SQLENS: FINE-GRAINED AND EXPLAINABLE ERROR DETECTION IN TEXT-TO-SQL

Anonymous authors
Paper under double-blind review

ABSTRACT

Text-to-SQL systems translate natural language (NL) questions into SQL queries, allowing non-technical users to perform complex data analytics. Large language models (LLMs) have shown promising results on the text-to-SQL task. However, these LLM-based text-to-SQL solutions often generate syntactically correct but semantically incorrect SQL queries, which yield undesired execution results. Additionally, most text-to-SQL solutions generate SQL queries without providing information on the quality or confidence in their correctness. Systematically detecting semantic errors in LLM-generated SQL queries in a fine-grained manner with explanations remains unexplored. In this paper, we propose SQLENS, a framework that leverages the given NL question as well as information from the LLM and database to diagnose the LLM-generated SQL query at the clause level. SQLENS can link problematic clauses to error causes, and predict the semantic correctness of the query. SQLENS effectively detects issues related to incorrect data and metadata usage such as incorrect column selection, wrong value usage, erroneous join paths, and errors in the LLM's reasoning process. SQLENS achieves an average improvement of 25.78% in F1 score over the best-performing LLM self-evaluation method in identifying semantically incorrect SQL queries on two public benchmarks. We also present a case study to demonstrate that SQLENS can localize and explain errors for subsequent automatic error correction.

1 INTRODUCTION

Text-to-SQL systems, that can translate a natural language (NL) question into a SQL query, democratize data access for non-technical users, serving as an entry point to a larger data science pipeline (Patel et al., 2024). The advent of Large Language Models (LLMs) has significantly advanced this field, and LLM-based text-to-SQL techniques (Talaei et al., 2024; Lee et al., 2024) have demonstrated promising results on public benchmarks such as BIRD (Li et al., 2023) and Spider (Yu et al., 2018). Recently, the LLM-based text-to-SQL solutions have been adopted in data platforms offered by AWS[1], Databricks[2], Snowflake[3], etc.

Despite these advancements, text-to-SQL remains a challenging problem. The best performing method on the BIRD leaderboard[4] only achieves an execution accuracy of around 73% on the dev set, still producing more than 400 incorrect SQL queries out of 1534 NL questions. LLM-based systems typically employ a multi-stage generation pipeline, consisting of a retrieval stage to collect contextual information, a generation stage to produce candidate SQL queries and a correction stage to regenerate SQL queries based on SQL parser errors as needed. While much attention has been given to the retrieval and generation stages, there is still a lack of fine-grained and explainable error detection in the correction stage. Namely, detecting semantic errors—where the SQL query executes successfully but returns incorrect results—remains challenging and largely unsolved. This is because semantic errors require a deep understanding of both the query logic and the database's structure. Most text-to-SQL solutions only produce a SQL query without providing any information on the quality or measures of confidence.

[1] Amazon Q generative SQL - `https://tinyurl.com/yjwcfwmc`
[2] Databricks Assistant - `https://tinyurl.com/cdva2bjx`
[3] Snowflake Copilot - `https://tinyurl.com/mtry8z7p`
[4] BIRD Leaderboard - Execution Accuracy (EX) - `https://bird-bench.github.io/`

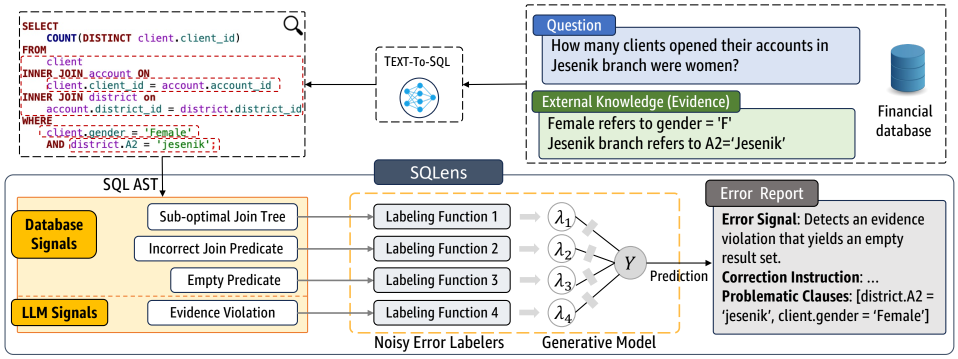

Figure 1: An overview of SQLens.

Existing LLM-based text-to-SQL methods, like DIN-SQL (Pourreza & Rafiei, 2023) and MCS-SQL (Lee et al., 2024), have a self-correction module that prompts an LLM to debug and correct a SQL query. Such modules detect potential errors or measure the confidence of a generated SQL query by generating multiple results and defining confidence e.g. based on LLM judgements or the number of consistent outputs. However, these approaches lack fine-grained semantic error information and explainability. They provide a confidence estimate for the entire SQL query based solely on the LLM's output but do not offer detailed insights into which part of the query might be incorrect and why it is potentially wrong. This lack of fine-grained and explainable error detection hinders both end users and the LLMs from effectively troubleshooting errors in LLM-generated SQL queries, ultimately undermining trust in LLM-based systems and their wider adoption (Brown, 2024).

In this paper, we develop SQLens, a fine-grained and explainable error detection framework for the text-to-SQL task. We concentrate on two specific challenges: (1) identifying potential error signals in a generated SQL query at the clause level, and (2) aggregating the error signals to determine whether the query could be semantically incorrect. SQLens is based on the intuition that a SQL query is reasonable for a question if the intermediate results of its clauses are reasonable (e.g., result sets are not empty, do not have too many missing values, etc.) and if the overall structure of the SQL follows meaningful join paths in the database schema.

As shown in Figure 1, SQLens parses a given SQL query into an abstract syntax tree (AST). For each SQL clause in the AST, SQLens exploits a variety of error signals - described in detail below - from both the database and the LLM to detect potential semantic errors. The database signals are lightweight and deterministic, assessing the correct usage of SQL clauses and evaluating their execution results over the database. The LLM-based signals are derived from LLM's comprehensive knowledge about SQL and semantic understanding of the given NL question. To mitigate the challenge of potentially noisy signals, SQLens is further equipped with a weakly-supervised training process that integrates both the database and the LLM signals to construct a labeled training dataset. SQLens then trains a supervised or unsupervised classification model to predict if the given SQL query is semantically correct, and generates an error report with detailed explanations. SQLens can also use feedback and any available labeled examples.

Fine-grained and Explainable. Detecting semantic errors in text-to-SQL is challenging because any misunderstandings about NL question or (meta-)data can lead to cascading errors throughout the SQL query generation process. We are the first to not only provide an overall estimation of a SQL query's semantic correctness but also identify potential error causes at the SQL clause level.

Robustness. Our approach can effectively handle noisy signals. The framework is capable of generating high-quality training data. As such, the system is domain-agnostic and can be adopted even under the absence of labeled data.

Applicability. SQLens is a general framework that can be seamlessly integrated with any text-to-SQL solution as it only takes as inputs an NL question and its corresponding SQL query. The fine-grained and explainable error report from SQLens can be utilized by any text-to-SQL solutions for error correction as demonstrated in Section 4.4.

Effectiveness. We provide extensive experimental results showing the effectiveness of SQLens on identifying semantically incorrect SQL queries (Section 4.2). On BIRD (Li et al., 2023) and

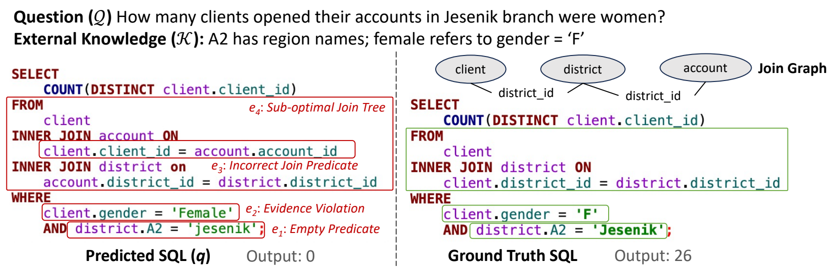

Figure 2: A running example on BIRD financial database.

Spider (Yu et al., 2018) benchmarks, SQLens achieves an average improvement of 25.78% in F1 score over the best-performing LLM self-evaluation method.

2 Preliminaries and Problem Statement

In this paper, a table T consists of a set of columns $\mathcal{C} = \{C_1, C_2, ..., C_n\}$. A join relationship J between two tables T_i and T_j is based on common attributes (i.e., joinable columns $T_i.C_m$ and $T_j.C_n$, respectively). A database instance $\mathcal{D} = \{(T_1, T_2, \ldots, T_n), \mathcal{J}\}$ comprises a set of tables and a set of join relationships $\mathcal{J}$ between these tables.

Definition 1 (Text-to-SQL Algorithm) *A text-to-SQL algorithm f takes as input a natural language question $\mathcal{Q}$, a database instance $\mathcal{D}$, and optionally external knowledge $\mathcal{K}$, and generates a SQL query $q = f(\mathcal{Q}, \mathcal{D}, \mathcal{K})$.*

Figure 2 presents a running example using a question from the BIRD benchmark, asking for the number of female clients who opened accounts at the Jesenik branch. The benchmark also provides external knowledge, such as annotations on a column name and a cell value. The predicted SQL query is generated by a text-to-SQL algorithm to answer the question.

Definition 2 (Semantic Error) *A semantic error e results in the SQL query q failing to correctly answer the natural language query $\mathcal{Q}$. Formally,*

$$do(e) \Rightarrow \mathcal{O}(q, \mathcal{D}) \neq \mathcal{O}(\mathcal{Q}, \mathcal{D})$$

The operation $do(e)$ denotes an intervention in the generation of q due to e, leading to a discrepancy between the observed output $\mathcal{O}(q, \mathcal{D})$ and the expected correct output $\mathcal{O}(\mathcal{Q}, \mathcal{D})$, thereby identifying e as the semantic error.

For example, the generated SQL query in Figure 2 is semantically incorrect with an output of 0, whereas the correct output is 26 based on the ground truth SQL query. First, the query incorrectly uses "*jesenik*" in the predicate, leading to the empty result. Secondly, the query violates the evidence specified in the evidence, using *gender='Female'* instead of *gender='F'*. Even with the correct predicates, the SQL query would still produce an incorrect result of 23 due to an incorrect join predicate and a suboptimal join strategy. Specifically, there is no valid join path between the *client* and *account* tables, indicating the join predicate *client.client_id = account.account_id* in the generated query is hallucinated. Furthermore, the *account* table involved in the join tree is redundant as the *client* and *district* tables can be directly joined to answer the question, as indicated in the join graph.

Problem 2.1 (Semantic Error Detection) *Given a natural language question $\mathcal{Q}$, a database instance $\mathcal{D}$, optionally external knowledge $\mathcal{K}$, and the output SQL query $q = f(\mathcal{Q}, \mathcal{D}, \mathcal{K})$ generated by a text-to-SQL algorithm f, the task is to identify a set of potential semantic errors $\mathcal{E}$ from q if q is semantically incorrect.*

3 SQLens Framework

To address Problem 2.1, we introduce SQLens, the first framework that provides fine-grained and explainable error detection for the text-to-SQL task. SQLens derives error signals from the SQL clauses by incorporating information from the database, the schema, intermediate execution results, and the LLM (Section 3.1). A robust error detection framework must reason about and establish

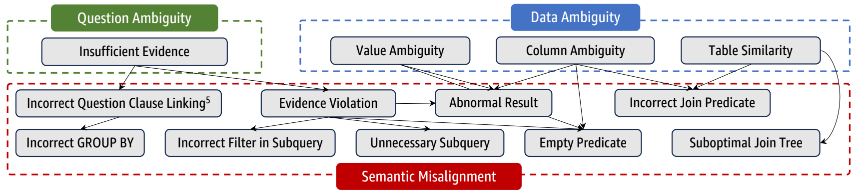

Figure 3: The causal graph of semantic errors and error signals in SQL queries.

relationships of these diverse error signals, where each could be noisy. Hence SQLENS employs a weak-supervision strategy to aggregate these signals to identify potential semantic errors and to predict the semantic correctness of the SQL query (Section 3.2).

3.1 ERROR SIGNALS

Precise detection of semantic errors from a SQL query is inherently challenging due to the complexity and ambiguity in NL queries, data, and database schemas. Our insight is that many semantic errors in LLM-generated queries exhibit common patterns that can be detected through carefully crafted signals. We categorize these errors as follows.

1. Question Ambiguity. The user's questions might inherently contain ambiguities, and can be interpreted in different ways. For example, if a user asks "*What were the total sales last quarter?*" in a database where the *sales* table has columns named both *gross_sales* and *net_sales*, either column could be selected to answer the question.
2. Data Ambiguity. In real-world databases, multiple tables or columns with similar or identical names could exist due to data integration, versioning, table transformations, and other factors, causing ambiguities as well. For example, a user might ask "*What are the average salaries by department?*", but the database contains both a *dept* table and a *department* table. Choosing the wrong table leads to incorrect query results.
3. Semantic Misalignment. Even when there is no ambiguity in NL questions nor databases, the semantic gap between the question and the data can still lead to misalignments between the generated SQL query and the given NL question. For instance, the SQL query shown in Figure 2 uses an incorrect join predicate (*client.client_id* = *account.account_id*) as the text-to-SQL algorithm fails to understand the join relationships in the financial database.

Definition 3 (SQL Error Signal) *An error signal s analyzes a SQL query q to identify potentially erroneous clauses $\mathcal{Q}'$ and their associated error causes $\mathcal{E}$. Formally, $(\mathcal{Q}', \mathcal{E}) = s(q)$.*

Intuitively, an error signal acts as a proxy for identifying SQL semantic errors. As depicted in Figure 3, we introduce an error causal graph that connects a diverse set of error signals to three common types of semantic errors described above. This graph resonates with the SQL error analysis conducted in recent text-to-SQL studies (Wang et al., 2024; Lee et al., 2024). Note that certain error signals can be more effectively and reliably extracted through database-driven analysis, particularly those related to semantic misalignment. In contrast, error signals that require nuanced interpretation of both the question and the SQL demand the deep semantic understanding capabilities from an LLM. Hence SQLENS incorporates both DB-based and LLM-based error signals to detect semantic errors effectively. Note that neither the aforementioned semantic errors nor the error signals described below are exhaustive, as error detection often involves a long tail of edge cases.

DB-based Error Signals. DB-based signals are designed to identify semantic misalignment and the inherent ambiguity within the data. In Table 1, we present examples of incorrect and correct SQL query pairs based on NL questions in the BIRD benchmark, highlighting the specific SQL clauses targeted by each signal. We design these DB-based signals by analyzing real-life and benchmark SQL queries such as TPC-DS[6], Redset (van Renen et al., 2024), BIRD (Li et al., 2023), etc. These signals can be efficiently obtained, without using LLMs, by (1) executing a subquery from the SQL query (e.g., *Empty Predicate, Abnormal Result*), (2) checking (meta-)data information from the underlying database (e.g., *Suboptimal Join Tree, Value Ambiguity*), or (3) leveraging general heuristics from the above query workloads (e.g., *Unnecessary Subquery*). Further details regarding all DB-based error signals are provided in Appendix A.1.

[5]Question Clause Linking refers to the mappings between the entities and expressions in an NL question to their corresponding clauses in the SQL query.

[6]`https://www.tpc.org/tpcds/`

Table 1: DB-based Error Signals.

Signal Name	Incorrect SQL Query	Correct SQL Query	SQL Clause
Abnormal Result (BIRD id=340)	SELECT c.id FROM cards c WHERE c.cardKingdomFoilId = c.cardKingdomId AND c.cardKingdomId IS NOT NULL AND c.hasFoil = 1 AND c.isFullArt = 0 AND c.isOversized = 0 AND c.isPromo = 0; *Output Size: 0*	SELECT id FROM cards WHERE cardKingdomFoilId IS NOT NULL AND cardKingdomId IS NOT NULL; *Output Size: 25061*	WHERE
Empty Predicate (BIRD id=223)	SELECT c.atom_id, c.atom_id2 FROM connected c JOIN bond b ON c.bond_id = b.bond_id WHERE b.bond_id = 'TR_000_2_5'; *'TR_000_2_5' is not in b.bond_id, should be 'TR000_2_5'*	SELECT T.atom_id FROM connected AS T WHERE T.bond_id = 'TR000_2_5';	WHERE
Incorrect Filter in Subquery (BIRD id=612)	SELECT Name FROM badges WHERE UserId = (SELECT Id FROM users WHERE DisplayName = 'Pierre');	SELECT Name FROM badges WHERE UserId IN (SELECT Id FROM users WHERE DisplayName = 'Pierre');	SUBQUERY
Incorrect GROUP BY (BIRD id=30)	SELECT s.City, f.Enrollment (K-12) FROM frpm f JOIN schools s ON f.CDSCode = s.CDSCode GROUP BY s.City, f.'Enrollment (K-12)' ORDER BY SUM(f.'Enrollment (K-12)') ASC LIMIT 5;	SELECT T2.City FROM frpm AS T1 INNER JOIN schools AS T2 ON T1.CDSCode = T2.CDSCode GROUP BY T2.City ORDER BY SUM(T1.'Enrollment (K-12)') ASC LIMIT 5;	GROUP BY
Incorrect Join Predicate (BIRD id=109)	SELECT (SELECT COUNT(DISTINCT client.client_id) FROM client INNER JOIN account ON client.client_id = account.account_id INNER JOIN district ON account.district_id = district.district_id WHERE district.a2 = 'Jesenik' AND client.gender = 'F') AS num_female_clients;	SELECT COUNT(T1.client_id) FROM client AS T1 INNER JOIN district AS T2 ON T1.district_id = T2.district_id WHERE T1.gender = 'F' AND T2.A2 = 'Jesenik';	ON
Suboptimal Join Tree (BIRD id=162)	SELECT d.a3 FROM client c JOIN disp di ON c.client_id = di.client_id JOIN account a ON di.account_id = a.account_id JOIN district d ON a.district_id = d.district_id WHERE c.client_id = 3541 LIMIT 1;	SELECT T1.a3 FROM district T1 INNER JOIN client T2 ON T1.district_id = T2.district_id WHERE T2.client_id = 3541;	FROM JOIN
Table Similarity (BIRD id=995)	SELECT AVG(r.points) AS avg_score FROM results r JOIN drivers d ON r.driverId = d.driverId JOIN races ra ON r.raceId = ra.raceId WHERE d.forename = 'Lewis' AND d.surname = 'Hamilton' AND ra.raceId IN (SELECT raceId FROM races WHERE name LIKE '%Turkish Grand Prix%');	SELECT AVG(T2.points) FROM drivers AS T1 INNER JOIN driverStandings AS T2 ON T1.driverId = T2.driverId INNER JOIN races AS T3 ON T3.raceId = T2.raceId WHERE T1.forename = 'Lewis' AND T1.surname = 'Hamilton' AND T3.name = 'Turkish Grand Prix';	SELECT FROM
Unnecessary Subquery (BIRD id=349)	SELECT (SELECT c.name FROM cards c WHERE c.uuid = (SELECT uuid FROM rulings)) AS card_name, (SELECT c.artist FROM cards c WHERE c.uuid = (SELECT uuid FROM rulings)) AS artist, (SELECT c.ispromo FROM cards c WHERE c.uuid = (SELECT uuid FROM rulings)) AS is_promo;	SELECT T1.name, T1.artist, T1.isPromo FROM cards AS T1 INNER JOIN rulings AS T2 ON T1.uuid = T2.uuid WHERE T1.isPromo = 1 GROUP BY T1.artist ORDER BY COUNT(DISTINCT T1.uuid) DESC LIMIT 1	SUBQUERY
Value Ambiguity (BIRD id=367)	SELECT c.'artist' FROM 'cards' c JOIN 'foreign_data' f ON c.'uuid'=f.'uuid' WHERE c.'watermark'='phyrexian' AND c.'artist' IS NOT NULL GROUP BY c.'artist';	SELECT T1.artist FROM cards AS T1 INNER JOIN foreign_data AS T2 ON T1.uuid = T2.uuid WHERE T2.language = 'Phyrexian';	SELECT WHERE

Table 2: LLM-based Error Signals.

Signal Name	Incorrect SQL Query	Correct SQL Query	SQL Clause
Column Ambiguity (BIRD id=50)	SELECT s.School, s.StreetAbr FROM satscores sat JOIN schools s ON sat.cds = s.CDSCode ORDER BY sat.AvgScrMath DESC LIMIT 1 OFFSET 5;	SELECT T2.MailStreet, T2.School FROM satscores AS T1 INNER JOIN schools AS T2 ON T1.cds = T2.CDSCode ORDER BY T1.AvgScrMath DESC LIMIT 5, 1;	SELECT
Evidence Violation (BIRD id=463)	Evidence: set of cards with "Angel of Mercy" in it refers to name = 'Angel of Mercy' SELECT COUNT(*) FROM set_translations WHERE setCode = 'UNH';	SELECT COUNT(DISTINCT translation) FROM set_translations WHERE setCode IN (SELECT setCode FROM cards WHERE name = 'Angel of Mercy') AND translation IS NOT NULL;	Any clause identified by LLM
Insufficient Evidence (BIRD id=215)	Q: How many atoms with iodine and sulfur type elements are there in single bond molecules? Evidence: with iodine element refer to element = 'i'; with sulfur element refers to element = 's'; single type bond refers to bond_type = '-'; *It is not clear what "single bond molecules" refers to.*	SELECT COUNT(DISTINCT CASE WHEN T1.element = 'i' THEN T1.atom_id ELSE NULL END) AS iodine_nums, COUNT(DISTINCT CASE WHEN T1.element = 's' THEN T1.atom_id ELSE NULL END) AS sulfur_nums FROM atom AS T1 INNER JOIN connected AS T2 ON T1.atom_id = T2.atom_id INNER JOIN bond AS T3 ON T2.bond_id = T3.bond_id WHERE T3.bond_type = '-';	Any clause identified by LLM
Incorrect Question Clause Linking (BIRD id=305)	Q: List the top 10 players' names whose heights are above 180 in descending order of average heading accuracy. SELECT p.player_name FROM Player p JOIN Player_Attributes pa ON p.player_api_id = pa.player_api_id WHERE p.height > 180 ORDER BY pa.heading_accuracy DESC LIMIT 10;	SELECT t1.player_name FROM Player AS t1 INNER JOIN Player_Attributes AS t2 ON t1.player_api_id = t2.player_api_id WHERE t1.height > 180 GROUP BY t1.id ORDER BY CAST(SUM(t2.heading_accuracy) AS REAL) / COUNT(t2.'player_fifa_api_id') DESC LIMIT 10;	Any clause identified by LLM
LLM Self-Check (BIRD id=365)	SELECT p.Paragraph_Text FROM Paragraphs p JOIN Documents d ON p.Document_ID = d.Document_ID WHERE d.Document_Name = 'Welcome to NY';	SELECT T1.paragraph_text FROM Paragraphs AS T1 JOIN Documents AS T2 ON T1.document_id =T2.document_id WHERE T2.document_name = 'Customer reviews';	Any clause identified by LLM

[1] Question clause linking is the process of linking entities and expressions in a user question to the corresponding clauses in a SQL query.

LLM-based Error Signals. DB-based signals primarily focus on extracting information from the SQL and the underlying data. However, for semantic errors due to insufficient evidence or LLM hallucination, we need to consider both the question and the SQL query simultaneously and understand the LLM's reasoning process. Therefore, SQLENS introduces LLM-based error signals that dig into question ambiguity and the LLM's reasoning process. The signals are listed in Table 2. Further details regarding all error signals are provided in Appendix A.2 and A.3.

3.2 AGGREGATING ALL SIGNALS USING WEAK SUPERVISION

The signals we collect are noisy, in that any signal may mistakenly flag correct SQL clauses as erroneous and vice versa. For example, the *empty predicate* signal, as shown in Figure 2, is to check whether the value "*jesenik*" is in the column "*A2*". However, it can lead to false positives if the absence of value in the column domain is intended. Therefore, it is crucial to consider these diverse signals collectively, for a specific application or database, when annotating the semantic correctness of a generated SQL query, thereby mitigating the impact of noise. To do this, we leverage a weak-supervision based framework that aggregates multiple sources of noisy labels – called labeling functions (LFs) – to approximate the true labels. These LFs can be heuristics or rules, each contributing partial information about the target label. Weak supervision not only leverages the collective wisdom of the LFs similarly to majority voting, but also goes further by learning the accuracy and correlations of these LFs (Ratner et al., 2017).

In the context of SQLENS, our diverse error signals are essentially the LFs, identifying potential issues with a SQL query but not providing a definitive verdict on its correctness. By combining these noisy error signals using weak supervision, we can infer the correctness of SQL queries, even in the absence of ground truth labels. Specifically, an error signal s maps SQL clauses in q to potential semantic errors. To determine whether any problematic SQL clauses are identified, we apply the LF λ_s, which converts $s(q)$ into a variable $\mathcal{I}$. Formally,

$$\mathcal{I} = \lambda_s(q) = \begin{cases} 1 & \text{if } |s(q)| > 0 \\ -1 & \text{if } |s(q)| = 0 \end{cases}$$

$\mathcal{I} = 1$ indicates that the error signal s has detected at least one problematic SQL clause, suggesting that the SQL is likely incorrect, while $\mathcal{I} = -1$ indicates that no issues are found. Note that error signals are designed to identify potential errors and can only suggest that a SQL query is incorrect. Relying solely on negative labelers in weak supervision can result in limited coverage, leaving the correctness of many SQL queries uncertain from labeling. To fully assess the correctness of a SQL, we also develop three positive labelers, which are derived from combinations of error signals to label a SQL as correct (i.e., $\mathcal{I} = 0$).

1. λ_{all}: labels a SQL query as correct if no error signals are detected.
2. λ_{db}: labels a SQL query as correct if no database-based signals are detected.
3. λ_{llm}: labels a SQL query as correct if no LLM-based signals are detected.

For a SQL query q, we have negative labelers derived from error signals that label a SQL as incorrect (1) and three positive labelers that label a SQL as correct (0). This results in a decision vector $\Lambda_q = \langle \lambda_{s_1}(q), ..., \lambda_{s_n}(q), \lambda_{all}(q), \lambda_{db}(q), \lambda_{llm}(q) \rangle$.

The goal of weak supervision is to learn a generative model, often called a label model, that estimates the joint distribution $p(\Lambda, Y)$, where Y represents the unobserved true labels and Λ denotes the observed noisy labels. The model's objective is to find the parameters θ that best describe the distribution by maximizing the likelihood of observing the labels provided by the LFs. To train the model without true labels, SQLENS minimizes the negative log marginal likelihood:

$$\theta_{opt} = \arg\min_{\theta} - \log \sum_{Y} p_\theta(\Lambda, Y)$$

Since the true labels Y are unknown, the model sums over all possible values that Y could take (Ratner et al., 2017). Using the generative model, we obtain probabilistic labels, $p(Y|\Lambda)$, representing the likelihood of a SQL query's correctness. SQLENS then uses these probabilistic labels to train a classifier that predicts the semantic correctness of SQL queries.

4 EXPERIMENTAL RESULTS

4.1 EXPERIMENTAL SETUP

Datasets. We evaluate our approach on the dev sets of two widely used text-to-SQL benchmarks: BIRD (Li et al., 2023) and Spider (Yu et al., 2018). These datasets provide a correspondence between the NL questions and the ground truth SQL queries.

Table 3: Statistics of generated SQL queries.

Approach	Benchmark	Accuracy 1[1]	Accuracy 2[2]	# SQL Queries	# Incorrect	# Syntax Error	# Semantic Error
Vanilla	BIRD	55.41	59.07	1534	684	95	589
DINSQL	BIRD	35.53	39.49	1534	989	154	835
MACSQL	BIRD	58.87	60.04	1534	631	30	601
CHESS	BIRD	67.60	67.91	1534	497	7	490
Vanilla	Spider	79.11	79.65	1034	216	7	209
DINSQL	Spider	76.31	77.66	1034	245	18	227
MACSQL	Spider	78.92	79.69	1034	218	10	208

[1] Accuracy over all queries [2] Accuracy over queries without syntax errors

For each dataset, we employ multiple text-to-SQL approaches, including Vanilla using a basic text-to-SQL prompt (See Appendix A.3.1), DIN-SQL (Pourreza & Rafiei, 2023), and MAC-SQL (Wang et al., 2024), to generate SQL queries. Claude 3 (Anthropic, 2024) is configured as the backbone LLM for all these methods. In addition, we also directly use the SQL queries on the BIRD dev set, provided by the authors of CHESS[7] (Talaei et al., 2024).

Statistics of Generated SQL Queries. Table 3 presents the statistics of generated SQL queries on two benchmarks using the above mentioned text-to-SQL approaches. The input to SQLENS is a set of generated SQL queries without syntax errors. We evaluate the correctness of a SQL query by comparing its execution result with the ground truth SQL query execution result.

Baselines. We consider the following baselines to identify semantically incorrect SQL queries. In both baselines, we use Claude 3 as the judge model, providing it with the question, database schemas, and the predicted SQL.

• LLM Self-Evaluation (Bool). Kadavath et al. (2022) found that LLMs can evaluate the validity of their own answers and are well-calibrated on True/False questions. In this baseline, we ask the LLM to determine whether the predicted SQL correctly answers the user question. The prompt we use is provided in Appendix A.3.2.

• LLM Self-Evaluation (Prob). Tian et al. (2023) showed that LLMs produce well-calibrated verbalized probabilities as confidence scores for their answers. In this baseline, we ask the LLM to output the probability (0.0 to 1.0) that the SQL is correct. The prompt is provided in Appendix A.3.3.

• SQLENS with Supervised Learning. We manually annotated a set of training data where the correctness of SQL queries has been evaluated against the ground truth. Each SQL query in the training set is associated with a decision vector generated by the SQLENS's LFs and a ground truth label g indicating whether the query is correct. The goal is to train a classifier $\mathcal{F}$ to predict the correctness of a SQL query based on the labeler outputs. We utilize AutoGluon (Erickson et al., 2020), an automated machine learning framework, to obtain the best performing classification model.

Metrics. We evaluate the performance of SQLENS using the following metrics: (1) Accuracy, (2) Area under the ROC Curve (AUC), and (3) Precision/Recall/F1. Accuracy and AUC assess the overall prediction power in determining if the predicted SQL correctly answers the question. Precision, recall and F1 evaluates the ability to identify semantically incorrect SQL queries. Note that relying solely on accuracy can be misleading because state-of-the-art text-to-SQL approaches have reasonable accuracy on Spider and BIRD, which means blindly predicting all SQL queries as correct can still have a high accuracy. Therefore, it is crucial to also consider precision, recall, and F1 in detecting incorrect SQL queries when evaluating the performance of different approaches.

4.2 OVERALL EFFECTIVENESS OF SQLENS

We first evaluate the overall effectiveness of SQLENS in predicting the correctness of SQL queries. Table 4 presents the results on BIRD, while the results on Spider are reported in Table 7 in Appendix A.4. We use 5-fold cross-validation to compute the statistics for all approaches.

On BIRD, SQLENS outperforms all baselines in terms of Accuracy, Recall and F1, indicating better performance in identifying erroneous SQL queries. While the LLM Self-Evaluation (Prob) achieves the highest precision on MAC-SQL, it does so at the expense of having the lowest recall. On Spider, we observe similar results, with SQLENS outperforming LLM self-evaluation methods in terms of both recall and F1 score (Table 7). A notable observation is that LLM self-evaluation tends to be

[7] https://tinyurl.com/mry73y24

overly confident in the generated SQL queries, leading to low recall in identifying incorrect queries. For instance, LLM Self-Evaluation (Prob) only identifies 5.16% of the incorrect SQL queries on those generated by MAC-SQL.

When aggregating various signals, The use of supervised learning in SQLens yields better accuracy, AUC, and precision compared to the weak supervision method. This advantage arises because supervised learning has access to gold labels during training, making it to more effectively assess the reliability of each signal. The presence of these true labels allows the model to learn which signals are more indicative of correctness, leading to slightly higher overall accuracy and precision.

On the other hand, aggregating all signals through weak supervision results in better recall and F1 scores. Weak supervision does not rely on ground truth labels. Instead, it depends on the agreement and conflicts among signals to gauge their reliability. This approach may result in lower accuracy and precision compared to supervised learning, as it lacks the direct guidance of gold labels. However, weak supervision achieves higher recall and F1 scores by trusting the majority of signals, particularly when they cover different aspects of the SQL queries and do not frequently conflict.

Table 4: Effectiveness of SQLens on BIRD (AUC=✗ when the classification is not threshold-based). We highlight the top two results in bold and mark the top-1 result using †.

	Method	Accuracy	AUC	Precision	Recall	F1
Vanilla	LLM Self-Evaluation (Bool)	60.53 (±2.30)	✗	57.70 (18.90)	10.02 (4.54)	16.97 (7.35)
	LLM Self-Evaluation (Prob)	59.76 (±1.10)	**64.23** (±2.98)	**64.22** (±22.97)	2.72 (±1.89)	5.17 (±3.49)
	SQLens w. Supervised Learning	**66.58**† (±2.56)	**65.12**† (±3.88)	**71.83**† (±9.55)	**31.77** (±7.38)	**43.32** (±6.47)
	SQLens	**64.63** (±1.97)	61.90 (±2.18)	58.11 (±2.80)	**48.74**† (±4.31)	**52.94**† (±3.24)
DIN-SQL	LLM Self-Evaluation (Bool)	61.52 (±1.20)	✗	**86.83** (±2.18)	42.99 (±2.72)	57.43 (±2.20)
	LLM Self-Evaluation (Prob)	49.57 (±1.56)	73.01 (±0.86)	**92.27**† (±5.05)	17.84 (±2.19)	29.83 (±3.09)
	SQLens w. Supervised Learning	**76.96**† (±2.10)	**83.55**† (±2.33)	85.90 (±2.14)	**74.13** (±3.27)	**79.53**† (±2.14)
	SQLens	**75.29** (±2.33)	**81.49** (±1.74)	81.64 (±2.17)	**76.41**† (±3.50)	**78.88** (±2.19)
MAC-SQL	LLM Self-Evaluation (Bool)	61.50 (±1.47)	✗	65.51 (±14.72)	7.83 (±2.16)	13.94 (±3.69)
	LLM Self-Evaluation (Prob)	61.37 (±0.81)	**64.60** (±2.19)	**72.69**† (±12.19)	5.16 (±1.53)	9.61 (±2.76)
	SQLens w. Supervised Learning	**67.09** (±3.56)	**65.07**† (±4.30)	**66.19** (±10.83)	**38.11** (±2.90)	**48.16** (±4.29)
	SQLens	**67.43**† (±4.38)	64.27 (±4.42)	63.27 (±8.64)	**45.10**†(±3.90)	**52.63**† (±5.62)
CHESS	LLM Self-Evaluation (Bool)	67.98 (±1.95)	✗	50.89 (±10.76)	15.71 (±3.96)	23.81 (±5.22)
	LLM Self-Evaluation (Prob)	68.50 (±0.82)	**64.23**† (±1.65)	**61.87** (±13.09)	4.90 (±1.63)	9.03 (±2.87)
	SQLens w. Supervised Learning	**72.10**† (±1.27)	62.95 (±3.23)	**72.43**† (±8.93)	**23.06** (±7.23)	**33.96** (±8.04)
	SQLens	**69.35** (±1.60)	**63.34** (±2.90)	52.54 (±2.91)	**44.69**† (±6.03)	**48.17**† (±4.33)

4.3 Effectiveness of Individual Signals

Table 5 shows detailed performance of individual signals for SQL queries generated by MAC-SQL, DIN-SQL, and CHESS on BIRD. N_w is the number of truly incorrect SQL queries identified by a signal. Detailed results on the other setups including those on Spider, are provided in Appendix A.5.

BIRD Overall Results. For the SQL queries generated by MAC-SQL, 13 out of 14 signals achieve over 60% precision. Notably, *Abnormal Result* identifies 40 incorrect SQL queries with 100% precision, while *Suboptimal Join Tree* identifies the highest number of incorrect queries with ∼62% precision. Signals such as *Empty Predicate* and *Incorrect Join Predicate* are shown to be effective across all three text-to-SQL solutions. On the other hand, some signals, such as *Unnecessary Subquery* and *Insufficient Evidence*, exhibit relatively lower precision on the queries generated by MAC-SQL and CHESS. Their impact on overall accuracy is not significant as they only identify a small number of SQL queries. Notably, the effectiveness of *LLM Self-Check* is more robust compared to other LLM-based error signals. This observation is consistent with the results on LLM Self-Evaluation presented in Table 4.

Spider Overall Results. We observe similar patterns on Spider, although the overall precision of signals is lower than on BIRD. This reduction in precision can be attributed to the higher accuracy of the text-to-SQL approaches on Spider, as shown in Table 3. With nearly 80% accuracy on Spider, only approximately 200 semantically incorrect queries remain for error detection, resulting in a long-tailed distribution of errors.

Table 5: Effectiveness of individual error signals on BIRD.

Signal Name	DIN-SQL			MAC-SQL			CHESS		
DB-based Error Signals	**Precision**	**Recall**	N_w	**Precision**	**Recall**	N_w	**Precision**	**Recall**	N_w
Abnormal Result	99.68	37.01	309	100	6.66	40	98.48	13.27	65
Empty Predicate	96.07	46.83	391	75.81	7.82	47	81.25	10.61	52
Incorrect Filter in Subquery	No queries detected			76	3.16	19	No queries detected		
Incorrect GROUP BY	68.75	5.27	44	66.67	3.0	18	50	0.2	1
Incorrect Join Predicate	100	0.12	1	92.86	2.16	13	100	0.41	2
Suboptimal Join Tree	73.86	15.57	130	62.24	10.15	61	55.13	8.78	43
Table Similarity	73.08	4.55	38	67.31	5.82	35	63.27	6.33	31
Unnecessary Subquery	92.31	1.44	12	62.77	10.15	61	100	0.2	1
Value Ambiguity	66.22	5.87	49	58.49	5.16	31	38.89	5.71	28
LLM-based Error Signals	**Precision**	**Recall**	N_w	**Precision**	**Recall**	N_w	**Precision**	**Recall**	N_w
Column Ambiguity	88.69	17.84	149	75.0	3.49	21	50	4.69	23
Evidence Violation	91.24	14.97	125	87.1	4.49	27	42.86	1.22	6
Insufficient Evidence	82.4	12.34	103	62.07	3.0	18	41.03	3.27	16
LLM Self-Check	86.71	43.0	359	65.28	7.82	47	50.33	15.71	77
Question Clause Linking	87	12.34	103	60.71	5.66	34	53.49	4.69	23

Additional Insights. The precision of *Suboptimal Join Tree* is lower with MAC-SQL and CHESS, because, in some cases, the generated SQL query includes a redundant table in the join tree, which does not affect the final outcome of the query. However, we observe that the optimal join tree identified by this signal often aligns with the ground truth query. Although a suboptimal join tree may not impact the semantic correctness of the results, it can adversely affect query execution performance. *Incorrect Join Predicate* achieves more than 90% precision across all datasets. Overall, DB-based error signals demonstrate higher coverage and precision compared to LLM-based signals.

4.4 CASE STUDY: USING DETECTED ERRORS TO CORRECT SQL QUERIES

SQLENS's error signals are associated with SQL clauses and provide a detailed error report, as depicted in Figure 2. Each error report consists of: (1) *signal description* and the conditions that trigger it; (2) *example(s)* (optional) to clarify the meaning of the signal; (3) *correction instruction* for correcting the SQL query; and (4) *problematic clauses* identified as potential sources of error. Such error report offers valuable insights for addressing the identified errors in SQL queries.

In this case study, we explore the potential of using error reports to automatically correct SQL queries. Specifically, we design an LLM-based SQL correction module that takes as inputs: (1) an NL question, (2) optional external knowledge, (3) the original SQL query generated by a text-to-SQL solution, and (4) an error report generated by SQLENS. The LLM-based correction module produces a new SQL query if any correction is needed. During the correction process, a syntactic error might be introduced into the updated SQL query. To handle this, we follow the common practice (Pourreza & Rafiei, 2023; Lee et al., 2024), which takes the parser error message from a database and iteratively prompts the LLM to revise the SQL query until it is syntactically correct. The prompt template for the SQL correction module can be found in Appendix A.7.

Table 6: Effectiveness of using detected errors to correct SQL queries (MAC-SQL on BIRD).

Signal Name	N_{fix}	N_{break}	N_{net}
DB-based Signals			
Abnormal Result	4	0	4
Empty Predicate	8	1	7
Incorrect Filter in Subquery	9	3	6
Incorrect GROUP BY	5	2	3
Incorrect Join Predicate	5	1	4
Suboptimal Join Tree	16	7	9
Table Ambiguity	2	1	1
Unnecessary Subquery	6	4	2
Value Ambiguity	4	1	3
LLM-based Signals			
Column Ambiguity	4	1	3
Evidence Violation	9	1	8
LLM Self-Check	10	3	7

We evaluate the SQL correction capability of each individual signal. For each signal, we input all SQL queries that a signal flags as problematic into the LLM-based correction module and measure three outcomes: the number of SQL queries successfully corrected (incorrect $\rightarrow$ correct), denoted as N_{fix}; the number of SQL queries that were correct but were made incorrect (correct $\rightarrow$ incorrect), denoted as N_{break}; and the net number of SQL queries corrected, calculated as $N_{\text{net}} = N_{\text{fix}} - N_{\text{break}}$.

Question Clause Linking and *Insufficient Evidence* signals were not included in this experiment as they require additional external information for corrections.

The result is shown in Table 6. Among DB-based signals, *Suboptimal Join Tree* exhibits the highest correction power, with $N_{\text{fix}} = 16$ and a net correction of 9 queries. *Empty Predicate* also shows high effectiveness, with a net correction of 7 ($N_{\text{fix}} = 8$, $N_{\text{break}} = 1$). Among the LLM-based signals, *Evidence Violation* performs the best, achieving a net correction of 8 queries. Overall, all signals result in a positive net number of corrected SQL queries, underscoring the efficacy of SQLENS.

Additionally, we calculate the total net number of successfully corrected SQL queries, $Q_{\text{net}} = |\bigcup_{i=1}^{n} S_{\text{fix}}^{i} - \bigcup_{i=1}^{n} S_{\text{break}}^{i}|$, where $Q_{\text{fix}} = |\bigcup_{i=1}^{n} S_{\text{fix}}^{i}|$ denotes the total number of queries corrected successfully and $Q_{\text{break}} = |\bigcup_{i=1}^{n} S_{\text{break}}^{i}|$ is the total number of SQL queries that were correct but became incorrect due to the LLM-based correction module. Here S_{fix}^{i} and S_{break}^{i} represent the set of SQL queries successfully corrected and broke by the signal s_i, respectively. For the 1,534 SQL queries generated by MAC-SQL on BIRD, we found $Q_{\text{fix}} = 67$, $Q_{\text{break}} = 22$, and $Q_{\text{net}} = 45$ (2.9% of total queries). These results demonstrate the potential of SQLENS to further enhance the performance of state-of-the-art text-to-SQL methods.

5 RELATED WORK

Text-to-SQL. Generating accurate SQL from natural language questions (Text-to-SQL) is a long-standing challenge due to the complexities in user question understanding, database schema comprehension, and SQL generation (Quamar et al., 2022; Katsogiannis-Meimarakis & Koutrika, 2023). Recently, large language models (LLMs) have demonstrated significant capabilities in natural language understanding as the model scale increases. LLM-based text-to-SQL solutions (Hong et al., 2024) have emerged. DIN-SQL (Pourreza & Rafiei, 2023) studies how decomposing the text-to-SQL task into smaller sub-tasks can be effective. MAC-SQL (Wang et al., 2024) presents a multi-agent collaborating framework. The selector preserves relevant tables for user questions, the decomposer breaks down user questions into sub-questions and provides solutions, and finally the refiner validates and refines the defective SQL. CHESS (Talaei et al., 2024) introduces a new pipeline that hierarchically retrieves relevant data and context, selects an efficient schema, and synthesizes correct and efficient SQL queries. MCS-SQL (Lee et al., 2024) leverages multiple prompts to explore a broader search space for possible answers and effectively aggregate them. However, the SQL queries generated by these methods can often be semantically incorrect. And such erroneous queries are only detected after being executed. These methods can benefit from our SQLENS by incorporating its fine-grained error detection capability.

LLM Self-Evaluation. As part of ongoing research to improve LLM's reliability and trustworthiness, LLM self-evaluation enables an LLM to assess the quality, accuracy, or relevance of its own response (Geng et al., 2024). Kadavath et al. (2022) explored the self-evaluation capabilities of LLMs. The findings show that LLMs are well-calibrated when answering multiple-choice and true/false questions. This ability improves further when the models consider multiple possible answers before deciding on one. Tian et al. (2023) introduced methods to obtain well-calibrated confidence scores from large language models (LLMs) that have been fine-tuned using reinforcement learning from human feedback (RLHF). Self-RAG (Asai et al., 2024) enhances the factual accuracy of LLMs by combining selective retrieval of external information with self-critiquing mechanisms. The model can decide when to retrieve information and critique its own outputs, leading to more reliable and verifiable results. However, these methods are designed for general tasks, which do not address the unique challenges in text-to-SQL.

6 CONCLUSION

In this paper, we introduce SQLENS, the first framework that exploits information from both the database and the LLM to achieve fine-grained and explainable error detection in text-to-SQL. DB-based signals identify semantic misalignment and the inherent ambiguity within the underlying data, while LLM-based signals consider the question and the SQL query simultaneously and examine the reasoning process of an LLM. SQLENS systematically diagnoses SQL queries at the clause level, aggregating noisy error signals through weak supervision to predict the semantic correctness of a SQL query. SQLENS significantly outperforms LLM self-evaluation methods in identifying semantic errors in SQL queries. Beyond error detection, we also demonstrate the effectiveness of using error signals to fix SQL queries automatically.

7 REPRODUCIBILITY

To ensure the reproducibility of SQLENS, we provide a detailed discussion regarding the experimental setup in Section 4.1, including the backbone LLM, dataset statistics, baselines, etc. The implementation of DB-based error signals is thoroughly described in Appendix A.1. Additionally, the prompts used for LLM-based error signals are available in Appendix A.3. We report additional experimental results in Appendix A.4 and A.5. Lastly, the prompts for the vanilla text-to-SQL approach and the SQL query correction module can be found in Appendix A.3.1 and A.7, respectively.

REFERENCES

Anthropic. Claude 3 model card, 2024. URL `https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf`. Accessed: 2024-08-12.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL `https://openreview.net/forum?id=hSyW5go0v8`.

Nik Bear Brown. Enhancing trust in llms: Algorithms for comparing and interpreting llms. *arXiv preprint arXiv:2406.01943*, 2024.

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data, March 2020. URL `http://arxiv.org/abs/2003.06505`. arXiv:2003.06505 [cs, stat].

Jiahui Geng, Fengyu Cai, Yuxia Wang, Heinz Koeppl, Preslav Nakov, and Iryna Gurevych. A survey of confidence estimation and calibration in large language models. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 6577–6595, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-long.366. URL `https://aclanthology.org/2024.naacl-long.366`.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. Next-generation database interfaces: A survey of llm-based text-to-sql. *CoRR*, abs/2406.08426, 2024. doi: 10.48550/ARXIV.2406.08426. URL `https://doi.org/10.48550/arXiv.2406.08426`.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, Scott Johnston, Sheer El-Showk, Andy Jones, Nelson Elhage, Tristan Hume, Anna Chen, Yuntao Bai, Sam Bowman, Stanislav Fort, Deep Ganguli, Danny Hernandez, Josh Jacobson, Jackson Kernion, Shauna Kravec, Liane Lovitt, Kamal Ndousse, Catherine Olsson, Sam Ringer, Dario Amodei, Tom Brown, Jack Clark, Nicholas Joseph, Ben Mann, Sam McCandlish, Chris Olah, and Jared Kaplan. Language Models (Mostly) Know What They Know, November 2022. URL `http://arxiv.org/abs/2207.05221`. arXiv:2207.05221 [cs].

George Katsogiannis-Meimarakis and Georgia Koutrika. A survey on deep learning approaches for text-to-sql. *VLDB J.*, 32(4):905–936, 2023. doi: 10.1007/S00778-022-00776-8. URL `https://doi.org/10.1007/s00778-022-00776-8`.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. MCS-SQL: Leveraging Multiple Prompts and Multiple-Choice Selection For Text-to-SQL Generation, May 2024. URL `https://arxiv.org/abs/2405.07467v1`.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs, November 2023. URL `http://arxiv.org/abs/2305.03111`. arXiv:2305.03111 [cs].

Liana Patel, Siddharth Jha, Carlos Guestrin, and Matei Zaharia. Lotus: Enabling semantic queries with llms over tables of unstructured and structured data. *arXiv preprint arXiv:2407.11418*, 2024.

Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction, April 2023. URL `https://arxiv.org/abs/2304.11015v3`.

Abdul Quamar, Vasilis Efthymiou, Chuan Lei, and Fatma Özcan. Natural language interfaces to data. *Found. Trends Databases*, 11(4):319–414, 2022. doi: 10.1561/1900000078. URL `https://doi.org/10.1561/1900000078`.

Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment*, 11(3):269–282, November 2017. ISSN 2150-8097. doi: 10.14778/3157794.3157797. URL `https://dl.acm.org/doi/10.14778/3157794.3157797`.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. CHESS: Contextual Harnessing for Efficient SQL Synthesis, June 2024. URL `http://arxiv.org/abs/2405.16755`. arXiv:2405.16755 [cs].

Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher D. Manning. Just Ask for Calibration: Strategies for Eliciting Calibrated Confidence Scores from Language Models Fine-Tuned with Human Feedback, October 2023. URL `http://arxiv.org/abs/2305.14975`. arXiv:2305.14975 [cs].

Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. Why tpc is not enough: An analysis of the amazon redshift fleet. In *VLDB 2024*, 2024. URL `https://www.amazon.science/publications/why-tpc-is-not-enough-an-analysis-of-the-amazon-redshift-fleet`.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL, June 2024. URL `http://arxiv.org/abs/2312.11242`. arXiv:2312.11242 [cs].

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3911–3921, Brussels, Belgium, October 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL `https://aclanthology.org/D18-1425`.

A Appendix

A.1 DB-based Error Signals

SQLens uses the following signals to identify semantic misalignment.

• **Suboptimal join tree** signal is introduced to determine whether a SQL query utilizes the optimal join tree for connecting the required tables to answer an NL question. SQLens first constructs a join graph $G = (\mathcal{D}, \mathcal{J})$ based on the database schema, where $\mathcal{D}$ represents the tables and $\mathcal{J}$ represents the join relationships. Let $\mathcal{D}_{\text{req}} \subseteq \mathcal{D}$ denote the subset of tables required to answer the question. The optimal join tree is defined as the minimum Steiner tree T^* that spans $\mathcal{D}_{\text{req}}$, indicating the minimal set of tables needed for the join. If the SQL query includes more tables than those in $\mathcal{D}_{\text{req}}$, the *Sub-optimal Join Tree* signal is flagged. For example, Figure 4 (right) shows an optimal join tree to connect the tables A and C.

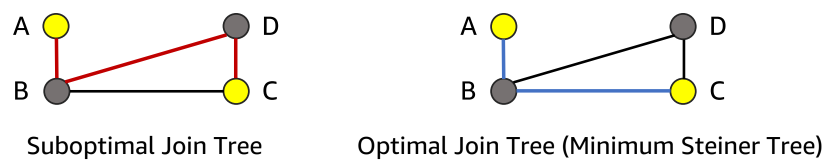

Figure 4: Steiner Trees spanning A and C

To implement this signal, SQLens first identifies the columns in a SQL query that are not involved in the Join clauses. Based on these columns, SQLens searches for the minimum Steiner Tree on the join graph built offline to connect the relevant tables. If the SQL query involves more tables in the join than necessary, compared to the minimum Steiner Tree, SQLens raises *Suboptimal Join Tree* flag. This signal can produce false positives when a suboptimal join strategy does not impact the correctness of the SQL query. For instance, if a query asks for the total sales from a specific store, it can directly select the *sales* column from the *store_sales* table. However, if the query unnecessarily joins the *store_sales* table with a *store_location* table, even though the *store_location* table is not needed to get the correct sales data, the result would still be correct but the join is suboptimal.

• **Incorrect join predicate** signal checks whether a SQL query uses an invalid join predicate. For example, in Figure 2, the predicted SQL incorrectly joins *client_id* with *account_id*, which does not exist in the corresponding schema graph. To implement this signal, SQLens first extracts all the join predicates from the SQL query. It then identifies two types of correct join predicates: (1) using a primary key-foreign key (PK/FK) join explicitly defined in the database schema, and (2) derived from PK/FK joins. Specifically, if two columns C_i and C_j both reference the same primary key C_k (i.e., they refer to the same entity), C_i and C_j can be used in a join predicate. This signal may generate false positives when the PK/FK relationships are not fully documented in the database.

• **Empty predicate** signal detects if there is a predicate within a SQL query that yields an empty result. This signal is useful to detect semantic misalignment errors, including wrong column selection, wrong value usage or wrong comparison operator. SQLens extracts all comparisons between a column and a literal from a given SQL, executes each of them individually and records the output size. If there is a predicate that yields empty rows, this signal is flagged. This signal may lead to false positives when an empty predicate is intentional.

• **Abnormal result** signal detects whether a SQL outputs an abnormal result that does not provide much information. SQLens executes the SQL and records its output. The output is considered abnormal if (1) it is empty, (2) it contains a column full of zeros, or (3) it contains a column full of Nulls. This signal extends beyond the empty predicate to evaluate the entire SQL output. In addition to detecting empty predicates, it can identify empty intermediate execution results, making it effective for catching semantic misalignment in the intermediate steps of a SQL query. This signal may incorrectly flag a SQL query when the abnormal result is intentional, though this scenario is rare in practice.

• **Incorrect filter in subquery** signal detects the problematic filtering in a subquery. Filters in a subquery often follow the pattern *column* = (Select...). When the subquery returns multiple rows, the filter condition becomes ambiguous (e.g., IN or '='), potentially leading to errors. SQLENS uses regular expressions to match this pattern and executes the extracted subquery separately. If it returns more than one row, the signal is flagged. Additionally, SQLite is lenient with SQL semantics, allowing *column* = (SELECT...) to match only the first value returned by the subquery. While the query might still be correct if the first value happens to be the desired one, relying on this behavior is generally considered poor practice in SQL writing.

• **Incorrect GROUP BY** signal detects any standalone GROUP BY clause without accompanying aggregate functions such as MAX, COUNT. A misused GROUP BY clause can change the SQL semantics and lead to an incorrect result. In SQLite, a standalone GROUP BY behaves the same as the DISTINCT operator. While the query may still be correct when using GROUP BY as a substitute for DISTINCT, this approach is generally considered poor practice.

• **Unnecessary subquery** signal indicates if there is an excessive use of subqueries in a SQL query, which leads to inefficiencies, increased complexity, and a higher likelihood of errors. This signal counts the number of subqueries in a SQL query and flags it as problematic if the count exceeds a specified threshold. In our evaluation, this threshold is set to 3. False positives may arise when the subqueries are necessary for performance optimization or specific logic, even though they exceed the threshold.

The following signals are designed to capture the inherent ambiguity within the data.

• **Value ambiguity** signal detects incorrect column selections when a value used in an NL question appears in multiple columns. For example, "New York" can appear in both "state" and "city" columns. To identify ambiguous value, this signal extracts all values used in the SQL and finds columns that contain a used value via an inverted index built offline. If there is an alternative column that is closer to the question semantically, the signal flags the corresponding value as ambiguous. It is possible that the originally selected column in the SQL is correct, as the semantic distance may not always accurately determine which column containing the value is the best candidate to answer the question.

• **Table similarity** signal detects potential errors in table selection by identifying alternative tables with similar structures. It extracts all columns used in the SQL, groups columns by their tables, and searches for other tables that contain the same groups of columns. If such a table is found, it suggests that the alternative table could have been used, indicating a potential mistake in the original chosen table. False positives can occur when the chosen table is actually correct, but another table with a similar structure exists, leading the system to incorrectly flag a possible error.

A.2 LLM-BASED ERROR SIGNALS

• **Evidence violation** signal identifies cases where the generated SQL query contradicts the evidence provided in the question or external knowledge. For example, if the question specifies retrieving rows only about *active employees*, but the SQL query does not include a condition to filter out inactive employees, this would trigger an evidence violation. The prompt used by SQLENS can be found in Appendix A.3.4.

• **Insufficient evidence** signal assesses whether the available evidence is adequate to confirm that the SQL query correctly answers the user's question. For example, if the question lacks a clear explanation of a domain-specific concept, the LLM is prone to hallucination. This signal essentially verifies that the LLM has enough information and context to provide a correct response. The prompt is shown in Appendix A.3.5.

• **Incorrect question clause linking** signal evaluates the LLM's confidence in the generated SQL clauses. This signal first prompts the LLM to map the concepts, entities, and expressions in the user question to the corresponding clauses in the SQL query. For each identified link, the LLM is then asked to indicate its confidence in the generated clause by responding with a simple *yes* or *no*. This signal is flagged when there is at least one clause with low confidence. The prompt can be found in Appendix A.3.6.

• **Column ambiguity** signal identifies whether there are columns in the database that are very similar to those used in the SQL query and could also be used to answer the user's question. This signal suggests that a SQL is prone to wrong column selection. The prompt for this signal is shared in Appendix A.3.7.

The above signals provide specific error causes for the LLM to detect. Given that studies have shown LLMs possess the ability to validate their own answers Kadavath et al. (2022); Tian et al. (2023), we also incorporate a signal that prompts the LLM to provide an overall assessment of its own output.

• **LLM self-check** asks an LLM to determine whether the proposed SQL correctly answers the user question considering the database and any available external knowledge. The prompt can be found in Appendix A.3.2.

False positives can occur for the LLM-based signals when the LLM misinterprets the question, has limited understanding of the specified knowledge, experiences hallucination, or exhibits bias when evaluating its own output.

A.3 Prompts

A.3.1 Prompt for Vanilla text-to-SQL

```
Role: You are an expert SQL database administrator responsible for
↪  crafting precise SQL queries to address user questions.

Context: You are provided with the following information:
1. A SQLite database schema
2. A user question
3. Relevant evidence pertaining to the user's question

Database Schema:
- Consists of table descriptions
- Each table contains multiple column descriptions
- Frequent values for each column are provided

Your Task:
1. Carefully analyze the user question, evidence and the database
↪  schema.
2. Write a SQL query that correctly answers the user question

Format your SQL query using the following markdown:
```sql
YOUR SQL QUERY HERE
```

[Question]
{question}

[Evidence]
{evidence}

[Database Info]
{db_desc_str}

[Answer]
```

A.3.2 Prompt for LLM Self-Check (True/False)

```
You are provided with a SQLite database schema, a user question, and a
↪  proposed SQL query intended to solve the user question. Your task is
↪  to determine whether the proposed SQL correctly answers the user
↪  question. Your answer should be in the form of a JSON object with
↪  two keys: "correct" and "explanation". Provide an explanation only
↪  if the SQL is incorrect.

You need to generate the answer in the following format:

[Answer]:
```json
{{
  "correct": false,
  "explanation": "your explanation of why the SQL is incorrect"
}}
```
Make sure you generate a valid json response.
=================================
Please start answering the following question:
[Question]
{question}. {evidence}
```

````
[Database Info]
{db_desc_str}

[SQL query]
{sql_query}

[Answer]
````

A.3.3 PROMPT FOR LLM SELF-CHECK (PROBABILITY)

````
You are provided with a user question, a SQLite database schema and a
↪  proposed SQL query intended to solve the user question.

Your task is to evaluate the proposed SQL query and provide the
↪  probability that it correctly answers the user question.

Provide this probability as a decimal number between 0 and 1.

You need to generate the answer in the following format:

[Answer]:
```json
{{
  "probability": <the probability between 0.0 and 1.0 that the SQL
  ↪  correctly answers the question>
}}
```
Make sure you generate a valid json response.
=================================
Please start answering the following question:
[Question]
{question}. {evidence}

[Database Info]
{db_desc_str}

[SQL query]
{sql_query}

[Answer]
````

A.3.4 PROMPT FOR EVIDENCE VIOLATION

````

You are provided with a user question and a proposed SQL query that
↪  solves the user question. Your task is to determine whether the
↪  proposed SQL query reflects all the evidence specified in the
↪  question.

Here is a typical example:

==== Example ====

Question] How many users are awarded with more than 5 badges? more than
↪  5 badges refers to Count (Name) > 5; user refers to UserId

[SQL query] SELECT UserId FROM ( SELECT UserId, COUNT(DISTINCT Name) AS
↪  num FROM badges GROUP BY UserId ) T WHERE T.num > 6;

[Answer]:

```json
{{
````
````

```
  "violates_evidence": true
  "explanation": "This SQL query violates the evidence in the question
  ↪  because it counts the number of users with more than 6 badges, not
  ↪  more than 5 badges. Additionally, it uses COUNT(DISTINCT Name)
  ↪  instead of COUNT(Name) as specified in the question.
}}
```
Question Solved. Make sure you generate a valid json response.
===============

Here is new example, please start answering:

[Question] {question}. {evidence}

[SQL query] {sql_query}

[Answer]
```

#### A.3.5 Prompt for Insufficient Evidence

```
You are provided with sqlite database schema, a user question, and a
↪ proposed SQL query that solves the user question. Your task it to
↪ determine whether you have sufficient evidence to determine whether
↪ the SQL answers the user question correctly.

Output a JSON object in the following format. Make sure you generate a
↪ valid json response.
[Answer]
```json
{{
  "insufficient_evidence": true/false
  "explanation": "why the evidence is not sufficient"
}}
```
Question Solved.
===============
Here is new example, please start answering:

[Question] {question}. {evidence}

{db_desc_str}

[SQL query]

{sql_query}

[Answer]
```

#### A.3.6 Prompt for Question-Clause Linking

```
You are given a SQLite database schema, a user question, and a proposed
↪ SQL query intended to address the user's question. Please follow
↪ these steps:

1. Link the concepts, entities, and expressions in the user question to
↪ the corresponding clauses in the SQL query.
2. For each link you have identified, indicate whether you are confident
↪ in the generated clause by answering "yes" or "no."

Output a JSON object in the following format. Make sure you generate a
↪ valid json response.
[Answer]
```json
```

```
{{
    "<(entity in the question, the corresponding SQL clause)>":
    ↪  "<yes/no>"
}}
```
```

Please start answering the following question:
[Question] {question}. {evidence}.

{db_desc_str}

[SQL query] {sql_query}

[Answer]
```

A.3.7 PROMPT FOR COLUMN AMBIGUITY

```
As an experienced and professional database administrator, you are
↪  provided with a SQLite database schema, a user question, and a
↪  proposed SQL query intended to solve the user question. The database
↪  schema consists of table descriptions, each containing multiple
↪  column descriptions.
Your task is to determine whether there are columns in the database that
↪  are very similar to the ones used in the SQL query and could also be
↪  used to answer the user's question.

Here is a typical example:

==== Example ====

[Question] Which state special schools have the highest number of
↪  enrollees from grades 1 through 12? State Special Schools refers to
↪  DOC = 31; Grades 1 through 12 means K-12

[DB_ID]california_schools
[Database Schema]
# Table: frpm
[
  (CDSCode, CDSCode.),
  (Enrollment (K-12), Enrollment (K-12).),
]
# Table: satscores
[
  (cds, cds. Column Description: California Department Schools),
  (sname, school name. Value examples: [None, 'Middle College High',
  ↪  'John F. Kennedy High', 'Independence High', 'Foothill High',
  ↪  'Washington High', 'Redwood High'].),
  (enroll12, enrollment (1st-12nd grade).),
]
# Table: schools
[
  (CDSCode, CDSCode.),
  (DOC, District Ownership Code. Value examples: ['54', '52', '00',
  ↪  '56', '98', '02'].),
]

[SQL query] <SQL> SELECT T2.sname FROM schools AS T1 INNER JOIN
↪  satscores AS T2 ON T1.CDSCode = T2.cds WHERE T1.DOC = '31' AND
↪  T2.enroll12 IS NOT NULL ORDER BY T2.enroll12 DESC LIMIT 1; </SQL>

[Answer]
```json
{{
```
```

```
  "alternative_column": true
  "explanation": "frpm.Enrollment (K-12) can also be used to determine
  ↪  the number of enrollees from grades 1 through 12. This column is
  ↪  very similar to satscores.enroll12 used in the proposed SQL."
}}
```
Question Solved. Make sure you generate a valid json response.
===============

Please start answering the following question.
[Question] {question}. {evidence}

{db_desc_str}

[SQL query] {sql_query}

[Answer]
```
```

A.4 EFFECTIVENESS OF SQLENS ON SPIDER

Table 7: Effectiveness of SQLENS on Spider (AUC=✗ when the classification is not threshold-based.). We highlight the top two results in bold and mark the top-1 result using †.

	Method	Accuracy	AUC	Precision	Recall	F1
Vanilla	LLM Self-Evaluation (Bool)	74.39 (±3.12)	✗	21.59 (±13.00)	9.13 (±5.21)	12.77 (±7.42)
	LLM Self-Evaluation (Prob)	**77.41** (±1.43)	57.42 (±1.30)	13.33 (±17.78)	2.90 (±3.89)	4.77 (±6.38)
	SQLENS w. Supervised Learning	**80.72**† (±1.06)	**61.02**† (±3.28)	**75.79**† (±17.24)	**10.56** (±4.53)	**17.79** (±6.62)
	SQLENS	74.49 (±4.86)	**59.49** (±5.91)	**34.91** (±14.18)	**29.22**† (±11.61)	**31.76**† (±12.75)
DIN-SQL	LLM Self-Evaluation (Bool)	75.69 (±2.46)	✗	40.38 (±11.33)	16.78 (±4.24)	23.64 (±6.01)
	LLM Self-Evaluation (Prob)	**76.58** (±1.23)	58.60 (±1.78)	41.07 (±17.97)	5.73 (±2.23)	9.90 (±3.72)
	SQLENS w. Supervised Learning	**82.48**† (±2.27)	**67.89**† (±3.66)	**73.68**† (±7.45)	**33.57** (±9.14)	**45.51** (±9.91)
	SQLENS	76.38 (±2.75)	**67.14** (±4.30)	**47.18** (±5.76)	**48.09**† (±6.91)	**47.59**† (±6.17)
MAC-SQL	LLM Self-Evaluation (Bool)	76.85 (±1.60)	✗	31.20 (±11.26)	**11.52** (±4.11)	**16.81** (±5.98)
	LLM Self-Evaluation (Prob)	**78.51** (±1.21)	57.81 (±4.42)	**42.52** (±30.00)	5.31 (±2.41)	9.07 (±3.74)
	SQLENS w. Supervised Learning	**79.98**† (±1.39)	**61.59**† (±3.11)	**54.83**† (±21.06)	9.65 (±8.11)	15.10 (±11.29)
	SQLENS	74.41 (±2.24)	**58.99** (±3.19)	35.88 (±5.22)	**32.23**† (±4.33)	**33.88**† (±4.49)

A.5 EFFECTIVENESS OF INDIVIDUAL ERROR SIGNALS ON BIRD AND SPIDER

Table 8: Individual error signal performance (Vanilla+BIRD).

Signal Name	Precision	Recall	N_w
DB-based Signals			
Abnormal Result	98.96	16.13	95
Empty Predicate	85.34	11.88	70
Incorrect Filter in Subquery	No Queries Detected		
Incorrect GROUP BY	40.0	2.38	14
Incorrect Join Predicate	100	1.87	11
Suboptimal Join Tree	45.9	14.26	84
Table Similarity	67.35	5.6	33
Unnecessary Subquery	33.33	0.34	2
Value Ambiguity	53.85	7.13	42
LLM-based Signals			
Column Ambiguity	68.75	1.87	11
Evidence Violation	61.11	1.87	11
Insufficient Evidence	29.03	1.53	9
LLM Self Check	60.82	10.02	59
Question Clause Linking	53.49	3.9	23

Table 9: Individual error signal performance (Vanilla+Spider).

Signal Name	Precision	Recall	N_w
DB-based Signals			
Abnormal Result	60.0	14.35	30
Empty Predicate	50.0	2.39	5
Incorrect Filter in Subquery	100.0	0.48	1
Incorrect GROUP BY	64.29	4.31	9
Incorrect Join Predicate	100.0	5.26	11
Suboptimal Join Tree	42.86	5.74	12
Table Similarity	No Queries Detected		
Unnecessary Subquery	100.0	0.48	1
Value Ambiguity	33.33	1.44	3
LLM-based Signals			
Column Ambiguity	33.33	0.48	1
Evidence Violation	100	1.44	3
Insufficient Evidence	7.14	1.44	3
LLM Self Check	20.65	9.09	19
Question Clause Linking	70.0	3.35	7

Table 10: Individual error signal performance (DIN-SQL+Spider).

Signal Name	Precision	Recall	N_w
DB-based Signals			
Abnormal Result	73.61	23.35	53
Empty Predicate	82.98	17.18	39
Incorrect Filter in Subquery	No Queries Detected		
Incorrect GROUP BY	43.33	5.73	13
Incorrect Join Predicate	92.86	11.45	26
Suboptimal Join Tree	33.33	5.73	13
Table Similarity	No Queries Detected		
Unnecessary Subquery	0	0	0
Value Ambiguity	25.0	1.32	3
LLM-based Signals			
Column Ambiguity	80	1.76	4
Evidence Violation	80	1.76	4
Insufficient Evidence	40	7.93	18
Question Clause Linking	53.33	3.52	8
LLM Self Check	39.58	16.74	38

Table 11: Individual error signal performance (MAC-SQL+Spider).

Signal Name	Precision	Recall	N_w
DB-based Signals			
Abnormal Result	58.82	14.42	30
Empty Predicate	52.94	4.33	9
Incorrect Filter in Subquery	100	1.44	3
Incorrect GROUP BY	50	3.37	7
Incorrect Join Predicate	100	3.85	8
Suboptimal Join Tree	38.71	5.77	12
Table Similarity	No Queries Detected		
Unnecessary Subquery	33.33	0.48	1
Value Ambiguity	25.0	1.44	3
LLM-based Signals			
Column Ambiguity	0	0	0
Evidence Violation	58.33	3.37	7
Insufficient Evidence	18.92	3.37	7
LLM Self Check	31.17	11.54	24
Question Clause Linking	40	2.88	6

A.6 EXAMPLE ERROR REPORT

A.6.1 EMPTY PREDICATE

```
{
    "signal description": "The SQL query contains a predicate that
    ↪  yields an empty result set.",

    "example": "<sql> SELECT * FROM students WHERE LOWER(students.name)
    ↪  = LOWER('mike')</sql> The predicate students.name = 'mike'
    ↪  yields an empty result set because it is case-sensitive. It
    ↪  should be students.name = 'Mike' or use a case-insensitive
    ↪  comparison.",

    "correction instruction": """
        1. For predicates that yield an empty result set, ensure you are
         ↪  using the correct value with the correct case. Consider
         ↪  using case-insensitive comparisons like LOWER(column_name) =
         ↪  LOWER(value).
        2. There might be typos in a user's question. Consider choosing
         ↪  values that are very similar to the user's question and that
         ↪  do appear in the database.
        3. Review the value examples provided in the database schema to
         ↪  ensure the format of the value is correct.
        4. Verify that the column name is correct. Refer to the database
         ↪  schema to find the correct column name.
        5. Ensure that the schema linking process is accurate, meaning
         ↪  that the entities mentioned in the question are correctly
         ↪  mapped to the corresponding database columns.
    """

    "problematic clauses": {
            "Predicates that yield empty results": [
            "bond.\"BOND_ID\" = 'TR_000_2_5'"
        ]
    }
}
```

A.6.2 SUBOPTIMAL JOIN TREE

```
{
    "signal description": "The SQL query uses more tables than necessary
    ↪  in the join clauses, which may lead to potential errors.",


    "correction instruction": """
       Review and revise the SQL query to include only the essential
        ↪  tables in the join clauses.
    """

    "problematic clauses": {
        "tables used in the JOIN clauses": ["client", "account",
         ↪  "district"],
        "optimal set of tables to join": ["client", "district"],

    }
}
```

A.7 PROMPT FOR SQL QUERY CORRECTION MODULE

```
Role: You are an experienced and professional database administrator
↪  tasked with analyzing and correcting SQL queries that are
↪  potentially wrong.

Context: You are provided with the following information:
1. A SQLite database schema
2. A user question
3. A proposed SQL query intended to answer the user question
4. An error report for the proposed SQL query. The error report suggests
↪  potential errors in the SQL.

Database Schema:
- Consists of table descriptions
- Each table contains multiple column descriptions
- Frequent values for each column are provided

Your Task:
1. Analyze the error report
2. Determine if the SQL query needs to be fixed. You can choose not to
↪  modify the SQL if it is correct.
3. If the proposed SQL is incorrect, generate a correct SQL query to
↪  answer the user question

Instructions:
1. Review the provided information carefully
2. Use SQL format in code blocks for any SQL queries
3. Explain your reasoning and any changes made to the query
4. Avoid using overly complex queries. For example, ... EXISTS (SELECT 1
↪  FROM table WHERE condition) can be substituted with JOIN.

[Question]
{question}

[Evidence]
{evidence}

[Database Info]
{db_desc}

[Old SQL]
```sql
{old_sql}
```

[Error Report]
{error_report}

Now, please analyze the error report, decide whether the SQL needs to be
↪ fixed and generate a correct SQL to answer the user question if you
↪ think the proposed SQL is indeed wrong.

[Correct SQL]
```