

# AMBI SQL: Interactive Ambiguity Detection and Resolution for Text-to-SQL

Zhongjun Ding

Alibaba Group

dingzhongjun.dzj@alibaba-inc.com

Yin Lin

Alibaba Group

yin.lin@alibaba-inc.com

Tianjing Zeng

Alibaba Group

zengtianjing.ztj@alibaba-inc.com

## ABSTRACT

Text-to-SQL systems translate natural language questions into SQL queries, providing substantial value for non-expert users. While large language models (LLMs) show promising results for this task, they remain error-prone. Query ambiguity has been recognized as a major obstacle for LLM-based Text-to-SQL systems, leading to misinterpretation of user intent and inaccurate SQL generation. We demonstrate AMBI SQL, an interactive system that automatically detects query ambiguities and guides users through intuitive multiple-choice questions to clarify their intent. Our approach introduces a fine-grained ambiguity taxonomy for identifying ambiguities that affect database element mapping and LLM reasoning, then incorporates user feedback to rewrite ambiguous questions. Evaluation on an ambiguous query dataset shows that AMBI SQL achieves 87.2% precision in ambiguity detection and improves SQL exact match accuracy by 50% when integrated with Text-to-SQL systems. Our demonstration showcases the significant performance gains and highlights the system’s practical usability. Code repo and demonstration are available at: <https://github.com/JustinzjDing/AmbiSQL>.

## 1 INTRODUCTION

Natural language interfaces for databases have been extensively studied in the database community. Recent advances [8, 13, 14, 19, 22] in utilizing large language models (LLMs) for Text-to-SQL translation have led to remarkable improvements, as evidenced by their performance on widely used benchmarks such as Spider [24] and BIRD [12].

Despite these promising results, *query ambiguity* remains a key error source in LLM-based Text-to-SQL systems, causing misalignment between the user’s actual intent and generated SQL queries [3, 8, 9, 25]. Given an ambiguous query, these systems might generate syntactically valid but semantically inaccurate SQL queries that do not meet the user’s expectations. This creates significant challenges for Text-to-SQL system development (e.g., difficulty in linking natural language questions to the correct database elements), evaluation (e.g., queries may have multiple valid interpretations while benchmarks provide only a single ground-truth answer), and real-world deployment (e.g., systems struggle to accurately parse and understand user intentions without additional context).

Research [15, 23] demonstrates that Text-to-SQL systems frequently encounter ambiguous questions in real-world scenarios. These ambiguities manifest in various forms, each presenting unique challenges for accurate SQL generation. For example, consider a query sampled from BIRD [12]: “What is the average number of test takers from Fresno schools that opened between 1/1/1980 and 12/31/1980?” The geographical reference “Fresno” can be mapped to the “City” or “County” column in the database by the LLM. The user has to explicitly use “schools in Fresno County” to resolve this

ambiguity. TAG [3], another widely used benchmark containing queries requiring complex LLM reasoning and external knowledge, presents even more diverse ambiguities. Consider the question “How many drivers born after the end of the Vietnam War have been ranked 2?”. The temporal constraint “end of the Vietnam War” may be interpreted at different levels of granularity (year vs. exact date). In addition, the ambiguous phrase “ranked 2” can be mapped to both the “position” and “rank” columns from different tables.

**Existing Works and Limitations.** Query ambiguity resolution techniques have been extensively studied. Existing approaches mainly follow two strategies: engaging users for clarification after ambiguity identification [4, 11], or leveraging LLMs to automatically generate candidate interpretations and disambiguate through subsequent selections [8, 10, 17]. However, these approaches face several limitations:

(1) *Limited Ambiguity Coverage.* In Text-to-SQL systems, phrases in the natural language query are mapped to relevant database columns and cell values, and utilize LLM knowledge to generate SQL queries. However, most existing studies [4, 10, 17] focus on resolving data-related and question-related ambiguities, especially those causing schema linking errors. These approaches often overlook ambiguities that affect LLM reasoning, such as determining the precise temporal interpretation of “end of the Vietnam War” in our previous example.

(2) *Efficiency and Generalization Concerns.* Several studies [8, 17] resolve ambiguity by generating extensive candidate interpretations (e.g., alternative SQLs), the majority of which prove irrelevant to the actual user intent. This approach incurs significant computational overhead and limits practical scalability. Additionally, methods relying on pre-trained ambiguity detection models [4, 17] require extensive annotated datasets and demonstrate limited cross-domain generalization capabilities.

(3) *Misalignment with User Intent.* Many automated approaches [8, 10, 17, 20] rely solely on LLMs to resolve ambiguities without user involvement, risking interpretations that are misaligned with actual user intent.

**Contributions.** To address these limitations, we propose AMBI SQL, an interactive system for ambiguity identification and query refinement through user clarifications. Our approach centers on a comprehensive taxonomy that specifies fine-grained ambiguity types, to detect and resolve ambiguities that affect database element mapping and LLM reasoning in Text-to-SQL systems.

The system employs in-context learning to automatically detect ambiguous phrases and classify them into ambiguity categories. Based on this classification, it generates targeted clarification questions with intuitive multiple-choice options, then rewrites queries according to user selections. Our demonstration showcases AMBI SQL’s user-friendly interface that guides non-expert users

through disambiguation. Users simply select from multiple-choice options without requiring SQL or database knowledge. We demonstrate how AMBISQL integrates seamlessly as a preprocessing module with existing Text-to-SQL systems, detecting ambiguities in user questions and improving SQL generation accuracy through user clarification.

## 1.1 Related Work

**Ambiguity Categories and Benchmarks.** Studies have systematically analyzed the various forms of ambiguity in Text-to-SQL benchmarks, revealing their adverse effects on SQL generation accuracy [15, 23]. Building on these findings, recent works have focused on constructing new benchmarks and datasets specifically designed to handle ambiguous queries. AmbiQT [2] interprets each ambiguous query with two plausible SQLs to enhance evaluation reliability, compared with previous benchmarks where each query is only annotated with a single ground truth SQL. PRACTIQ [6] innovatively constructs a practical conversational Text-to-SQL dataset with ambiguous and unanswerable queries, providing a training corpus for Text-to-SQL systems aimed at engaging users for clarification. PYTHIA [20] proposes a training-based large-scale data generation system to automatically identify and annotate database-linked (e.g., schema, column, cell values) ambiguities in natural language inputs. Ambrosia [16] constructs a human-annotated Text-to-SQL dataset containing ambiguous queries. Although all of these benchmarks, together with some exploratory studies [7, 21], attempt to classify ambiguities in Text-to-SQL, their discussions focus mainly on ambiguities related to queries and databases. None of them have addressed ambiguities that affect LLM reasoning.

While these benchmarks provide a foundation for evaluating the ability of Text-to-SQL systems to handle ambiguous and unanswerable questions, significant challenges remain in achieving practical and generalizable ambiguity resolution for real-world deployments.

**Ambiguity Resolution.** Query disambiguation has been extensively studied, with approaches operating at different stages of the Text-to-SQL pipeline. *Preprocessing approaches* resolve ambiguity within the natural language inputs, such as APA [11], which fine-tunes LLMs to generate clarification requests for ambiguous input questions, enabling disambiguation through user interaction prior to SQL generation. *Inprocessing approaches* target ambiguity identification during SQL generation, such as RTS [4], which focuses on the schema linking phase, employing pre-trained classifiers on LLM hidden layers and applying conformal prediction techniques to provide probabilistic guarantees for error detection during the translation process. *Postprocessing approaches* focus on fixing generated SQLs after initial translation, including SQLens [10], which trains a classifier for error prediction followed by error correction and auditing to enhance SQL quality, and [17], which generates multiple interpretations using LLMs and employ a pre-trained model to fill missing information, producing clearer SQL outputs.

## 2 PRELIMINARIES

### 2.1 Ambiguity Taxonomy

Existing works [2, 5, 7, 10, 18] have proposed taxonomies of ambiguities in SQL generation. However, their classifications primarily focus on ambiguities that cause misalignments between natural

language questions and database elements. However, recent Text-to-SQL benchmarks like BIRD [12] and TAG [3] have expanded SQL generation to include LLM reasoning capabilities, where external knowledge evidence is incorporated during SQL generation [12] or LLM-based knowledge retrieval is required during query execution [3]. This introduces new types of ambiguity that fall beyond the scope of previous taxonomies.

We present a comprehensive taxonomy that classifies ambiguities based on two primary dimensions, providing a foundation for addressing ambiguities arising from both database element mapping and LLM reasoning in Text-to-SQL systems.

The first dimension is (1) **DB-related ambiguity**: Ambiguity arising from unclear or underspecified references to database schema or content in natural language questions, leading to incorrect or incomplete data retrieval. This dimension encompasses the following subcategories:

- **Unclear schema reference**: The question lacks sufficient context to determine which table or column to use for operations like filtering, ranking, or aggregation, resulting in multiple plausible interpretations (e.g., "oldest user" could refer to age or registration date).
- **Unclear value reference**: The question refers to a value that does not correctly correspond to the actual values stored in the database, making it unclear how to formulate the WHERE clause condition and potentially causing relevant results to be omitted or producing inaccurate results (e.g., querying for "New York City" when the database stores "NYC" or asking for posts about "COVID-19" when the database contains "coronavirus").
- **Missing SQL-related keywords**: Key terms clarifying the intended SQL operation are absent, creating ambiguity about the desired operation (e.g., "Show me users by registration date" could mean ORDER BY for sorting, GROUP BY for grouping, or WHERE for filtering).

The second dimension is (2) **LLM-related ambiguity**: Ambiguity that results in the misuse of LLM reasoning, causing difficulties in correctly retrieving or applying information beyond the database content. This dimension includes the following subcategories:

- **Unclear knowledge source**: The question fails to specify whether required information should be retrieved from the database or inferred through LLM reasoning (e.g., for "female employees," whether to query a gender column or use semantic analysis of name fields).
- **Insufficient reasoning context**: The question lacks adequate information to guide LLM reasoning effectively (e.g., requesting "current exchange rate" without specifying the target currencies or date).
- **Conflicting knowledge**: Knowledge assumptions embedded within the question contradicts real-world facts or database contents (e.g., querying participants in events that never occurred).
- **Ambiguous temporal/spatial scope**: Spatial or temporal constraints are underspecified, resulting in multiple possible interpretations at different granularities (e.g., "after the 2018 World Cup" could mean immediately after the final match or after the entire tournament year).

## 2.2 Problem Statement

We model a Text-to-SQL system as a function  $f : (q, \mathcal{D}) \mapsto Q$  that maps a natural language question  $q$  and the database schema  $\mathcal{D}$  to a SQL query  $Q$ . Given our ambiguity taxonomy, we define the set of ambiguity types as  $\mathcal{A} = \{a_1, a_2, \dots, a_n\}$ , where each  $a_i$  represents a distinct ambiguity category that may introduce errors in Text-to-SQL translation.

Given a natural language question  $q$ , a Text-to-SQL system  $f$ , and our predefined ambiguity types  $\mathcal{A}$ , the *ambiguity detection problem* aims to identify: (1) ambiguous phrases  $P = \{p_1, p_2, \dots, p_k\}$  within  $q$ , where each  $p_i$  is a substring of  $q$ , and (2) the corresponding ambiguity type  $a_i \in \mathcal{A}$  for each phrase  $p_i$ , such that these ambiguous phrases cause  $f(q, \mathcal{D})$  to generate an incorrect SQL query  $Q \neq Q^*$ , where  $Q^*$  is the ground truth SQL.

The *ambiguity resolution problem* then seeks to clarify these ambiguous phrases and rewrite  $q$  as  $q'$  such that  $f(q', \mathcal{D}) \mapsto Q^*$ , producing the correct SQL query.

## 3 SYSTEM ARCHITECTURE

### 3.1 Method Overview

We present the overview of the AMBiSQL system in Figure 1. The system employs a two-stage pipeline: *Ambiguity Identification* and *Iterative Refinement*. The first stage aims to identify ambiguities in the original user query, while the second stage focuses on iteratively refining the query via user clarifications.

**Stage 1: Ambiguity Identification.** Given a natural language query and the corresponding database schema, the system first employs the ambiguity detection module to identify potentially ambiguous phrases (e.g., "rank 2" and "end of the Vietnam War" in our running example), and then classifies them according to our ambiguity taxonomy. These identified ambiguities are subsequently passed to the clarification question generation module, which formulates each ambiguous aspect into a targeted multiple-choice question, often supplemented by relevant descriptions or database schema snippets (cf. grey background panel in Stage 1). The generated clarification questions, along with their accompanying descriptions, are then presented to the user for response.

**Stage 2: Iterative Refinement.** After the user provides answers to the clarification questions, both the questions and their corresponding answers are used as inputs for Stage 2. Additionally, AMBiSQL allows users to specify *additional constraints* (e.g., "drivers need to be German") as extra conditions to refine their original query and clarify their intent. The question refinement module integrates the original query with any additional constraints to produce an updated query. This refined query, together with the user clarifications, is then re-examined by the ambiguity detection module to ensure no new ambiguities were introduced during the clarification process. Once all ambiguities are resolved, the refined query and user clarifications are submitted to the underlying Text-to-SQL system, which generates a SQL query reflecting the user's intent.

Section 3.2 provides detailed descriptions of all core system modules: the ambiguity detection module (used in both stages), the clarification question generation module (Stage 1), and the preference update and question refinement modules (Stage 2).

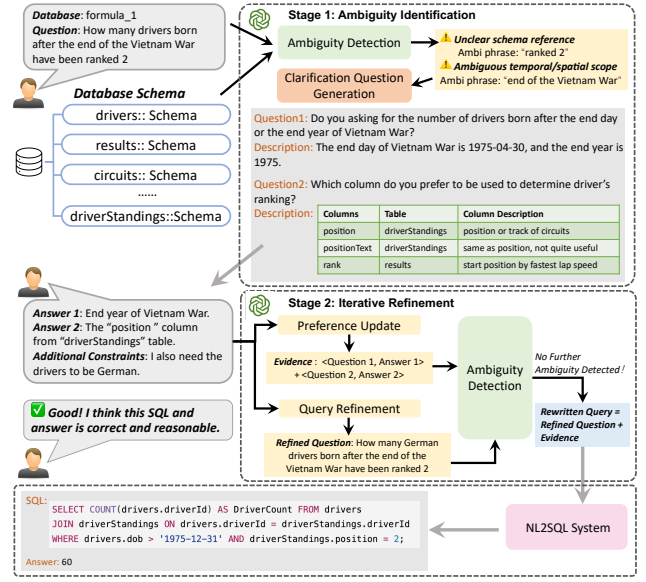


Figure 1: Overview of the AMBiSQL System.

### 3.2 AMBiSQL Design

**Ambiguity Detection.** As revealed in [7], advanced LLMs exhibit a similar agreement rate with human annotators on ambiguity detection tasks. Based on this observation, we design our ambiguity detection module in AMBiSQL to employ pre-trained LLMs to identify ambiguities in user questions through in-context learning. This process benefits from the comprehensive and clear ambiguity definitions and taxonomy introduced in Section 2.

The main challenge in using LLMs for ambiguity identification lies in instructing the model to detect potentially ambiguous phrases from queries and accurately categorize them. To address this, we construct prompts that include an explicit ambiguity definition, a concise taxonomy, and representative examples to guide the LLM. For instance, given a question such as "Which city is the largest one?", we provide an accompanying explanation highlighting its ambiguous aspect (which demonstrates a DB-related unclear schema reference ambiguity: the user does not specify whether 'largest' refers to area or population) to the LLM. With clear instructions and illustrative examples, the LLM can generalize and recognize similar ambiguities in unseen questions.

**Clarification Question Generation.** After detecting the ambiguous phrases and their categories, we aim to generate explicit and descriptive questions to help users clarify their intent. This is achieved by the clarification question generation module. For each detected ambiguity, the module rephrases it into a targeted clarification question accompanied by a brief description. The description either summarizes relevant knowledge from the database or parametric knowledge of the LLM, or provides a relevant database snippet when necessary. For example, for DB-related ambiguities, we provide a database snippet and ask which schema, value, or key terms should be used. For LLM-related ambiguities, we ask which source the relevant information should be retrieved from, or specifically which exact value reference should be mapped to ambiguous

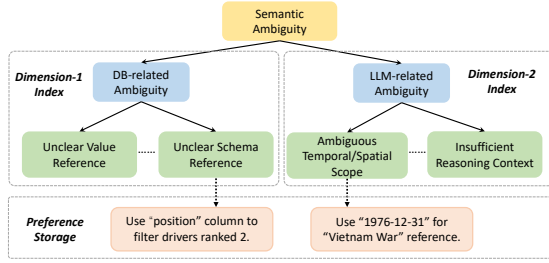


Figure 2: User Preference Tree.

phrases. Each follow-up question is presented in a multiple-choice format, which helps reduce answer uncertainty compared with free-text answers.

**Preference Update.** Based on the hierarchical taxonomy, we use a tree-like structure to index and store user clarifications in the preference update module. Each branch indexes a specific ambiguity subcategory, and each leaf node stores user clarifications linked to that subcategory. Following the example in Figure 1, when ambiguities such as "unclear schema reference" or "ambiguous temporal/spatial scope" are detected, the user's answers to clarification questions are stored in the corresponding leaf nodes. Specifically, the user's column selection is stored under the "unclear schema reference" node, while an exact date selection for a temporal reference is stored under the "ambiguous temporal/spatial scope" node, as illustrated in Figure 2.

If a semantic contradiction occurs within a leaf node, the LLM automatically updates the preferences in the data cells. It merges the new and previous preferences through in-context learning, using conflict definitions and illustrative examples as prompts. For example, if a user changes their preference from the "position" column to the "rank" column for driver ranking, the system updates the recorded preference accordingly to reflect this latest intent. This ensures that the recorded preference is always up to date.

**Question Refinement.** This module is designed to handle users' *additional constraints* and is responsible for synthesizing these constraints into the original query. When conflicts arise between the original and additional constraints, the module always prioritizes and retains the constraint specified in the additional constraints, reflecting the user's most recent intent. Considering *additional constraints* may be ambiguous whereas clarification answers are generally unambiguous, we handle them separately from the preference update module to facilitate ambiguity detection.

## 4 DEMONSTRATION AND EVALUATION

### 4.1 Demonstration Plan

We demonstrate AMBISQL's operation on five databases from the BIRD benchmark [12]: *California Schools*, *Debit Card Specializing*, *Formula One*, *Codebase Community*, and *European Football*. We also construct a set of 40 ambiguous queries from BIRD [12] and TAG [3] that covers all ambiguity categories defined in Section 2.1. Each query includes ground-truth SQL and human-annotated ambiguity categories. We integrate AMBISQL with XiYan-SQL [8] as the downstream Text-to-SQL system.

We demonstrate AMBISQL's user interface and key features through three functional panels (①–③) shown in Figure 3.

**User Input Panel ①.** Users can interact with AMBISQL in two ways. First, they can specify custom queries using the User Input Panel ①, which accepts three inputs: a natural language query, the target database dialect, and the specific database to be queried. Alternatively, to facilitate easy exploration, we provide an example selection feature that allows users to browse queries from our ambiguous dataset. Selecting an example automatically populates the user input panel with the corresponding parameters, enabling users to conveniently explore our system using default queries.

**Ambiguity Resolution Panel ②.** Upon submission, AMBISQL analyzes the inputs for ambiguity detection (Stage 1 in Figure 1). For each identified ambiguity, the system generates a corresponding multiple-choice clarification question. Each question is presented in this panel, with all candidate options accessible via dropdown menus. As demonstrated in Figure 3, interpretations related to "the end of Vietnam War" at different granularities (e.g., start date, end date, or year, each annotated with the exact time reference) are presented as options. For the unclear schema reference ambiguity caused by the phrase "ranked 2", three relevant column choices are presented for the user to clarify, each accompanied by a relevant database snippet for reference (cf. the grey dropdown menu in Figure 3). Users may optionally provide *additional constraints* in an auxiliary input field to further refine their query. In the example, we further constrain the driver's nationality to Germany.

After the user submits clarifications, AMBISQL iteratively refines the original query, generating a clarified and unambiguous rewritten query (Stage 2 in Figure 1). If no further ambiguities are detected during the user-system interaction, Panel ② remains disabled. The rewritten query is then passed to the downstream Text-to-SQL system for SQL generation, as shown in Panel ③.

**SQL Generation Panel ③.** This panel shows the SQL statements generated by the Text-to-SQL system. The output from XiYan-SQL without ambiguity resolution and the output with AMBISQL are displayed side by side. As shown, both output SQL queries capture the essential information from the original user query and the additional constraint. They preserve the initial filter conditions and add an extra user-specified filter on the driver's nationality.

The *Compare* button checks each generated SQL against a predefined ground truth SQL. This comparison highlights the benefits of ambiguity resolution. When given the ambiguous query, the basic XiYan-SQL model may misinterpret the user's intent. For example, it might link "ranked 2" to the wrong column, or fail to apply the correct temporal filter for "after the end of the Vietnam War." As a result, inaccurate SQL statements may be generated. In contrast, by leveraging ambiguity resolution and explicit user preferences, AMBISQL enables XiYan-SQL to generate accurate SQL statements that consistently capture the user's requirements.

### 4.2 Experimental Observations

We evaluated AMBISQL on two key aspects: (1) end-to-end SQL generation improvements when integrated with existing Text-to-SQL systems, and (2) accuracy of ambiguity detection and classification. For SQL generation, we report accuracy improvements achieved by AMBISQL on our constructed dataset using Exact Match accuracy [12] to measure correctness. For ambiguity detection, we evaluate AMBISQL's precision, recall, and F<sub>1</sub>-score in identifying and



Question	Database Name	DB Dialect
How many drivers born after the end of Vietnam War have been ranked 2?	formula_1	SQLite
Name all drivers in the 2010 Singapore Grand Prix order by their position stands.	formula_1	SQLite

Figure 3: User Interface of AMBiSQL.

classifying ambiguous phrases according to our taxonomy. All experiments are conducted using GPT-4o [1] as the underlying LLM. For evaluation, each query is processed without any additional user-specified constraints.

**SQL Generation Accuracy Improvement.** As demonstrated in Table 1, on the entire dataset of 40 ambiguous queries, XiYan-SQL [8] alone achieves an exact match accuracy of 42.5%. By leveraging interactive clarification via AMBiSQL, the accuracy increases remarkably to 92.5%, underscoring the effectiveness of ambiguity resolution.

For the TAG samples, the exact match accuracy increases substantially from 20.0% to 85.0% with the help of AMBiSQL. This improvement can be primarily attributed to the inherent complexity of TAG queries, which often involve intricate intents, requirements for external knowledge, and multi-table reasoning. Our observations indicate that independent ambiguity resolution is particularly necessary when both complex query interpretation and ambiguity identification are required in Text-to-SQL process. In particular, XiYan-SQL without AMBiSQL falls short when external knowledge is required for SQL generation with limited reasoning context, frequently failing to retrieve or correctly interpret such knowledge due to various LLM-related ambiguities (as defined in Section 2.1). For instance, when queried for "counties in Silicon Valley," the model is unable to generate a comprehensive county list owing to the ambiguous definition of geographical boundaries, an issue that AMBiSQL effectively addresses by eliciting clarification from users.

For the BIRD samples, XiYan-SQL attains an exact match accuracy of 75.0%, which is further improved to 100.0% with AMBiSQL. This result is largely due to the clear, explicit structure of BIRD

Table 1: Exact Match Accuracy of Text-to-SQL Tasks (based on XiYan-SQL) with and without AMBiSQL.

	Text-to-SQL w/o Ambi. Resolution	AMBiSQL
Average Acc.	42.5%	92.5%
TAG Samples	20.0%	85.0%
BIRD Samples	75.0%	100.0%

queries and straightforward query-to-schema mapping. The primary source of error in the BIRD samples stems from incorrect column mapping caused by insufficient database metadata descriptions. These findings demonstrate that AMBiSQL is effective in handling both complex and straightforward ambiguous queries.

**Ambiguity Detection Performance.** We further report the ambiguity detection and classification accuracy of AMBiSQL. As presented in Table 2, AMBiSQL achieves an overall precision of 87.2%, recall of 89.1%, and an  $F_1$ -score of 88.2%, indicating the effectiveness of ambiguity detection of AMBiSQL based on our taxonomy.

Further evaluation across different ambiguity dimensions reveals that, for the *DB-related ambiguity*, the model achieves higher recall but comparatively lower precision. This suggests that AMBiSQL successfully identifies most instances in this category, albeit with a slight increase in over-detection. In contrast, for the *LLM-related ambiguity*, the system demonstrates higher precision but lower recall, indicating a more conservative strategy that prioritizes accuracy over coverage. These patterns can be attributed to the inherent challenges in detecting LLM-induced ambiguities: while DB-related ambiguities can often be resolved through static, unambiguous database information, LLM-related ambiguities often arise from the model’s reliance on inconsistent or conflicting external knowledge sources, making detection more difficult.

**Table 2:** Ambiguity Detection Accuracy of AMBISQL.

	Precision	Recall	F <sub>1</sub> -Score
<b>Overall</b>	87.2%	89.1%	88.2%
<b>DB-related Ambiguity</b>	83.9%	92.9%	88.1%
Unclear schema reference	84.2%	88.9%	86.5%
Unclear value reference	80.0%	100.0%	88.9%
Missing SQL-related keywords	100.0%	100.0%	100.0%
<b>LLM-related Ambiguity</b>	93.8%	83.3%	88.2%
Unclear knowledge source	100.0%	66.7%	80.0%
Insufficient reasoning context	66.7%	100.0%	80.0%
Conflicting knowledge	100.0%	100.0%	100.0%
Ambiguous temporal/spatial scope	100.0%	81.8%	90.0%

Among subcategories, "unclear schema reference" emerges as the most prevalent across the dataset. Our analysis reveals that errors caused by this kind of ambiguity mainly stem from vague or incomplete descriptions in the database metadata. Notably, some subcategories achieve perfect coverage ( $F_1$ -score = 100%), indicating that the ambiguity taxonomy has been effectively internalized by the LLM through our in-context learning instructions.

## 5 FINAL REMARKS

In this paper, we demonstrate AMBISQL, a user-friendly interactive framework for ambiguity identification and query refinement in Text-to-SQL systems. We propose a systematic taxonomy covering fine-grained ambiguities that arise from both database element mapping and LLM reasoning. Our system employs LLMs to detect and classify ambiguities in user questions, then uses user feedback to refine queries through targeted clarification questions.

We envision several application scenarios for AMBISQL:

- **Enhancing Benchmark Robustness:** Most existing Text-to-SQL benchmarks only provide a single answer for each question, even when multiple interpretations may be valid due to ambiguity. These benchmarks typically employ exact match metrics, treating any SQL output not equivalent to the ground truth as an error, regardless of the question ambiguity. This leads to the evaluation of systems on such benchmarks being underestimated and inaccurate. AMBISQL can help identify ambiguous questions in existing benchmarks and generate multiple reasonable interpretations with corresponding SQL queries, enhancing benchmark robustness.

- **Improving Text-to-SQL Systems:** It has been shown [7] that a great percentage of Text-to-SQL errors correspond to ambiguous questions. AMBISQL can be integrated with these systems to enhance their capability in handling ambiguous queries. Compared to approaches that require fine-tuning, our method is more lightweight, does not rely on any task-specific training data, and is easier to adapt to new domains.

- **Interactive Query Clarification:** In real-world database querying scenarios, users naturally formulate ambiguous questions due to their limited familiarity with database schemas or inherent complexity in their needs. Rather than rejecting such queries, AMBISQL embraces this nature by allowing users to start with ambiguous questions and then guiding them through an interactive clarification process. The system identifies potential ambiguities and engages users to progressively refine their intent, ultimately producing accurate SQL queries while making database querying more accessible.

## REFERENCES

- [1] Open AI. 2024. GPT-4o System Card. *CoRR* abs/2410.21276 (2024).
- [2] Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. 2023. Benchmarking and Improving Text-to-SQL Generation under Ambiguity. In *EMNLP*.
- [3] Asim Biswal, Liana Patel, Siddharth Jha, Amog Kamsetty, Shu Liu, Joseph E. Gonzalez, Carlos Guestrin, and Matei Zaharia. 2025. Text2SQL is Not Enough: Unifying AI and Databases with TAG. In *CIDR*.
- [4] Kaiwen Chen, Yueting Chen, Nick Koudas, and Xiaohui Yu. 2025. Reliable Text-to-SQL with Adaptive Abstention. In *SIGMOD*.
- [5] Ziru Chen, Shijie Chen, Michael White, Raymond J. Mooney, Ali Payani, Jayanth Srinivasa, Yu Su, and Huan Sun. 2023. Text-to-SQL Error Correction with Language Models of Code. In *ACL*.
- [6] Mingwen Dong, Nischal Ashok Kumar, Yiqun Hu, Anuj Chauhan, Chung-Wei Hang, Shuaichen Chang, Lin Pan, Wuwei Lan, Henghui Zhu, Jiarong Jiang, Patrick Ng, and Zhiguo Wang. 2025. PRACTIQ: A Practical Conversational Text-to-SQL dataset with Ambiguous and Unanswerable Queries. In *NAACL*.
- [7] Avriella Floratou, Fotis Psallidas, Fuheng Zhao, Shaleen Deep, Gunther Hagleither, Wangda Tan, Joyce Cahoon, Rana Alotaibi, Jordan Henkel, Abhik Singla, Alex Van Grootel, Brandon Chow, Kai Deng, Katherine Lin, Marcos Campos, K. Venkatesh Emani, Vivek Pandit, Victor Shnayder, Wenjing Wang, and Carlo Curino. 2024. NL2SQL is a solved problem... Not!. In *CIDR*.
- [8] Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. 2025. A Preview of XiYan-SQL: A Multi-Generator Ensemble Framework for Text-to-SQL. arXiv:2411.08599
- [9] Parker Glenn, Parag Dakle, Liang Wang, and Preethi Raghavan. 2024. BlendSQL: A Scalable Dialect for Unifying Hybrid Question Answering in Relational Algebra. In *ACL (Findings)*.
- [10] Yue Gong, Chuan Lei, Xiao Qin, Kapil Vaidya, Balakrishnan Narayanaswamy, and Tim Kraska. 2025. SQLens: An End-to-End Framework for Error Detection and Correction in Text-to-SQL. arXiv:2506.04494
- [11] Hyuhng Joon Kim, Youna Kim, Cheonbok Park, Junyeob Kim, Choonghyun Park, Kang Min Yoo, Sang-goo Lee, and Taekuk Kim. 2024. Aligning Language Models to Explicitly Handle Ambiguity. In *EMNLP*.
- [12] 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 Chen-Chuan 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 *NeurIPS*.
- [13] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. 2025. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. In *ICLR*.
- [14] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *NeurIPS*.
- [15] Mohammadreza Pourreza and Davood Rafiei. 2023. Evaluating Cross-Domain Text-to-SQL Models and Benchmarks. In *EMNLP*.
- [16] Irina Saparina and Mirella Lapata. 2024. Ambrosia: A benchmark for parsing ambiguous questions into database queries. In *NeurIPS*.
- [17] Irina Saparina and Mirella Lapata. 2025. Disambiguate First Parse Later: Generating Interpretations for Ambiguity Resolution in Semantic Parsing. arXiv:2502.18448
- [18] Jiawei Shen, Chengcheng Wan, Ruoyi Qiao, Jiazhen Zou, Hang Xu, Yuchen Shao, Yueling Zhang, Weikai Miao, and Guguang Pu. 2025. A Study of In-Context-Learning-Based Text-to-SQL Errors. *CoRR* abs/2501.09310 (2025).
- [19] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHES: Contextual Harnessing for Efficient SQL Synthesis. *CoRR* abs/2405.16755 (2024).
- [20] Enzo Veltri, Gilbert Badaro, Mohammed Saeed, and Paolo Papotti. 2023. Data Ambiguity Profiling for the Generation of Training Examples. In *ICDE*.
- [21] Bing Wang, Yan Gao, Zhoujun Li, and Jian-Guang Lou. 2023. Know What I don't Know: Handling Ambiguous and Unknown Questions for Text-to-SQL. In *ACL (Findings)*.
- [22] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In *COLING*.
- [23] Niklas Wretblad, Fredrik Riseby, Rahul Biswas, Amin Ahmadi, and Oskar Holmström. 2024. Understanding the Effects of Noise in Text-to-SQL: An Examination of the BIRD-Bench Benchmark. In *ACL*.
- [24] 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 *EMNLP*.
- [25] Fuheng Zhao, Divyakant Agrawal, and Amr El Abbadi. 2025. Hybrid Querying Over Relational Databases and Large Language Models. In *CIDR*.