Automated Validating and Fixing of Text-to-SQL Translation with Execution Consistency

YICUN YANG, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China
ZHAOGUO WANG*, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China
YU XIA, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China
ZHUORAN WEI, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China
HAORAN DING, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China
RUZICA PISKAC, Department of Computer Science, Yale University, USA
HAIBO CHEN, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China
JINYANG LI, Department of Computer Science, New York University, USA

State-of-the-art Text-to-SQL models rely on fine-tuning or few-shot prompting to help LLMs learn from training datasets containing mappings from natural language (NL) queries to SQL statements. Consequently, the quality of the dataset can greatly affect the accuracy of these Text-to-SQL models. Unlike other NL tasks, Text-to-SQL datasets are prone to errors despite extensive manual efforts due to the subtle semantics of SQL. Our study has found a non-negligible (>30%) portion of incorrect NL to SQL mapping cases exists in popular datasets Spider and BIRD.

This paper aims to improve the quality of Text-to-SQL training datasets and thereby increase the accuracy of the resulting models. To do so, we propose a necessary correctness condition called execution consistency. For a given database instance, an NL to SQL mapping satisfies execution consistency if the execution result of an NL query matches that of the corresponding SQL. We develop SQLDriller to detect incorrect NL to SQL mappings based on execution consistency in a best-effort manner by crafting database instances that likely result in violations of execution consistency. It generates multiple candidate SQL predictions that differ in their syntax structures. Using a SQL equivalence checker, SQLDriller obtains counterexample database instances that can distinguish non-equivalent candidate SQLs. It then checks the execution consistency of an NL to SQL mapping under this set of counterexamples. The evaluation shows SQLDriller effectively detects and fixes incorrect mappings in the Text-to-SQL dataset, and it improves the model accuracy by up to 13.6%.

CCS Concepts: • **Information systems → Structured Query Language**; • **Computing methodologies → Machine translation**; • **Human-centered computing → Natural language interfaces**.

*Corresponding author (zhaoguowang@sjtu.edu.cn)

Authors' Contact Information: Yicun Yang, yangyicun@sjtu.edu.cn, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; Zhaoguo Wang, zhaoguowang@sjtu.edu.cn, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; Yu Xia, rainxxy@sjtu.edu.cn, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; Zhuoran Wei, 84461810@sjtu.edu.cn, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; Haoran Ding, nhaorand@sjtu.edu.cn, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; Ruzica Piskac, ruzica.piskac@yale.edu, Department of Computer Science, Yale University, New Haven, Connecticut, USA; Haibo Chen, haibochen@sjtu.edu.cn, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China; Jinyang Li, jinyang@cs.nyu.edu, Department of Computer Science, New York University, New York, New York, USA.

ACM 2836-6573/2025/6-ART134
https://doi.org/10.1145/3725271

Additional Key Words and Phrases: Text-to-SQL; Execution Consistency; SQL Query Equivalence; Language Model; Natural Language Interface for Databases

ACM Reference Format:
Yicun Yang, Zhaoguo Wang, Yu Xia, Zhuoran Wei, Haoran Ding, Ruzica Piskac, Haibo Chen, and Jinyang Li. 2025. Automated Validating and Fixing of Text-to-SQL Translation with Execution Consistency. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 134 (June 2025), 28 pages. https://doi.org/10.1145/3725271

1 Introduction

Text-to-SQL leverages the recent advances of NLP machine learning models to translate a natural language (NL) query into a SQL query [2, 23, 40]. This approach holds great promise to enable a vast population of non-technical users to effectively and intuitively retrieve information in databases [2, 10, 40]. As such, Text-to-SQL finds application in many scenarios, including business intelligence (BI) and data analysis systems, virtual assistants, and educational tools [18, 21, 32, 39, 43, 52].

To learn to translate NL queries to SQL, a machine learning model must have access to a large training dataset containing many instances of NL to SQL mappings. Natural Language Processing researchers have curated several such datasets like Spider [57] and BIRD [30]. With the recent success of large language models (LLM), state-of-the-art Text-to-SQL models are based on either fine-tuning using a Text-to-SQL training dataset [27–29, 45, 47, 51] or few-shot prompting with examples extracted from the training dataset [20, 28, 42]. Upon training with the training dataset, a test dataset is utilized to assess and measure the model's accuracy.

To improve Text-to-SQL accuracy, researchers have invested great amounts of effort to improve the underlying model architecture [6, 16, 20, 27–29, 42, 45, 47, 51, 61]. However, the quality (aka correctness) of the dataset also critically affects the model's accuracy. Unfortunately, the issue of dataset quality has so far been overlooked in the research community. Unlike many task-specific NL datasets such as question answering or basic mathematical reasoning, it can be intricate to ensure the correctness of instances in Text-to-SQL datasets due to the subtle and complex semantics of SQL. Figure 1 shows an example of incorrect NL to SQL mapping in the Spider training dataset. The NL query aims to list the name of each aircraft and the number of flights it has completed. This SQL is incorrect because it leaves out any aircraft with no flights recorded in the Flight table by using INNER JOIN instead of LEFT JOIN. There are 17 similar mappings with incorrect JOIN types among each set of 500 mappings randomly selected from Spider's training and test datasets, correspondingly. Such incorrect SQL mappings in the datasets affect Text-to-SQL model accuracy from two aspects. In the test dataset, the incorrect SQL mappings may cause imprecise model accuracy evaluation results. Such mappings accidentally misjudge the correctness of SQL predictions from models and cause imprecise accuracy evaluation results. In the training dataset, the non-negligible amount of incorrect SQL mappings leads to mispredictions of models. We attribute several test mispredictions to this training dataset error in Figure 1 such as a test NL query aiming to list the stadium name and the number of concerts of each one. More broadly, we have studied 168 incorrect predictions of 3 top-ranking models [20, 27, 42] that use the Spider dataset. Among these, 42 (25%) are due to errors in the training dataset.

As the error in Figure 1 demonstrates, the incorrect JOIN type exhibits a subtle semantic distinction from the correct counterpart, which is rather tricky to detect. Since the datasets usually contain a large number of mappings crafted mostly by non-expert-level SQL users with limited experience[1], it is understandable that errors arise despite a great amount of manual care and effort (e.g., the Spider authors have spent more than 450 man-hours reviewing all the 10,000 instances of the whole dataset [57]).

[1] e.g., Spider dataset is mostly written by 11 undergraduate students

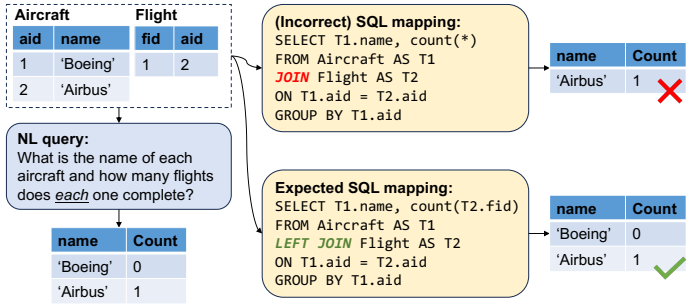

Fig. 1. An example of incorrect Text-to-SQL mapping in Spider's training dataset. Using INNER JOIN instead of LEFT JOIN, the incorrect SQL would leave out any aircraft with no flight recorded in the Flight table.

This paper aims to improve the quality of Text-to-SQL datasets. We develop a tool, SQLDriller, to automatically flag likely incorrect NL to SQL mappings in the training dataset and propose a fix. As the test dataset serves as the ground truth to assess model accuracy, we perform a meticulous manual review of every mapping after SQLDriller has done its fix to maximize correctness. Efforts involved in the manual review are manageable since the test dataset is much smaller than the whole dataset's size (e.g. 20% in Spider).

The central challenge faced by an error-detection tool such as SQLDriller lies in the lack of any formal semantics for NL statements. To tackle this, our insight is to make the user's intent for an NL query explicit using concrete database instances. Specifically, we propose a correctness condition called **execution consistency**. An NL to SQL mapping ($q_{nl} \rightarrow q_{sql}$) satisfies execution consistency in a given database instance if the "execution result" of q_{nl} matches that of q_{sql}. Execution consistency is a necessary but not sufficient correctness condition for NL to SQL mapping. It is a particularly useful property because it can be automatically checked on any concrete database instance by leveraging LLMs' NL reasoning capability to "execute" an NL query and compare that to SQL's execution result. Based on execution consistency, we develop SQLDriller to detect incorrect mappings in a best-effort manner by crafting database instances that likely cause the violation of execution consistency.

In designing SQLDriller, we make the key observation that incorrect SQL mappings usually differ in both syntax and keyword usage from the correct counterparts. Thus, SQLDriller starts by using a base Text-to-SQL model to generate multiple SQL candidates for a given NL query in addition to the original SQL. Next, SQLDriller introduces a SQL equivalence checker to generate counterexamples (concrete database instances) that can distinguish non-equivalent SQLs in this candidate set. By checking execution consistency of the given NL to SQL mapping using the counterexamples, SQLDriller can detect incorrect SQL mapping and propose a correct fix. False negatives sometimes occur if the SQL candidate set does not contain any correct SQL or inaccurate LLM NL execution happens to produce results that match those of an incorrect SQL. False positives (judging correct SQLs as incorrect) may also happen if inaccurate NL execution produces results that do not match those of a correct SQL. Based on the improved training datasets, we further optimize and improve the prediction accuracy of Text-to-SQL models using SQLDriller. In particular, we change the models to generate multiple SQL predictions and rely on SQLDriller to pick the optimal SQL, which is the one satisfying execution consistency with the NL query on the largest number of generated counterexamples.

To summarize, this paper has the following contributions:

- It introduces a necessary correctness condition for Text-to-SQL, upon which a new tool, SQLDriller, is developed to automatically detect and fix potential errors in Text-to-SQL translations.

Table 1. The error rate of various hardness levels in Spider's and BIRD's 500 sampled training mappings. The NL and SQL query complexity increases from "Easy" to "Extra" level.

Dataset	Error Rate				
	All	Easy	Medium	Hard	Extra
Spider train	183/500 (36.6 %)	19/145 (13.1 %)	84/220 (38.2 %)	53/93 (57.0 %)	27/42 (64.3 %)
BIRD train	272/500 (54.4 %)	2/3 (66.7 %)	119/298 (39.9 %)	52/83 (62.7 %)	99/116 (85.3 %)

- It presents a systematic study on the quality of Text-to-SQL datasets and introduces a new test dataset derived from the Spider and BIRD benchmarks, offering a fair and precise evaluation.
- It successfully improves the quality of Spider and BIRD training dataset by automatically fixing the errors which can improve the accuracy of 6 top-ranking models by up to 13.6%.
- It provides comprehensive lessons based on the incorrect Text-to-SQL mappings discovered by SQLDriller, which hopefully can shed light on the developers who trying to build new datasets.

2 Background and Motivation

2.1 Preliminaries

Text-to-SQL Model Training. Text-to-SQL techniques enable users to query databases in a more intuitive and user-friendly manner [2, 10, 40]. Its basic idea is to allow users to query databases using natural language (NL) and employ Text-to-SQL models to translate the NL query into an executable SQL query to execute in the database. Text-to-SQL models are machine-learning models trained with a Text-to-SQL dataset. A Text-to-SQL dataset consists of a set of NL to SQL translation mappings and is typically divided into a training and a test (development) set for model training and accuracy evaluation, respectively [30, 57].

With the rise of Large Language Models (LLMs) [13, 25, 34, 44, 50], developers utilize them as the pre-trained models to perform supervised fine-tuning [27–29, 47, 51]. The training mappings are fed to the model and used to fine-tune the model's internal parameters to fit in Text-to-SQL tasks. Additionally, other models are tuned through dedicated prompting on powerful generative LLMs such as GPT models [37]. They apply few-shot in-context learning [15], which incorporates existing training mappings as examples to the prompt context of LLMs for instruction [20, 28, 42]. Both fine-tuned and prompted models learn from and heavily rely on the training dataset to derive significant performance in Text-to-SQL.

SQL Equivalence Verification. SQL equivalence verification checks whether a pair of SQL is semantically equivalent, i.e., outputting the same execution result for arbitrary given input tables. It is widely employed in scenarios such as validating the correctness of SQL rewrites [8, 9, 54] and providing counterexamples for database testing and bug detection [9, 22]. Existing works have developed automated tools to verify SQL equivalence relationships [14, 22, 54]. Such a tool constructs first-order logic formulas to model the semantics of SQL queries. It calls an SMT solver [4, 12] to automatically verify the formula denoting that two SQLs output the same result for arbitrary input tables, then determines equivalence relationships according to the solver's response. For non-equivalent SQL pairs, the SMT solver returns a concrete solution that assigns values to each variable in the formula. Such a value assignment violates the formula and is called a *counterexample* to this formula. By converting the value assignment into the records of input tables, the formula's

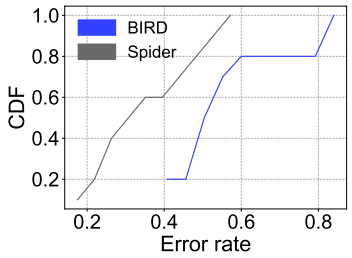

Fig. 2. CDF of error rates of all the schemas.

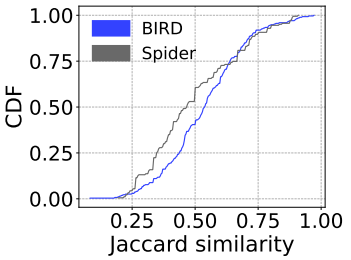

Fig. 3. CDF of SQL pairs' Jaccard similarity.

counterexample can be transformed into a counterexample of the SQL pair in the form of a concrete database instance [9, 22].

2.2 Text-to-SQL Dataset Quality

As is the case with all machine learning models, the quality of the training and test dataset is crucial to the accuracy and prediction capability of Text-to-SQL models. Existing works [30, 57, 61] have demonstrated that a Text-to-SQL dataset with diverse and comprehensive NL to SQL mappings can significantly improve the model's prediction capability. In addition to traditional models, recent LLM-based models use the training dataset for fine-tuning or prompt engineering. [27, 28, 47, 51] have shown that fine-tuning existing pre-trained language models can achieve notable accuracy. Other works design dedicated prompts with NL to SQL mapping examples from the training dataset and improve accuracy by over 10% compared to baselines without careful prompting [16, 20, 42].

Dataset Quality vs. Model Accuracy. The datasets play a crucial role in Text-to-SQL research, yet there has been little work focusing on improving their quality. To evaluate the adequacy of existing datasets and address potential oversights, we conducted an in-depth study of Spider [57] and BIRD [30], which are known as the leading Text-to-SQL datasets. These datasets, developed and maintained by a large team over several years, have served as the benchmarks for evaluation in the field and are applied by over 80 models according to their website rankings [1, 36]. They both consist of more than 10,000 NL to SQL mappings for model training and testing, with 200 and 80 distinct schemas in total, respectively. Following a careful study of the datasets, several key observations have emerged.

First, there exists a non-negligible portion of incorrect mappings within the dataset. We randomly sample 500 mappings from Spider and BIRD's training datasets respectively for a detailed study of all associated NL to SQL mappings. These samples span over all the hardness levels defined by Spider [57] shown in Table 1, whose distribution roughly represents that of the whole dataset. Among these samples, an overall error rate of over 35% is identified. Spider's training dataset has 183 errors of 500 mappings (36.6%) across 10 schemas, while the number is 272 of 500 mappings (54.4%) in BIRD's samples across 10 schemas. As shown in Table 1, the error rate increases with the rising hardness[2] of mappings. The error rates for Spider and BIRD both exceed 38% in medium-level mappings, 57% in hard ones, and even reach above 60% for extra-hard ones. Furthermore, Figure 2 shows that almost all the schemas exhibit a non-negligible portion of errors, indicating that these errors are not skewed in their distribution across the schemas. Schemas in Spider's training dataset have the highest error rate of 57.1%, the lowest of 13.1%, and the average of 30.1%. Corresponding numbers of schemas in BIRD are 84.0%, 36.0%, and 50.0%.

[2]Hardness levels defined by Spider are also applied to BIRD since BIRD's defined levels are not available in its training set (only manually annotated in development set.

Table 2. Model accuracy results evaluated with original SQL mappings (Original Accuracy) and with corrected SQL mappings (Re-evaluated Accuracy).

Model	Dataset	Original Accuracy (%)	Re-evaluated Accuracy (%)
DAIL-SQL (5 shots)	Spider test	83.7	61.8 (↓ 21.9)
DIN-SQL	Spider test	83.0	61.3 (↓ 21.7)
RESDSQL (T5-3b)	Spider test	76.2	52.3 (↓ 23.9)
Graphix-T5 (T5-large)	Spider test	64.8	46.3 (↓ 18.5)
SFT CODES (3b)	Bird dev	46.3	38.1 (↓ 8.2)
CODES (7b, 5 shots)	Bird dev	37.0	31.4 (↓ 5.6)

Second, incorrect mappings in the dataset play a decisive role in model prediction errors. In Spider's entire training dataset, more than 150 mappings exhibit a similar pattern of misusing INNER JOIN as LEFT JOIN like the example in Figure 1. These error mappings span over 60 distinct schemas. Consequently, despite using 3 top-ranking Text-to-SQL models developed under Spider [20, 27, 42], none could accurately infer the NL query resembling the one in Figure 1. Specifically, when tasked with predicting SQL with 10 similar NL queries under different schemas, all the models consistently generate SQLs with INNER JOIN and fail to produce the correct results. Fixing these incorrect mappings and retraining the models allows them to generate the correct SQLs.

Third, detecting and fixing errors poses a significant challenge due to the intricate semantics of SQL. Despite the authors dedicating 450 man-hours to reviewing all the mappings in Spider [57], a substantial portion of errors persist in the dataset. To assess whether the incorrect mappings are typically syntactically close to the correct ones and are easy to detect and fix, we carefully fix the errors in the sampled mappings and measure the similarity between SQLs before and after fixing. We apply Jaccard similarity [55] that calculates the intersection of the commonly used SQL keywords, tables, and columns. Figure 3 reveals that 53.0% and 40.4% of all the errors identified within Spider and BIRD's 500 training samples exhibit a similarity score below 0.5, signifying dissimilarity between the SQL pairs. The variety of dissimilarity patterns among these errors further complicates the reviewing process for developers.

Real Model Accuracy with Fixed Test Dataset. The test dataset provides the ground truth for assessing Text-to-SQL model accuracy. Since the test dataset represents a small portion of the overall dataset (e.g., 20% in Spider), it is more tractable to undertake a meticulous manual review of the Spider test and BIRD development dataset (BIRD's test dataset is not open-sourced). We carefully review every NL to SQL mapping after SQLDriller has provided its fix to maximize the correctness of the test dataset. This work is completed by 4 senior students who are familiar with SQL language, taking 80 man-hours for each dataset. In total, we have found 810 (37.7%) errors among 2147 mappings in the Spider test dataset and 768 (50.1%) among 1534 mappings in the BIRD development dataset.

We re-evaluate the models on the modified test dataset. As depicted in Table 2[3], the models get an accuracy drop from 5.6% to 23.9% compared with what they originally reported. The consistent accuracy drop is related to fixing incorrect SQL mappings in the original test dataset, which excludes

[3]Original Accuracy of BIRD models is lower than those in the original paper, since we check SQL result equivalence on bag semantics but not set semantics used in BIRD.

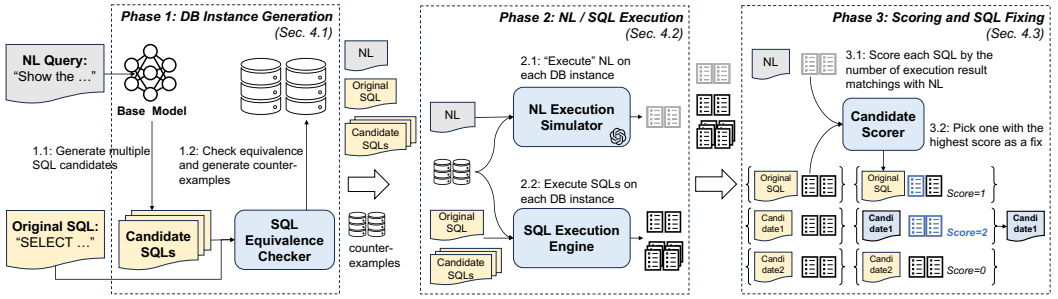

Fig. 4. An overview of SQLDriller workflow.

misjudgments of SQL predictions' correctness. Just like incorrect SQL mappings resembling the example in Figure 1, they accidentally misjudged incorrect SQL predictions with INNER JOIN as correct and those with LEFT JOIN as incorrect. For the predictions of RESDSQL whose accuracy drops the most, there are 103 such misjudgments concerning the JOIN type. We fix all such mappings by modifying INNER JOIN into LEFT JOIN. The predictions with INNER JOIN are thus judged incorrect. For other fixed errors, the comprehensive case study in Section 6.4 can be followed. In this paper, the re-evaluated accuracy results serve as our baselines and optimization targets.

3 System Overview

To improve the quality of Text-to-SQL training datasets, we aim to develop a tool to automatically flag and fix likely incorrect NL to SQL mappings. To achieve that, we first propose a new correctness condition, namely execution consistency. Based on this, we develop an error-detecting and fixing algorithm for Text-to-SQL datasets.

3.1 Execution Consistency

The primary challenge for an error-detecting tool lies in the absence of formal semantics for NL statements. To address this, our approach is to clarify the user's intent for an NL query using concrete database instances. Taking Figure 1 as an example, given a concrete database instance with records of two aircrafts and one flight, the execution result of NL query should return 1 flight for the aircraft "Airbus" and 0 flight for "Boeing", while the SQL execution result only returns the flight count of "Airbus". The inconsistency in their execution result implies that the NL to SQL mapping may contain errors. Based on the above insight, we propose a correctness condition termed **execution consistency** with the following definition:

Definition 1 (***Execution Consistency in Text-to-SQL***)**.** An NL to SQL mapping ($q_{nl} \rightarrow q_{sql}$) satisfies execution consistency under a given database instance if the "execution result" of q_{nl} matches that of q_{sql}.[4]

Based on the definition of execution consistency, we develop a tool designed to automatically flag and fix incorrect NL to SQL mappings, namely SQLDriller.

3.2 Basic Workflow

Detecting and fixing incorrect NL to SQL mappings poses three primary challenges: firstly, how to create database instances that are likely to reveal violations of execution consistency; secondly, how to really "execute" a natural language query on concrete database instances; lastly, how to automatically and efficiently fix the identified incorrect mappings.

[4]In this definition, we consider the NL query does not have ambiguity that may correspond to multiple SQL answers.

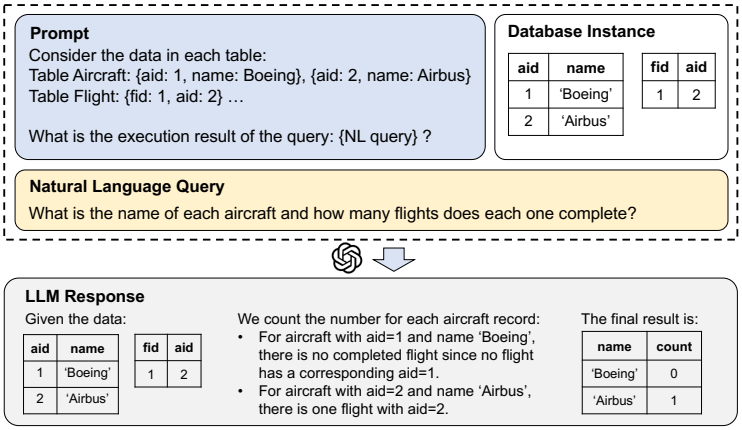

Fig. 5. A example of executing an NL query with LLM.

Firstly, the absence of formal semantics for NL queries presents a challenge in automatically constructing database instances that differentiate between NL and SQL query semantics. Unlike SQL queries with well-defined semantics, NL queries lack a formal and structured representation of semantics, which even leads to ambiguity. For example, the NL query "Show the number of packages sent." cannot determine whether to return a counting number of all packages or the package number of each package. This hinders the systematic construction of database instances where the NL query's intention yields different results from SQL. As a result, no existing tool could precisely construct such database instances. To tackle this, we focus on constructing database instances that differentiate between non-equivalent SQLs predicted by the same NL query instead. SQLDriller employs a SQL equivalence checker to construct such database instances. As the workflow shown in Figure 4, given an NL query, a base Text-to-SQL model is used to generate a set of candidate SQLs that differ from the original SQL. The SQL equivalence checker then checks SQL equivalence and generates counterexamples for non-equivalent SQLs. Counterexamples are concrete database instances that can identify non-equivalent SQLs within the set. Non-equivalent SQLs show different execution results on the counterexamples, thus reflecting semantic distinctions between them. Using such counterexamples, SQLDriller checks execution consistency and detects incorrect NL to SQL mappings.

Secondly, SQLDriller utilizes large language models (LLMs) to "execute" NL queries on concrete database instances. The recent success of LLMs empowers users to directly execute an NL query on a specific database instance through dedicated prompting. Figure 5 shows an example, where it directly integrates the concrete database instance into the prompt context and makes the user's intent for executing the NL query on the instance explicit to ask for LLM's response. According to our micro-benchmark evaluation in Section 4.2, prompting LLMs to execute NL queries achieves remarkable accuracy, up to 91.0% and 83.2% accuracy for Spider and BIRD datasets, respectively. This indicates the promising capability of LLMs in NL query execution. The detailed prompts and micro-benchmark designs will be discussed in Section 4.2.

Lastly, SQLDriller introduces a scoring algorithm to rank the SQL candidates. The candidate with the highest score is deemed likely correct and will be taken as a recommendation. This is because execution consistency is promising to tag the potential incorrect mappings but is not sufficient to judge whether a SQL is correct. An incorrect SQL may also satisfy execution consistency on particular database instances. In the example of Figure 6, the first two SQLs are incorrect but SQL_1 still satisfies execution consistency on the first generated counterexample, which is the same

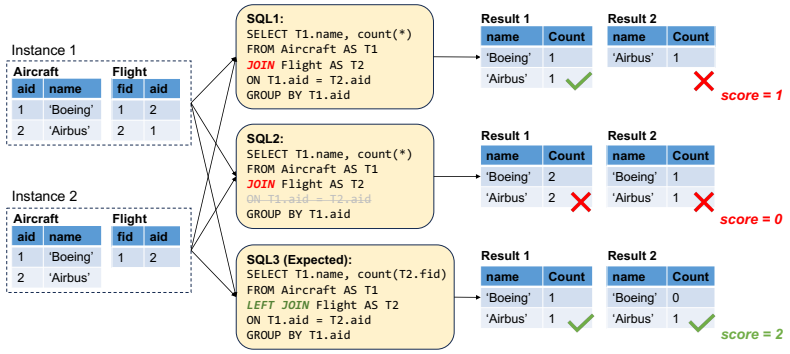

Fig. 6. An example of performing the scoring algorithm on various SQL candidates.

as the expected SQL_3. Therefore, more counterexamples are necessary to further differentiate between the correct and incorrect SQLs. The second counterexample successfully differentiates SQL_3 from other incorrect ones and allows SQLDriller to judge execution results among multiple instances and identify the SQL that is more likely to be correct. As Figure 4 shows, the scoring and ranking mechanism is designed to assign a score to each SQL based on the times its execution results match the NL query. The SQL candidate with the highest score is selected as the final choice.

4 SQLDriller Design

This section describes the workflow of SQLDriller and provides details of its key components.

4.1 Database Instance Generation

To effectively identify incorrect mappings in the dataset, SQLDriller must reveal violations of execution consistency on specific database instances. To make SQLDriller effective, such instances should be prone to triggering the violation if the mapping exactly contains an error. However, the lack of formal semantics for NL queries makes it challenging to directly construct instances to detect the errors. Fortunately, we have observed a key insight to address this issue. For a Text-to-SQL model, asking it to generate multiple SQL predictions that vary in syntax structures increases the chance of producing at least one correct SQL to the NL query. In our experiments on several models [16, 20, 28, 42, 48], when prompted to output multiple predictions as opposed to a single one, we observe an increase from 62.3% to 81.8% in the probability of producing one correct SQL within the set. In this way, the correct SQL differs in its semantics from other predictions (or the original SQL). Since counterexamples serve to distinguish non-equivalent SQLs, they can effectively differentiate and pick the correct SQL from incorrect ones. Therefore, SQLDriller first generates a set of SQL candidates, then uses a SQL equivalence checker to check equivalence and generate counterexamples as database instances to detect errors. These generated SQLs also serve as candidates for automated SQL fixing.

Generating Preliminary SQL Candidates. SQLDriller uses a base Text-to-SQL model to produce a set of initial SQL candidates. These candidates must satisfy two key requirements. First, they should be syntactically and semantically diverse to yield counterexamples for error detection, as well as potentially correct SQL candidates. Second, their semantics should closely align with the natural language query's semantics, avoiding significant deviations.

As LLM-based Text-to-SQL techniques have evolved, we design a dedicated prompt shown in Figure 7 that instructs the model to generate multiple SQL queries satisfying the above requirements. For the first requirement, based on our observation, a potentially correct SQL can differ from the original one in both syntax structures and keywords used. Therefore, we explicitly instruct the

… (Previous prompt contents)
Please provide N candidate SQL queries for the previous question that you believe are the correct answers. These candidate SQL queries are encouraged to **differ greatly from each other in their syntax structures and used keywords**, as long as you believe they are correct corresponding to the previous question.

List the N candidate SQLs one by one in JSON format, which should be like:
{"answers": ["SELECT...", "SELECT...", ..., "SELECT..."] }"

Fig. 7. The added prompt to ask a Text-to-SQL model to generate multiple diverse SQL candidates.

model to produce multiple SQL queries (with a parameter N as the number of candidates) that differ in syntax and keyword usage, as highlighted in the bold text of Figure 7. For the second requirement, we instruct the model to generate SQL statements it deems potentially correct responses to the NL query, in order to mitigate excessive deviation from the natural language query semantics. Finally, we instruct the model to output SQL in a specific JSON format, which simplifies the extraction and collection of the generated SQLs. Furthermore, unlike most Text-to-SQL models that rely on fine-tuning or few-shot prompting, SQLDriller employs a zero-shot model for SQL generation to avoid potential negative impacts of training datasets that may include incorrect mappings and negatively affect the SQL predictions.

Using this prompt design to generate syntactically diverse SQL candidates enables the model to produce semantically different ones as well, enabling counterexample generation for subsequent error detection. For instance, in Figure 6, the SQL candidates syntactically vary in their JOIN types and JOIN conditions and exhibit different semantics. We apply the prompt to several top-ranking models and conduct a preliminary experiment on 500 sampled mappings of Spider and BIRD. When N is 10, the generated candidate set contains an average of 3.2 semantically distinct groups of SQLs for each NL query, with a maximum of up to 8 groups. N is empirically set to 10 since a smaller value yields fewer distinct groups (2.6 when N is 5), and a larger one does not significantly increase the number (3.6 when N is 20) but with substantial time overhead.

Design of a Hybrid Checker. After collecting SQL candidates, SQLDriller creates a set of database instances for execution. It proposes a SQL equivalence checker to check the equivalence between each SQL candidate and the original SQL, then generates counterexamples for non-equivalent pairs as database instances. However, designing the checker is challenging because the capabilities of existing tools [9, 14, 22, 54] cannot satisfy our requirements for equivalence checking and counterexample generation.

There are two main types of SQL equivalence verification tools: full-equivalence and bounded-equivalence verifiers. Full-equivalence verifiers [14, 54] prove whether two SQL queries are equivalent across all possible input database instances. They guarantee that two SQLs are equivalent if the result is *Equivalent*, but they are unable to generate counterexamples when SQLs are non-equivalent due to inherent limitations [14, 54]. In contrast, bounded-equivalence verifiers [9, 22] check equivalence within a specific bound of input table sizes and can provide counterexamples when SQLs are *Non-Equivalent*. However, they are prone to timeouts when evaluating equivalent SQL queries, as they repeatedly increase the bounds to search for counterexamples, which leads to significant verification time costs.

To tackle this challenge, SQLDriller integrates two types of verifiers into its SQL equivalence checker to benefit from their strengths and enhance the verification capability. Specifically, it integrates SQLSolver [14] and VeriEQL [22] which are popular full- and bounded-equivalence verifiers with strong capabilities, respectively. As shown in Figure 8, it begins with SQLSolver to check if two SQL queries are equivalent. If SQLSolver confirms equivalence, the checker immediately classifies them as equivalent. Otherwise, if it returns *Non-Equivalent* or *Unknown*, the equivalence

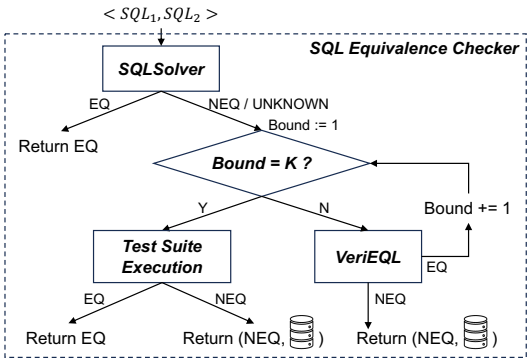

Fig. 8. The workflow of hybrid SQL equivalence checker.

of the SQL queries remains uncertain. At this point, VeriEQL is called to check if the queries are *Non-Equivalent* and try to generate counterexamples. Since it has been observed that counterexamples for most non-equivalent SQLs are typically in a small size [22], VeriEQL sets up the bound from 1 to a value K, which is empirically set as 10 to efficiently search for counterexamples since a larger size can easily incur timeouts. VeriEQL performs a round of verification under each bound size and increments the size by 1 if the SQL pair is verified *Equivalent* in the current round. It terminates when it either finds a counterexample or confirms the pair as *Equivalent* under the maximum bound. Finally, as a backup for timeouts or unreliable verification results from the verifiers, our checker uses the Test Suite Execution [60] technique. This involves running two SQL queries on a series of generated database instances. If the execution results differ on any instance, the SQLs are confirmed non-equivalent, and such instance is returned as a counterexample.

To avoid generating redundant counterexamples and reduce the time cost, each time two SQLs come, the checker first runs them on existing generated counterexamples. If execution results differ, they are directly judged non-equivalent without calling the verifiers.

Making Counterexamples Practical. SQLDriller uses an LLM to simulate NL query executions on generated counterexamples. However, current tools are not tailored to produce counterexamples with realistic and meaningful values, which are essential for accurate LLM execution. Specifically, meaningless or invalid column values may cause the LLM to misunderstand and produce unexpected results, leading to misjudgments in consistency checks.

Figure 9 presents the examples. In the first example, a record has an unrealistic "age" of 2147483648. During NL execution, the LLM dismisses this record as invalid and omits it from the average age calculation, whereas SQL includes it. Given that such a situation should not occur in real-world scenarios, the execution difference between LLM and SQL does not indicate the incorrectness of SQL. In the second example, unusual values of the "name" column cause the LLM to mistakenly treat them as dirty values and exclude them from processing, also leading to unexpected NL execution results. To prevent such issues and capture realistic counterexamples, we add extra "CHECK" constraints for numeric- and string-type columns in the schema definitions to keep the values within a practical range.

```
CREATE TABLE Dogs(
  dog_id INT PRIMARY KEY,
  name VARCHAR,
  age INT,
  CHECK name IN ('Jack', 'Bella', 'Leo', ...),
  CHECK age BETWEEN 0 AND 100
);
```

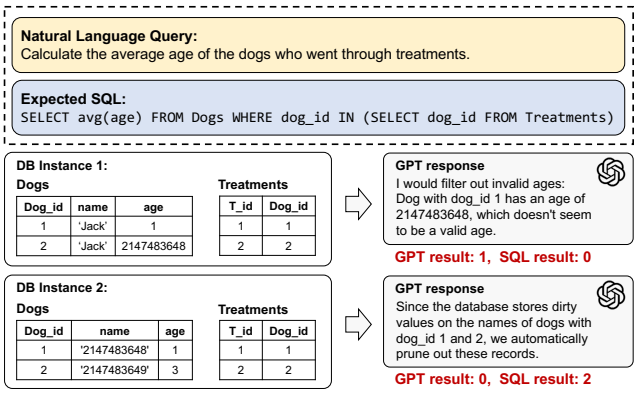

Fig. 9. Examples of the unrealistic counterexamples.

In this example, we constrain dog names to realistic ones and the ages between 0 and 100. For string-type columns, we extract all constant strings from the SQL queries, along with 10 random extra practical values generated by a GPT model based on the table and column names. Numeric-type columns should not contain excessively large values, so we set their range from 0 to 100 by default to provide enough value candidates for counterexample generation. If a SQL predicate contains constant values, the range should cover all these values. We set the range from 0 to $2 \times c$, where c represents the maximal positive value that appeared. Setting it twice helps provide value candidates to handle predicates like "WHERE col > c" for counterexample generation. Likewise, a minimal negative value $c < 0$ sets the range from $2 \times c$ to 0, correspondingly.

4.2 Natural Language Execution

Besides an off-the-shelf SQL execution engine to execute SQLs, SQLDriller develops an "executor" to simulate NL query execution. This NL executor should directly understand the NL query semantics, manipulate data, and produce the final outcome. Fortunately, the tiny size of counterexamples provides an opportunity to leverage LLM's inherent reasoning capability to execute NL queries accurately. The tiny size consumes a compact LLM context window, facilitating LLM's understanding of records in the instance and enabling precise data manipulations. Therefore, SQLDriller instructs the LLM to directly traverse each record in the counterexample and perform query-specific data manipulation to form the final result.

We design a dedicated prompt for the NL query execution task in Figure 10. First, the prompt provide detailed schema information including table names, column names, and integrity constraints (primary keys and foreign keys). It also sequentially lists every record in the tables, presents the NL query, and asks for its execution result on this counterexample. Then, we instruct the LLM to traverse each record and think step-by-step about how to manipulate the data and produce the final result. In this way, the LLM outputs each step of its cognitive process and data manipulations and formulates the output format of the execution result as a list of tuples.

When an LLM executes NL queries on the database instances, it may overlook specific execution rules that fall beyond its knowledge. This leads to unexpected mismatching with SQL's execution results. For instance, the SQL standard specifies that aggregations like SUM, AVG, or MAX/MIN on an empty set should return NULL values. However, the LLM would unexpectedly return 0 by its common sense. To tackle this, we integrate a post-processing step in the prompt that instructs the LLM to adjust the execution results. In Figure 10, we provide specific instructions for the LLM to modify its preliminary results by itself to follow specific execution rules.

```
Now we need to query a database storing the following tables:
Table T1: {id: v1, col2: v2}, {id: v1, col2: v2} …
Table T2: {id: v1, col2: v2, col3: v3}, {id: v1, col2: v2, col3: v3} …
Here, T1.id is the primary key of T1; T2.id is the primary key of T2.

For the natural language query: Show …, what is the execution result on the tables?
Please carefully learn relationships of tables by primary keys and foreign keys, think step-by-step, then analyze and
infer the execution result.
Specifically, you first schedule a step-by-step plan of result construction.
Then choose tables with columns to be output, scan over each record of the table, and perform query-specific data
manipulations to construct the result.

Finally, list out the result. Each record is a Python-type tuple with one or more columns:   List input tables, describe
``` #result: [(record0), (record1), (record2), ...] ```                                      the task, and specify output format

After you have decided the final result, you should analyze and try to fix the execution result for any potential issues.

- Instruction 1: check aggregation
When the query asks for aggregation results (e.g. count, sum, max, min, average), please return a number as the
result rather than some records.
If no record participates in aggregation when calculating an aggregation result, do not return empty set as aggregation
result. Return 0 when calculating count, and return None when calculating sum, max, min or average values.
…   Postprocess the execution result
```

Fig. 10. The simplified prompt of executing an NL query on a database instance.

To assess the accuracy in NL query execution with the prompt design, we construct a micro-benchmark derived from existing datasets. The off-the-shelf LLMs achieve remarkable accuracy of up to 91.0% and 83.2% for Spider's and BIRD's micro-benchmarks, as shown in Table 3. Specifically, we generate 500 random database instances on 100 NL to SQL mappings (5 for each) sampled from our fixed Spider test and BIRD development dataset, respectively. The table size of the instances is limited to 5 to match the size of most generated counterexamples. The evaluated LLMs excel in mappings of various hardness levels. GPT-4-turbo, the most outstanding one, shows 92.2%, 88.3%, 94.4%, and 92.9% accuracy for Spider's easy, medium, hard, and extra levels, and 92.7%, 76.1%, and 60.0% for BIRD's simple, moderate, and challenging levels.

### 4.3 Execution Result Scoring

SQLDriller checks whether an NL to SQL mapping satisfies or violates execution consistency after retrieving execution results of the NL and SQL queries. If execution consistency is violated, SQLDriller tries to pick a potentially better-suited SQL candidate as a fix for the original SQL. The preferred SQL candidate should match the execution results of the NL query on more database instances than the original SQL.

To achieve this, SQLDriller employs a scorer to evaluate each SQL and picks the one with the highest score. Given a set of database instances, the scorer treats the NL query's execution results on them as the oracle. It then checks, for each SQL, how many instances where its execution result is the same as the oracle. The score of a SQL is assigned as the exact number of such instances. If the highest score surpasses that of the original SQL, the corresponding SQL candidate is picked and returned as a fix. Otherwise, SQLDriller preserves the original SQL, as no better-suited candidate is available in the SQL candidate set.

It is possible that multiple SQL candidates score higher than the original SQL and share the same highest score. In this case, more database instances are needed to further differentiate which candidate is better. SQLDriller reruns the previous steps only on these highest-scoring SQL candidates. It calls the SQL equivalence checker to generate counterexamples among them and scores on the execution results of these counterexamples. Finally, a SQL candidate that outperforms the others is picked and returned by SQLDriller.
```

Table 3. LLMs' NL execution accuracy on the micro-benchmarks derived from existing Text-to-SQL datasets.

Model	Spider test (500 instances)	BIRD dev (500 instances)
Claude 3.5 Sonnet	451 (90.2 %)	391 (78.2 %)
GPT-4o	455 (91.0 %)	402 (80.4 %)
o1-preview	444 (88.8 %)	416 (83.2 %)
GPT-4-turbo	455 (91.0 %)	415 (83.0 %)

5 Optimizing Model Inference

Besides fixing errors and improving the dataset quality, SQLDriller can also be utilized to improve Text-to-SQL model inference accuracy. As illustrated in Section 4.1, it has been observed that generating multiple SQL predictions increases the possibility of containing the correct answer within the prediction set. With this observation, precisely selecting the correct SQL within a prediction set generated by the model is the key to accuracy improvements. Since SQLDriller's underlying workflow is to effectively select the best-suited SQL within a candidate set, it can improve model accuracy by selecting the best-suited SQL within multiple model predictions.

To apply SQLDriller for optimization, the paradigm of Text-to-SQL model inference changes from generating one single prediction to "brainstorming" and generating multiple SQL predictions given an NL query. The prompt design in Figure 7 can be applied to the models to achieve that. After collecting a set of SQL predictions, SQLDriller employs execution consistency and performs its scoring algorithm on them. Specifically, since the original ground truth SQL is not available in the scenario of model inference, SQLDriller checks equivalence and generates counterexamples only among the generated SQL predictions. It then executes the NL query on these counterexamples and scores each SQL prediction based on the number of consistent execution results with the NL query. Finally, the highest-score SQL is selected to be the model's final prediction result. As evaluated in Section 6.2, this optimization consistently improves all the models' accuracy, by up to 9.2%.

6 Evaluation

We aim to answer the following questions in our evaluation:
Q1. How much accuracy improvement can the repaired datasets bring to Text-to-SQL models?
Q2. How effective is SQLDriller in detecting and fixing incorrect NL to SQL mappings in existing Text-to-SQL datasets?
Q3. What lessons can we learn from incorrect mappings on NL to SQL translation?

6.1 Setup

Datasets. We evaluate on the popular Text-to-SQL datasets, Spider [57] and BIRD [30]. They both have a training set of over 8,000 mappings and a development set of over 1000 mappings. Spider has open-sourced its 2147 test mappings. We view BIRD's development set as the test set since its real test set is not open-sourced.
Models. We evaluate accuracy on these models: DAIL-SQL [20], DIN-SQL [42], RESDSQL [27] and Graphix-T5 [29] for Spider, SFT CODES and CODES [28] for BIRD. These models are the highest-ranked open-sourced ones on the leaderboards. DAIL-SQL uses NL similarity to pick training mappings as examples for few-shot prompting [20]. DIN-SQL is also few-shot prompted, with its examples statically encoded and unchanged for each prediction [42]. We directly fix incorrect examples in DIN-SQL's prompt for evaluation. CODES applies few-shot prompting after fine-tuning

a model. RESDSQL [27], Graphix-T5 [29] and SFT CODES [28] fine-tune pre-trained models with the training mappings. RESDSQL's fine-tuned T5-3b, Graphix-T5's fine-tuned T5-large, and SFT CODES's 3b models are evaluated. Prompting-based models can generate multiple SQLs by adding the prompt in Figure 7 to their prompt template. RESDSQL and CODES output a sequence of SQL predictions during inference [27, 28], but lack support on such promptings. So we directly collect all the SQL predictions during their inference. C3 [16] and CHESS [48] are not evaluated since 0-shot prompted models do not access training data.

Accuracy Metric. We use Test Suite Accuracy metric [60] to evaluate accuracy. This metric checks whether the predicted and ground truth SQL execute consistently on a large set of database instances [60]. It has much fewer false positives [60] than Execution Accuracy metric, which executes the SQLs on only one given instance. Exact Match Accuracy [57] checks SQL syntax structure for equivalence. Its strict equivalence condition is prone to false negatives [60] that judge equivalent SQLs as non-equivalent. So we do not apply it.

Baseline. We compare SQLDriller with a baseline of LLM consistency technique on accuracy improvement and error fixing. The baseline selects the most "confident" SQL among the candidates using prompt-agreement confidence estimation in [41], which evaluates each response's LLM log probability across diverse prompts. We use GPT-3 to generate 100 diverse prompt templates that ask for selecting one of the SQL candidates given the NL query. We randomly pick 20 templates for each Text-to-SQL mapping, encode the NL query and SQL candidates into each template, and call GPT-4 to select and return the selection's log probability. The SQL with the highest average log probability in 20 LLM calls is finally selected.

Implementation Details. SQLDriller uses a base model to generate the SQL candidate set given an NL. We apply the prompt design of C3 [16] and CHESS [48] for Spider and BIRD respectively, since they rank first among 0-shot prompted models in the leaderboards. We add the prompt in Figure 7 to their prompt templates. For NL execution, we call GPT-4 for its high accuracy and affordable price. We call 5 times for each database instance and pick the majority of these execution results to return. We use SQLite v3.45.1 [11] as the SQL execution engine.

Testbed. We fine-tune the models on a server with 8 NVIDIA A100 (80G) GPUs, 1 Intel(R) Xeon(R) Gold 6348 CPU, 1 TB memory, CentOS Linux 7 operating system.

6.2 Accuracy Improvement

We fine-tune the models with the original and fixed datasets and evaluate their accuracy. All the training mappings are automatically checked and fixed by SQLDriller. For rare cases with NL ambiguity (<2% in Spider and BIRD) that SQLDriller gives another SQL as a fix, we keep the original SQL as the answer in accuracy evaluation.

As shown in Table 4, SQLDriller brings accuracy improvements ranging from 3.6% to 13.6% to these models. Specifically, 4 models achieve substantial improvements of more than 10.0%, where RESDSQL and SFT CODES achieve the highest in Spider and BIRD, up to 13.6% and 11.2%, respectively. According to the analysis in Figure 11, RESDSQL and SFT CODES's improvements mainly come from the fixed higher-quality training dataset, which brings 7.2% and 6.5% improvements to them. We attribute this to the fact that fine-tuned models are more susceptible to the quality of training data. Besides, DAIL-SQL and DIN-SQL gain comparable improvements of 12.9% and 12.3%, respectively. The improvements to these prompting models (using GPT-4) mainly come from the optimization of execution consistency (9.2% of 12.9% and 9.1% of 12.3% in Figure 11). It reveals that such GPT-based prompting models benefit from the prompt that asks for generating multiple diverse SQL predictions, substantially increasing the possibility of containing correct SQLs within the set. So they gain more accuracy improvements than RESDSQL and SFT CODES from execution consistency. The prompting CODES model has a 6.1% improvement with 2.8% from the

Table 4. The evaluation of Text-to-SQL model accuracy improvements brought by SQLDriller. For BIRD, we use levels that are manually annotated by its developers: "Simple", "Moderate", and "Challenging", which are different from those of Spider.

Model	Dataset	Training Scheme	Accuracy Improvement (%)				
			All	Easy / Simple	Medium / Moderate	Hard / Challenging	Extra
DAIL-SQL (5 shots)	Spider test	Prompting	+ 12.9 (61.8 → 74.7)	− 0.4 (90.2 → 89.8)	+ 9.4 (68.5 → 77.9)	+ 24.8 (41.5 → 66.3)	+ 23.5 (34.5 → 58.0)
DIN-SQL	Spider test	Prompting	+ 12.3 (61.3 → 73.6)	− 1.1 (87.9 → 86.8)	+ 9.3 (66.9 → 76.2)	+ 24.8 (41.9 → 66.7)	+ 20.2 (38.9 → 59.1)
RESDSQL (T5-3b)	Spider test	Fine-tune	+ 13.6 (52.3 → 65.9)	+ 1.9 (86.0 → 87.9)	+ 8.4 (60.3 → 68.7)	+ 29.2 (28.3 → 57.5)	+ 21.3 (19.6 → 40.9)
Graphix-T5 (T5-large)	Spider test	Fine-tune	+ 3.6 (46.3 → 49.9)	+ 0.6 (79.8 → 80.4)	+ 3.8 (53.6 → 57.4)	+ 3.9 (22.9 → 26.8)	+ 3.4 (15.4 → 18.8)
SFT CODES (3b)	BIRD dev	Fine-tune	+ 11.2 (38.1 → 49.3)	+ 12.1 (47.0 → 59.1)	+ 11.8 (26.2 → 38.0)	+ 4.9 (22.2 → 27.1)	/
CODES (7b, 5 shots)	BIRD dev	Prompting	+ 6.1 (31.4 → 37.5)	+ 6.7 (40.7 → 47.4)	+ 6.3 (19.2 → 25.5)	+ 2.8 (13.2 → 16.0)	/

optimization of execution consistency, relatively lower than that of DAIL-SQL and DIN-SQL. We attribute this to the stronger capability of GPT-4 than the base language model used by CODES and the greater difficulty of Text-to-SQL mappings in BIRD than Spider. Graphix-T5's modest 3.6% improvement is due to our decision not to apply execution consistency's optimization, as it lacks inherent support for outputting multiple SQL predictions in inference. To conclude, fixing SQL mappings to improve the dataset quality and optimization of execution consistency bring substantial accuracy improvements to models despite their diverse model designs.

SQLDriller brings accuracy improvements to varied Text-to-SQL hardness levels. As shown in Table 4, the improvement for Spider's hard and extra-hard mappings reaches over 20%. Easy mappings do not gain that much improvement since the evaluated models have effectively tackled these mappings and achieved high accuracy. For BIRD, SQLDriller consistently improves accuracy for simple and moderate mappings (>11% for SFT CODES and >6% for CODES). It achieves moderate improvement for challenging ones due to their inherent challenges of semantic understanding and SQL query translation to the models.

Execution Consistency vs. LLM Consistency. We compare execution consistency with LLM consistency on their impact on accuracy improvements. Figure 11 shows that execution consistency brings higher improvements than LLM consistency to all the models. LLM consistency even negatively impacts 3 models, with an accuracy drop of up to 0.6% on DIN-SQL. We attribute LLM consistency's weakness to its lack of awareness of SQL's subtle execution semantics. For example, among the SQL candidates in Figure 1, LLM consistency selects the one with INNER JOIN, which is its most "confident" option based on the log probability of LLM inference. It merely decides based on the textual statements of SQL and fails to delve into SQL execution semantics. Execution consistency, instead, selects based on SQLs' execution results on concrete database instances. It explicitly reveals the subtle semantic difference between LEFT JOIN and INNER JOIN by presenting a database instance with an aircraft record having no completed flight and revealing the necessity of LEFT JOIN to preserve such record. Execution consistency thus brings more accurate semantic matching between NL and SQL queries and gains higher accuracy improvements.

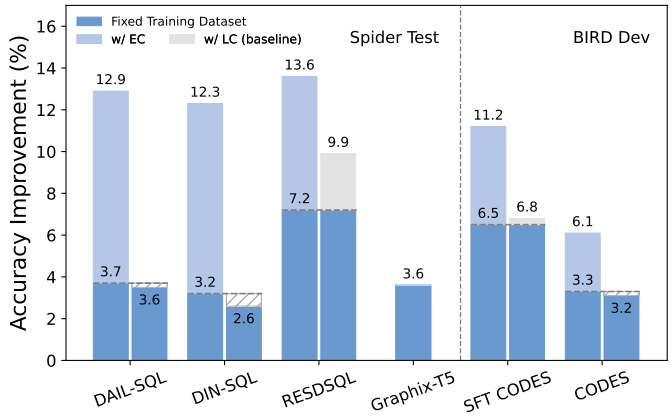

Fig. 11. Accuracy improvements analysis. "w/ EC" and "w/ LC" mean using Execution Consistency and LLM Consistency, respectively. Negative impacts of LLM Consistency are depicted by grey diagonal lines. Graphix-T5 does not apply them for accuracy improvement since it cannot output multiple SQLs in inference.

Impact of Ambiguity. Ambiguous NL queries in Spider and BIRD may inevitably exist, but these constitute a minority. 25 and 33 mappings in the Spider test and BIRD development dataset are detected to have ambiguity. To assess the impact of ambiguous mappings on benchmarking, we consider both the original and fixed SQL queries as ground truth answers to re-evaluate the accuracy. The models' original and improved accuracy in Table 4 all increase by less than 0.5% and 0.9% for Spider and BIRD datasets, respectively, exhibiting limited impact on the accuracy outcomes. Hence, the conclusion that SQLDriller brings significant accuracy improvements still holds. For ambiguous mappings in training sets, we also preserve the original SQL. Actually, mapping the NL to any correct SQL does not notably degrade the dataset quality due to their low proportion.

6.3 Detecting and Fixing Dataset Errors

This section analyzes SQLDriller's effectiveness of using execution consistency to automatically detect and fix incorrect NL to SQL mappings. To do this, we manually review the sampled 500 training mappings in Spider and BIRD's training dataset, providing ground truth to check whether detections and fixes are correct or not.

Detecting Errors. Table 5 shows the results. Among the 183 and 272 incorrect mappings of Spider and BIRD, 149 (81.4%) and 207 (76.1%) violate execution consistency on at least one database instance and are detected by SQLDriller. Despite a few false negatives (34 and 48 for Spider and BIRD), the high proportion of detected errors (high true positive rate) reveals that execution consistency can effectively detect incorrect NL to SQL mappings. On the other hand, SQLDriller inevitably introduces a few false positives. For Spider and BIRD, 282 of 317 and 200 of 228 correct mappings satisfy execution consistency on the generated counterexamples. The remaining ones violate execution consistency, showing a false positive rate of 11.0% and 12.3% for Spider and BIRD, respectively. We consider the low false positive rate acceptable compared to the high true positive rate of execution consistency for error detection.

We dig into the reasons for these misjudgments. Of the 34 false negatives in Spider that overlook the error, 19 lack necessary counterexamples and fail to trigger the violation of execution consistency. The other 15 mappings stem from inaccurate NL executions of LLM that accidentally match the incorrect SQL's execution results and claim compliance with execution consistency. Among 65 BIRD's false negatives, 45 lack counterexamples and 20 suffer from inaccurate NL

Table 5. Analysis of error detection. "# Incorrect / Correct" represents manual judgments of the mapping's correctness. "# Inconsistent / Consistent" means violating or satisfying execution consistency.

Dataset	# Incorrect	# Inconsistent	# Correct	# Consistent
Spider train	183	149 (81.4%)	317	282 (89.0%)
BIRD train	272	207 (76.1%)	228	200 (87.7%)

executions. On the other hand, the 35 and 28 false positives that misjudge correct mappings as violating execution consistency are also caused by inaccurate NL executions. To conclude, the main cause of misjudgments lies in both the lack of counterexamples (mostly due to lacking correct SQL candidates generated) and inaccurate LLM-based NL executions. Thus, enhancing the SQL candidate generation capability along with LLM's NL execution accuracy is regarded as subsequent optimization pathways for SQLDriller.

Automated SQL Fixing. SQLDriller tries to fix the SQL for each NL to SQL mapping that violates execution consistency. Table 6 shows the results of automated SQL fixing on Spider and BIRD's 500 samples. For Spider, SQLDriller fixes 138 of 183 incorrect mappings, among which 110 (60.1%) are fixed to be completely correct. The remaining 28 are partially or wrongly fixed. For BIRD, it fixes 185 of 272 incorrect mappings, where 142 (52.2%) are completely fixed. On the other hand, for 317 and 228 correct mappings in Spider and BIRD, SQLDriller preserves the original SQL for 309 (97.5%) and 218 (95.6%) mappings, respectively. It introduces a low false positive rate of 2.5% and 4.4% on SQL fixing for Spider and BIRD, which are significantly lower than the corresponding true positive rates. This contributes to SQLDriller's effect of improving the quality of datasets by fixing incorrect mappings while introducing few false positives, thus bringing model accuracy improvements illustrated in Section 6.2.

We analyze the root cause of the misjudgments. Among 73 incorrect Spider mappings that are not correctly fixed, 41 lack correct SQL candidates within the set to pick, which is limited by the capability of the model used to generate multiple SQLs. In the remaining 32 mappings, the correct SQLs are present within the set. 11 fail to generate counterexamples to differentiate between the original and the correct SQL due to the limited checking capability of the SQL equivalence checker. 21 result from inaccurate LLM NL executions that hinder SQLDriller from scoring the highest to the correct SQL. For 130 unfixed BIRD mappings, 79 lack correct candidates, 15 lack counterexamples to differentiate, and 36 for inaccurate NL executions. Inaccurate NL execution is also the root cause of the 8 and 10 false positives that mistakenly fix the original correct SQL.

Compared to error detection, the SQL fixing task exhibits a lower false positive rate (<5% v.s. >10%) along with a lower true positive rate (≈60% v.s. ≈80%). We attribute this to the scoring mechanism in SQLDriller. There are cases where the NL to SQL mapping violates execution consistency but the SQL remains unfixed. When it occurs on correct mappings, such violations stem from inaccurate NL executions, causing SQL candidates to execute differently from the NL query and lose corresponding scores. It prevents SQL miscorrections and thus contributes to a lower false positive rate of ≈ 8%. As to incorrect mappings, the SQL candidates fail to surpass the original SQL's score due to the absence of a better-suited SQL. Correct fixes are not provided on such detected incorrect mappings, leading to SQL fixing's lower true positive rate of ≈ 20%. To conclude, SQLDriller cannot fix all its detected errors, but its scoring mechanism mitigates the introduction of SQL miscorrections.

Execution Consistency vs. LLM Consistency. We compare execution consistency with LLM consistency baseline on their effectiveness of SQL fixing. As shown in Table 6, execution consistency

Table 6. Analysis of SQL fixing. "# Incorrect / Correct" represents manual judgments of the mapping's correctness. "# Correctly Fixed / Remain Unfixed (EC / LC)" are SQLDriller (i.e., Execution Consistency) or LLM Consistency baseline's choice of correctly fixing it with another SQL or remaining the original SQL.

Dataset	# Incorrect	# Correctly Fixed (EC)	# Correctly Fixed (LC)	# Correct	# Remain Unfixed (EC)	# Remain Unfixed (LC)
Spider train	183	110 (60.1%)	42 (23.0%)	317	309 (97.5%)	283 (89.3%)
BIRD train	272	142 (52.2%)	78 (28.7%)	228	218 (95.6%)	162 (71.1%)

outperforms LLM consistency on its higher true positive rate and lower false positive rate in both Spider and BIRD. It successfully fixes 37.1% and 23.5% more incorrect mappings than LLM consistency, while also introducing 8.2% and 24.5% fewer miscorrections. This also stems from execution consistency's capability to delve into SQL execution semantics by executing concrete database instances, as illustrated in the baseline comparison in Section 6.2. It thus provides more accurate SQL fixes, such as effectively picking the SQL candidate with LEFT JOIN in Figure 1 as the fix, while applying LLM consistency remains the original SQL with INNER JOIN unfixed. Therefore, execution consistency exhibits stronger effectiveness in error fixing, which contributes to higher-quality Text-to-SQL datasets.

6.4 Analysis and Lessons

In this section, we provide an analysis of the detected incorrect NL to SQL mappings, followed by a case study as lessons to Text-to-SQL developments.

Distribution. We analyze the distribution of all 2393 Spider training mappings that violate execution consistency across their different NL and SQL complexities. In this way, we try to understand the potential characteristics of NL and SQL that are prone to errors.

Figure 12 shows the trend between consistency violation rate and mapping complexity (measured by number of tokens in NL and SQL query). Each point represents the violation rate of mappings with a token number between the previous X-axis scale and the current one (each interval range of the X-axis scale is 3). It shows that the violation rate of NL to SQL mappings arises as the number of tokens in either the NL or the SQL query increases. The violation rate of mappings exceeding 30 tokens in NL or SQL query (the point of the largest X-axis value) rises to an extremely high value, over 0.6 and 0.4, respectively. It reveals that developers tend to make mistakes on more complex NL to SQL mappings. A longer NL query is harder to understand, leading to challenges in writing the SQL. A longer SQL is prone to errors since it typically consists of multiple sub-clauses to write. Developers have to consider the correctness of not only all the sub-clauses but also the whole SQL query made up of each sub-clause.

Case Study on Errors. The incorrect Text-to-SQL mappings exhibit various error patterns. We present a summary of our detected common patterns as lessons for developers in future developments, as shown in Table 7.

***Pattern 1**: Improper use of JOIN causes records omitted.* An NL query asks for aggregate results for each entity record (record with a primary key in a table) should include all records, and omitting any record misleads the user that these records do not exist. In the first example of Table 7, a teacher with no course should not be excluded by the join condition. Instead, his/her number of courses should be 0 to return. The improper INNER JOIN accidentally omits such teacher records and can be fixed by using LEFT JOIN instead, which is the same as the example in Figure 1. SQLDriller provides another fix in this case: a scalar sub-query on the SELECT clause counting the number of

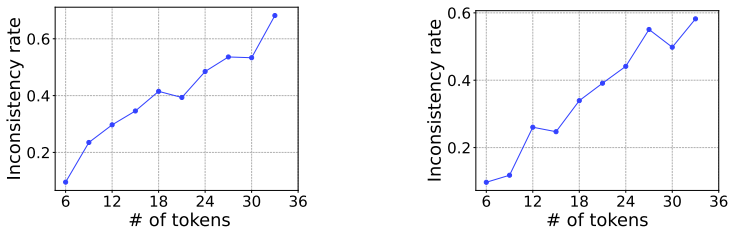

Fig. 12. Relationships between the Execution Consistency violation rate and the mapping complexity (# of tokens in NL and SQL).

courses for each teacher. Such error occurs not only on SQLs with JOIN but also on those with a single table that represents an *N-to-N* relationship of two entities.

Pattern 2*: Improper duplication or deduplication.* The records returned to the user should be properly displayed without redundant duplication or deduplication. When an NL query asks for some attributes of each entity record, the returned record of attributes should correspond one-to-one with all the entity records. An intuitive example is using SQL "SELECT fname, age FROM student" to answer the NL query "Show the first name and age of each student". However, in the second example of Table 7, an error of duplication occurs. An INNER JOIN is performed on table "Student" with another table "Has_Pet" representing an *N-to-N* relationship between tables "Student" and "Pets". This JOIN operation unexpectedly returns multiple duplicated records with the same student's first name and age. For instance, if a 20-year-old student "Bob" owns ten pets, the record ("Bob", 20) is presented ten times in the result set. Simply adding a DISTINCT keyword for deduplication could exclude duplicated combinations of first name and age from different students, which unexpectedly misses information about some students owning pets. Such results are improper and could mislead users or downstream applications about which students or *how many* students own pets due to redundant or missing records. To solve it, an IN sub-query in the predicate should be used to properly return the required records.

Pattern 3*: Improper attribute selection on Aggregation and Set operations.* A primary key uniquely identifies one record of the entity represented by the table, while other attributes cannot serve as a unique identifier. However, some mappings improperly use non-key attributes for SQL operators like set operations and aggregation. The first and third examples in Table 7 illustrate such an issue. Using the non-key attribute "name" on GROUP BY or EXCEPT operator, they overlook a special condition that records in the operated table share the same "name". This leads to incorrect execution results of the counts or set differences and fails to meet the requirement of NL queries. The first example incorrectly calculates counts from different teachers with the same name, while the third one unexpectedly excludes employees who lack a certain certificate but share the same name with others having that certificate. Hence, the primary key should be used on GROUP BY or replace EXCEPT with a NOT IN sub-query in the predicate to correctly answer the NL query.

Pattern 4*: Incomplete Top-K selection.* The problem of SQL that retrieves records with the maximal or minimal value of an attribute has been widely discussed [30, 57, 59]. Both operators LIMIT 1 and RANK OVER() = 1 are originally considered acceptable. However, if a column has multiple maximal/minimal values, the operator LIMIT 1 may choose different records in different situations. For the original SQL of the fourth example in Table 7, it returns a status with the largest number of competitions. If using another column "c.city_id" on the Equi-JOIN condition as the GROUP BY key, the total order of the records is different, and another status with the same largest number will be returned. Using LIMIT 1 to return only one of the statuses does not meet the requirement and makes the user unaware of the existence of multiple records with the same maximal/minimal

Table 7. The selected examples of common incorrect Text-to-SQL translations.

Natural Language Query	Original SQL	Corrected SQL	Description
What are the names of the teachers and how many courses do they teach?	`SELECT t.name, COUNT(*) FROM course_arrange c JOIN teacher t ON c.teacher_id = t.id GROUP BY t.name`	`SELECT t.name, (SELECT COUNT(*) FROM course_arrange WHERE teacher_id = t.id) FROM teacher t`	1. Improper use of JOIN causes record omitted 2. Improper GROUP BY attributes for Aggregation
Find the first name and age of students who have a pet.	`SELECT s.fname, s.age FROM student s JOIN has_pet h ON s.stu_id = h.stu_id`	`SELECT fname, age FROM student WHERE stu_id IN (SELECT stu_id FROM has_pet)`	Improper duplication or deduplication
Show names for all employees who do not have certificate of 'Boeing 737'.	`SELECT name FROM employee EXCEPT SELECT e.name FROM employee e JOIN certificate c ON e.eid = c.eid JOIN aircraft a ON a.aid = c.aid WHERE a.name = 'Boeing 737'`	`SELECT name FROM employee WHERE eid NOT IN (SELECT eid FROM certificate WHERE aid IN (SELECT aid FROM aircraft WHERE name = 'Boeing 737'))`	Improper attribute selection on Set operations
Show the status of the city that has hosted the greatest number of competitions.	`SELECT c.status FROM city c JOIN farm_competition fc ON c.city_id = fc.host_city_id GROUP BY fc.host_city_id ORDER BY COUNT(*) DESC LIMIT 1`	`SELECT status FROM ( SELECT c.status, RANK() OVER(ORDER BY COUNT(*) DESC) AS rk FROM city c JOIN farm_competition fc ON c.city_id = fc.host_city_id GROUP BY fc.host_city_id ) AS ranked WHERE rk = 1`	Incomplete top-K selection
Find the ID and cell phone of the professionals who operate two or more types of treatments. ('treatments.treatment_type_code' is an attribute that identifies a unique treatment type)	`SELECT p.prof_id, p.cell_number FROM professionals p JOIN treatments t ON p.prof_id = t.prof_id GROUP BY p.prof_id HAVING COUNT(*) >= 2`	`SELECT p.prof_id, p.cell_number FROM professionals p JOIN treatments t ON p.prof_id = t.prof_id GROUP BY p.prof_id HAVING COUNT(DISTINCT t.treatment_type_code) >= 2`	Domain-specific natural language query misunderstanding

values. Thus, all the statuses with the most competitions should be returned by using the RANK OVER() operator. It is worth noting that the incomplete selection issue is not limited to Top-1 selections, but includes arbitrary Top-K selections (LIMIT K) as well.

Other Patterns*: Domain-specific query misunderstanding.* In addition to the above patterns, a significant portion of mappings exhibit domain-specific query misunderstanding and semantic mismatching in their SQLs. Such misunderstandings lead to diverse incorrect SQL patterns, including incorrect table scans and joins, selected columns, aggregate functions, expressions in predicates, etc. Take the last case in Table 7 as an example, the NL query asks for "two or more types of treatments" but the SQL only counts the number of treatments regardless of their types. A corrected SQL modifies the parameter of the COUNT function to the column treatment.treatment_type_code and counts distinct values of this column. This example illustrates that a subtle difference in the SQL can lead to incorrectness and a wrong answer to the NL query.

7 Discussion

In this section, we provide a discussion on potential directions for the optimizations and applications of SQLDriller.

Ambiguity in Text-to-SQL. Natural language's ambiguity causes diverse interpretations of NL queries, leading to one NL query being mapped to multiple SQL queries. However, state-of-the-art datasets Spider and BIRD follow a one-to-one mapping between NL and SQL queries without accounting for NL ambiguity by default [30, 57]. While a few ambiguous cases exist, they constitute a minority and have limited impact on our evaluation, as shown in Section 6.2. Our system, SQLDriller, is also primarily designed to detect errors that do not stem from NL ambiguity, such as the examples in the case study in Section 6.4. In SQLDriller, ambiguous mappings are flagged as errors to return: it identifies them by selecting one of the interpretations of the NL query to execute and triggering the violation of execution consistency between NL and SQL queries. If the LLM's NL execution does not select other interpretations, potential ambiguity may not be effectively detected. For future datasets with more ambiguous contexts, we believe an advanced Text-to-SQL correctness condition should be proposed. What's more, enhancing the LLM's capability of exploring more interpretations to execute the NL query remains a compelling avenue for future research.

Scaling to Enterprise Dataset. It is interesting to scale NL execution in SQLDriller to real-world enterprise datasets, which mainly involves two aspects: schema complexity (number of tables and columns and their relationships) and table size (number of records). NL execution scales better than SQL prediction in schema complexity. We test on BEAVER [7], a benchmark that open-sources 48 mappings of an enterprise dataset with 99 tables and 16 columns for each table on average. The NL execution accuracy of GPT-4 reaches 42.1% on the generated random database instances with a bounded table size of 5 records. While not notably high compared to that of Spider and BIRD in Table 3, it exceeds 12.1% compared to predicting SQL using the top-ranking model [48] to execute instances (30.0%). Incorrect NL executions stem from a lack of comprehensive understanding of domain-specific knowledge and complex relationships among tables and columns. For further improvement, specific techniques can be applied to the NL executor, such as calculating the embedding similarity between NL query and table or column names to precisely select tables and columns to operate [48].

On the other hand, NL execution does not scale well as the number of table records increases. When the table size is increased to 100, the NL execution accuracy drops to 17.5% due to a longer context for LLM to understand. BEAVER's originally given database, which has over 10 thousand records in the tables, even exceeds the context window size of GPT models. Thus, the long LLM context bottlenecks NL execution accuracy when dealing with large database instances, which is left as part of our future work to solve. In SQLDriller, NL execution remains effective thanks to the inherent tiny size of counterexamples between non-equivalent SQLs.

Adopting LLM Consistency Checking. LLM consistency checking is a well-studied technique in the NLP domain [3, 17, 41, 53, 62]. It optimizes model inference by encouraging consistent responses across perturbations of input prompts, aiming to output reliable and accurate predictions. Although not as effective as execution consistency in revealing semantic differences and detecting errors, we believe it presents opportunities to optimize NL execution in SQLDriller, by picking the most reliable result it evaluates. Adopting the simple self-consistency [53] is feasible, by repeatedly calling LLM with the same prompt and picking one from multiple results via majority voting. Advanced consistency-checking techniques automatically generate multiple prompt variants of the same intent and use advanced metrics to evaluate LLM responses (e.g., average log probability of LLM) [3, 41, 62]. However, they are specifically designed for multiple-choice questions, and applying them to NL execution presents challenges: first, for the dedicated prompt design of NL

execution, it is hard to automatically generate prompt variants and guarantee their effectiveness; second, the metrics are designed for multiple-choice questions with pre-determined answer choices but do not show effectiveness on open-ended questions [41, 62] like NL execution. Restructuring diverse NL execution results into a multiple-choice question and applying the techniques achieves comparable accuracy to the original result while incurring more complexity and overhead. Thus, developing effective consistency-checking techniques targeting NL execution results becomes the main challenge and is left as part of future work.

Exploring Broader Applications. Besides fixing errors in existing datasets, SQLDriller has the potential to facilitate training data generation by automatically generating and validating SQL mappings given NL queries. It effectively saves manual efforts but falls slightly short in fine-tuning models compared to fixing errors in existing datasets. The quality of generated training data is constrained by the inherent prediction capability of the model used to generate SQL candidates. There are cases where the model fails to generate correct SQL in the candidate set, leading to inevitable errors in the training dataset. We test by using SQLDriller to generate and validate SQL mappings and retraining the models. The generated Spider and BIRD training dataset has 73.0% and 58.0% correct mappings in our 500 samples respectively, lower than the proportion achieved by fixing existing datasets: 83.8% and 72.0%[5] for Spider and BIRD in Table 6. Retrained models naturally show lower accuracy than those achieved by fixing existing datasets (dark-blue segments in Figure 11), with SFT CODES having the greatest drop of 1.7%, from 44.6% to 42.9%. Nevertheless, we believe SQLDriller holds promise in training data generation scenarios, such as automatically validating human-crafted SQLs in crowdsourcing.

Exploring potential applications of SQLDriller to other tabular data scenarios is also interesting. In Table QA and fact-checking scenarios, existing works typically design and prompt an LLM-based agent to schedule a plan across data retrieval, manipulation, and intricate reasoning [58, 66]. Throughout their workflow, some sub-steps could involve generating SQL queries to retrieve essential data from the given tables [58, 66]. SQLDriller can help generate accurate SQL queries given specific contexts and precisely retrieve data in the tables with large data volumes for subsequent reasoning and answer generation.

8 Related Work

Text-to-SQL. Text-to-SQL tasks are typically accomplished using large language models [13, 34, 37, 44, 49]. These models undergo supervised fine-tuning on a Text-to-SQL dataset [27–29, 47, 51]. LLM also allows direct prompt engineering without fine-tuning. DAIL-SQL [20], DIN-SQL [42] and CODES [28] are few-shot prompted, which pick NL to SQL examples for LLM in-context learning. Chain-of-Thought reasoning is also adopted in [16, 42, 48], decomposing Text-to-SQL into sub-steps (e.g., schema linking, query classifying, and SQL generation) to perform. DB-GPT [65] automatically generates effective input LLM prompts by generating and ranking prompt candidates with a customized scoring function. These works focus on model designs or training schemes, while we target the underlying dataset quality, which is orthogonal to them.

Text-to-SQL Dataset. Several Text-to-SQL datasets are created for model development. Earlier datasets [19, 24, 26, 56, 61] suffer from low SQL complexity and domain diversity. Spider [57] and BIRD [30] are state-of-the-art cross-domain datasets with more complex and diverse NL to SQL mappings. They are manually crafted following several rounds of manual reviewing. BEAVER [7] and QATCH [38] target proprietary enterprise databases with a large number of tables and columns. AmbiQT [5] and AMBROSIA [46] are datasets specifically evaluating ambiguous NL queries. [33]

[5]It includes both incorrect mappings that are correctly fixed and correct ones that remain unfixed in Table 6. E.g., 309 plus 110 out of 500 mappings in Spider (83.8%).

demonstrates the challenge of obtaining high-quality training data due to noisy manual annotations. It also provides a Text-to-SQL error analysis taxonomy that targets error localization and cause.

SQL Equivalence Verifying. Developing automated tools to verify SQL equivalence is a well-studied problem. There are mainly two types of such verification tools. Full-equivalence verifiers [8, 14, 54, 63, 64] check whether SQLs are equivalent under any input tables. Bounded-equivalence ones [9, 22] check the equivalence under a bounded table size. They can provide counterexamples for non-equivalent SQL pairs. SQLDriller hybrids the verifiers to maximize the capability of SQL equivalence checking and provide reliable outputs.

LLM Consistency Checking. LLM consistency checking is a well-studied problem in the NLP field. Self-consistency [53] is applied to Text-to-SQL model inference [16] by picking one from multiple SQL predictions via majority voting. [3, 41, 62] propose advanced consistency checking techniques with prompt consistency, which encourages consistent model behavior across various prompts. They provide potential optimizations for more accurate model responses.

Applications for Tabular Data. Besides Text-to-SQL, applications for tabular data like Table QA and fact-checking are emerging. ReAcTable [58] and AutoTQA [66] design LLM-based agents to schedule a plan to retrieve data from the given tables, perform analysis, and generate the answer. Table-GPT [31] targets fine-tuning an LLM to understand tabular and columnar text better. [35] mentions using visual language models to process visual tabular inputs.

9 Conclusion

This paper develops a tool to automatically flag and fix likely incorrect NL to SQL mappings in Text-to-SQL datasets and improve model accuracy. We first introduce execution consistency, a new correctness condition for NL to SQL mapping, to facilitate checking. Based on this, we develop SQLDriller to flag likely incorrect NL to SQL mappings and provide a potential SQL fix. The evaluation shows that SQLDriller effectively detects and fixes incorrect mappings in popular datasets and improve the model accuracy by up to 13.6%.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable comments. This work is supported by the National Natural Science Foundation of China (No. 62422209, 62132014, and 62272304) and the National Science Foundation under Grant No. 2220407. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Alibaba DAMO Academy. 2024. BIRD Website Homepage. https://bird-bench.github.io/.

[2] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases - an introduction. *Nat. Lang. Eng.* 1, 1 (1995), 29–81. https://doi.org/10.1017/S135132490000005X

[3] Simran Arora, Avanika Narayan, Mayee F. Chen, Laurel J. Orr, Neel Guha, Kush Bhatia, Ines Chami, and Christopher Ré. 2023. Ask Me Anything: A simple strategy for prompting language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=bhUPJnS2g0X

[4] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022* (Munich, Germany). Springer, Berlin, Heidelberg, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[5] Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. 2023. Benchmarking and Improving Text-to-SQL Generation under Ambiguity. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*. Association for Computational Linguistics, 7053–7074. https:

//doi.org/10.18653/v1/2023.emnlp-main.436

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[7] Peter Baile Chen, Fabian Wenz, Yi Zhang, Moe Kayali, Nesime Tatbul, Michael J. Cafarella, Çagatay Demiralp, and Michael Stonebraker. 2024. BEAVER: An Enterprise Benchmark for Text-to-SQL. *CoRR* abs/2409.02038 (2024). arXiv:2409.02038 https://doi.org/10.48550/arXiv.2409.02038

[8] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proc. VLDB Endow.* 11, 11 (July 2018), 1482–1495. https://doi.org/10.14778/3236187.3236200

[9] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[10] E. F. Codd. 1974. Seven Steps to Rendezvous with the Casual User. In *Data Base Management, Proceeding of the IFIP Working Conference Data Base Management, Cargèse, Corsica, France, April 1-5, 1974*. North-Holland, 179–200.

[11] The SQLite Consortium. 2024. SQLite Release 3.45.1. https://www.sqlite.org/releaselog/3_45_1.html.

[12] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[14] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proc. ACM Manag. Data* 1, 4, Article 227 (Dec. 2023), 26 pages. https://doi.org/10.1145/3626768

[15] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, Xu Sun, and Zhifang Sui. 2024. A Survey on In-context Learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*. Association for Computational Linguistics, 1107–1128. https://aclanthology.org/2024.emnlp-main.64

[16] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot Text-to-SQL with ChatGPT. *CoRR* abs/2307.07306 (2023). arXiv:2307.07306 https://doi.org/10.48550/arXiv.2307.07306

[17] Yanai Elazar, Nora Kassner, Shauli Ravfogel, Abhilasha Ravichander, Eduard H. Hovy, Hinrich Schütze, and Yoav Goldberg. 2021. Measuring and Improving Consistency in Pretrained Language Models. *Trans. Assoc. Comput. Linguistics* 9 (2021), 1012–1031. https://doi.org/10.1162/tacl_a_00410

[18] Ahmed Elgohary, Saghar Hosseini, and Ahmed Hassan Awadallah. 2020. Speak to your Parser: Interactive Text-to-SQL with Natural Language Feedback. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 2065–2077. https://doi.org/10.18653/v1/2020.acl-main.187

[19] Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir R. Radev. 2018. Improving Text-to-SQL Evaluation Methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Association for Computational Linguistics, 351–360. https://aclanthology.org/P18-1033/

[20] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.* 17, 5 (Jan. 2024), 1132–1145. https://doi.org/10.14778/3641204.3641221

[21] Fadi H. Hazboun, Majdi Owda, and Amani Yousef Owda. 2021. A Natural Language Interface to Relational Databases Using an Online Analytic Processing Hypercube. *AI* 2, 4 (2021), 720–737. https://www.mdpi.com/2673-2688/2/4/43

[22] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 132 (April 2024), 29 pages. https://doi.org/10.1145/3649849

[23] Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. 1978. Developing a natural language interface to complex data. *ACM Trans. Database Syst.* 3, 2 (June 1978), 105–147. https://doi.org/10.1145/320251.320253

[24] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Association for Computational Linguistics, 963–973. https://doi.org/10.18653/v1/P17-1089

[25] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703

[26] Fei Li and H. V. Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 73–84. https://doi.org/10.14778/2735461.2735468

[27] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*. AAAI Press, 13067–13075. https://doi.org/10.1609/aaai.v37i11.26535

[28] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. *Proc. ACM Manag. Data* 2, 3, Article 127 (May 2024), 28 pages. https://doi.org/10.1145/3654930

[29] Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023. Graphix-T5: mixing pre-trained transformers with graph-aware layers for text-to-SQL parsing. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'23/IAAI'23/EAAI'23)*. AAAI Press, Article 1467, 9 pages. https://doi.org/10.1609/aaai.v37i11.26536

[30] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1835, 28 pages.

[31] Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2024. Table-GPT: Table Fine-tuned GPT for Diverse Table Tasks. *Proc. ACM Manag. Data* 2, 3, Article 176 (May 2024), 28 pages. https://doi.org/10.1145/3654979

[32] Shuqin Li, Kaibin Zhou, Zeyang Zhuang, Haofen Wang, and Jun Ma. 2023. Towards Text-to-SQL over Aggregate Tables. *Data Intell.* 5, 2 (2023), 457–474. https://doi.org/10.1162/dint_a_00194

[33] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? *CoRR* abs/2408.05109 (2024). arXiv:2408.05109 https://doi.org/10.48550/arXiv.2408.05109

[34] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[35] Weizheng Lu, Jing Zhang, Ju Fan, Zihao Fu, Yueguo Chen, and Xiaoyong Du. 2025. Large language model for table processing: a survey. *Frontiers Comput. Sci.* 19, 2 (2025), 192350. https://doi.org/10.1007/s11704-024-40763-6

[36] LILY Lab of Yale University. 2024. Spider Website Homepage. https://yale-lily.github.io/spider.

[37] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). arXiv:2303.08774 https://doi.org/10.48550/arXiv.2303.08774

[38] Simone Papicchio, Paolo Papotti, and Luca Cagliero. 2023. QATCH: benchmarking SQL-centric tasks with table representation learning models on your data. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1348, 20 pages.

[39] Parth Parikh, Oishik Chatterjee, Muskan Jain, Aman Harsh, Gaurav Shahani, Rathin Biswas, and Kavi Arya. 2022. Auto-Query - A simple natural language to SQL query generator for an e-learning platform. In *IEEE Global Engineering Education Conference, EDUCON 2022, Tunis, Tunisia, March 28-31, 2022*. IEEE, 936–940. https://doi.org/10.1109/EDUCON52537.2022.9766617

[40] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces* (Miami, Florida, USA) *(IUI '03)*. Association for Computing Machinery, New York, NY, USA, 327. https://doi.org/10.1145/604045.604120

[41] Gwenyth Portillo Wightman, Alexandra Delucia, and Mark Dredze. 2023. Strength in Numbers: Estimating Confidence of Large Language Models by Prompt Agreement. In *Proceedings of the 3rd Workshop on Trustworthy Natural Language Processing (TrustNLP 2023)*. Association for Computational Linguistics, Toronto, Canada, 326–362. https://aclanthology.org/2023.trustnlp-1.28

[42] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: decomposed in-context learning of text-to-SQL with self-correction. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1577, 10 pages.

[43] Abdul Quamar, Fatma Özcan, Dorian Miller, Robert J Moore, Rebecca Niehus, and Jeffrey Kreulen. 2020. Conversational BI: an ontology-driven conversation system for business intelligence applications. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3369–3381. https://doi.org/10.14778/3415478.3415557

[44] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* 21, 1, Article 140 (Jan. 2020), 67 pages.

[45] Daking Rai, Bailin Wang, Yilun Zhou, and Ziyu Yao. 2023. Improving Generalization in Language Model-based Text-to-SQL Semantic Parsing: Two Simple Semantic Boundary-based Techniques. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*. Association for Computational Linguistics, 150–160. https://doi.org/10.18653/v1/2023.acl-short.15

[46] Irina Saparina and Mirella Lapata. 2024. AMBROSIA: A Benchmark for Parsing Ambiguous Questions into Database Queries. *CoRR* abs/2406.19073 (2024). arXiv:2406.19073 https://doi.org/10.48550/arXiv.2406.19073

[47] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 9895–9901. https://doi.org/10.18653/v1/2021.emnlp-main.779

[48] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. *CoRR* abs/2405.16755 (2024). arXiv:2405.16755 https://doi.org/10.48550/arXiv.2405.16755

[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023). arXiv:2302.13971 https://doi.org/10.48550/arXiv.2302.13971

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[51] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 7567–7578. https://doi.org/10.18653/v1/2020.acl-main.677

[52] Weiguo Wang, Sourav S. Bhowmick, Hui Li, Shafiq Joty, Siyuan Liu, and Peng Chen. 2021. Towards Enhancing Database Education: Natural Language Generation Meets Query Execution Plans. In *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1933–1945. https://doi.org/10.1145/3448016.3452822

[53] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=1PL1NIMMrw

[54] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 94–107. https://doi.org/10.1145/3514221.3526125

[55] Wikipedia. 2024. Jaccard Similarity. Retrieved from https://en.wikipedia.org/wiki/Jaccard_index.

[56] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 63 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133887

[57] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. Association for Computational Linguistics, 3911–3921. https://doi.org/10.18653/v1/d18-1425

[58] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. 2024. ReAcTable: Enhancing ReAct for Table Question Answering. *Proc. VLDB Endow.* 17, 8 (April 2024), 1981–1994. https://doi.org/10.14778/3659437.3659452

[59] Fuheng Zhao, Lawrence Lim, Ishtiyaque Ahmad, Divyakant Agrawal, and Amr El Abbadi. 2023. LLM-SQL-Solver: Can LLMs Determine SQL Equivalence? *CoRR* abs/2312.10321 (2023). arXiv:2312.10321 https://doi.org/10.48550/arXiv.2312.10321

[60] Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic Evaluation for Text-to-SQL with Distilled Test Suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*. Association for Computational Linguistics, 396–411. https://doi.org/10.18653/v1/2020.emnlp-main.29

[61] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017). arXiv:1709.00103 http://arxiv.org/abs/1709.00103

[62] Chunting Zhou, Junxian He, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. 2022. Prompt Consistency for Zero-Shot Task Generalization. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*. Association for Computational Linguistics, 2613–2626. https://doi.org/10.18653/v1/2022.findings-emnlp.192

[63] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proc. VLDB Endow.* 12, 11 (July 2019), 1276–1288. https://doi.org/10.14778/3342263.3342267

[64] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2735–2748. https://doi.org/10.1109/ICDE53745.2022.00250

[65] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. DB-GPT: Large Language Model Meets Database. *Data Sci. Eng.* 9, 1 (2024), 102–111. https://doi.org/10.1007/s41019-023-00235-6

[66] Jun-Peng Zhu, Peng Cai, Kai Xu, Li Li, Yishen Sun, Shuai Zhou, Haihuang Su, Liu Tang, and Qi Liu. 2024. AutoTQA: Towards Autonomous Tabular Question Answering through Multi-Agent Large Language Models. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3920–3933. https://doi.org/10.14778/3685800.3685816

Received October 2024; revised January 2025; accepted February 2025