MindSweeper: Looking for the Unknowns

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have made significant progress in assisting users with querying databases in natural language. While LLM-based techniques provide state-of-the-art results on many standard benchmarks, their performance drops significantly in interactive settings, where the system interacts with the user to resolve possible ambiguities in the user's query. To mitigate this problem, we propose a method that performs a feedback loop based on SQL equivalence checking to drive the dialog with the user. We develop a new framework that consistently outperforms state-of-the-art techniques on the BIRD-Interact benchmark by up to 10% across several underlying LLMs.

1 Introduction

Text-to-SQL is one of the central tasks in many interactive systems that require retrieving data from databases (Amazon, 2025; Amazon Web Services, 2024; BitsAI, 2025; Splunk, 2025). Since data are typically stored in databases for efficient access, it is almost a necessity for a system agent, such as a chatbot, to communicate with database interfaces. Indeed, there has been substantial investment in this area from major companies, including Google (Pourreza et al., 2024), Databricks (Ali et al., 2025), and Snowflake (Yao et al., 2026).

Most prior work in Text-to-SQL has focused on single-turn interaction, where the user asks a question and the system agent generates a SQL query, executes it against a database, and returns the result (Liu et al., 2023; Dong et al., 2023; Chang and Fosler-Lussier, 2023; TiDBCloud, 2020; Shkapenyuk et al., 2025; Talaei et al., 2024; Gao et al., 2025; Sequeda et al., 2023; Sheng and Xu, 2025; Liu et al., 2025; Zhai et al., 2025; Li et al., 2024; Lei et al., 2024). However, in practice, this setting is often unrealistic, as users cannot always precisely express their intent in the initial question. Therefore, the agent must guess the intended meaning, which frequently leads to incorrect outputs (Wretblad et al., 2024; Hui, 2024).

To address this limitation, recent work has introduced multi-turn text-to-SQL settings, where at each turn, the system agent decides whether to ask a clarification question or to terminate by producing a SQL query (Huo et al., 2025). While this setting enables the agent to acquire additional relevant information, it also introduces a major epistemic challenge: in order to ask questions to fill its knowledge gap, the agent must know what it does not yet know. However, relying on the agent itself to determine what remains ambiguous is circular: ambiguities outside the agent's current interpretation of the user question are precisely the ones it is least likely to report. Indeed, state-of-the-art LLM-based agents still perform poorly in interactive text-to-SQL settings, with success rates around 30%, and to our knowledge, no prior work has proposed techniques to specially address this challenge.

In this work, we investigate the multi-turn Text-to-SQL task. We argue that current low performance stems in part from the agent's difficulty in independently assessing whether it has resolved all relevant ambiguities. We propose to give the agent access to an external formal-reasoning oracle, which soundly pinpoints the differences between candidate SQL queries compatible with the current dialogue context. Based on this external feedback, the agent can ask targeted clarification questions to gradually uncover the missing information. We show that this approach can significantly improve performance in multi-turn settings.

2 Preliminary

Text-to-SQL task. Given an NL query N from a user U, a database D with schema M, and additional knowledge K, the task is to map N to an SQL query Q such that executing Q on D produces the answer to N. There are single-turn and multi-turn variants of this task. In the multi-turn setting, clarification questions can be asked of the user, as the original query contains multiple ambiguities.

User query ambiguity types. We consider common types of ambiguities (Huo et al., 2025). *Superficial* ambiguities arise directly from how the user phrases the request, e.g., the intent is vague or important implementation details are missing.

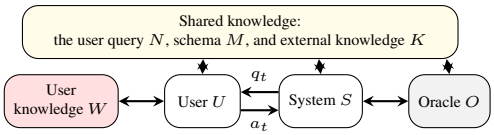

Figure 1: Representation of U and S agents interactions.

Knowledge ambiguities are caused by missing or incomplete external knowledge, where the system cannot reliably infer needed facts without additional clarification. *Environmental* ambiguities are introduced by noise or irregularities in the data environment, e.g., NULL values.

Text-to-SQL evaluation. The main evaluation mechanism consists of two metrics (Huo et al., 2025). The first metric is **success rate**, SR. For each natural language query N over a database, there exists a gold SQL query Q that represents the human-labeled ground truth translation of N into SQL. Given a generated SQL query P and the corresponding gold query Q, the evaluation performs the following check: $\text{SR}(P, Q, D_\text{t}) = 1$ if $set(P(D_\text{t})) = set(Q(D_\text{t}))$ and 0 otherwise, where D_t is a test database provided by the benchmark set, and SR compares whether the two tables, $P(D_\text{t})$ and $Q(D_\text{t})$, contain the same rows. The second evaluation metric is **efficiency** E, which is defined as the number of turns used before the system agent emits a generated query P.

3 Motivation

We consider a model of multi-turn interactions between a user U and a system agent S in an environment $\mathcal{E}$. An environment $\mathcal{E}$ represents static knowledge and contains a database D, its schema M, and additional knowledge of two types: external public knowledge K and user-specific private knowledge W. Figure 1 shows a schematic representation of the interaction (we ignore the oracle O for now). Both U and S have access to the initial ambiguous user question N, the schema M, and external knowledge K. In addition, the user U has access to private knowledge W. At each turn t, the system S asks the user U a clarification question q_t aimed at resolving ambiguity in the user question, and the user responds with an answer a_t. The interaction proceeds for at most T turns, where T is a given constant. Each turn of the interaction is represented as a pair $\langle q_t, a_t \rangle$.

Definition 1. *A dialog $\mathcal{D}_t$ is a sequence of t pairs $\langle q_1, a_1 \rangle, \ldots, \langle q_t, a_t \rangle$, where q_t and a_t are NL text representing a question and its answer.*

$\mathcal{D}_t$ represents knowledge accumulated during the interaction up to time t.

```
CREATE TABLE scores (
   name TEXT PK,
   biology INTEGER,
   chemistry INTEGER,
   physics INTEGER);
```

name	biol.	chem.	phys.
Alice	90	100	100
Bob	NULL	100	100
Claire	100	90	100

Figure 2: Schema M and the `scores` table.

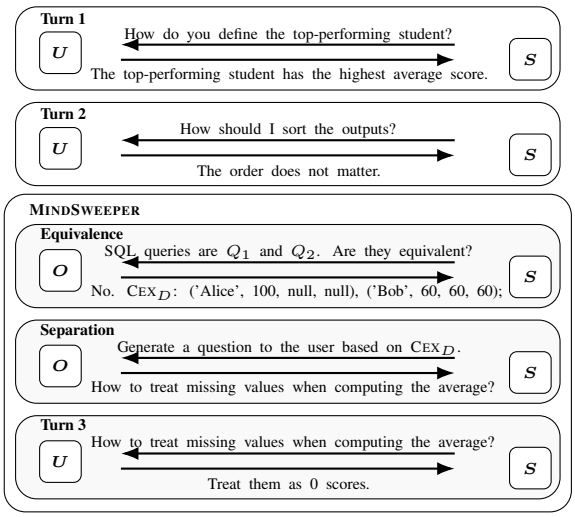

Figure 3: Resolving ambiguities in Example 1.

Example 1. *Suppose we have an environment $\mathcal{E}$ that contains a database D describing exam scores of students in biology, chemistry, and physics. Figure 2 shows an example of a* `scores` *table and the schema of D. The table has four columns that represent a student name and their exam scores in the three subjects, on a scale from 0 to 100. Some students did not take an exam, so their score is NULL. K is empty. Suppose the user asks,* 'Show the top-performing students in the class.' *This question contains two ambiguities. First is* superficial ambiguity *as the notion of a* `top-performing` *student is undefined. By the user's choice, a top-performing student is one with the highest average exam score. The second is* environmental ambiguity. *The treatment of* `NULL values` *when operators are applied is undefined. By the user's choice, NULL values should be treated as 0. Note that under standard SQL semantics, the average is computed by* ignoring *NULL values. Hence, it is important to understand the user-intended behavior to generate the final SQL.* □

Since the user query is ambiguous, the goal of the system agent S is to detect and resolve these ambiguities by asking the user questions.

Example 2. *Figure 3 shows a possible conversation between U and S from Example 1 (we ignore the* MINDSWEEPER *box for now). The interaction runs for two turns. After the second turn is completed, S outputs an* SQL *query Q_1 shown in*

```sql
WITH avg_scrs AS (
   SELECT Name, AVG(score) AS avg_scr
   FROM (
      SELECT Name, Biology AS score FROM scores
      UNION ALL
      SELECT Name, Chemistry FROM scores
      UNION ALL
      SELECT Name, Physics FROM scores)
   GROUP BY Name)
SELECT Name FROM avg_scrs
WHERE avg_scr = (SELECT MAX(avg_scr) FROM avg_scrs)
```

(a) Query Q_1. Standard AVG semantics: NULLs are ignored

```sql
WITH avg_scrs AS (
   SELECT Name, AVG(COALESCE(score, 0)) AS avg_scr
   FROM (
      SELECT Name, Biology AS score FROM scores
      UNION ALL
      SELECT Name, Chemistry FROM scores
      UNION ALL
      SELECT Name, Physics FROM scores)
   GROUP BY Name)
SELECT Name FROM avg_scrs
WHERE avg_scr = (SELECT MAX(avg_scr) FROM avg_scrs)
```

(b) Query Q_2. User's choice: NULL values are treated as zero

Figure 4: Two SQL formulations for computing the top-performing student under different semantics

Figure 4a. Based on Q_1, Bob's score is 100, and Alice's and Claire's scores are 96.67. Bob is returned as the top-performing student, which is incorrect w.r.t. the user's intent of treating NULLs as 0. □

4 MINDSWEEPER

The main challenge for the S agent is to resolve all ambiguities in the user query. However, as shown in Example 2, the system agent has no way to know whether it has resolved all ambiguities, whether it should ask additional clarification questions, or whether it should stop and produce the final SQL query Q. In practice, these challenges are left to the system agent to resolve based on the user query and the system's knowledge base. However, state-of-the-art results demonstrate that this leads to very poor performance (BIRD-Interact Mini, 2025). Our main idea is to help the system understand what it does not know by using an external oracle that has the capability for formal reasoning. An oracle, denoted O, is an agent that takes two SQL queries as input and decides whether these queries are semantically equivalent. More concretely, an oracle takes two SQL queries Q_1 and Q_2 as input and performs two actions: *(1) Equivalence check:* Given two SQL queries Q_1 and Q_2, it decides whether they are semantically equivalent. If so, it outputs 'equivalent'; otherwise, it outputs a counterexample database CEX_D; *(2) Separation question:* Given two SQL queries Q_1 and Q_2 and a counterexample database CEX_D, it formulates a question for the user that separates these two queries to identify which one corresponds to the user intent.

At any point in the conversation, the system S can query the oracle with a pair of SQL queries and ask for feedback, i.e., whether they are equivalent and how to ask the user a question to separate them. There are two main advantages of such an oracle. First, it helps S determine whether it is confident enough to produce semantically equivalent SQL queries, and thus provides guidance for the stopping criterion. Second, if the SQL queries are not equivalent, the oracle helps S formulate a separation question to reveal the next ambiguity. Figure 1 shows the oracle component in gray. The oracle can access the shared knowledge, and the system can communicate with the oracle at any turn.

Algorithm 1 MINDSWEEPER workflow

Require: Maximum number of turns T, number of samples s, user question N, schema M, external knowledge K, user knowledge W
Ensure: An SQL query P

$\mathcal{D} \leftarrow [\,]$ ▷ Sequence of pairs $\langle q_i, a_i \rangle$
$t \leftarrow 1$ ▷ Turn number
while $t < T$ **do**
  if $\neg S.\text{USEORACLE}()$ **then** ▷ Ask the next question (or output SQL)
    $st, q_t \leftarrow S.\text{QUESTIONORFINALSQL}(\mathcal{D}, K, M, N)$
    if st **then return** q_t **else** $a_t \leftarrow U.\text{ANSWER}(W, M, N, q_t)$
  else ▷ Consult the oracle
    $A \leftarrow \emptyset$
    for $i \leftarrow 1$ to s **do**
      $A \leftarrow A \cup \{S.\text{SAMPLESQL}(\mathcal{D}, K, M, N, i)\}$
    $q_t \leftarrow \emptyset$; $a_t \leftarrow \emptyset$
    for all pairs (Q_i, Q_j) with $Q_i, Q_j \in A$ and $i < j$ **do**
      $(\textit{equiv}, \text{CEX}_D) \leftarrow O.\text{EQUIVALENCE}(M, Q_i, Q_j)$
      if $\neg$*equiv* **then**
        $q_t \leftarrow O.\text{SEPARATION}(\mathcal{D}, K, M, N, Q_i, Q_j, \text{CEX}_D)$
        $a_t \leftarrow U.\text{ANSWER}(W, M, N, q_t)$
        break
    if q_t is $\emptyset$ **then** ▷ No separating counterexample found
      $st, q_t \leftarrow S.\text{QUESTIONORFINALSQL}(\mathcal{D}, K, M, N)$
      if st **then return** q_t **else** $a_t \leftarrow U.\text{ANSWER}(W, M, N, q_t)$
  $\mathcal{D} \leftarrow \mathcal{D} + [\langle q_t, a_t \rangle]$
  $t \leftarrow t + 1$
return $S.\text{FINALSQL}(\mathcal{D}, K, M, N)$ ▷ Output a final SQL as $t = T$.

Algorithm 1 describes the workflow of MINDSWEEPER, which iterates for at most T turns. At each turn, S operates in one of two modes. In *system-driven* mode, it asks a clarifying question or outputs a final SQL query. In *oracle-driven* mode, it samples s candidate SQL queries and invokes O to identify semantically non-equivalent pairs; for each such pair, the oracle produces a separation question that S poses to the user. The two modes interleave, progressively narrowing the space of valid SQL queries toward the user's intent (see Appendix A for a detailed description).

Example 3. *Continuing Example 2, after the second turn S enters oracle-driven mode. It samples two candidate queries Q_1 and Q_2 (Figure 4); the oracle finds a counterexample database that separates them and suggests a clarifying question, which S asks in Turn 3 (Figure 3). The user response resolves the last ambiguity of the user query.*

System S	User U	Baseline		MINDSWEEPER			
				$O_{\texttt{llm}}$		$O_{\texttt{ver}}$	
		SR	E	SR	E	SR	E
GPT-5-nano	GPT-5.1	19.2%	2.9	19.2%	5.8	**22.4%**	6.7
GPT-5-mini	GPT-5.1	26.7%	3.1	27.1%	4.7	**32.3%**	5.9
O4-mini	GPT-5.1	38.4%	3.7	**48.7%**	5.0	43.7%	5.9
Sonnet-4.5	Haiku-4.5	34.6%	4.6	**57.7%**	7.7	48.6%	7.4

Table 1: Results on a reduced dataset (78 instances).

See Appendix A for the full walkthrough. □

5 Experimental evaluation

Setup. We evaluated the proposed techniques on a set of benchmarks from the BIRD-interact Mini dataset (BIRD-interact Mini, 2025). The dataset contains 300 instances of ambiguous questions and the corresponding queries in SQLite, and it provides a user simulator. We used two state-of-the-art models to power the user agent: GPT-5.1 and Haiku-4.5. We used weak models (GPT-5-nano, GPT-5-mini) and strong models (O4-mini, Sonnet-4.5) to power the system agent. Due to cost limitations, we performed extensive experiments on a subset of benchmarks, where we took the first three queries from each dataset, giving us 78 instances in this reduced dataset. The final experiments were performed on the entire BIRD-interact mini dataset. We use standard evaluation metrics, **success rate** SR and **efficiency** E, described in Section 2.

Since SQL equivalence is undecidable (Mohamed et al., 2024), we use incomplete oracles: $O_{\texttt{ver}}$, a bounded verifier (He et al., 2024) augmented with a per-database fuzzer that we developed using Cursor (Anysphere, 2025) for queries beyond its fragment, and $O_{\texttt{llm}}$, an LLM-based fallback. In practice, $O_{\texttt{ver}}$ resolves about 80% of checks and $O_{\texttt{llm}}$ the remaining 20%. See Appendix B for more details on configurations.

Results. Table 1 shows our results on the reduced dataset. Each number is averaged over three runs. The variance is below 2% for all experiments, so we do not report it in the table. It is clear from the table that the oracle helps to drive the conversation and improve the accuracy of the model on these benchmarks by up to 23.1%. Note that oracle-driven conversations also take more turns, showing they can reveal more information about the user's intent. For example, the system agent powered by O4-mini takes about one extra step but increases SR by 10% compared to the baseline. We also notice that weaker models (e.g., GPT-5-nano) cannot serve as powerful oracles of type $O_{\texttt{llm}}$, while stronger models, such as O4-mini or Sonnet-4.5, are more capable of performing equivalence checking using $O_{\texttt{llm}}$. Therefore, for the final experiments on all benchmarks, we use $O_{\texttt{ver}}$ for system agents powered by weaker models and $O_{\texttt{llm}}$ for system agents powered by stronger models.

System S	User U	Baseline		MINDSWEEPER	
		SR	E	SR	E
GPT-5-nano ($O_{\texttt{ver}}$)	GPT-5.1	23.3%	2.9	**29.1%**	6.6
GPT-5-mini ($O_{\texttt{ver}}$)	GPT-5.1	29.7%	3.1	**34.3%**	5.8
O4-mini ($O_{\texttt{llm}}$)	GPT-5.1	38.3%	3.7	**45.3%**	5.3
Sonnet-4.5 ($O_{\texttt{llm}}$)	Haiku-4.5	39.7%	4.9	**49.7%**	7.7

Table 2: Results on a BIRD-interact mini dataset.

Table 2 shows our results on the BIRD-interact Mini dataset. Using MINDSWEEPER yields consistent improvements in performance across all system and user models. We report results for one run only, as the experiment is very costly. The best SR is achieved with the system using Sonnet-4.5, reaching SR $= 49.7\%$, which is a significant improvement over the baseline. The system agent powered by O4-mini also shows strong performance (45.3%) and uses about one extra step on average compared to the baseline. We ran an additional experiment with the baseline approach in which we enforced the interaction to run for at least 5 steps before it can produce the final SQL, to verify whether simply running the baseline longer leads to improvement. In this case, the baseline approach with S powered by O4-mini achieved SR $= 39.3\%$, which demonstrates that MINDSWEEPER takes meaningful advantage of the additional turns.

Ablation: number of sampled SQL queries. We study the sensitivity of results to the number of sampled SQL queries s on the reduced dataset, using the system agent powered by O4-mini. We varied $s \in \{1, 2, 3, 4\}$ and found that $s = 3$ is sufficient: increasing to $s = 4$ or more provides no measurable improvement in SR while increasing computational cost proportionally.

6 Conclusion and Future work

We show that interactive systems for the Text-to-SQL task can benefit from sound feedback obtained through logical reasoning. Our reasoning oracle solves SQL equivalence as an intermediate subproblem, which helps the system agent gain information about ambiguities in user questions and recover the user's intent. This allows MINDSWEEPER to effectively search the SQL semantic space guided by counterexamples produced by the oracle. One important future direction is to investigate how formal-reasoning tools can better support multi-turn conversations involving complex real-world SQL queries.

Limitations

Oracle incompleteness. The oracle is sound but incomplete in solving the SQL equivalence problem: none of the methods we employ, i.e., bounded verification, fuzzing, and LLM judgment, can guarantee that they detect non-equivalence between two SQL queries. In such cases, the system may stop asking questions before all ambiguities are resolved.

Formal equivalence checking coverage. Currently, SOTA formal equivalence checkers (He et al., 2024) can handle about 25% of query pairs in our experiments; queries with JSON, window functions, or recursion had to fall back to fuzzing or LLM judgment.

SQL sampling diversity. Repeated LLM sampling cannot guarantee that all plausible query interpretations are explored. Some ambiguities may remain undetected if they do not appear among the sampled queries.

Ethical Considerations

This work focuses on improving interactive Text-to-SQL systems to better understand user intent using state-of-the-art LLMs and formal-methods tools. We do not foresee ethical concerns specific to this work.

References

Alnur Ali, Ashutosh Baheti, Jonathan Chang, Ta-Chung Chi, Brandon Cui, Andrew Drozdov, Jonathan Frankle, Abhay Gupta, Pallavi Koppol, Sean Kulinski, Jonathan Li, Dipendra Misra, Krista Opsahl-Ong, Jose Javier Gonzalez Ortiz, Matei Zaharia, and Yue Zhang. 2025. A state-of-the-art sql reasoning model using rlvr. *Preprint*, arXiv:2509.21459.

Amazon. 2025. Build a text-to-sql solution for data consistency in generative ai using amazon nova.

Amazon Web Services. 2024. How msd uses amazon bedrock to translate natural language into sql for complex healthcare databases. Accessed: 2025-09-21.

Anysphere. 2025. Cursor - the ai code editor. Accessed: February 19, 2026.

BIRD-interact Mini. 2025. Bird-interact: Re-imagine text-to-sql evaluation via lens of dynamic interactions. https://huggingface.co/datasets/birdsql/mini-interact. Accessed: 2025-06-01.

Team BIRD-Interact Mini. 2025. Bird-interact: Re-imagine text-to-sql evaluation via lens of dynamic interactions. https://bird-interact.github.io/. Accessed: 2025-06-01.

Team BitsAI. 2025. Bits ai:scale your teams with autonomous ai agents across monitoring, development, and security.

Shuaichen Chang and Eric Fosler-Lussier. 2023. How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings. *arXiv preprint arXiv:2305.11853*.

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. *Preprint*, arXiv:2307.07306.

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. *Preprint*, arXiv:2411.08599.

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).

Binyuan Hui. 2024. Panel of BIRD Annotation Issues. https://github.com/AlibabaResearch/DAMO-ConvAI/issues/39. Accessed: May 2024.

Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipei Lin, Bowen Qin, Binyuan Hui, Xiaolong Li, Ge Qu, Shuzheng Si, Linheng Han, Edward Alexander, Xintong Zhu, Rui Qin, Ruihan Yu, Yiyao Jin, Feige Zhou, Weihao Zhong, Yun Chen, and 5 others. 2025. Bird-interact: Re-imagining text-to-sql evaluation for large language models via lens of dynamic interactions. *Preprint*, arXiv:2510.05318.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, and 1 others. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*.

Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2024. BIRD_SQ. https://bird-bench.github.io/. Accessed: Sep 2025.

Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S. Yu. 2023. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability. *Preprint*, arXiv:2303.13547.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2025. Xiyan-sql: A novel multi-generator framework for text-to-sql. *Preprint*, arXiv:2507.04701.

Mudathir Mohamed, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2024. Verifying sql queries using theories of tables and relations. In *Proceedings of 25th Conference on Logic for Pro*, volume 100, pages 445–463.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *Preprint*, arXiv:2410.01943.

Juan Sequeda, Dean Allemang, and Bryon Jacob. 2023. A benchmark to understand the role of knowledge graphs on large language model's accuracy for question answering on enterprise sql databases. *Preprint*, arXiv:2311.07509.

Lei Sheng and Shuai-Shuai Xu. 2025. Csc-sql: Corrective self-consistency in text-to-sql via reinforcement learning. *Preprint*, arXiv:2505.13271.

Vladislav Shkapenyuk, Divesh Srivastava, Theodore Johnson, and Parisa Ghane. 2025. Automatic metadata extraction for text-to-sql. *Preprint*, arXiv:2505.19988.

Splunk. 2025. Ai assistant in observability cloud.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *Preprint*, arXiv:2405.16755.

TiDBCloud. 2020. Get Started with Chat2Query API. `https://docs.pingcap.com/tidbcloud/use-chat2query-api`. Accessed: May 2024.

Niklas Wretblad, Fredrik Gordh 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. *Preprint*, arXiv:2402.12243.

Zhewei Yao, Guoheng Sun, Lukasz Borchmann, Gaurav Nuti, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2026. Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql. *Preprint*, arXiv:2505.20315.

Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. 2025. Excot: Optimizing reasoning for text-to-sql with execution feedback. *Preprint*, arXiv:2503.19988.

A MINDSWEEPER Algorithm

Algorithm 1 describes the workflow of MINDSWEEPER. The algorithm maintains a dialog history $\mathcal{D}$ and iteratively extends it for at most T turns. Both the system and the user agents are powered by an LLM to perform natural-language steps, such as formulating clarification questions in the case of S and answering questions in the case of U. At each turn, S chooses whether to use the oracle (line 4). The choice of the USEORACLE() function is a hyperparameter. If S does not use the oracle, it follows a system-driven mode, where S either asks the next clarifying question or stops and directly outputs a final SQL query based on the current dialog and available knowledge (lines 5–6). Otherwise, S enters an oracle-driven mode (line 7). In this mode, the system samples s candidate SQL queries that are consistent with the current dialog $\mathcal{D}_t$ (line 10). Sampling is implemented by invoking the same LLM prompt multiple times with identical parameters; the stochastic nature of the LLM is sufficient to produce diverse candidate queries without requiring more sophisticated strategies (see Section C for the prompt). It then examines pairs of sampled queries using the oracle O to check whether they are equivalent. If the oracle finds a counterexample database for a pair, the oracle generates a separation question that distinguishes between the two queries (lines 15–16). The S poses this question to the U and records the response in the dialog. After each turn, the newly obtained pair is appended to the dialog (line 21), and the process continues. If no separating counterexample is found among the sampled candidates, the S falls back to asking a clarifying question (or producing a final SQL). Overall, the algorithm interleaves system-guided clarification with oracle-guided equivalence checking, enabling the system S to progressively reduce the space of valid SQL queries and converge toward a query that matches the user's intent.

Example 4. *Let us continue Example 2 and consider how* MINDSWEEPER *operates on this example. Suppose that after the second turn, the system enters the oracle-driven mode. First, it samples the space of SQL queries given the dialog $\mathcal{D}$ and the shared knowledge. Figure 4 shows examples of two possible sampled queries: Q_1 and Q_2.*

Figure 3 illustrates a possible interaction with the oracle in the MINDSWEEPER *box. First, the system S asks the oracle O to check the equivalence of Q_1 and Q_2. The oracle finds a counterexample and provides a counterexample database* CEX_D *(see the "Equivalence" box in Figure 3). On this database, Q_1 returns* `Alice` *as the top student, since her average score is 100 when NULL values are ignored, while Bob's average score is 60. However, Q_2 returns* `Bob`*, since NULL values are treated as 0 before averaging, and Alice's score is $avg(100, 0, 0) = 33.3$ in this case.*

Second, the system asks the oracle to generate a question for the user that separates these two queries (the "Separation" box in Figure 3). Based on this suggestion, the system asks the next clarification question in Turn 3. □

B Experimental Evaluation

Setup: Algorithm 1. In terms of input parameters of Algorithm 1, we set S.USEORACLE() to always return `True` to investigate the extreme case in which the oracle drives the conversation at every turn (line 4). Note that calling the oracle at each interaction step represents the most aggressive configuration; heuristically deciding when to invoke the oracle is a promising direction to reduce computational cost while maintaining performance, and we leave this as future work.

The number of SQL samples s is set to 3 just by calling LLM three times. The baseline approach is the vanilla interaction procedure described in Section 3. The number of turns was determined as in the BIRD-interact mini evaluation, based on the number of ambiguities in the user query with a patience parameter of 5.

Setup: LLM calls. We used default parameters for all calls to LLMs.

Cost. We spent approximately $3000 US dollars on these experiments.

Implementation of the oracle equivalence-checking procedure. Equivalence checking of SQLs is generally undecidable (Mohamed et al., 2024). Therefore, we consider relaxations. First, we use the LLM that powers the system agent to judge the equivalence between two queries (denoted $O_{\texttt{llm}}$). Second, we use a bounded verification approach (He et al., 2024) combined with a simple, per-database fuzzer generated using Cursor (Anysphere, 2025) (denoted $O_{\texttt{ver}}$). If formal verification fails or does not support the complexity of the SQL query (e.g., because many queries

contain JSON structures), we use the fuzzer. These are sound but incomplete methods that allow us to experiment with a formal-reasoning oracle. If both fail, we fall back to using an LLM to judge equivalence. In our experiments, approximately 80% of equivalence checks are resolved by $O_{\texttt{ver}}$ (formal verification or fuzzing), while the remaining 20% rely on $O_{\texttt{llm}}$. VERIEQL formally verifies approximately 25% of query pairs; the remainder handled by $O_{\texttt{ver}}$ uses the fuzzer, as many queries involve JSON structures or other constructs beyond VERIEQL's supported fragment.

C Prompts

The vanilla system prompt (below) is used by the system agent in both the baseline and the system-driven mode of MINDSWEEPER. The SQL sampling prompt is used by S.SAMPLESQL in oracle-driven mode: it is invoked s times with identical parameters, relying on the stochastic nature of the LLM to produce diverse candidate queries.

Vanilla System Prompt for Text-to-SQL

You are a good data scientist with great SQL writing ability. You have a DB called "[[DB name]]". You are given the DB schema information below:
DB Schema Info: [[DB schema]]
And you are given some useful external knowledge about this DB below: # External Knowledge: “‘json [[external kg]] “‘
Instructions:
You are a good data scientist who is tasked with generating SQLite to solve the user task below. However, the user’s query may not be clear enough. Then you need to ask for clarification about ambiguities in the user task below. You only have [[max turn]] turns to ask for clarification. At each turn, you can only ask one question with a few sentences. After using up all turns or if you are clear enough, you can provide the final SQLite.
You have the following choices at each turn:
Ask for Clarification: Enclose your question between <s>and </s>.
Generate Final SQL: Enclose your SQL between <t >sqlite and </t >.
NOTE: If you think you have asked enough questions or used up all turns, you MUST provide the final SQLite. User Task:
[[user query]] Turn 1 ([[max turn]] turns left):
Format: "<s >[YOUR-ONLY-ONE-QUESTION] </s>" or "<t >sqlite [FILL-YOUR-SQL] </t >"

SQL Sampling Prompt for Oracle-Driven Mode (S.SAMPLESQL)

Consider your conversation with the user to resolve the user question ambiguities so far.
Here is the user question: [[user query]]
Here is your conversation with the user so far:
[[dialog]]
Schema of the database with the data:
[[DB schema]]
Here is the SQL query that you have generated based on the user task and conversation so far:
{pred_sql}
Task
You are given a natural-language user question, the conversation so far, and a SQL query that you have generated as intermediate-result. However, the user question contains unresolved ambiguity.
The current SQL necessarily makes implicit assumptions about:
1. output shape and row granularity, 2. semantic categories and thresholds, 3. derived metric formulas, 4. time windows and reference points, 5. grouping level, 6. NULL and division-by-zero handling, etc
Your task is to:
Identify at least one such unresolved ambiguity that was silently guessed in the current SQL. Replace it with a different but still plausible interpretation. Generate a new SQL query that implements this alternative interpretation and is likely to produce a different result on at least some database instances. Do not reuse the same assumptions as the original SQL. Explain your assumptions briefly in the SQL comments.
Purpose
The goal is meaningful divergence in order to reveal missing user intent.
###Format your SQL: Then you MUST enclose your SQlite between <t>sqlite and </t>, for example <t>sqlite [FILL-YOUR-SQL] </t>. No explanations only SQLite query.