatly enhanced natural language understanding, resolving approximately 80% of cases in the Spider dataset [44]. However, accuracy drops to about 50% on the extra hard cases of Spider [45]. In addition, these models still face challenges in handling complex schemas.

Remark: PLMs vs. LLMs Figure 4 shows the key differences between LLMs and PLMs. LLMs are a subset of PLMs, distinguished by their advanced language understanding and emergent capabilities [46], [47]. The emergent abilities allow LLMs to perform NL2SQL tasks directly using prompts. In contrast, PLMs generally require additional pre-training or fine-tuning for acceptable NL2SQL performance.

4) **LLM-based Stage:** LLMs demonstrate unique emergent capabilities that surpass traditional PLMs in NLP tasks, marking a new paradigm for NL2SQL solutions. These LLM-based NL2SQL methods have become the most representative solutions in the current NL2SQL landscape [5], [6], [48], [49]. Current research focuses on optimizing prompt design [6] and fine-tuning LLMs [48]. For example, DAIL-SQL [6] utilizes the GPT-4 with effective prompt engineering techniques, achieving strong results on the Spider dataset [44]. Meanwhile, CodeS [48] builds an LLM specifically for NL2SQL tasks by pretraining StarCoder [50] on a large NL2SQL-related corpus, showing solid performance on benchmarks like BIRD [51]. At this stage, LLMs’ emergent capabilities have significantly improved natural language understanding, shifting the task’s focus toward database-specific challenges. New benchmarks like BIRD [51] and BULL [49] emphasize handling massive tables and domain-specific solutions (see Figure 3(b)-④).

NL2SQL Solutions in LLMs Era. Broadly speaking, there are two major approaches to leverage the capabilities of LLMs for NL2SQL: 1) in-context learning, and 2) pre-train/fine-tune LLMs specialized for NL2SQL.

In-Context Learning for NL2SQL. For in-context learning NL2SQL methods, the goal is to optimize the prompt function P to guide the LLMs, which can be formulated as follows:

$$\mathcal{F}_{\text{LLM}}(P \mid \text{NL}, \text{DB}, \text{K}) \rightarrow \text{SQL},$$

where K denotes additional information or domain-specific knowledge related to NL or DB. P is a *prompt function* that transforms the input (NL, DB, K) into a suitable *textual prompt* for the LLMs. An appropriately designed P can effectively guide the LLMs to perform the NL2SQL task more accurately.

Employing in-context learning strategies for NL2SQL treats LLMs as *off-the-shelf* tools, without modifying their parameters. However, if users have sufficient training data or hardware resources, calibrating the LLMs’ parameters can enhance performance and accuracy, allowing the model to be more closely tailored to the specific NL2SQL task.

Pre-train and Fine-tune LLMs for NL2SQL. Fully optimizing the parameters of LLMs for NL2SQL involves two key stages:

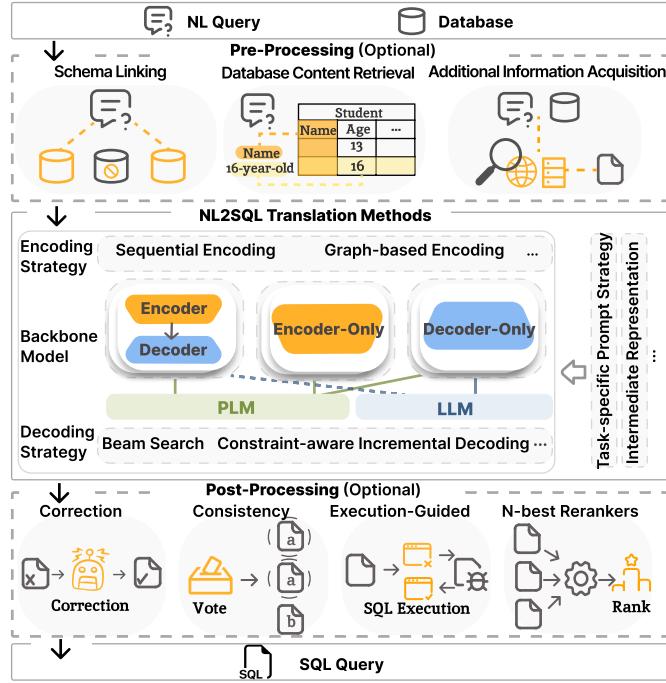


Fig. 5: An Overview of NL2SQL Modules in the LLM Era.

pre-train and fine-tune, which can be formulated as follows:

$$\text{LLM}^* = \mathcal{F}_{\text{fine-tune}}(\mathcal{F}_{\text{pre-train}}(\text{LLM}, \mathcal{D}_p), \mathcal{D}_f)$$

During pre-train, the LLM is trained on a large-scale and diverse dataset \mathcal{D}_p that includes a broad range of linguistic patterns and domain-general knowledge, enabling the model to develop robust understanding capabilities.

In the subsequent fine-tuning stage, the pre-trained model is further adjusted on a more specialized dataset \mathcal{D}_f , which is closely aligned with the NL2SQL task. This targeted training refines the model's capabilities, enabling it to more effectively interpret and generate SQL based on natural language queries.

III. LANGUAGE MODEL-POWERED NL2SQL OVERVIEW

In this section, we first summarize the key modules of PLM- and LLM-powered NL2SQL solutions, as shown in Figure 5. Next, we compare the key differences between the modules of existing NL2SQL solutions in Table I.

Pre-Processing Methods. Pre-processing serves as an enhancement to the model's inputs in the NL2SQL parsing process. Although not strictly necessary, pre-processing significantly contributes to the refinement of NL2SQL parsing [52].

- Schema Linking: This key module identifies the most relevant tables and columns from NL2SQL (Section IV-A).
- Database Content Retrieval: This key module accesses the appropriate database contents or cell values needed for formulating SQL (Section IV-B).
- Additional Information Acquisition: This key module enriches the contextual backdrop by integrating domain-specific knowledge (Section IV-C).

NL2SQL Translation Methods. This is the core of NL2SQL solution, responsible for converting input NL queries into SQL.

- Encoding Strategy: This crucial module transforms the input NL and database schema into an internal representation, capturing both the semantic and structural information of the input data (Section V-A).
- Decoding Strategy: This key module transforms the internal representation into SQL queries (Section V-B).
- Task-specific Prompt Strategy: This module provides tailored guidance for the NL2SQL model, optimizing the NL2SQL translation workflow (Section V-C).
- Intermediate Representation: This module serves as a bridge between NL and SQL translation, providing a structured approach to abstract, align, and optimize NL understanding, simplify complex reasoning, and guide the generation of accurate SQL queries (Section V-D).

Post-Processing Methods. Post-processing is a crucial step to refine the generated SQL queries for better accuracy.

- SQL Correction Strategy: This aims to identify and correct syntax errors in generated SQL (Section VI-A).
- Output Consistency: This module ensures the uniformity of SQL by sampling multiple reasoning results and selecting the most consistent result (Section VI-B).
- Execution-Guided Strategy: It uses the execution results of SQL to guide subsequent refinements (Section VI-C).
- N-best Rankers Strategy: It aims to rerank the top- k results generated by the NL2SQL model to enhance query accuracy (Section VI-D).

IV. PRE-PROCESSING STRATEGIES FOR NL2SQL

The pre-processing step is crucial in the NL2SQL translation process, as it identifies relevant tables and columns (*i.e.*, Schema Linking) and retrieves necessary database contents or cell values (*i.e.*, DB Content Retrieval) to support SQL query generation. What's more, it enriches context by incorporating domain-specific knowledge (*i.e.*, Additional Information Acquisition), which can improve the understanding of the query context and correct errors to prevent their propagation.

A. Schema Linking

Schema linking aims to identify the tables and columns relevant to the given NL query, ensuring accurate mapping and processing of key information within the limited input. This step is essential for improving the performance of the NL2SQL task. In the LLM era, schema linking has become even more critical due to the input length limitations of LLMs.

We categorize existing schema linking strategies into three groups based on their characteristics: 1) *string matching-based schema linking*, 2) *neural network-based schema linking*, and 3) *in-context learning-based for schema linking*.

1) *String Matching-based Schema Linking*: Early research [38], [79], [80] primarily focused on string matching techniques for schema linking. These methods use similarity measures between the NL queries and schemas to identify relevant mappings. Both *exact* and *approximate* matching techniques are commonly employed in this approach.

Exact matching, as used by IRNet [79], identifies matches when candidates are identical or when one is a substring

TABLE I: Comparisons of Existing NL2SQL Solutions.

Methods	Years	Finetuning	Pre-Processing			NL2SQL Translation Methods						Post-Processing		
			Schema	DB Content	Additional Information Aquisition	Backbone Model	Encoding Strategy	Intermediate Representation	Task-specific Prompt Strategy	Decoding Strategy	Correction	Consistency	Execution -Guided	N-best Rerankers
CHESS [53]	2024	-	✓	✓	✓	Decoder-Only	Sequential Encoding	-	COT	Greedy Search	✓	✓	✓	-
CodeS [48]	2024	-	✓	✓	✓	Decoder-Only	Sequential Encoding	-	-	Greedy Search	-	-	✓	-
SFT Codes [48]	2024	✓	✓	✓	✓	Decoder-Only	Sequential Encoding	-	-	Greedy Search	-	-	✓	-
FInSQL [49]	2024	✓	✓	-	✓	Decoder-Only	Sequential Encoding	-	-	Greedy Search	✓	✓	-	-
DTS-SQL [54]	2024	✓	✓	-	-	Decoder-Only	Sequential Encoding	-	-	Greedy Search	-	-	-	-
TA-SQL [55]	2024	-	✓	-	-	Decoder-Only	Sequential Encoding	Sketch Structure	COT	Greedy Search	-	-	-	-
SuperSQL [45]	2024	-	✓	✓	✓	Decoder-Only	Sequential Encoding	-	-	Greedy Search	-	✓	-	-
ZeroNL2SQL [43]	2024	✓	-	-	-	Encoder-Decoder	Sequential Encoding	Sketch Structure	Decomposition	Beam Search	✓	-	✓	-
PET-SQL [56]	2024	✓	✓	-	✓	Decoder-Only	Sequential Encoding	-	-	Greedy Search	-	✓	-	-
CoE-SQL [57]	2024	-	-	-	✓	Decoder-Only	Sequential Encoding	-	CoT	Greedy Search	✓	-	-	-
PURPLE [58]	2024	-	✓	-	✓	Decoder-Only	Sequential Encoding	-	-	Greedy Search	✓	✓	-	-
MetaSQL [59]	2024	-	✓	-	✓	Decoder-Only	Sequential Encoding	-	Decomposition	Greedy Search	-	-	✓	-
DEA-SQL [60]	2024	-	✓	-	✓	Decoder-Only	Sequential Encoding	-	Decomposition	Greedy Search	✓	-	-	-
DIN-SQL [51]	2023	-	✓	-	✓	Decoder-Only	Sequential Encoding	Syntax Language	Decomposition	Greedy Search	✓	-	-	-
DAIL-SQL [6]	2023	-	-	✓	✓	Decoder-Only	Sequential Encoding	-	-	Greedy Search	✓	✓	-	-
C3-SQL [61]	2023	-	✓	-	-	Decoder-Only	Sequential Encoding	-	COT	Greedy Search	-	✓	-	-
RIESDSQL [7]	2023	✓	✓	-	-	Encoder-Decoder	Sequential Encoding	Syntax Language	Decomposition	Beam Search	-	-	-	-
T5-3B-NatSQL+Token Preprocessing [62]	2023	✓	✓	-	-	Encoder-Decoder	Sequential Encoding	Syntax Language	-	Greedy Search	✓	-	-	-
ACT-SQL [63]	2023	-	✓	-	✓	Decoder-Only	Sequential Encoding	-	CoT	Greedy Search	-	-	-	-
ODIS [64]	2023	-	-	✓	-	Decoder-Only	Sequential Encoding	-	-	Greedy Search	-	-	-	-
MAC-SQL [65]	2023	-	✓	-	-	Decoder-Only	Sequential Encoding	-	Decomposition	Greedy Search	✓	-	-	-
SC-Prompt [1]	2023	✓	-	-	-	Encoder-Decoder	Separate Encoding	Sketch Structure	-	Beam Search	✓	-	-	-
CatSQL [66]	2023	✓	-	-	-	Encoder-Only	Sequential Encoding	Sketch Structure	-	Beam Search	✓	-	-	-
SQLFormer [67]	2023	✓	✓	-	-	Encoder-Decoder	Graph-based Encoding	-	-	Beam Search	-	-	-	-
GR [68]	2023	✓	✓	✓	-	Encoder-Only	Graph-based Encoding	-	COT	Beam Search	-	-	✓	-
Graphix-T5 [42]	2022	✓	✓	✓	-	Encoder-Decoder	Graph-based Encoding	-	-	Constraint-aware Incremental	-	-	-	-
SHIP [69]	2022	✓	-	✓	-	Encoder-Decoder	Graph-based Encoding	-	-	Constraint-aware Incremental	-	-	-	-
N-best List Rerankers [70]	2022	✓	✓	✓	-	Encoder-Decoder	Sequential Encoding	-	-	Constraint-aware Incremental	-	-	✓	-
RASAT [71]	2022	✓	-	✓	-	Encoder-Decoder	Graph-based Encoding	-	-	Constraint-aware Incremental	-	-	-	-
PICARD [72]	2022	✓	-	✓	-	Encoder-Decoder	Sequential Encoding	-	-	Constraint-aware Incremental	-	-	-	-
TKK [73]	2022	✓	-	✓	-	Encoder-Decoder	Separate Encoding	Sketch Structure	Decomposition	Constraint-aware Incremental	-	-	-	-
SFSQL [74]	2022	✓	✓	✓	-	Encoder-Only	Graph-based Encoding	-	-	Greedy Search	-	-	-	-
RAT-SQL [75]	2021	✓	✓	✓	-	Encoder-Only	Graph-based Encoding	Syntax Language	-	Beam Search	-	-	-	-
SmBaP [76]	2021	✓	-	✓	-	Encoder-Only	Graph-based Encoding	-	-	Beam Search	-	-	-	-
RaSaP [77]	2021	✓	✓	✓	-	Encoder-Only	Graph-based Encoding	-	-	Beam Search	-	-	-	-
BRIDGE [78]	2020	✓	-	✓	-	Encoder-Only	Sequential Encoding	-	-	Others	-	-	-	-

of the other. While this method can detect straightforward links, it may lead to false positives when candidates share common words. Approximate string matching techniques, like the Damerau–Levenshtein distance [81] used in ValueNet [82], help identify matches despite spelling variations or errors.

However, these methods struggle with handling synonyms and are not robust enough to manage vocabulary variations, limiting their effectiveness in complex NL2SQL tasks.

2) Neural Network-based Schema Linking: To alleviate the above limitations, researchers have employed deep neural networks to align database schemas with natural language queries [7], [52], [75], [83]. These methods can better parse complex semantic relationships between NL queries and database structures. DAE [83] frames schema linking as a sequential tagging problem, using a two-stage anonymization model to capture semantic relationships between schema and NL. However, without an annotated corpus, DAE does not evaluate schema linking's impact on NL2SQL performance. To address this, SLSQL [52] annotates schema linking information in Spider dataset [44], enabling a systematic, data-driven study. RESDSQL [7] introduces a ranking-enhanced encoding framework for schema linking, using a cross-encoder to prioritize tables and columns based on classification probabilities. FinSQL [49] uses a parallel cross-encoder to retrieve relevant schema elements, significantly reducing linking time.

However, neural network-based methods often struggle to generalize across databases with diverse schemas or domains, especially when training data is scarce.

3) In-Context Learning for Schema Linking: With the advancement of LLMs like GPT-4, research is exploring how to leverage their strong reasoning capabilities for schema linking, *i.e.*, directly identifying and linking relevant database schema components from the NL query. A key technique is In-Context Learning (ICL) technique [84], which utilizes LLMs' ability to understand complex language patterns and relationships within data schemas, enabling a more dynamic and flexible schema linking process [5], [53], [61], [65], [85].

C3-SQL [61] employs zero-shot prompts with GPT-3.5 using self-consistency for table and column linking. For table linking, tables are ranked by relevance and listed; for column linking, columns are ranked within relevant tables and outputted as a dictionary, prioritizing matches with question terms or foreign keys. MAC-SQL [65] proposes a multi-agent collaborative framework for NL2SQL, where the *Selector* agent handles schema linking, activated only when the database schema prompt exceeds a specified length. CHESS [53] utilizes GPT-4 to extract keywords from both NL and evidence (additional information from BIRD [51]), implementing a three-stage schema pruning protocol with different prompts.

Employing ICL for schema linking has shown promising performance. However, LLMs have inherent limitations in the amount of context they can process, meaning complex schemas with many tables and columns may exceed this limit.

B. Database Content Retrieval

Database content retrieval focuses on efficiently extracting cell values for specific SQL clauses such as WHERE. We

categorize existing database content retrieval strategies into three groups based on their characteristics: 1) *String Matching-based Methods*, 2) *Neural Network-based Methods*, and 3) *Index Strategy for Database Content Retrieval*.

1) String Matching-based Methods: String matching-based methods identify and compare cell values related to the NL query through string matching [7], [42], [72], [78], [79], [82].

IRNet [79] uses n-grams, treating text between quotes as cell values. RESDSQL [7] uses the Longest Common Substring algorithm [86] to find the longest sequence shared between strings. BRIDGE [78] advances this with an anchor text matching technique that automatically extracts cell values from the NL. Using heuristics, it calculates the maximum sequence match to define matching boundaries, excluding irrelevant substrings and adjusts thresholds for accuracy.

However, while string matching methods are effective, they struggle with synonyms and can be computationally expensive when handling large databases.

2) Neural Network-based Methods: These methods aim to capture complex data and semantic features through layers of nonlinear transformations, helping to resolve synonym issues.

TABERT [87] uses a method called *database content snapshots* to encode relevant database content for an NL query, employing attention mechanisms to manage information across cell value representations in different rows. Another approach leverages graph relationships to represent database content. For example, IRNet [79] uses the knowledge graph ConceptNet [88] to find and link relevant cell values, assigning types based on exact or partial matches. RAT-SQL [75] further enhances structural reasoning by modeling the relationship between cell values and the NL query, identifying column-value relationships where the query value is part of the column's candidate cell value.

While these methods capture semantic features, they may struggle with ambiguous or context-dependent NL, leading to inaccurate cell value retrieval. Moreover, the training of neural networks demands substantial computational resources.

3) Index Strategy for Database Content Retrieval: Efficiently retrieving relevant cell values is crucial for the performance of NL2SQL systems, especially with large datasets. Indexing is a key method for improving retrieval efficiency by enabling faster access to relevant cell values [48], [53].

CHESS [53] uses a Locality-sensitive Hashing [89] for approximate nearest neighbor searches, indexing unique cell values to quickly find the top matches related to the NL query. This approach speeds up the process of comparing edit distances and semantic embeddings. CodeS [48] employs a coarse-to-fine matching strategy. It uses BM25 [90] to build an index for coarse-grained searches, identifying candidate values, which are then refined by applying the Longest Common Substring algorithm [86] to assess similarity with the NL, thereby pinpointing the most relevant cell values.

While indexing significantly improves retrieval efficiency, building indexes is time-consuming, and frequent changes to database content require continuous updates, adding overhead.

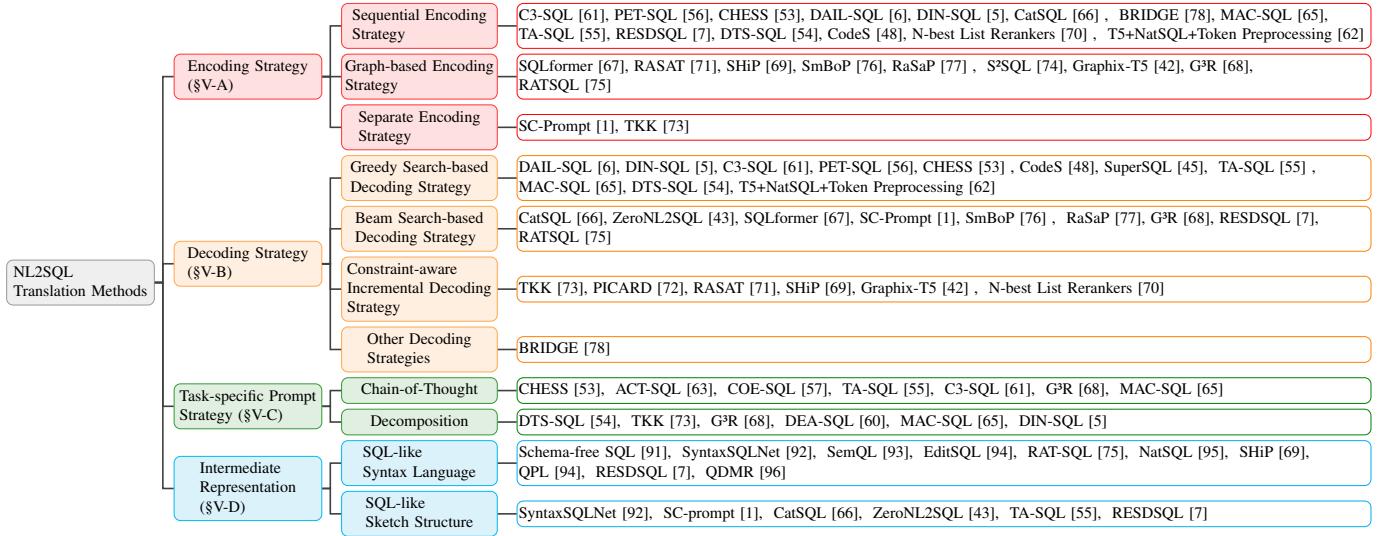


Fig. 6: A Taxonomy of NL2SQL Translation Methods based on their Design Choices.

C. Additional Information Acquisition

Additional information, such as domain knowledge, plays a crucial role in enhancing NL2SQL models' understanding of NL queries, schema linking, and overall NL2SQL translation. This information can provide demonstration examples, domain knowledge, formulaic evidence, and format information for the NL2SQL backbone model or specific modules, thereby enhancing the quality of generated results. We categorize existing strategies into the following two groups: 1) *Sample-based Methods*, and 2) *Retrieval-based Methods*.

1) *Sample-based Methods*: With advancements in LLMs and in-context learning techniques, researchers often incorporate additional information into the textual inputs (*i.e.*, prompts) alongside demonstration examples. DIN-SQL [5] integrates additional information through few-shot learning across multiple stages, such as schema linking, query classification, task decomposition, and self-correction. This enables DIN-SQL to handle challenges like complex schema links, multiple table joins, and nested queries. Other works [48], [53], [64] employ similar strategies. In real-world databases, cross-domain knowledge or evidence is often available. For example, BIRD [51] contains domain knowledge which is crucial for various NL2SQL works [6], [45], [48], [53], [54].

2) *Retrieval-based Methods*: Extracting relevant knowledge and few-shot examples from extensive domain knowledge bases can increase token usage, impacting efficiency and computational cost [5], [6]. To enhance accuracy and efficiency, some researchers employ similarity-based retrieval methods. For example, PET-SQL [56] builds a pool of question frames and question-SQL pairs, selecting the k most similar examples to the target query, which are then used in prompts.

When databases lack text-based additional information, researchers devise methods to retrieve and convert external knowledge into natural language. For example, RE-GROUP [97] creates a formulaic knowledge base across domains (*e.g.*, finance, transportation) and uses Dense Passage Retriever [98] to compute similarity scores, integrating related entities with NL and schema through an Erasing-Then-Awakening model [99]. ReBoost [100] uses a two-phase Explain-Squeeze Schema Linking strategy, first presenting a

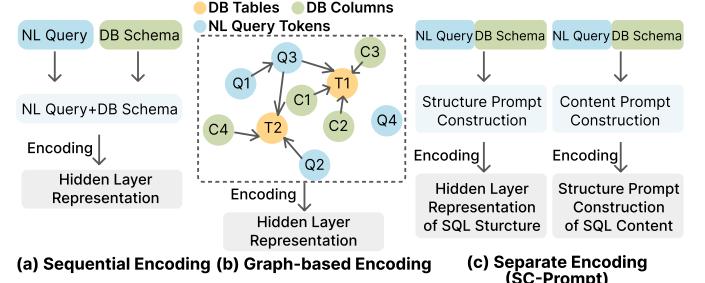


Fig. 7: An Overview of the Encoding Strategies.

generalized schema to LLMs, then applying targeted prompts to improve query-to-entity mapping accuracy.

Retrieval-based methods improve the effectiveness of acquiring additional information but increase computing costs. Moreover, current research mostly relies on domain-specific text, with limited use of structured knowledge. Thus, integrating diverse information sources could further enhance NL2SQL performance, especially for domain-specific databases.

V. NL2SQL TRANSLATION METHODS

In this section, we elaborate on NL2SQL translation methods using language models. As shown in Figure 6, we will detail their encoding (Section V-A), decoding (Section V-B), and task-specific prompt strategies (Section V-C). Moreover, we will discuss how the intermediate representation can benefit the NL2SQL translation process (Section V-D).

A. Encoding Strategy

In the NL2SQL task, encoding refers to transforming NL and database schema into a structured format suitable for language model processing. This step is essential for converting unstructured data into a form usable for SQL generation, capturing the NL's semantics and the schema's structure to help the model map user intent to appropriate SQL. As shown in Figure 7, primary encoding strategies include 1) *Sequential Encoding*, 2) *Graph-based Encoding*, and 3) *Separate Encoding*.

1) Sequential Encoding Strategy: Sequential encoding is a strategy in NL2SQL models where both the NL and the database schema are treated as token sequences. The key idea is to linearize the input data, enabling sequence-based models to effectively capture semantic and syntactic information, as shown in Figure 7(a). For example, models like T5 [41] are used to encode NL and database schema sequentially in works [62], [72] BRIDGE [78] improves the alignment between the NL and database schema by representing both as a tagged sequence and inserting matched database cell values (called anchor texts) next to corresponding fields. Similarly, N-best List Rerankers [70] appends database content to column names. RESDSQL [7] uses a ranking-enhanced encoder to sort and filter schema items, prioritizing the most relevant ones and reducing schema linking complexity. CatSQL [66] leverages the pre-trained GraPPa encoding network [36] to concatenate NL, schema, and additional information into an input sequence.

Although LLM-based NL2SQL methods don't explicitly define an input encoding strategy, most rely on processing input sequences using the self-attention mechanism. This mechanism functions similarly to an encoding process, predicting the next word based on preceding ones in the sequence.

While sequential encoding is simple and intuitive, it may struggle to capture the complex relationships between the database schema and the NL query, potentially limiting the model's ability to understand and generate more complex SQL.

2) Graph-based Encoding Strategy: Graph-based encoding in NL2SQL models represents both NL and database schema as interconnected graphs, leveraging the relational structure of databases and inter-dependencies in the input data, as shown in Figure 7(b) [36], [42], [67], [69], [71], [74]–[77]. Unlike sequential encoding, this approach preserves the schema's topology, offering richer context for each element and enhancing the model's ability to produce accurate SQL queries.

RAT-SQL [75] introduces a relation-aware self-attention mechanism, explicitly using relational information in a graph structure to jointly encode the question and schema, enhancing the model's understanding of structural information. S²SQL [74] injects syntactic structure information at the encoding stage using the ELECTRA [101] model, enhancing semantic understanding. G³R [68] uses the LGESQL [102] encoder and Graph Attention Network (GAT) [103] to capture multi-source heterogeneous information. SQLformer [67] introduces learnable embeddings for tables and columns to select the most relevant schema elements.

Graph-based encoding effectively captures complex relationships between input data, enabling the model to generate more accurate SQL queries with multiple relationships and conditions. However, it requires more sophisticated algorithms for graph construction and processing, and its full potential may only be realized with large amounts of training data, making it less suitable for data-limited scenarios.

3) Separate Encoding Strategy: Separate encoding strategy is an approach in NL2SQL models where different parts of the NL (such as clauses and conditions) are encoded separately and then combined to form the final SQL, as shown in Figure 7(c).

Early models, such as SQLNet [104] and Seq2SQL [105], used separate encoding for the NL and database schema due to

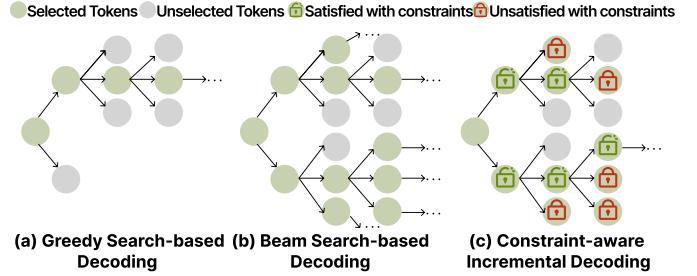


Fig. 8: An Overview of the Decoding Strategies.

the format mismatch between the two. However, this approach hindered schema linking and is now less commonly used. Despite this, separate encoding allows for tailored handling of different data types, leveraging the strengths of various encoding techniques. TKK [73] employs task decomposition and multi-task learning strategies by breaking down the complex NL2SQL task into subtasks and progressively integrating knowledge. Similarly, SC-Prompt [1] divides text encoding into two stages: structure and content, each encoded separately.

While the separate encoding strategy requires multiple processing of input data, which may extend the training and inference time of the model, it allows for more refined handling and understanding of different aspects of queries. This provides the model with the flexibility to handle various query tasks, thereby enhancing overall performance.

B. Decoding Strategy

Decoding is a crucial step in NL2SQL translation, transforming encoder-generated representations into SQL. An effective decoding strategy ensures that the generated SQL are not only syntactically correct but also semantically aligned with the NL queries, while optimizing SQL execution efficiency. Figure 8 introduces several key decoding strategies, as discussed below.

1) Greedy Search-based Decoding Strategy: The greedy search-based decoding strategy is simple and fast, selecting the token with the highest probability at each decoding step. This strategy builds the final output sequence by continuously choosing the locally optimal solution, as shown in Figure 8(a).

Since GPT models (e.g., GPT-4) default to greedy search-based decoding, many NL2SQL solutions using GPT fall into this category. DTS-SQL [54], based on DeepSeek LLM [106], uses the same approach. Early models like SQLNet [104] and Seq2SQL [105] also rely on greedy search for SQL generation.

Greedy search is popular due to its fast decoding speed and simplicity. However, it only considers the locally optimal solution, potentially overlooking long-term dependencies and global optimization. This can lead to suboptimal SQL queries, particularly for complex queries, where early mistakes may propagate and accumulate throughout the decoding process.

2) Beam Search-based Decoding Strategy: This strategy explores a larger search space based on beam search, leading to potentially better results. Instead of selecting only one optimal token at each step, beam search retains multiple candidate sequences ("beams") and expands the top- k tokens with the highest probabilities until a complete output sequence is generated, as shown in Figure 8(b).

Given its advantages, several NL2SQL models employ beam search [7], [66], [77]. RAT-SQL [75] combines relation-aware graph structure encoding with beam search to generate multiple SQL candidates, reranking them based on graph structure information. Unlike RAT-SQL, EditSQL [94] uses beam search alongside dialogue history to generate and refine candidate SQL queries. SmBoP [76] employs a semi-autoregressive bottom-up decoding approach, improving efficiency by parallelizing sub-tree construction and scoring, with logarithmic time complexity. ZeroNL2SQL [43] retains the top- k hypotheses during the SQL sketch generation stage, which are then refined for query and predicate calibration.

Compared to greedy search, beam search retains multiple candidate sequences, which increases memory and computational demands, slowing decoding speed. However, by considering multiple candidates at each step, beam search explores a wider search space, making it more likely to generate accurate and complex SQL queries.

3) Constraint-aware Incremental Decoding Strategy: The constraint-aware incremental decoding strategy, introduced by PICARD [72] (Parsing Incrementally for Constrained Auto-Regressive Decoding), is designed for NL2SQL tasks. This strategy aims to ensure the generation of syntactically correct SQL queries by incorporating constraints during the decoding process, as shown in Figure 8(c).

Roughly speaking, NL2SQL models using this strategy generate SQL queries incrementally, applying SQL grammar rules at each step. As the model predicts the next token, it also checks the syntactic correctness of the partial query against a set of constraints derived from SQL grammar rules. This ensures that each token adheres to the required syntax, reducing the chances of generating invalid or incomplete SQL queries and improving overall translation accuracy and reliability. PICARD [72] ensures that each token added adheres to the correct SQL syntax, significantly reducing errors and incomplete queries. Many models [42], [69]–[73] have demonstrated better performance by using PICARD’s decoding strategy.

While this strategy requires more computational resources and processing time due to incremental decoding and constraint application, it effectively ensures syntactic correctness, making it ideal for generating structurally complex SQL queries while minimizing errors.

4) Other Decoding Strategies: Beyond the three common decoding strategies, some models introduce specialized approaches to improve decoding accuracy. For example, BRIDGE [78] introduces Schema-Consistency Guided Decoding, which ensures that generated SQL queries align with the database schema. It does this by continuously verifying the match between the query and schema during decoding and adjusting the decoding path accordingly based on the results.

C. Task-specific Prompt Strategy

In the era of LLMs, prompt engineering has become a powerful method for harnessing LLM capabilities across diverse tasks [107]. In NL2SQL, task-specific prompts are crafted to guide LLMs in optimizing NL2SQL translation, enhancing the accuracy of translating complex NL queries into precise SQL

queries. Broadly speaking, there are two main types of task-specific prompt strategies: 1) *Chain-of-Thought prompting*, and 2) *Decomposition Strategy*.

1) Chain-of-Thought (CoT) Prompting: The CoT prompting [108], known for its effectiveness, showcases the LLM’s reasoning process, improving both the accuracy and interpretability of the generated results. In NL2SQL tasks, CoT enhances model performance and ensures that the generated SQL statements are more aligned with human expectations [109]. For example, CHESS [53] transforms NL into SQL statements through a streamlined pipeline that utilizes LLMs and CoT. This process includes entity and context retrieval, schema selection, SQL generation, and revision.

In addition, the integration of CoT with other techniques can enhance the performance of NL2SQL models. These techniques include in-context learning [57], [63], logical synthesis [55], calibration with hints [61], [68] and multi-agent system [65]. Specifically, in-context learning and logical synthesis enrich CoT by embedding a deeper linguistic understanding, enabling precise semantic mapping to SQL constructs [57], [63]. Calibration with hints fine-tunes model responses, aligning them closely with NL nuances for accurate intent translation [61], [68]. Furthermore, integrating the multi-agent framework with CoT fosters a collaborative approach, with specialized agents handling tasks like schema linking and SQL generation, which speeds up reasoning and enhances adaptability [65].

Overall, these techniques create a more robust NL2SQL framework, offering better precision and reliability in translating complex NL queries into accurate SQL statements.

2) Decomposition Strategy: The decomposition strategy divides the NL2SQL task into sequential subtasks, allowing each sub-module to concentrate on a specific generation step, thereby enhancing accuracy, quality, and interpretability.

Different approaches vary in subtask decomposition granularity [54], [60], [65], [68], [73]. For example, TKK [73] applies finer-grained decomposition by breaking down NL2SQL parsing into subtasks like mapping NL to SELECT, FROM, and WHERE clauses. This approach helps the model concentrate on each clause, enhancing understanding of the problem, schema, and SQL alignment. Similar strategies are used in G³R [68] and DEA-SQL [60]. Moreover, decomposition also reduces model complexity. For example, MAC-SQL [65] introduces a Decomposer agent to split the user’s query into subproblems, making SQL generation for each part more manageable.

In general, the decomposition strategy divides the NL2SQL translation task into multiple subtasks, enabling each sub-module to focus on enhancing its specific output. However, this approach also raises computational costs, making model training and deployment more complex and resource-intensive.

D. Intermediate Representation for NL2SQL Translation

The NL2SQL task is challenging due to the complexity and ambiguity of NL queries, coupled with the syntax-constrained nature of SQL. To simplify this process, researchers have developed a *grammar-free* intermediate representation (IR) to bridge the “free-form” NL question and the “constrained and formal” SQL. This IR provides a structured yet flexible format,

Intermediate Representation:

```

SELECT film.title
WHERE count(film_actor.*)>5
    (a) SQL-like Syntax Language (e.g. NATSQL)

SELECT [column] ..... [column] title
FROM [table] ..... [table] film
JOIN [table] ..... [table] film_actor
ON [table].[column] ..... [table].[column] film.film_id
=[table].[column] ..... [table].[column] film_actor.film_id
GROUP BY [column] ..... [column] film_id
HAVING count([column]) > n [column] * [n] 5
    (b) SQL-like Sketch Structure (e.g. SC-Prompt)

```

Fig. 9: An Example of the Intermediate Representation.

capturing the essential components and relationships within an NL query without the strict syntax requirements of SQL. Figure 9 shows two types of IR strategies, discussed below.

1) *SQL-like syntax language*: As shown in Figure 9(a), SQL-like syntax language is a simplified SQL-like structure. Early approaches used information retrieval techniques to map the original question and schema data into this syntax [91], [110]. Subsequent research efforts have focused on consolidating or eliminating partial clauses or operations in SQL queries to simplify SQL-like syntax language [92], [93]. For example, EditSQL [94] adds WHERE and HAVING conditions. Recent research has focused on simplifying syntax languages to improve parsing efficiency [111]. NatSQL (Natural SQL) [95], a widely used SQL-like syntax language, eliminates uncommon SQL operators and keywords, streamlining schema linking by minimizing necessary schema items. Combined with PLMs, NatSQL has achieved strong results on various benchmarks [7], [62]. Additionally, Question Decomposition Meaning Representation [96], [112] breaks down the original NL into atomic questions, each serving as an intermediate representation that translates into small formal operations like selecting entities, retrieving attributes, or aggregating data.

SQL-like syntax languages have demonstrated potential in bridging user queries and databases. However, previous studies face challenges due to high complexity and limited coverage of database structures [95]. As databases grow in size and domain specificity, maintaining the simplicity of SQL-like syntax languages becomes increasingly difficult. Moreover, some of these languages require manual construction and adjustments, raising deployment costs and complexity.

2) *SQL-like sketch structure*: Leveraging the structural characteristics of SQL, researchers have developed SQL-like sketches that mirror SQL structure for parsing, enabling diverse NL queries to be mapped into a specific sketch space, as shown in Figure 9(b). This approach reduces parsing complexity.

Early works applied fixed sketch rules and neural networks to map the NL into SQL-like sketch structure [92], [113]. For example, SyntaxSQLNet [92] uses a syntax tree and a corresponding decoder, dividing the decoding into nine sub-modules that separately predict operators, keywords, and entities before combining them to generate the final SQL. In recent years, the development of language models has allowed researchers to design more elaborate SQL-like sketch structures for parsing [1], [43], [55], [66]. CatSQL [66] constructs a more general template sketch with slots serving as initial placeholders. Its base model focuses on the parsing of NL to fill

these placeholders, consequently decreasing the computational cost. Moreover, several recent works cover both SQL-like syntax language and SQL-like sketch transition methods. For instance, RESDSL [7] introduces a rank-enhanced encoding and skeleton-aware decoding framework. During the decoding phase, its decoder initially generates the SQL skeleton and then transforms it into the actual SQL query. When combined with NatSQL, RESDSL demonstrates the ability further to enhance the quality of SQL query generation.

In general, SQL-like sketch structure can be more easily combined with other strategies, such as decomposition strategy or SQL-like syntax language strategy. In addition, it can more fully utilize the comprehension and cloze capabilities of existing LLMs and reduce the dependence on professionals.

VI. POST-PROCESSING STRATEGIES FOR NL2SQL

After the NL2SQL model generates the SQL, post-processing can refine it to better meet user expectations. This step involves leveraging additional information or models to enhance the SQL, with a focus on SQL correction, ensuring output consistency, and execution-guided checking.

A. *SQL Correction Strategies*

The SQL generated by NL2SQL models may contain syntax errors or unnecessary keywords like DESC, DISTINCT, and Aggregate functions. DIN-SQL [5] introduces a self-correction module that operates in a zero-shot setting, where the model receives only the faulty SQL and attempts to correct it. Two prompts are used: a general prompt for CodeX, which directly asks for error identification and correction, and a mild prompt for GPT-4, which seeks potential issues without presuming errors. To handle errors in predicate predictions, such as incorrect columns or values, ZeroNL2SQL [43] employs a multi-level matching approach. This method incrementally expands matching across columns, tables, and databases, allowing matched values to be returned to the LLMs to generate SQL queries consistent with the database content.

While these methods focus on fixing syntax errors, they often overlook semantic errors, such as incorrect table joins, misaligned conditions, or inaccurate aggregations, which are essential for improving accuracy.

B. *Output Consistency*

To enhance output consistency, self-consistency [114] has been introduced, based on the idea that complex reasoning tasks may have multiple valid paths to a single correct answer. This approach samples various reasoning paths and selects the most consistent answer to improve output quality.

C3-SQL [61] introduces a Consistency Output (CO) component to reduce LLM output randomness and improve zero-shot NL2SQL performance. CO samples multiple reasoning paths, executes them, filters errors, and applies voting to select the most consistent SQL result. DAIL-SQL [6] integrates self-consistency, achieving a 0.4% performance improvement over configurations without it. To reduce LLM randomness, Fin-SQL [49] generates n candidate SQL queries in parallel,

clusters them based on keyword consistency, and selects a query from the largest cluster. The self-consistency method enhances LLM output diversity by raising the temperature and then using majority voting to select the final result.

However, recent studies [115] suggest that a single model may not provide sufficient diversity. To address this, PET-SQL [56] introduces a cross-consistency strategy, in which multiple LLMs generate SQL at lower temperatures and vote based on execution results.

While these methods improve consistency across multiple runs with single or multiple LLMs, reducing model randomness and enhancing accuracy, they significantly increase inference cost and time.

C. Execution-Guided Strategies

In NL2SQL tasks, the execution result of an SQL query provides critical feedback on NL2SQL translation accuracy. For example, errors or `NULL` values in execution results can signal potential issues with the SQL query.

To reflect human behavior in writing complex SQL queries, CHESS [53] provides LLMs with the database schema, question, candidate SQL queries, and their execution results. CHESS starts with a draft query and refines it based on execution feedback, adjusting for syntax errors as needed. CodeS [48] on the other hand, generates a complete SQL statements through beam search, producing four SQL candidates and selecting the first executable one as the final result.

Execution-Guided Strategies refine SQL based on execution results, ensuring the query retrieves data correctly. However, this approach can significantly increase SQL generation time, especially with large databases.

D. N-best Rerankers Strategies

In NL2SQL tasks, especially in cross-domain scenarios, generated SQL queries can vary subtly in structure and semantics. N-best reranking reorders the top- N model outputs, often leveraging a larger model or additional knowledge sources. For example, fine-tuning a BERT-based reranker, as demonstrated by Bertrand-dr on the Spider dataset [116], has effectively improved the performance of several NL2SQL models.

However, the effectiveness of Bertrand-dr's reranking can be unstable and sensitive to threshold settings, sometimes even yielding negative effects. To address these limitations, G³R [68] introduces a feature-enhanced reranker using PLM-based hybrid prompt tuning, which bridges domain gaps without extra parameters. Contrastive learning then sharpens distinctions between candidate queries [117]. Similarly, ReF-SQL [118] retrieves the most relevant results from the retriever and generator modules to improve final answer quality.

While N-best reranking is widely used in PLM-based methods to refine SQL candidates, it is less common in LLM-based methods, which typically have stronger inference capabilities.

VII. NL2SQL BENCHMARKS

In this section, we will first elaborate on the different types of NL2SQL datasets, highlighting their characteristics, as shown in Figure 10 (Section VII-A). We will then perform an in-depth analysis of existing NL2SQL datasets (Section VII-B).

A. An Overview of NL2SQL Benchmarks

With advancements in NL2SQL, various datasets have emerged to address the evolving challenges, as shown in Figure 10. These range from single-domain databases with simple queries to cross-domain, multi-turn, multilingual, and domain-specific scenarios, reflecting the progress and the emergence of new challenges for NL2SQL solutions.

Single-Domain NL2SQL Datasets. Early NL2SQL datasets focused on specific domains with relatively simple SQL queries, such as ATIS [119] for flight information and GeoQuery [120] for U.S. geographical facts. Recently, larger single-domain datasets [49], [128], [130], [139], [144], [145] have been introduced, featuring more complex databases and SQL queries tailored to specific scenarios. This shift reflects an increased emphasis on assessing NL2SQL systems' performance and practical utility within particular domains.

Cross-Domain NL2SQL Datasets. After the development of early single-domain datasets, the NL2SQL field shifted toward cross-domain datasets to test systems' generalization across diverse SQL queries and databases. WikiSQL [105] was the first cross-domain dataset, drawing tables from Wikipedia across various domains. Subsequently, Spider [44] was introduced, containing more complex relational databases with multiple tables. Recently, the BIRD [51] has further advanced complexity by including SQL functions and operations absent in Spider, providing an even greater challenge for NL2SQL.

Multi-Turn NL2SQL Datasets. With advancements in NL2SQL, multi-turn datasets have been developed to support interactive dialogues. SParC [125] is a cross-domain and multi-turn dataset with about 4.3K NL questions, totaling over 12K (NL, SQL) pairs, each NL questions requiring contextual understanding across turns. CoSQL [126], collected using a Wizard-of-Oz setup, includes over 30K turns and introduces additional challenges like unanswerable questions, further testing context comprehension. CHASE [134], the first multi-turn Chinese dataset, reduces context-independent questions and simple SQL queries, advancing dialogue-centered NL2SQL research.

NL2SQL Datasets with Robustness Testing. In real-world applications, NL2SQL systems must handle diverse user groups and databases, emphasizing robustness. Spider-Syn [135] simulates user unfamiliarity with schemas by using synonyms in NL questions, while Dr.Spider [142] applies 17 types of perturbations to databases, NL questions, and SQL queries for comprehensive robustness evaluation.

NL2SQL Datasets with SQL Efficiency Testing. Real-world databases often hold vast amounts of data, and multiple SQL queries can answer a single user question, each with varying SQL execution efficiency. BIRD [51] introduces a metric for evaluating SQL execution efficiency called the Valid Efficiency Score (VES), which will be further discussed in Section VIII.

Knowledge-Augmented NL2SQL Datasets. Domain-specific knowledge is essential for NL2SQL systems to perform well in real-world applications. KaggleDBQA [138] includes database documents, such as column and table descriptions. Similarly, Spider-DK [136] expands the Spider development set by

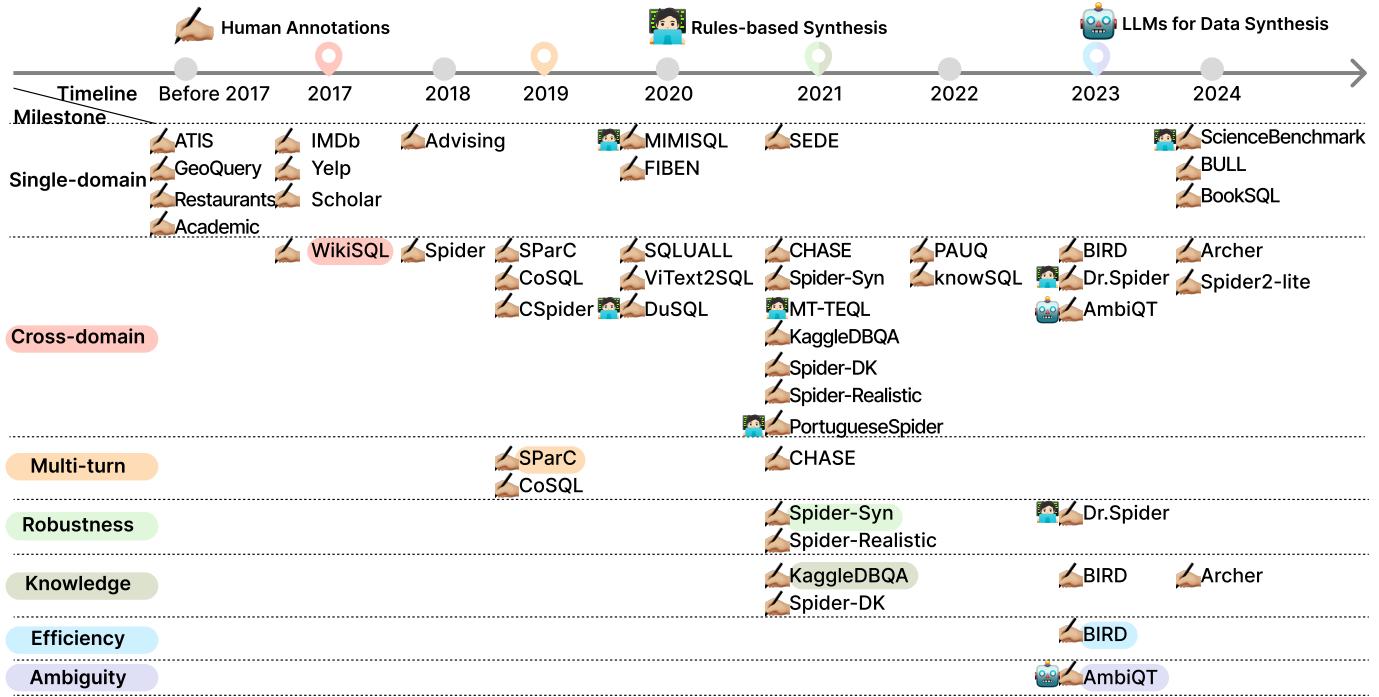


Fig. 10: Timeline for NL2SQL Benchmarks.

TABLE II: Statistics of NL2SQL Benchmarks.

Dataset	Redundancy Measure			DB Complexity				Query Complexity				
	#-Questions	#-Unique SQLs	#-Questions / #SQLs	#-DBs	#-Tables	#-Tables / DB	#-Cols / Table	#-Records / DB	Tables	Selects	Agg Func	Math Comp
ATIS [119]	5280	947	5.6	1	25	25	5.24	162243	8.39	1.79	0.22	0
GeoQuery [120]	877	246	3.6	1	7	7	4.14	937	2.22	2.19	0.92	0
Restaurants [121]	378	23	16.4	1	3	3	4.00	19295	2.43	1.17	0.35	0
Academic [110]	196	185	1.1	1	17	17	3.12	58249674	3.48	1.04	0.54	0
IMDb [122]	131	89	1.5	1	17	17	3.94	40147386	2.91	1.01	0.30	0
Yelp [122]	128	110	1.2	1	8	8	5	4823945	2.41	1	0.45	0
Scholar [123]	817	193	4.2	1	10	10	2.50	147416275	3.38	1.02	0.68	0
WikiSQL [105]	80654	80257	1	26531	26531	1	6.34	17	1	1	0.28	0
Advising [124]	4387	205	21.4	1	15	15	7.40	332596	3.41	1.21	0.40	0
Spider [44]	11840	6448	1.8	206	1056	5.13	5.01	8980	1.83	1.17	0.54	0
SParC [125]	10228	8981	1.1	166	876	5.28	5.14	9665	1.58	1.10	0.44	0
CoSQL [126]	8350	8007	1	166	876	5.28	5.14	9665	1.54	1.11	0.42	0
CSParC [127]	11840	6408	1.8	206	1056	5.13	5.01	8980	1.83	1.17	0.54	0
MIMICSQL [128]	20000	10000	2	-	-	-	-	-	1.74	1	0.84	0
SQuALL [129]	11276	8296	1.4	2108	4028	1.91	9.18	71	1.22	1.29	0.40	0.03
FIBEN [130]	300	233	1.3	1	152	152	2.46	11668125	5.59	1.56	0.97	0
ViText2SQL [131]	9693	5223	1.9	166	876	5.28	5.14	9665	1.17	1.12	0.54	0
DuSQL [132]	25003	20308	1.2	208	840	4.04	5.29	20	1.49	1.25	0.73	0
PortugueseSpider [133]	9693	5275	1.8	166	876	5.28	5.14	9665	1.85	1.17	0.54	0
CHASE [134]	15408	13900	1.1	350	1609	4.60	5.19	4594	1.81	1.16	0.31	0
Spider-Syn [135]	1034	550	1.9	166	876	5.28	5.14	9665	1.68	1.17	0.59	0
Spider-DK [136]	535	283	1.9	169	887	5.25	5.14	9494	1.71	1.16	0.54	0
Spider-Realistic [137]	508	290	1.8	166	876	5.28	5.14	9665	1.79	1.21	0.50	0
KaggleDBQA [138]	272	249	1.1	8	17	2.12	10.53	595075	1.25	1.05	0.69	0
SEDE [139]	12023	11421	1.1	1	29	29	7.28	-	1.90	1.29	0.94	0.49
MT-TEQL [140]	489076	4525	108.1	489076	3279004	6.70	5.51	-	1.69	1.15	0.53	0
PAUQ [141]	9876	5497	1.8	166	876	5.28	5.14	9693	1.82	1.17	0.53	0
knowSQL [97]	28468	-	-	488	-	-	-	-	-	-	-	-
Dr.Spider [142]	15269	3847	4	549	2197	4	5.54	28460	1.81	1.19	0.52	0
BIRD [51]	10962	10840	1	80	611	7.64	7.14	4585335	2.07	1.09	0.61	0.20
AmbiQT [143]	3046	3128	1	166	876	5.28	5.14	9665	1.85	1.17	0.51	0
ScienceBenchmark [144]	5031	3654	1.4	-	-	-	-	-	1.45	1	0.24	0
BULL [49]	7932	5864	1.4	3	78	26	14.96	85631	1.22	1	0.18	0.42
BookSQL [145]	78433	39530	2	1	7	7	8.86	1012948	1.25	1.12	0.78	0.39
Archer [146]	518	260	2	10	68	6.8	6.81	31365.3	3.89	3.07	1.77	0.1
Spider2-Lite [147]	527	527	1	264	6259	23.71	35.61	-	6.53	5.10	3.57	1.60

adding five types of domain knowledge to NL questions, testing systems' ability to understand and use this information.

NL2SQL Datasets with Ambiguous Questions. In real-world NL2SQL tasks, ambiguities often arise, such as se-

mantic ambiguities in NL and overlapping database schemas, making ambiguity-focused evaluation increasingly important. AmbiQT [143] is the first dataset designed to assess ambiguity coverage, comprising four ambiguity types. Each NL question maps to two valid SQL queries, reflecting specific ambiguities.

Synthetic NL2SQL Datasets. MIMICSQL [128] employs a template-based approach to generate initial template questions and corresponding SQL queries, though manual refinement is required to make questions more natural. ScienceBenchmark [144] also uses templates for initial SQL generation but leverages GPT-3 for SQL-to-NL translation.

B. In-depth Analysis of Existing NL2SQL Datasets

To analyze and compare NL2SQL datasets complexity, we use the NL2SQL360 [45] system for statistical evaluation, as shown in Table II. We measure the *Redundancy*, including the number of NL questions, SQL queries, and their ratio. DB Complexity covers the total databases, total tables, average tables per database, average columns per table, and average records per database. Query Complexity measures the average number of tables, SELECT keywords, aggregate functions, scalar functions, and mathematical computations in each SQL query. For datasets without public dev/test splits, such as CHASE [134], only statistics for public splits are reported. For datasets without publicly available data, like knowSQL [97], values in Table II are marked with “–”.

From the Redundancy Measure perspective, we observe a trend from early datasets to recent ones where datasets have grown in size. Specifically, MT-TEQL [140] stands out with the highest number of NL questions and the largest ratio of NL questions to SQL queries due to its automated transformation of NL questions, generating a large volume of variants.

In terms of Database Complexity, the number of databases and tables within each dataset aligns with its intended task. Single-domain datasets, such as BookSQL [145], generally contain fewer databases, while those aimed at robustness evaluation, like Dr.Spider [142] and MT-TEQL [140], include a larger number of databases.

Regarding Query Complexity, datasets like FIBEN [130] and SEDE [139] feature SQL queries with multiple tables and aggregate functions, mirroring complexities in real-world financial domains and Stack Exchange sites. Recent datasets also emphasize Scalar Functions and Mathematical Computations, adding structural challenges.

Discussion. Despite the increasing number of datasets proposed by the NL2SQL community, a gap in SQL complexity remains compared to real-world scenarios. Current datasets typically feature fewer SELECT keywords, indicating a lack of nested queries and complex set operations. Additionally, challenges involving Scalar Functions and Mathematical Computations require further focus. We encourage the community to propose datasets addressing these complexities.

VIII. EVALUATION AND ERROR ANALYSIS

In this section, we introduce key evaluation metrics for NL2SQL solutions (Section VIII-A), review toolkits for low-cost and comprehensive evaluation (Section VIII-B), and provide an error taxonomy for analyzing SQL errors in the NL2SQL process (Section VIII-C).

A. Evaluation Metrics

Evaluation metrics are crucial for assessing the performance of NL2SQL solution. We define N as the dataset size, Q_i as the NL question of the i -th example, V_i as the execution result set of the ground-truth SQL query Y_i and \hat{V}_i as the execution result set of the SQL query \hat{Y}_i generated by the NL2SQL solution.

Execution Accuracy (EX) [44]. This metric evaluates the performance of the NL2SQL system by comparing whether the execution result sets of the ground-truth SQL queries and the predicted SQL queries are identical. It can be computed by: $EX = \frac{\sum_{i=1}^N \mathbb{1}(V_i = \hat{V}_i)}{N}$, where $\mathbb{1}(\cdot)$ is an indicator function that equals 1 if the condition inside is satisfied, and 0 otherwise. Note that false negatives could occur because different SQL queries corresponding to semantically different NL queries may produce identical execution result sets.

String-Match Accuracy (SM) [105]. This metric, also called Logical Form Accuracy, simply compares whether the ground-truth SQL query and the predicted SQL query are identical as strings. It may penalize SQL queries that produce the correct execution result sets but do not have the exact string match with the ground-truth SQL queries. It can be computed as follows: $SM = \frac{\sum_{i=1}^N \mathbb{1}(Y_i = \hat{Y}_i)}{N}$.

Component-Match Accuracy (CM) [44]. This metric evaluates the detailed performance of the NL2SQL system by measuring the exact matching of different SQL components such as SELECT, WHERE and others between the ground-truth SQL query and the predicted SQL query. For a specific SQL component C . The computation can be formalized as follows:

$$CM^C = \frac{\sum_{i=1}^N \mathbb{1}(Y_i^C = \hat{Y}_i^C)}{N},$$

where Y_i^C is the component of SQL query Y_i . To correctly determine if an SQL component matches, some SQL components (e.g., WHERE) do not consider order constraints.

Exact-Match Accuracy (EM) [44]. This metric is based on the Component-Match Accuracy (CM) and measures whether all SQL components $\mathcal{C} = \{C_k\}$ of the predicted SQL query match the ground-truth SQL query. It can be computed as follows:

$$EM = \frac{\sum_{i=1}^N \mathbb{1}(\bigwedge_{C_k \in \mathcal{C}} Y_i^{C_k} = \hat{Y}_i^{C_k})}{N}.$$

Valid Efficiency Score (VES) [51]. This metric measures the execution efficiency of valid SQL queries. It considers both the accuracy and efficiency of SQL execution, which can be computed as follows:

$$VES = \frac{\sum_{i=1}^N \mathbb{1}(V_i = \hat{V}_i) \cdot \mathbf{R}(Y_i, \hat{Y}_i)}{N}, \quad \mathbf{R}(Y_i, \hat{Y}_i) = \sqrt{\frac{\mathbf{E}(Y_i)}{\mathbf{E}(\hat{Y}_i)}},$$

where $\mathbf{R}(\cdot)$ measures the relative execution efficiency of the predicted SQL query compared to the ground-truth SQL query, eliminating uncertainties due to machine status. $\mathbf{E}(\cdot)$ measures the efficiency of specific SQL query, which can be refer to execution time, memory usage and more.

Query Variance Testing (QVT) [45]. This metric measures the robustness of an NL2SQL system in handling variations in NL queries. For a given SQL query Y_i , there are often multiple corresponding NL queries, represented as pairs $\{(Q_1, Y_i), (Q_2, Y_i), \dots, (Q_m, Y_i)\}$. The QVT metric is calculated as:

$$QVT = \frac{1}{N} \sum_{i=1}^N \left(\frac{\sum_{j=1}^{m_i} \mathbb{1}(\mathcal{F}(Q_{ij}) = Y_i)}{m_i} \right),$$

where m_i is the number of different NL variations for the SQL query Y_i , and $\mathcal{F}(Q_{ij})$ is the predicted SQL query for the j -th NL variation of Y_i .

B. NL2SQL Evaluation Toolkits

Recent NL2SQL solutions have achieved remarkable performance on various NL2SQL benchmarks. However, in real-world applications, variations in NL query styles, database schemas, and SQL query characteristics across domains make it difficult to fully assess system robustness using standard benchmark metrics alone. To address this, recent toolkits [45], [140] have been developed to provide a more comprehensive evaluation of NL2SQL systems in practical scenarios.

MT-TEQL [140] is a unified framework for evaluating the performance of NL2SQL systems in handling real-world variations in NL queries and database schemas. It is based on a metamorphic testing approach, implementing semantic-preserving transformations of NL queries and database schemas to generate their variants without manual efforts automatically. It includes four types of transformations for NL queries (e.g. *Prefix Insertion*) and eight types of transformations for database schemas: (e.g. *Table Shuffle*).

NL2SQL360 [45] is a multi-angle evaluation framework offering fine-grained assessments of NL2SQL systems across diverse scenarios (Figure 1(c)). Unlike MT-TEQL, it emphasizes varied SQL query characteristics in different applications, such as aggregate functions, nested queries, or top- k queries typical of the Business Intelligence scenario. Comprising six core components, *i.e.*, *Dataset*, *Model Zoo*, *Metrics*, *Dataset Filter*, *Evaluator*, and *Analysis*, NL2SQL360 provides a unified, model-agnostic interface for systematic evaluations. Users can apply both public and private datasets, customize metrics for specific scenarios, and analyze performance on subsets with scenario-specific SQL characteristics, offering valuable insights into NL2SQL system effectiveness across applications.

C. A Taxonomy for NL2SQL Errors Analysis

Error analysis involves examining model errors to identify limitations and guide corrective actions for improved performance. In this section, we first review the existing NL2SQL error taxonomy. We then propose design principles and introduce a two-level NL2SQL errors taxonomy.

Existing Taxonomies for NL2SQL Errors Analysis. Recent NL2SQL research [5], [53], [148]–[150] has increasingly incorporated error analysis, proposing various error taxonomies.

Ning et al. [149] introduced a detailed error taxonomy based on two dimensions: (1) *Syntactic dimension* identifies specific SQL parts where errors occur, organized by keywords

such as WHERE and JOIN. (2) *Semantic dimension* indicates misinterpretations of the natural language description, such as errors in understanding table names. SQL-PaLM [150] categorizes errors into five types: (1) *Schema Linking*, where the model fails to select the relevant tables required for NL; (2) *Misunderstanding Database Content*, where the model is not able to accurately interpret the data within the tables; (3) *Misunderstanding Knowledge Evidence*, where the model fails to use or ignores human-annotated evidence; (4) *Reasoning*, which includes an inadequate understanding of the question, leading to queries that lack the necessary reasoning steps to generate correct results; and (5) *Syntax-Related Errors*, where the generated SQL contains syntax errors.

Our Taxonomy for NL2SQL Errors Analysis. Current error taxonomies in NL2SQL are often specific to particular datasets, limiting their general applicability. To address these issues, a standardized and effective taxonomy is essential. We propose the following principles to guide the development of an NL2SQL error taxonomy:

- *Comprehensiveness*: It should cover all possible error types in the NL2SQL translation process.
- *Mutual Exclusivity*: Each error type should be clearly distinct to avoid classification ambiguity.
- *Extensibility*: The taxonomy should be adaptable to include emerging error types as NL2SQL evolves.
- *Practicality*: It should be practical, enabling users to diagnose and address errors in real-world scenarios.

Following these principles, we developed a two-level NL2SQL error analysis taxonomy:

- *Error Localization*: The first level identifies specific SQL components where errors occur, such as the SELECT or WHERE clause. Pinpointing error locations enables targeted adjustments and enhances correction efficiency.
- *Cause of Error*: The second level focuses on the underlying reasons for the error. For instance, errors in the WHERE clause values may indicate the model's limitations in database content retrieval or interpretation.

A Case Study of Error Analysis. We collected and classified errors from DIN-SQL [5] on the Spider [44] using our proposed taxonomy. As shown in Figure 1(d), the results suggest that this taxonomy is effective. However, we acknowledge that achieving a comprehensive NL2SQL error taxonomy requires ongoing refinement. We invite the research community to contribute to the continuous improvement of this taxonomy.

IX. PRACTICAL GUIDANCE FOR NL2SQL SOLUTIONS

In this section, we provide practical guidance for developing NL2SQL solutions, considering key factors and scenarios.

A. Data-Driven Roadmap of Optimizing LLMs for NL2SQL

In Figure 11(a), we outline a strategic roadmap designed to optimize LLMs for NL2SQL task, based on data privacy and data volume. Data privacy affects the choice of open-source and closed-source LLMs, while data volume affects the strategies for optimization for training and inference.

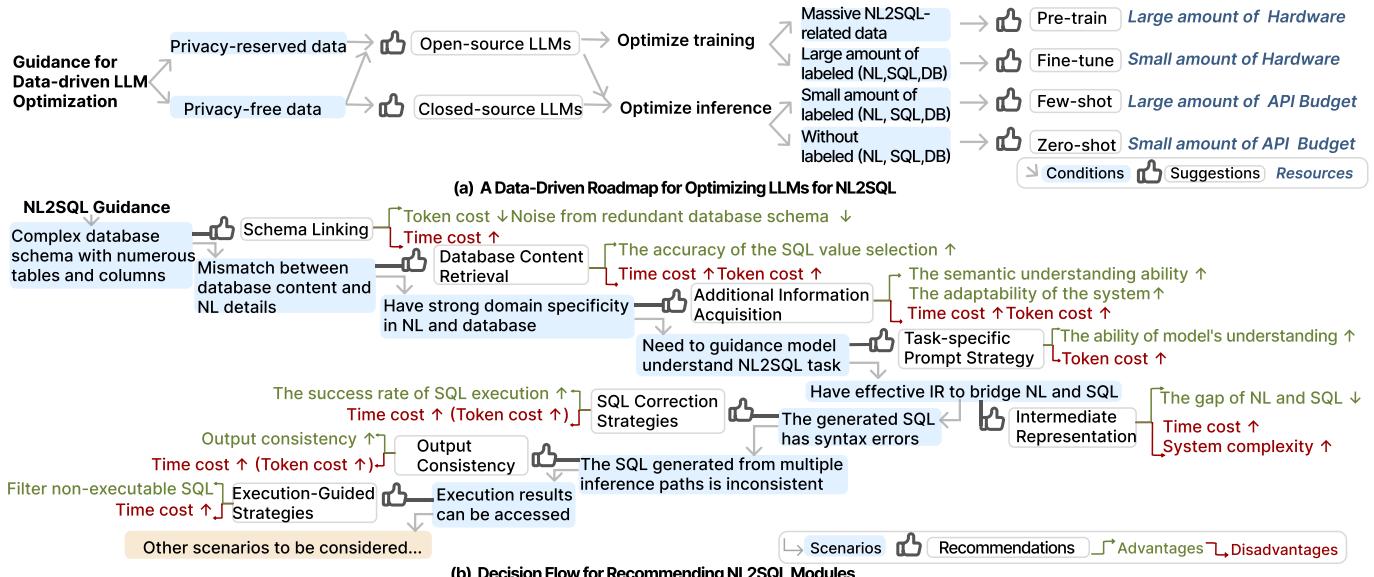


Fig. 11: A Data-Driven Roadmap and a Decision Flow for Recommending NL2SQL modules.

Condition 1: Data Privacy. For privacy-sensitive data, open-source LLMs are preferable, as closed-source models typically use external APIs, potentially exposing data to external servers. Open-source models allow full control over local training and inference, providing stronger data privacy protection.

Condition 2: Data Volume. For open-source LLMs, optimization is possible in both training and inference phases, while closed-source LLMs allow only inference-stage optimization due to limited access. With extensive NL2SQL data, pre-training enhances performance; fine-tuning is suitable for datasets with hundreds to thousands of (NL, SQL) pairs. In low-data scenarios, few-shot learning is recommended, while zero-shot methods are essential when labeled data is unavailable. Hardware resources and API costs are also important considerations in selecting the best optimization strategy.

B. Decision Flow of Selecting NL2SQL Modules

In Figure 11(b), we present recommendations for choosing NL2SQL modules based on specific scenarios, highlighting both benefits and trade-offs. Below, we outline two examples.

Scenario 1: Complex Database Schema with Numerous Tables and Columns. In this case, using Schema Linking strategies is advisable. This reduces token costs and minimizes noise from irrelevant schema elements, enhancing efficiency. However, it also incurs additional time costs.

Scenario 2: Execution Results Can be Accessed. Here, Execution-Guided Strategies are recommended, as they improve system performance by filtering out non-executable SQL queries. The downside is the increased time required for query execution, which can be substantial with large databases.

In summary, while each module offers unique advantages for specific NL2SQL scenarios, it is essential to balance these benefits with the potential drawbacks in system design.

X. OPEN PROBLEMS

Open NL2SQL Problem. In real-world scenarios like government open data platforms, citizens may ask questions that require querying multiple databases and aggregating results. For example, a citizen might ask, “What is the average processing time for tax returns in the last five years?”. Answering this question involves retrieving relevant tables from different databases, such as tax records, processing logs, and statistical reports, and then generating multiple SQL queries over them.

The Open NL2SQL problem requires translating NL queries into SQL queries that access not just a single fixed database but also identify relevant databases (or tables) within a large-scale DBMS. Unlike traditional NL2SQL, where a single target database is specified by the user, Open NL2SQL may need to generate multiple SQL queries that access different databases for a single user question. Thus, the Open NL2SQL problem introduces unique challenges, including: (1) *database retrieval*: accurately identifying and retrieving relevant databases from a vast array of data sources; (2) *handling heterogeneous schemas*: integrating data with varied structures and terminologies, requiring advanced schema matching and linking techniques; (3) *answer aggregation*: inferring final answers from multiple SQL queries across databases, which demands methods to plan query order, resolve conflicts, and ensure consistency; (4) *domain adaptation*: generalizing models across domains to address differences in terminology and structure; (5) *scalability and efficiency*: managing large data volumes while maintaining performance; and (6) *evaluating and benchmarking*: developing metrics and datasets that accurately reflect real-world complexity for open NL2SQL solutions.

Develop Cost-effective NL2SQL Methods. LLM-based NL2SQL methods show great potential but are limited by high token consumption, leading to increased costs and slower inference times. In contrast, PLM-based NL2SQL methods excel at handling complex SQL queries and accurately interpreting

database schemas. A promising approach is to combine the strengths of both, developing modular NL2SQL solutions or using a multi-agent framework to integrate LLMs and PLMs for the NL2SQL task. This hybrid strategy can manage complex queries while conserving tokens and reducing costs. However, effectively balancing LLM and PLM usage to optimize performance and resource efficiency remains an open challenge.

Make NL2SQL Solutions Trustworthy. Ensuring NL2SQL solutions are trustworthy is essential for generating accurate and reliable SQL, mitigating risk, and reducing the need for manual intervention. Key topics include the following:

Interpreting NL2SQL Solutions. Understanding the reasoning behind a NL2SQL model's performance enhances confidence in its reliability. Explainable AI techniques [151], [152], such as surrogate models [153] and saliency maps [154], can help reveal the decision-making processes within the model. However, their effectiveness in NL2SQL contexts, especially with combined LLM and PLM, remains an open question. In addition, multi-agent frameworks using LLMs [155] can improve reliability by dividing the NL2SQL task into specialized sub-tasks, each handled by an agent optimized for a particular function. This approach leverages the strengths of multiple agents to enhance overall robustness and accuracy. Nevertheless, effectively coordinating these agents to work harmoniously and optimize their combined performance poses significant challenges in NL2SQL contexts.

NL2SQL Debugging Tools. Inspired by code compilers with step-by-step debugging features, a visual debugger for NL2SQL could improve accuracy and reliability by measuring semantic and syntactic errors in generated SQL queries. Such tools would detect potential errors, enable users to examine the SQL generation process, identify mismatches, and understand the logic behind generated SQL. However, while traditional compilers capture syntactic errors, NL2SQL debugging tools would also need to address semantic errors, *i.e.*, ensuring that generated SQL align with the intent of the original NL query.

Interactive NL2SQL Tools. These tools are essential for empowering professional users (*e.g.*, DBAs) to create complex SQL queries that span multiple databases, often exceeding 50 lines of code. A key feature is the model's ability to decompose complex queries into manageable sub-queries, reducing cognitive load and enabling DBAs to focus on each part before reassembling them. Supporting both bottom-up and top-down approaches, these tools allow DBAs to build queries according to their workflow preferences and to correct and refine generated SQL as needed. This collaborative process aligns complex SQL output with user intent, combining the model's capabilities with the DBA's expertise.

Adaptive Training Data Synthesis. Current learning-based NL2SQL methods struggle to adapt to new, unseen domains, highlighting a gap in generalizability across data domains. In addition, the performance of these methods heavily relies on the quality, coverage, diversity, and amount of the training data. Therefore, an interesting research problem is to automatically and incrementally generate (NL, SQL) pairs based on the model performance. The key idea is to dynamically synthesize or augment the training data by leveraging feedback

from NL2SQL evaluation results. Specifically, by incorporating insights from evaluation metrics and evaluation results, we can identify specific weaknesses of the model. Using this information, we can synthesize training data that continually evolves with the help of LLMs to cover a broader range of scenarios and domains.

XI. CONCLUSION

In this paper, we provide a comprehensive review of NL2SQL techniques from a lifecycle perspective in the era of LLMs. We formally state the NL2SQL task, discuss its challenges, and present a detailed taxonomy of solutions based on the language models they rely on. We summarize the key modules of language model-powered NL2SQL methods, covering pre-processing strategies, translation models, and post-processing techniques. We also analyze NL2SQL benchmarks and evaluation metrics, highlighting their characteristics and typical errors. Furthermore, we outline a roadmap for practitioners to adapt LLMs for NL2SQL solutions. Finally, we maintain an online NL2SQL handbook to guide researchers and practitioners in the latest NL2SQL advancements and discuss the research challenges and open problems for NL2SQL.

REFERENCES

- [1] Z. Gu, J. Fan, N. Tang, and et al., “Few-shot text-to-sql translation using structure and content prompt learning,” *SIGMOD*, 2023.
- [2] Z. Chen, S. Chen, M. White, R. J. Mooney, and et al., “Text-to-sql error correction with language models of code,” in *ACL*, 2023.
- [3] L. Wang and et al., “Proton: Probing schema linking information from pre-trained language models for text-to-sql parsing,” in *KDD*, 2022.
- [4] A. Liu, X. Hu, L. Lin, and L. Wen, “Semantic enhanced text-to-sql parsing via iteratively learning schema linking graph,” in *KDD*, 2022.
- [5] M. Pourreza and D. Rafiei, “Din-sql: Decomposed in-context learning of text-to-sql with self-correction,” *NeurIPS*, 2024.
- [6] D. Gao, H. Wang, Y. Li, and et al., “Text-to-sql empowered by large language models: A benchmark evaluation,” *Proc. VLDB Endow.*, 2024.
- [7] H. Li, J. Zhang, C. Li, and H. Chen, “Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql,” in *AAAI*, 2023.
- [8] N. Tang, C. Yang, J. Fan, and et al., “Verifai: Verified generative AI,” in *CIDR*. www.cidrdb.org, 2024.
- [9] Y. Ye, J. Hao, Y. Hou, and et al., “Generative ai for visualization: State of the art and future directions,” *Visual Informatics*, 2024.
- [10] Y. Zhu, S. Du, B. Li, and et al., “Are large language models good statisticians?” *CoRR*, 2024.
- [11] Y. Xie, Y. Luo, G. Li, and N. Tang, “Haichart: Human and AI paired visualization system,” *Proc. VLDB Endow.*, 2023.
- [12] L. Shen, E. Shen, Y. Luo, and et al., “Towards natural language interfaces for data visualization: A survey,” *IEEE TVCG.*, 2023.
- [13] Y. Luo, X. Qin, C. Chai, and et al., “Steerable self-driving data visualization,” *IEEE Trans. Knowl. Data Eng.*, 2022.
- [14] Y. Luo, N. Tang, G. Li, and et al., “Natural language to visualization by neural machine translation,” *IEEE Trans. Vis. Comput. Graph.*, 2022.
- [15] J. Tang, Y. Luo, M. Ouzzani, and et al., “Sevi: Speech-to-visualization through neural machine translation,” in *SIGMOD*, 2022.
- [16] Y. Luo, N. Tang, G. Li, and et al., “Synthesizing natural language to visualization (NL2VIS) benchmarks from NL2SQL benchmarks,” in *SIGMOD*, 2021.
- [17] X. Qin, Y. Luo, N. Tang, and et al., “Making data visualization more efficient and effective: a survey,” *VLDB J.*, 2020.
- [18] Y. Luo, X. Qin, N. Tang, and et al., “Deepeye: Towards automatic data visualization,” in *ICDE*. IEEE Computer Society, 2018.
- [19] Y. Luo, X. Qin, N. Tang, G. Li, and X. Wang, “Deepeye: Creating good data visualizations by keyword search,” in *SIGMOD*. ACM, 2018.
- [20] X. Zhou, Z. Sun, and G. Li, “DB-GPT: large language model meets database,” *Data Sci. Eng.*, 2024.
- [21] S. Amer-Yahia and et al., “From large language models to databases and back: A discussion on research and education,” *SIGMOD*, 2023.

- [22] W. Zhang, Y. Wang, Y. Song, and et al., “Natural language interfaces for tabular data querying and visualization: A survey,” *TKDE*, 2024.
- [23] G. Katsogiannis-Meimarakis and G. Koutrika, “A survey on deep learning approaches for text-to-sql,” *The VLDB Journal*, 2023.
- [24] N. Deng, Y. Chen, and Y. Zhang, “Recent advances in text-to-sql: A survey of what we have and what we expect,” in *COLING*, 2022.
- [25] H. Kim, B.-H. So, W.-S. Han, and et al., “Natural language to sql: Where are we today?” *Proc. VLDB Endow.*, 2020.
- [26] L. Shi, Z. Tang, N. Zhang, X. Zhang, and Z. Yang, “A survey on employing large language models for text-to-sql tasks,” 2024.
- [27] A. Mohammadjafari, A. S. Maida, and R. Gottumukkala, “From natural language to sql: Review of llm-based text-to-sql systems,” 2024.
- [28] X. Zhu, Q. Li, L. Cui, and Y. Liu, “Large language model enhanced text-to-sql generation: A survey,” 2024.
- [29] Z. Hong, Z. Yuan, Q. Zhang, and et al., “Next-generation database interfaces: A survey of llm-based text-to-sql,” *arXiv:2406.08426*, 2024.
- [30] F. Özcan, A. Quamar, J. Sen, and et al., “State of the art and open challenges in natural language interfaces to data,” in *SIGMOD*, 2020.
- [31] Y. Li and D. Rafie, “Natural language data management and interfaces: Recent development and open challenges,” in *SIGMOD*, 2017.
- [32] G. Katsogiannis-Meimarakis and et al., “Natural language interfaces for databases with deep learning,” *Proc. VLDB Endow.*, 2023.
- [33] G. Katsogiannis-Meimarakis and G. Koutrika, “A deep dive into deep learning approaches for text-to-sql systems,” in *SIGMOD*, 2021.
- [34] N. Rajkumar and et al., “Evaluating the text-to-sql capabilities of large language models,” *CoRR*, 2022.
- [35] F. Li and H. V. Jagadish, “Nalir: an interactive natural language interface for querying relational databases,” in *ACM SIGMOD*, 2014.
- [36] T. Yu, C.-S. Wu, X. V. Lin, and et al., “Grappa: Grammar-augmented pre-training for table semantic parsing,” in *ICLR*, 2021.
- [37] C. Xiao, M. Dymetman, and C. Gardent, “Sequence-based structured prediction for semantic parsing,” in *ACL*, 2016.
- [38] K. Lin, B. Bogin, M. Neumann, and et al., “Grammar-based neural text-to-sql generation,” *arXiv:1905.13326*, 2019.
- [39] B. Bogin, M. Gardner, and et al., “Representing schema structure with graph neural networks for text-to-sql parsing,” *arXiv:1905.06241*, 2019.
- [40] J. Devlin and et al., “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *NAACL*, 2019.
- [41] C. Raffel, N. Shazeer, A. Roberts, and et al., “Exploring the limits of transfer learning with a unified text-to-text transformer,” *JMLR*, 2020.
- [42] J. Li and et al., “Graphix-t5: Mixing pre-trained transformers with graph-aware layers for text-to-sql parsing,” *arXiv:2301.07507*, 2023.
- [43] Z. Gu, J. Fan, N. Tang, and et al., “Interleaving pre-trained language models and large language models for zero-shot nl2sql generation,” *arXiv:2306.08891*, 2023.
- [44] T. Yu, R. Zhang, K. Yang, and et al., “Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task,” in *EMNLP 2018*, 2018.
- [45] B. Li, Y. Luo, C. Chai, and et al., “The dawn of natural language to sql: Are we fully ready?” *Proc. VLDB Endow.*, 2024.
- [46] W. X. Zhao, K. Zhou, J. Li, and et al., “A survey of large language models,” *arXiv:2303.18223*, 2023.
- [47] S. Minaee, T. Mikolov, N. Nikzad, and et al., “Large language models: A survey,” *arXiv:2402.06196*, 2024.
- [48] H. Li, J. Zhang, H. Liu, and et al., “Codes: Towards building open-source language models for text-to-sql,” *SIGMOD*, 2024.
- [49] C. Zhang and et al., “Finsql: Model-agnostic llms-based text-to-sql framework for financial analysis,” in *SIGMOD*, 2024.
- [50] R. Li, L. B. Allal, Y. Zi, and et al., “Starcoder: may the source be with you!” *arXiv:2305.06161*, 2023.
- [51] J. Li, B. Hui, G. Qu, and et al., “Can LLM already serve as A database interface? A big bench for large-scale database grounded text-to-sqls,” in *NeurIPS*, 2023.
- [52] W. Lei, W. Wang, Z. Ma, and et al., “Re-examining the role of schema linking in text-to-sql,” in *EMNLP*, 2020.
- [53] S. Talaei, M. Pourreza, Y.-C. Chang, and et al., “Chess: Contextual harnessing for efficient sql synthesis,” *arXiv:2405.16755*, 2024.
- [54] M. Pourreza and D. Rafie, “Dts-sql: Decomposed text-to-sql with small large language models,” *arXiv:2402.01117*, 2024.
- [55] G. Qu, J. Li, B. Li, and et al., “Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation,” *arXiv:2405.15307*, 2024.
- [56] Z. Li and et al., “Pet-sql: A prompt-enhanced two-stage text-to-sql framework with cross-consistency,” *arXiv:2403.09732*, 2024.
- [57] H. Zhang, R. Cao, H. Xu, and et al., “Coe-sql: In-context learning for multi-turn text-to-sql with chain-of-editions,” *arXiv:2405.02712*, 2024.
- [58] T. Ren, Y. Fan, Z. He, , and et al., “Purple: Making a large language model a better sql writer,” *arXiv:2403.20014*, 2024.
- [59] Y. Fan, Z. He, T. Ren, and et al., “Metasql: A generate-then-rank framework for natural language to sql translation,” 2024.
- [60] Y. Xie, X. Jin, T. Xie, and et al., “Decomposition for enhancing attention: Improving llm-based text-to-sql through workflow paradigm,” *arXiv:2402.10671*, 2024.
- [61] X. Dong, C. Zhang, Y. Ge, and et al., “C3: Zero-shot text-to-sql with chatgpt,” *arXiv:2307.07306*, 2023.
- [62] D. Rai, B. Wang, Y. Zhou, and et al., “Improving generalization in language model-based text-to-sql semantic parsing: Two simple semantic boundary-based techniques,” in *ACL*, 2023.
- [63] H. Zhang, R. Cao, L. Chen, and et al., “Act-sql: In-context learning for text-to-sql with automatically-generated chain-of-thought,” in *Findings of EMNLP*, 2023.
- [64] S. Chang and E. Fosler-Lussier, “Selective demonstrations for cross-domain text-to-sql,” in *Findings of EMNLP*, 2023.
- [65] B. Wang, C. Ren, J. Yang, and et al., “Mac-sql: A multi-agent collaborative framework for text-to-sql,” *CoRR*, 2023.
- [66] H. Fu, C. Liu, B. Wu, and et al., “Catsql: Towards real world natural language to sql applications,” *Proc. VLDB Endow.*, 2023.
- [67] A. Bazaga, P. Liò, and et al., “Sqlformer: Deep auto-regressive query graph generation for text-to-sql translation,” *arXiv:2310.18376*, 2023.
- [68] Y. Xiang, Q.-W. Zhang, X. Zhang, and et al., “G3r: A graph-guided generate-and-rerank framework for complex and cross-domain text-to-sql generation,” in *Findings of ACL*, 2023.
- [69] Y. Hu, Y. Zhao, J. Jiang, and et al., “Importance of synthesizing high-quality data for text-to-sql parsing,” in *Findings of ACL*. Association for Computational Linguistics, 2023.
- [70] L. Zeng, S. H. K. Parthasarathi, and D. Hakkani-Tur, “N-best hypotheses reranking for text-to-sql systems,” in *SLT*, 2023.
- [71] J. Qi, J. Tang, Z. He, and et al., “Rasat: Integrating relational structures into pretrained seq2seq model for text-to-sql,” *arXiv:2205.06983*, 2022.
- [72] T. Scholak and et al., “Picard: Parsing incrementally for constrained auto-regressive decoding from language models,” 2021.
- [73] C. Gao, B. Li, W. Zhang, and et al., “Towards generalizable and robust text-to-sql parsing,” *arXiv:2210.12674*, 2022.
- [74] B. Hui and et al., “S2sql: Injecting syntax to question-schema interaction graph encoder for text-to-sql parsers,” in *Findings of ACL*, 2022.
- [75] B. Wang, R. Shin, X. Liu, and et al., “RAT-SQL: relation-aware schema encoding and linking for text-to-sql parsers,” in *ACL*, 2020.
- [76] O. Rubin and J. Berant, “Smbop: Semi-autoregressive bottom-up semantic parsing,” *arXiv:2010.12412*, 2020.
- [77] J. Huang, Y. Wang, Y. Wang, and et al., “Relation aware semi-autoregressive semantic parsing for nl2sql,” *arXiv:2108.00804*, 2021.
- [78] X. V. Lin and et al., “Bridging textual and tabular data for cross-domain text-to-sql semantic parsing,” *arXiv:2012.12627*, 2020.
- [79] J. Guo, Z. Zhan, Y. Gao, and et al., “Towards complex text-to-sql in cross-domain database with intermediate representation,” in *ACL*, 2019.
- [80] T. Yu, Z. Li, Z. Zhang, and et al., “Typesql: Knowledge-based type-aware neural text-to-sql generation,” *arXiv:1804.09769*, 2018.
- [81] F. J. Damerau, “A technique for computer detection and correction of spelling errors,” *Communications of the ACM*, 1964.
- [82] U. Brunner and K. Stockinger, “ValueNet: A natural language-to-sql system that learns from database information,” in *ICDE*. IEEE, 2021.
- [83] Z. Dong, S. Sun, H. Liu, and et al., “Data-anonymous encoding for text-to-sql generation,” in *EMNLP-IJCNLP*, 2019.
- [84] T. Brown, B. Mann, N. Ryder, and et al., “Language models are few-shot learners,” *NeurIPS*, 2020.
- [85] D. Lee, C. Park, J. Kim, and et al., “Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation,” *arXiv:2405.07467*, 2024.
- [86] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” *Communications of the ACM*, 1975.
- [87] P. Yin, G. Neubig, W.-t. Yih, and et al., “Tabert: Pretraining for joint understanding of textual and tabular data,” *arXiv:2005.08314*, 2020.
- [88] R. Speer, C. Havasi et al., “Representing general relational knowledge in conceptnet 5,” in *LREC*, 2012.
- [89] P. Indyk and R. Motwani, “Approximate nearest neighbors: towards removing the curse of dimensionality,” in *STOC*, 1998.
- [90] S. Robertson, H. Zaragoza et al., “The probabilistic relevance framework: Bm25 and beyond,” *FOUND TRENDS INF RET*, 2009.
- [91] F. Li and et al., “Schema-free sql,” in *SIGMOD*, 2014.
- [92] T. Yu and et al., “Syntaxqlnet: Syntax tree networks for complex and cross-domain text-to-sql task,” *arXiv:1810.05237*, 2018.
- [93] J.-O. Lee and D.-K. Baik, “Semql: a semantic query language for multidatabase systems,” in *CIKM*, 1999.

- [94] R. Zhang and et al., “Editing-based sql query generation for cross-domain context-dependent questions,” *arXiv:1909.00786*, 2019.
- [95] Y. Gan and et al., “Natural SQL: Making SQL easier to infer from natural language specifications,” in *Findings of EMNLP*, 2021.
- [96] T. Wolfson, M. Geva, A. Gupta, and et al., “Break it down: A question understanding benchmark,” *TACL*, 2020.
- [97] L. Dou, Y. Gao, X. Liu, and et al., “Towards knowledge-intensive text-to-sql semantic parsing with formulaic knowledge,” in *EMNLP*, 2022.
- [98] V. Karpukhin, B. Oğuz, S. Min, and et al., “Dense passage retrieval for open-domain question answering,” *arXiv:2004.04906*, 2020.
- [99] Q. Liu, D. Yang, J. Zhang, J. Guo, B. Zhou, and J. Lou, “Awakening latent grounding from pretrained language models for semantic parsing,” in *ACL/IJCNLP (Findings)*, 2021.
- [100] G. Sui, Z. Li, Z. Li, and et al., “Reboost large language model-based text-to-sql, text-to-python, and text-to-function—with real applications in traffic domain,” *arXiv:2310.18752*, 2023.
- [101] K. Clark, “Electra: Pre-training text encoders as discriminators rather than generators,” *arXiv:2003.10555*, 2020.
- [102] R. Cao and et al., “Lgesql: line graph enhanced text-to-sql model with mixed local and non-local relations,” *arXiv:2106.01093*, 2021.
- [103] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, “Graph attention networks,” *arXiv:1710.10903*, 2017.
- [104] X. Xu and et al., “Sqlnet: Generating structured queries from natural language without reinforcement learning,” *arXiv:1711.04436*, 2017.
- [105] V. Zhong and et al., “Seq2sql: Generating structured queries from natural language using reinforcement learning,” *CoRR*, 2017.
- [106] X. Bi, D. Chen, G. Chen, and et al., “Deepseek Ilm: Scaling open-source language models with longtermism,” *arXiv:2401.02954*, 2024.
- [107] X. Liu, J. Wang, J. Sun, and et al., “Prompting frameworks for large language models: A survey,” *arXiv:2311.12785*, 2023.
- [108] J. Wei, X. Wang, D. Schuurmans, and et al., “Chain-of-thought prompting elicits reasoning in large language models,” *NeurIPS*, 2022.
- [109] C.-Y. Tai, Z. Chen, T. Zhang, and et al., “Exploring chain-of-thought style prompting for text-to-sql,” *arXiv:2305.14215*, 2023.
- [110] F. Li and H. V. Jagadish, “Constructing an interactive natural language interface for relational databases,” *Proc. VLDB Endow.*, 2014.
- [111] B. Eyal, A. Bachar, O. Haroche, and et al., “Semantic decomposition of question and sql for text-to-sql parsing,” *arXiv:2310.13575*, 2023.
- [112] T. Wolfson and et al., “Weakly supervised mapping of natural language to sql through question decomposition,” *Findings of NAACL*, 2021.
- [113] D. Lee, “Clause-wise and recursive decoding for complex and cross-domain text-to-sql generation,” *arXiv:1904.08835*, 2019.
- [114] X. Wang and et al., “Self-consistency improves chain of thought reasoning in language models,” *arXiv:2203.11171*, 2022.
- [115] M. Renze and E. Guven, “The effect of sampling temperature on problem solving in large language models,” *arXiv:2402.05201*, 2024.
- [116] A. Kelkar, R. Relan, V. Bhardwaj, and et al., “Bertrand-dr: Improving text-to-sql using a discriminative re-ranker,” *arXiv:2002.00557*, 2020.
- [117] Y. Luo, Y. Zhou, N. Tang, and et al., “Learned data-aware image representations of line charts for similarity search,” *SIGMOD*, 2023.
- [118] K. Zhang and et al., “Refsql: A retrieval-augmentation framework for text-to-sql generation,” in *Findings of EMNLP*, 2023.
- [119] D. A. Dahl, M. Bates, M. K. Brown, and et al., “Expanding the scope of the atis task: The atis-3 corpus,” in *Workshop on HLT*, 1994.
- [120] J. M. Zelle and R. J. Mooney, “Learning to parse database queries using inductive logic programming,” in *AAAI*, 1996.
- [121] L. R. Tang and R. J. Mooney, “Automated construction of database interfaces: Integrating statistical and relational learning for semantic parsing,” in *EMNLP*, 2000.
- [122] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, “Sqlizer: query synthesis from natural language,” *Proc. ACM Program. Lang.*, 2017.
- [123] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, “Learning a neural semantic parser from user feedback,” in *ACL*, 2017.
- [124] C. Finegan-Dollak, J. K. Kummerfeld, L. Zhang, and et al., “Improving text-to-sql evaluation methodology,” in *ACL*, 2018.
- [125] T. Yu, R. Zhang, M. Yasunaga, and et al., “Sparc: Cross-domain semantic parsing in context,” in *ACL*, 2019.
- [126] T. Yu, R. Zhang, H. Er, and et al., “Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases,” in *EMNLP-IJCNLP*, 2019.
- [127] Q. Min, Y. Shi, and Y. Zhang, “A pilot study for chinese SQL semantic parsing,” in *EMNLP-IJCNLP*, 2019.
- [128] P. Wang, T. Shi, and C. K. Reddy, “Text-to-sql generation for question answering on electronic medical records,” in *WWW*, 2020.
- [129] T. Shi and et al., “On the potential of lexico-logical alignments for semantic parsing to SQL queries,” in *Findings of EMNLP*, 2020.
- [130] J. Sen, C. Lei, A. Quamar, and et al., “Athena++ natural language querying for complex nested sql queries,” *Proc. VLDB Endow.*, 2020.
- [131] A. T. Nguyen, M. H. Dao, and D. Q. Nguyen, “A pilot study of text-to-sql semantic parsing for vietnamese,” in *Findings of EMNLP*, 2020.
- [132] L. Wang, A. Zhang, K. Wu, and et al., “Dusql: A large-scale and pragmatic chinese text-to-sql dataset,” in *EMNLP*, 2020.
- [133] M. A. José and F. G. Cozman, “mrat-sql+ gap: a portuguese text-to-sql transformer,” in *BRACIS*, 2021.
- [134] J. Guo, Z. Si, Y. Wang, and et al., “Chase: A large-scale and pragmatic chinese dataset for cross-database context-dependent text-to-sql,” in *ACL/IJCNLP*, 2021.
- [135] Y. Gan, X. Chen, Q. Huang, and et al., “Towards robustness of text-to-sql models against synonym substitution,” in *ACL/IJCNLP*, 2021.
- [136] Y. Gan, X. Chen, and M. Purver, “Exploring underexplored limitations of cross-domain text-to-sql generalization,” in *EMNLP*, 2021.
- [137] X. Deng, A. H. Awadallah, C. Meek, and et al., “Structure-grounded pretraining for text-to-sql,” in *NAACL-HLT*, 2021.
- [138] C. Lee, O. Polozov, and M. Richardson, “Kaggledbqa: Realistic evaluation of text-to-sql parsers,” in *ACL/IJCNLP*, 2021.
- [139] M. Hazoom and et al., “Text-to-sql in the wild: A naturally-occurring dataset based on stack exchange data,” *arXiv:2106.05006*, 2021.
- [140] P. Ma and S. Wang, “Mt-teql: Evaluating and augmenting neural NLIDB on real-world linguistic and schema variations,” *VLDB*, 2021.
- [141] D. Bakshandaeva, O. Somov, E. Dmitrieva, V. Davydova, and E. Tuttubalina, “PAUQ: text-to-sql in russian,” in *Findings of EMNLP*, 2022.
- [142] S. Chang, J. Wang, M. Dong, and et al., “Dr.spider: A diagnostic evaluation benchmark towards text-to-sql robustness,” in *ICLR*, 2023.
- [143] A. Bhaskar, T. Tomar, A. Sathe, and S. Sarawagi, “Benchmarking and improving text-to-sql generation under ambiguity,” in *EMNLP*, 2023.
- [144] Y. Zhang, J. Deriu, G. Katsogiannis-Meimarakis, and et al., “Sciencedebenchmark: A complex real-world benchmark for evaluating natural language to SQL systems,” *Proc. VLDB Endow.*, 2023.
- [145] R. Kumar, A. R. Dibbu, S. Harsola, and et al., “Booksq: A large scale text-to-sql dataset for accounting domain,” *arXiv:2406.07860*, 2024.
- [146] D. Zheng and et al., “Archer: A human-labeled text-to-sql dataset with arithmetic, commonsense and hypothetical reasoning,” in *EACL*, 2024.
- [147] F. Lei, J. Chen, Y. Ye, and et al., “Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows,” 2024.
- [148] A. Narechania, A. Fournier, B. Lee, and et al., “Diy: Assessing the correctness of natural language to sql systems,” in *IUI*, 2021.
- [149] Z. Ning, Z. Zhang, T. Sun, and et al., “An empiricmacsql study of model errors and user error discovery and repair strategies in natural language database queries,” in *IUI*, 2023.
- [150] R. Sun, S. O. Arik, H. Nakhost, and et al., “Sql-palm: Improved large language modeladaptation for text-to-sql,” *arXiv:2306.00739*, 2023.
- [151] S. Ali, T. Abuhmed, S. H. A. El-Sappagh, and et al., “Explainable artificial intelligence (XAI): what we know and what is left to attain trustworthy artificial intelligence,” *Inf. Fusion*, 2023.
- [152] H. Zhao, H. Chen, F. Yang, and et al., “Explainability for large language models: A survey,” *ACM Trans. Intell. Syst. Technol.*, 2024.
- [153] Y. Chen, R. Zhong, N. Ri, and et al., “Do models explain themselves? counterfactual simulatability of natural language explanations,” *arXiv:2307.08678*, 2023.
- [154] A. Shrikumar, P. Greenside, and A. Kundaje, “Learning important features through propagating activation differences,” in *ICML*, 2017.
- [155] J. Zhang, J. Xiang, Z. Yu, and et al., “Aflow: Automating agentic workflow generation,” 2024.