TS-SQL: Test-driven Self-refinement for Text-to-SQL

Wenbo Xu[1], Haifeng Zhu[1], Liang Yan[1,4], Chuanyi Liu*[1,2], Peiyi Han*[1,2], Shaoming Duan[2], Jeff Z. Pan[3]

[1]Harbin Institute of Technology (Shenzhen)
[2]Peng Cheng Laboratory
[3]The University of Edinburgh
[4]Inspur Cloud Information Technology Co., Ltd

Abstract

Large Language Model (LLM)-based self-refinement has advanced Text-to-SQL, but it struggles with SQL semantic errors, such as omitted conditions and misinterpreted requirements. This is because self-refinement depends on LLMs' semantic understanding of questions, a process prone to hallucination-induced biases, leading to uncorrectable errors. To solve this problem, we propose **T**est-driven **S**elf-refinement for Text-to-SQL (**TS-SQL**). It leverages a collaborative LLM agent framework to automatically synthesize high-quality test cases, including test data and test code. The test cases are further employed to provide execution feedback for LLM self-refinement towards SQL semantic errors. Rigorous evaluation shows the superiority of TS-SQL: for BIRD-dev, TS-SQL improves at least **6%** over existing SQL self-refinement methods; for Spider-dev, TS-SQL identifies and corrects **131** gold SQL errors, exposing system flaws in benchmark rigor. For reproducibility, we release the modified Spider-dev benchmark to foster further research. [1]

1 Introduction

The prevalence of large language models (LLMs) has amplified the demand for scalable data analysis research in real-world business applications, such as data annotation (Tan et al., 2024), data wrangling (Li and Döhmen, 2024), and Text-to-SQL (Zhu et al., 2024; Shen et al., 2024). Limited by the hallucination and unreliability of LLMs (Pan et al., 2023; Huang et al., 2025), data analysts have to manually check the usability of their output results, which is labor-intensive and time-consuming. Recent studies suggest that incorporating the feedback of self-refinement techniques can substantially improve the usability of LLMs on various tasks (Madaan et al., 2024). Therefore, various LLM self-refinement techniques (Wang et al., 2025) have been applied to detect SQL errors that cause unexpected output, including prompt instruction (Pourreza and Rafiei, 2024; Caferoğlu and Ulusoy, 2024; Talaei et al., 2024; Pourreza et al., 2024) and agent inference (Wang et al., 2023; Askari et al., 2024).

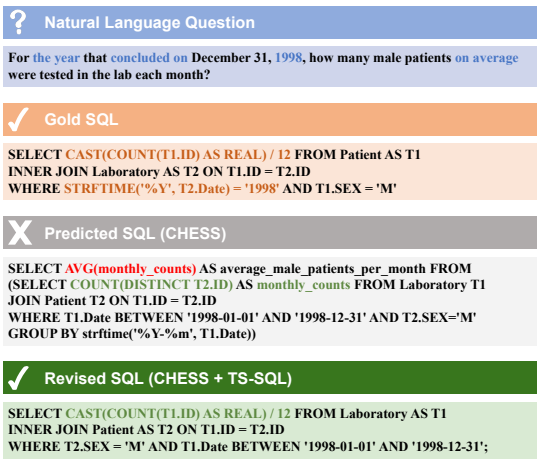

Figure 1: An example of an *SQL semantic error*, where an LLM demonstrates a biased understanding.

Despite the efficacy of these methods, their self-refinement depends on the validity of LLM's semantic understanding of input questions. Indeed, LLM hallucinations can introduce significant biases in this understanding (Qu et al., 2024), making it difficult to address *SQL semantic errors* (Cen et al., 2024; Liu et al., 2025), such as omitting essential conditions or misinterpreting the question requirements. Figure 1 illustrates an example of semantic errors. The LLM insists on interpreting the phrase `'on average'` as the SQL function `'`$AVG(monthly\_counts)$`'`, leading to an incorrect SQL query. This issue arises because LLMs have trouble identifying flaws in their outputs (Kamoi et al., 2024; Tyen et al., 2024).

In contrast, test cases have been used in software engineering to verify whether a system satisfies

*Corresponding authors: liuchuanyi@hit.edu.cn, hanpeiyi@hit.edu.cn

[1]https://github.com/prosperhitfz/TS-SQL

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 2864–2889

specific requirements, including inputs, execution conditions, testing procedures, and expected outcomes (573, 2010). Applying this paradigm to Text-to-SQL not only validates whether the generated SQL aligns with the question requirements, but also provides corrective feedback to help the LLM rectify semantic understanding bias, thereby improving SQL execution accuracy. Therefore, **this paper focuses on generating test cases to examine and provide validation feedback for SQL self-refinement.**

Unlike prior research (Li and Xie, 2024), we argue that the most urgent problem in generating Text-to-SQL test cases is the inability to directly derive the desired outcome from a question, since LLMs are not adept at complex multi-step reasoning when relying solely on natural language inference (Strachan et al., 2024; Dziri et al., 2024). Therefore, **we propose to decompose the test case generation task into two components: generating test data for the question and generating test code for the test data.** This decomposition brings three technical challenges: **(1) Ensuring high coverage and non-duality of test data.** High coverage means the test data should contain all the relevant data involved in the question. Non-duality means the execution result of the test data should correspond to the test code uniquely. Violating either property results in no testing ability of the test case. For example, the test data is empty or omits necessary tables. **(2) Ensuring validity of test code.** The test code should be consistent and comply with the question requirements. Since the LLM is sensitive to hyperparameters, there are potential discrepancies across each output. Such inconsistency in the output test code can cause the failure to adequately assess the SQL according to the test data, such as the lack of correspondence between the test code and test data, or wrong logic in the test code. **(3) Ensuring maintainability of test code.** The test code should be simple to read and modify. Without this property, the test code cannot examine SQL due to the failure of modifications, such as generating suggestions only without corrected test code or buggy test code that necessitates manual review.

To address these challenges, we propose test-driven self-refinement for Text-to-SQL (**TS-SQL**). The core objective is to design test cases that verify the accuracy of the generated SQL and provide valuable feedback based on the test results, thus assisting the LLM in rectifying semantic understanding bias and enhancing Text-to-SQL performance. As shown in Figure 2, TS-SQL consists of two key techniques: the **test case generation phase** and the **feedback-driven SQL generation phase.** In the test case generation phase, TS-SQL leverages a collaborative framework with three LLM-based agents to provide high-quality test cases. The data generation agent generates test data through the relevant linking and gold SQL execution. The code generation agent and code inspection agent cooperate to generate corresponding test code with the test data through iterative interactions of the generation-execution-regeneration and voting mechanisms. In the feedback-driven SQL generation phase, TS-SQL refines its SQL output based on the logic of the test case within the execution feedback. This helps rectify LLM's semantic understanding bias and improve SQL accuracy.

We propose four research questions (RQs) to conduct evaluation experiments. On the Spider development set, TS-SQL discovers **131** (**12.67%**) incorrect gold SQL due to benchmark issues (wrong data types, values, and semantic logic), verifying the error-checking capability. On the BIRD benchmark, TS-SQL shows competitive results with an execution accuracy of **70.93%** (**69.20%**) on the development (test) set. Case studies further reveal the superiority of TS-SQL, which is at least **6%** better than other self-refinement techniques in correcting the generated BIRD-dev SQL, with most errors including deduplication, incorrect tables, columns, and filter conditions. Our contributions can be summarized as follows:

1. We propose TS-SQL, which assists the LLM in rectifying semantic understanding bias and self-refining SQL semantic errors.

2. We rectify **12.67%** of incorrect gold SQL queries of the Spider development set using the test cases of TS-SQL and release a more rigorous version of the modified Spider-dev set.

3. Experiments demonstrate the competitive performance of TS-SQL, which improves at least **6%** compared to state-of-the-art SQL self-refinement methods and achieves an accuracy of **70.93%** and **69.20%** on the BIRD development and test set.

2 Related Work

2.1 Test Case for LLM-based Code Generation

In software engineering code generation tasks, test cases are typically included in problem descriptions to verify the correctness of generated code. Existing LLM-based code generation research leverages

test cases in two key ways. Some methods directly utilize the provided test cases to correct code errors. CYCLE (Ding et al., 2024) trains the code language model to better fix errors in the generated code based on available feedback, including test case execution results. RLEF (Gehring et al., 2024) uses an end-to-end reinforcement learning approach to teach models how to exploit execution feedback for code generation. Some methods try to generate new test cases to examine the output code. AgentCoder (Huang et al., 2023) designs the Test Design Agent to generate test cases, thus enriching available test cases and enhancing the feedback effectiveness. LDB (Zhong et al., 2024) captures detailed test case feedback by partitioning the code into basic blocks and tracking intermediate variable values after each block, allowing a thorough code examination. MGDebugger (Shi et al., 2024) employs a bottom-up, layered strategy for debugging. It decomposes the code into sub-functions and tests each sub-function by generating test cases from the code's provided test cases.

Unfortunately, such methods cannot be applied directly to Text-to-SQL tasks because the answer to the question can only be obtained indirectly through SQL queries from a database. To introduce test cases for Text-to-SQL, our method entails the partitioning of the test case generation into test data and test code generation, thus guiding the LLM to self-refine SQL errors with generated test cases.

2.2 Self-refine for LLM-based Text-to-SQL

Existing studies have discussed the self-refinement technique in helping LLMs get better results across different tasks (Chen et al., 2023; Madaan et al., 2024). Therefore, various LLM-based Text-to-SQL research seeks to further enhance task performance via self-refinement, including two categories: **prompt instruction** and **agent inference**. Prompt instruction methods directly prompt the LLM to self-refine utilizing natural language constraints. DIN-SQL (Pourreza and Rafiei, 2024) and E-SQL (Caferoğlu and Ulusoy, 2024) provide the LLM with guidance and instructions in the input prompt for SQL self-correction. CHESS (Talaei et al., 2024) and CHASE-SQL (Pourreza et al., 2024) ask the LLM to examine and correct the logical reasonability of the generated SQL query. Agent inference methods construct LLM agents to self-refine, which utilizes the tool usage ability of LLMs. MAC-SQL (Wang et al., 2023) designs the refiner agent by executing the SQL query and checking SQLite syntax errors. MAGIC (Askari et al., 2024) designs three specialized agents (manager, correction, and feedback agent) to iteratively generate and refine a self-correction guideline tailored to LLM mistakes. SQLFixAgent (Cen et al., 2024) designs a three-agent collaboration framework (SQLRefiner, SQLReviewer, and QueryCrafter) to address syntax errors, semantic errors, and runtime errors. Li et al. (Li and Xie, 2024) initially present test cases for re-ranking, aiming to select SQL queries with expected execution results. The construction of test cases involves filtering subtables from the database and using LLMs to predict the execution results of SQL within the sub-tables.

However, the shortcoming of existing methods is that their refinement process depends on LLM's semantic understanding of questions without verification. If the LLM exhibits bias in understanding the question semantics, these methods will encounter difficulties in rectifying SQL semantic errors. Unlike them, our method constitutes a novel self-refinement technique that introduces test cases to verify the accuracy of the generated SQL. It provides valuable feedback to assist the LLM in rectifying its own semantic understanding bias towards questions and self-refining its generated SQL.

3 TS-SQL

Current LLM-based Text-to-SQL methods struggle to self-refine semantic errors in their generated SQL queries, primarily due to hallucination-induced biases of LLMs in understanding question semantics. In response to this issue, we propose TS-SQL, whose objective is to design test cases that verify the accuracy of the generated SQL and provide valuable feedback based on the test results, thus assisting the LLM in rectifying semantic understanding bias and enhancing Text-to-SQL performance. **Our hypothesis is that the test cases generated by TS-SQL play a vital role in identifying SQL semantic errors, despite the fact that they cannot be completely accurate in reflecting the semantic meaning of the original question.** Therefore, it poses a significant challenge to generate high-quality test cases that are semantically aligned with the questions as much as possible. This includes the necessity of ensuring high coverage and non-duality of test data, as well as the validity and maintainability of the test code. As shown in Figure 2, TS-SQL designs two key techniques to generate high-quality test cases for Text-to-SQL self-refinement effectively:

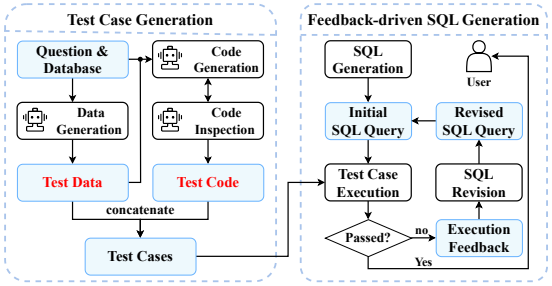

Figure 2: Overall framework of TS-SQL, encompassing two core phases: test case generation and feedback-driven SQL generation.

1. **Test case generation phase.** This phase generates high-quality test cases for Text-to-SQL through three-LLM-based agent collaboration. The data generation agent generates test data based on the question and database schema. The code generation agent and code inspection agent generate the corresponding test code based on the test data, which ensures the reliability of the test case through multiple interactions of generation and inspection.
2. **Feedback-driven SQL generation phase.** This phase utilizes execution feedback derived from the test cases to improve the accuracy of SQL generation. In the SQL generation session, existing LLM-based Text-to-SQL methods are introduced to provide generation guidelines and avoid potential errors. In the SQL revision session, the execution feedback is provided to the LLM for comparison with the generated SQL and the test case, thus determining the correctness of the generated SQL. This further enables the revision of the SQL in accordance with the logic of the test case, thereby obtaining the desired solution to the question.

3.1 Test Case Generation Phase

To help LLMs correct semantic errors in SQL queries, the generated test cases must align with the requirements of the original natural language question. However, Text-to-SQL test case generation is unique because question answers cannot be directly derived from databases (unlike standard unit tests). Besides, the generated SQL requires complex logical reasoning based on the input question. This leads to the difficulty in ensuring the accuracy of LLM in generating test cases since LLM is not adept at completing complex multi-step reasoning operations (Strachan et al., 2024; Dziri et al., 2024). In comparison to reasoning with natural language text in a predefined format, the support of code for control and data flow allows LLMs to solve complex tasks with their pre-trained programming knowledge (Wang et al., 2024). Therefore, we divide the test case generation for Text-to-SQL into two principal components: generating test data for the question and generating test code for the test data. The former requires LLM to generate test data with Python code in the format of `pandas.dataframe`. The latter requires LLM to generate test code (Python code snippets) according to the test data.

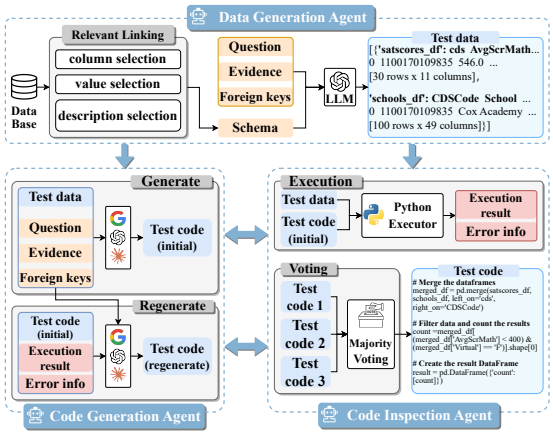

Figure 3: Pipeline of the test case generation phase, involving three collaborative LLM agents: data generation, code generation, and code inspection.

Specifically, the goal of the test case generation phase is to generate test cases that can be used to assess the correctness of SQL. This poses two technical challenges. First, the generated test data should encompass all data involved in the natural language question (**high coverage**), and ensure that the execution result of the test data corresponds to the test code uniquely (**non-duality**). Second, the generated test code should be straightforward to test and comply with the original requirements of the question (**validity**), as well as simple to read and modify (**maintainability**). As shown in Figure 3, this phase comprises three LLM agents for data generation, code generation, and code inspection.

3.1.1 Data generation agent

To ensure that the generated test data is high coverage, we design relevant linking to obtain the database schema associated with the question, which consists of three parts (Figure 3):

1. **Relevant column selection.** We leverage the LLM to extract keywords in the question and select column names $cols$ relevant to each keyword kw from the database.
2. **Relevant content value selection.** For each keyword kw in the question, we choose the top 5 database content values $vals = \{val_1, ..., val_5\}$

with Levenshtein distance smaller than 0.3. Next, we use OpenAI text-embedding-3-small model[2] to obtain the embedding vectors Emb_{kw}, Emb_{val_i} of each question keyword kw and database content value val_i, respectively. Then we calculate the dot product between the embedding vectors and select the highest-ranked content value.

$$vals = \{val_i | Lev(kw, val_i) \leq 0.3\} \quad (1)$$

$$sel\_val = val | max(Emb_{kw} \cdot Emb_{vals} \geq 0.6) \quad (2)$$

3. **Relevant database description selection.** Similarly, we obtain the Euclidean (L2) distance of the embedding vectors Emb_{kw}, Emb_{des} for each keyword kw and description des. Then we select the highest-ranked database description.

$$sel\_des = des | max(L2(Emb_{kw}, Emb_{des})) \quad (3)$$

To optimize the speed of the relevant linking in huge databases, we index database content values using Locality Sensitive Hashing (LSH) and retrieve database descriptions using the embedding vector database (Talaei et al., 2024).

After the relevant linking, we provide the relevant information, foreign keys, question, and evidence as inputs to LLM, and output test data in `pandas.dataframe` format. Finally, the result is used to execute the gold SQL to check non-duality, ensuring that the test data and the gold SQL are aligned without nulls, errors, or other problems.

3.1.2 Code generation agent

As shown in Figure 3, the code generation agent includes two parts to guarantee the validity of the test code: generation and regeneration. During the generation process, the agent accepts the question, evidence, foreign keys, and test data (generated by the data generation agent) as input and outputs a test code for interaction with the code inspection agent. During the regeneration process, the agent receives the test code execution results and error messages from the code inspection agent as supplementary input and adjusts the test code to continue interacting with the code inspection agent.

Previous research (Jiang et al., 2024; Du et al., 2024) shows the potential of using diverse LLMs to improve the accuracy of code generation. In light of these findings, we introduce three LLMs to optimize the generated test code as much as possible: Gemini 1.5 pro, GPT-4o, and Claude 3.5 sonnet.

[2]https://platform.openai.com/docs/models/embeddings

3.1.3 Code inspection agent

As shown in Figure 3, the code inspection agent includes two parts to ensure the maintainability of the test code: execution and voting. In the execution process, the agent uses test data to evaluate the test code output to ensure that the test code is executed correctly and contains natural language comments that facilitate comprehension of the code. If the execution result is anomalous, the agent will output the execution result and error message to the code generation agent for further interaction. In the voting process, the agent combines the test code generation results of the three LLMs and selects the test code with the same generation result as the final result of the test case. If the generation results are different, we default to output the test code of Gemini 1.5 pro to mitigate probable bias.

It should be noted that we have tried two distinct voting methods: majority voting and fine-tuned LLM selection. Future work includes the exploration to enhance the performance of test case generation through the design of a voting mechanism, such as self-consistency (Wang et al., 2022), LLM-based agent committee review (Zhang et al., 2024). We will discuss the performance impact of the voting mechanisms in Section 4.3.2.

3.2 Feedback-driven SQL Generation Phase

Existing self-refinement techniques for LLM-based Text-to-SQL only provide coarse-grained feedback, which hinders LLMs from identifying the root cause of SQL errors, thus necessitating a novel test case-driven self-refinement approach. The technical challenge is to determine how the test cases can be used to provide information that is genuinely useful for SQL self-refinement. Although the generated test cases can express the semantic information correctly, there is no guarantee that the SQL generated by the same LLM will pass the test case in the actual execution. This discrepancy between the test case generation and the SQL generation highlights the different aspects of the LLM's capabilities in code generation.

To overcome this obstacle, we combine the test case verification logic and relevant information as a supplement to improve the generation-execution-revision framework. As shown in Figure 2, the feedback-driven SQL generation phase consists of 2 primary sessions: SQL generation and SQL revision. **Algorithm 1** shows the overall workflow. First, we prompt the LLM to generate an initial SQL based on the question, evidence, and database

Algorithm 1: Feedback-driven SQL Generation

Input:
Question, q; Evidence, e;
Database schema, DS; Foreign keys, fk;
Test case, $TC = \{data, code\}$.
Output:
Result SQL, SQL_{res}.

$SQL_{ini} \leftarrow Generate(q, e, DS, fk)$;
for each $i \in [1, max\_interation = 3]$ **do**
  $TC_{ini} \leftarrow Execute(SQL_{ini}|data)$;
  $TC_{code} \leftarrow Execute(code|data)$;
  if $TC_{ini} == TC_{code}$ **then**
    $SQL_{res} \leftarrow SQL_{ini}$;
    break;
  else
    $Feedback \leftarrow \{SQL_{ini}, TC_{ini}, code, TC_{code}\}$;
    $SQL_{ini} \leftarrow Revise(q, e, DS, fk, Feedback)$;
return SQL_{res};

schema (**step 1**, Appendix A.1). Next, we execute the generated SQL and test code on the test data to obtain the SQL execution result and the test case result, respectively (**step 2-4**). If the results are the same, it will be regarded as the final output (**step 5-7**). Otherwise, we start the SQL revision session by integrating the test code, test case result, initial SQL query, and SQL execution result as feedback, which prompts the LLM to refine semantic errors in the generated SQL based on the comparison results and the logic of the test code (**step 8-11**, Appendix A.2). Such process is repeated until the SQL execution result is the same as the test case result or the number of revisions reaches the upper limit.

It's important to note that there is no potential risk of information leakage in this phase. As shown in Appendix A.1 and A.2, the information provided to the LLM during the feedback-driven SQL self-refinement is entirely dependent on the database schema and regenerated test data samples, thereby ensuring that no potentially sensitive information from the original database is exposed.

4 Evaluation

We utilize two datasets for the experiments: **Spider** (Yu et al., 2018) and **BIRD** (Li et al., 2024). The evaluation metric is execution accuracy (**EX**), which judges the correctness of the predicted SQL by comparing their execution results with those of the gold (ground-truth) SQL. Further details are presented in Appendix B. Specifically, we focus on the following research questions (RQs):

RQ0: How effective is TS-SQL in verifying the correctness of gold SQL of existing benchmarks?

RQ1: What is the effectiveness of TS-SQL compared with state-of-the-art Text-to-SQL baselines?

RQ2: Does each main component in TS-SQL contribute to the overall effectiveness?

RQ3: What is the advantage of TS-SQL compared with other self-refinement methods?

4.1 Benchmark Error Checking (RQ0)

To verify the error-checking ability of TS-SQL, we first apply our method to the Spider dev set to examine the gold SQL errors. As shown in Table 1, manual analysis shows that TS-SQL examines 131 (12.67%) incorrect gold SQL queries with 120 correct and 11 wrong test cases, consistent with recent findings that Text-to-SQL benchmarks often lack rigor in complex reasoning tasks (Zheng et al., 2024). According to detailed case studies in Appendix C, the most common errors detected by TS-SQL include wrong values (32.5%), wrong semantic logic (21.7%), and data type errors (18.3%). For complex SQL benchmarks, we also conduct manual checks on the first three databases of the BIRD development set based on the results of Section 4.2. Table 1 reveals that TS-SQL examines 7, 17, and 22 gold SQL errors, separately. According to Figure 14 in Appendix E, the distribution of the detected common error types is slightly different from that of the Spider-dev set, including table join errors, wrong semantic logic, and wrong selected columns. In addition, the error distribution of each database in the BIRD-dev set varies from each other. For example, the table join error and the wrong semantic logic error exist in the 'financial' and 'toxicology' databases, while they do not occur in the 'california_schools' database. In general, TS-SQL can help identify gold SQL errors in existing benchmarks, confirming the validity of the generated test cases and its error-checking effectiveness.

Test Case / Gold SQL		right / wrong	wrong / wrong	Sum
Spider Dev		120	11	131/1034
Bird Dev	california schools	5	2	7/89
	financial	16	1	17/106
	toxicology	21	1	22/145

Table 1: Statistics on reasons for gold SQL failing TS-SQL's test cases (Spider-dev and BIRD-dev subsets, including counts of correct/wrong test cases).

We further provide the modified Spider-dev dataset with a more rigorous gold SQL standard by

correcting the above gold SQL errors of the Spider-dev set. Table 2 shows that TS-SQL achieves an EX of 88.88% on the modified Spider-dev set, outperforming both DIN-SQL (Pourreza and Rafiei, 2024) and MAC-SQL (Wang et al., 2023). It should be noted that the reason why Li et al. (Li and Xie, 2024) have no results is that it is a closed-source LLM-based method that obtains test data by random selection from the original database and directly infers the expected output with the question and test data. Different from this test case-based research, TS-SQL constructs the test case by generating test data and test code, which supports control flow and data flow to help LLM handle multi-step reasoning tasks based on pre-trained programming knowledge (Wang et al., 2024). Therefore, TS-SQL can check the gold SQL error and perform better in SQL self-refinement. In Appendix D, we will analyze why the EX of MAC-SQL and DIN-SQL drop on the modified Spider-dev set. In Appendix F, we will also provide the results on the ordinary and modified datasets of the first three databases in the BIRD-dev set.

Method	Spider Dev	
	EX (Original)	EX (Modified)
GPT-4o	75.92	77.08
MAC-SQL	75.44	75.05
DIN-SQL	81.72	80.75
(Li and Xie, 2024)	80.31	-
TS-SQL	77.76	**88.88**

Table 2: Execution accuracy (EX) of different methods on the original vs. modified Spider-dev sets (TS-SQL outperforms baselines on the refined benchmark).

4.2 Main Results (RQ1)

To validate the effectiveness of TS-SQL for SQL generation, we conduct experiments using the BIRD benchmark. As shown in Table 3, TS-SQL achieves an EX score of 67.60% with majority voting and 70.93% with fine-tuned LLM for test code selection on the BIRD development set, verifying the validity of test case for SQL self-refinement. On the BIRD hold-out test set, TS-SQL achieves 69.20%, demonstrating competitive performance over state-of-the-art Text-to-SQL baselines.

4.3 Ablation Studies (RQ2)

4.3.1 Effectiveness of agents

To evaluate the necessity of each agent in TS-SQL, we conduct ablation experiments. Without the data generation agent (**w/o DG**), we provide the original database schema as input to support test code generation. Without the code inspection agent (**w/o CI**), the generated test code is used directly without inspection. Without the code generation agent (**w/o CG**), the LLM directly generates test cases based on the data and semantic reasoning. In the extreme case where all agents are removed (**w/o all**), test case generation becomes infeasible, relying solely on the LLM's SQL generation capabilities.

Method	EX (Dev)	EX (Test)
non-fine-tuned methods		
CHESS$_{IR+CG+UT}$	**68.31**	**71.10**
TS-SQL (majority voting)	67.60	67.79
E-SQL + GPT-4o	65.58	66.29
MCS-SQL + GPT-4	63.36	65.45
MAC-SQL + GPT-4	57.56	59.59
DAIL-SQL + GPT-4	54.76	57.41
DIN-SQL + GPT-4	50.72	55.90
fine-tuned methods		
XiYan-SQL	73.34	**75.63**
CHASE-SQL + Gemini	**74.46**	74.79
TS-SQL (fine-tuned selection)	70.93	69.20
CHESS$_{IR+SS+CG}$	65.00	66.69
CodeS-15B + SQLFixAgent	-	64.62
Dubo-SQL, v1	59.71	60.71
DTS-SQL + DeepSeek 7B	55.80	60.31

Table 3: Execution accuracy (EX) of TS-SQL and state-of-the-art (SOTA) baselines on BIRD-dev and BIRD-test (TS-SQL achieves 70.93%/69.20% with fine-tuned test code selection).

As shown in Table 4, removing the data generation, code inspection, and code generation agents decreases the EX of the generated SQL by 10.7%, 5.2%, and 41.5%, respectively, proving that each agent makes a significant contribution. The data generation agent provides high coverage and non-duality test data, which helps the LLM better understand the specific meanings of database tables and columns. The code inspection agent plays a pivotal role in ensuring the validity of the test code, and the code generation agent facilitates the LLM's ability to perform logical and numerical operations, significantly improving the test case's accuracy.

Method	Simple	Mod.	Chall.	All
TS-SQL	**75.35**	**64.73**	**62.50**	**70.93**
w/o DG	65.51	52.04	52.78	60.23(↓10.7)
w/o CI	71.46	58.49	52.08	65.71(↓ 5.2)
w/o CG	31.46	26.67	25.69	29.47(↓41.5)
w/o all	63.68	50.54	47.22	58.15(↓12.8)

Table 4: Ablation studies after removing each agent on the BIRD dev set. For brevity, "Mod." stands for "Moderate" while "Chall." denotes "Challenging".

To further assess the impact of test data generated by the data generation agent, we conduct com-

parison experiments by replacing the test data with 1/10 of the data values randomly sampled from the original databases. As shown in Figure 4, the EX score of TS-SQL reaches 61.99% on the BIRD development set, which decreases by 8.94% after replacing the test data. This demonstrates the necessity of test data for SQL self-refinement because the generated test data owns strong relevance to the original questions, which improves their utility in guiding the test case construction and test-driven self-refinement.

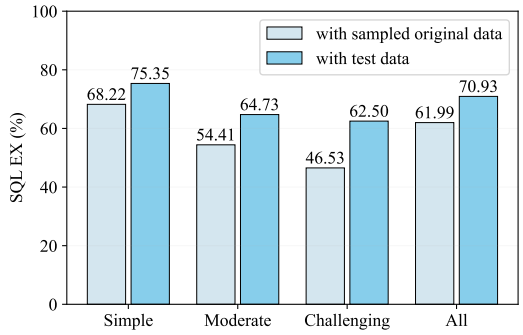

Figure 4: Performance comparison between using test data and randomly sampled original data.

4.3.2 Effectiveness of voting mechanisms

To analyze the contribution of different voting mechanisms to the accuracy of test cases, we compare test cases generated by three different LLMs, majority voting, and fine-tuned LLM selection.

As shown in Figure 5, the accuracy of test cases with majority voting and fine-tuned LLM selection improves by at least 1.63% and 5.74%, respectively. Clearly, utilizing these two voting mechanisms improves the accuracy of test cases and corresponding SQL queries. However, the highest test case accuracy of the voting mechanism only reaches 71.32%, which limits TS-SQL in further improving SQL EX. This highlights the potential of test cases in SQL self-refinement, and we anticipate further research to reduce this huge gap between the LLM-generated and manual test cases.

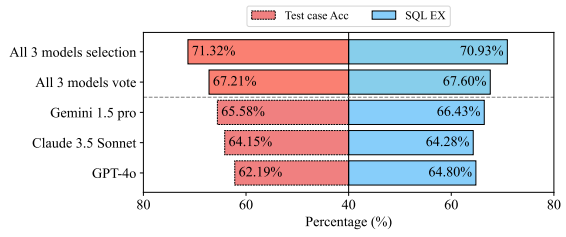

Figure 5: Test case accuracy and SQL EX on the BIRD dev set with single LLM and voting mechanisms.

4.4 Discussion of TS-SQL (RQ3)

4.4.1 Self-refine performance comparison

To highlight the strength of test case feedback-driven self-refinement, we compare self-refinement techniques of three methods with TS-SQL to correct the SQL generated by zero-shot Gemini 1.5 pro. As shown in Table 5, TS-SQL provides a clear correctness criterion with highly readable and fine-grained test cases, resulting in a significant 12.8% improvement, outperforming other self-refinement methods by 6% at least.

Method	Simple	Mod.	Chall.	All
Gemini 1.5 Pro	63.68	50.54	47.22	58.15
+ DIN-SQL	65.30	52.69	52.78	60.30(↑ 2.2)
+ MAC-SQL	67.68	56.56	52.78	62.91(↑ 4.8)
+ CHESS	70.70	56.99	53.47	64.93(↑ 6.8)
+ TS-SQL	**75.35**	**64.73**	**62.50**	**70.93(↑12.8)**
DIN-SQL	67.14	49.89	45.14	59.84
+ TS-SQL	75.57	64.30	65.28	71.19(↑11.4)
MAC-SQL	69.19	49.89	51.39	61.67
+ TS-SQL	75.14	63.87	63.89	70.66(↑ 9.0)
$\text{CHESS}_{\text{IR+SS+CG}}$	70.92	53.76	53.47	64.08
+ TS-SQL	75.89	65.16	63.89	71.51(↑ 7.4)
OpenSearch-v2	74.05	62.15	61.11	69.23
+ TS-SQL	75.89	64.09	64.58	71.25(↑ 2.0)

Table 5: Comparison across self-refinement methods and applicability of TS-SQL on the BIRD dev set.

4.4.2 Robustness and applicability

To assess the robustness and applicability of TS-SQL across different state-of-the-art (SOTA) Text-to-SQL methods, we apply the generated test cases to correct the SQL generated by non-finetuned (DIN-SQL, MAC-SQL, CHESS) and finetuned baselines (OpenSearchSQL-v2 (Xie et al., 2025)). The evaluation results on the BIRD dev set are also presented in Table 5. In Table 6 and Appendix G, we further evaluate the token cost to compare the inference consumption between TS-SQL and other methods with similar performance. For TS-SQL, the token cost includes all steps of test case generation and SQL self-refinement (provided in Appendix A). For other methods, the token cost is calculated based on the prompt templates provided in their papers.

Table 5 and 6 indicate that TS-SQL can stably improve the EX of various Text-to-SQL methods to around 71%, with a smaller token cost of 12.8% at least. Among TS-SQL's all successful semantic and syntactic error corrections towards DIN-SQL, MAC-SQL, CHESS, and OpenSearchSQL-v2 shown in Table 5, semantic error correction

leads to performance improvements of 8.2%, 9.1%, 6.9%, and 1.5%, respectively. These results showcase TS-SQL's robustness and broad applicability in correcting SQL semantic errors.

Method	Token Cost per SQL Query	Growth Rate
TS-SQL	2,369	-
CHESS	2,672	↑12.8%
E-SQL	3,237	↑36.6%

Table 6: Token cost of generating a single SQL query across methods with similar performance.

4.4.3 Boundaries of self-refining ability

On the BIRD development set, our method successfully refines 17.9% (275) of the generated SQL and fails 5.14% (79) of them. To further acknowledge the borderline of TS-SQL in SQL self-refinement, we analyze the reason for the successful and misleading correction from one-tenth of the above 354 cases, including 28 successful and 8 misleading samples. As shown in Figure 6, TS-SQL uses correct test cases to refine five types of SQL semantic errors and one type of syntactic error. Meanwhile, TS-SQL causes two table selection errors with wrong test cases. These results indicate that TS-SQL is good at refining the incorrect filtering condition and incorrect column errors, while it may encounter difficulties in refining incorrect table errors. It should be noted that, despite using correct test cases, TS-SQL still leads to two deduplication errors because the question description is ambiguous about the need to eliminate duplication, affecting the self-refinement results of TS-SQL. Appendix H shows typical case studies in detail.

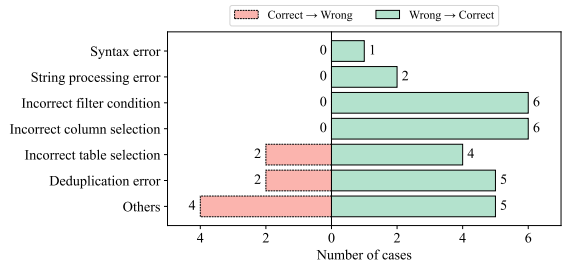

Figure 6: SQL error type of TS-SQL's successful vs misleading correction on the sampled BIRD-dev set.

5 Conclusion

In this paper, we propose TS-SQL (Test-driven Self-refinement for Text-to-SQL), a novel framework that uses automatically generated test cases to help LLMs correct semantic understanding biases and fix SQL semantic errors. Extensive experiments on the Spider and BIRD benchmark show the superior performance of TS-SQL compared to other LLM-based Text-to-SQL self-refinement methods. The modified Spider-dev set further verifies the unique advantage of TS-SQL in identifying incorrect Text-to-SQL benchmark instances and generating more rigorous SQL results.

In terms of practical applications, TS-SQL has the potential to help deployment of Text-to-SQL in real-world use cases. For example, financial analysts use Text-to-SQL to query transaction data (e.g., "2024 Q1 credit card fraud rates"). TS-SQL can auto-generate test cases to verify SQL correctness (e.g., test data for "fraud vs. non-fraud" labels) without exposing sensitive transaction data (test data is synthetic), reducing audit time (no manual SQL checks) and complies with data privacy regulations (e.g., GDPR). In healthcare, people can use Text-to-SQL to query patient lab data (e.g., "1998 male patients' monthly lab tests"). TS-SQL's synthetic test data avoids leaking real patient information, while test code ensures SQL aligns with medical logic (e.g., "monthly average" vs. AVG()), thus balancing usability (no manual SQL verification) and data security (critical for HIPAA compliance).

Limitations

Due to the performance upper bound of existing large language models (LLMs), the accuracy of the generated test cases severely limits our method's performance in SQL self-refinement. This further limits the performance gains to apply our method across all baselines on the leaderboard of the BIRD benchmark. Although we have tried majority voting and fine-tuning techniques to obtain more accurate test cases, the huge gap between the LLM-generated and manual test cases still remains. Moreover, the generated test cases of our method have not considered performance issues, such as the assessment of execution time. We anticipate further research to provide valuable insights, such as integrating various LLMs for test case generation and exploring different methods for test case selection.

Ethics Statement

Our method utilizes several closed-source large language models (LLMs) to conduct various experiments, including GPT-4o, Claude 3.5 sonnet, and Gemini 1.5 pro. It's noteworthy that LLMs depend on substantial computing power, causing electric power consumption and carbon dioxide emissions.

Acknowledgments

This study is supported by the National Key Research and Development Program of China under Grant 2023YFB3106504, Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies under Grant 2022B1212010005, the Major Key Project of PCL under Grant PCL2024A04, Shenzhen Science and Technology Program under Grant ZDSYS20210623091809029 and RCBS20221008093131089, the project of Guangdong Power Grid Co., Ltd. under Grant 037800KC23090005 and GDKJXM20231042, the China Postdoctoral Science Foundation under Grant Number 2024M751555.

References

2010. Iso/iec/ieee international standard - systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418.

Arian Askari, Christian Poelitz, and Xinye Tang. 2024. Magic: Generating self-correction guideline for in-context text-to-sql. *arXiv preprint arXiv:2406.12692*.

Hasan Alp Caferoğlu and Özgür Ulusoy. 2024. E-sql: Direct schema linking via question enrichment in text-to-sql. *arXiv preprint arXiv:2409.16751*.

Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. 2024. Sqlfixagent: Towards semantic-accurate sql generation via multi-agent collaboration. *arXiv preprint arXiv:2406.13408*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. 2024. Cycle: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):392–418.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. 2024. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2024. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Wenyu Huang, Pavlos Vougiouklis, Mirella Lapata, and Jeff Z. Pan. 2025. Masking in Multi-hop QA: An Analysis of How Language Models Perform with Context Permutation. In *In the Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL 2025)*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Ryo Kamoi, Sarkar Snigdha Sarathi Das, Renze Lou, Jihyun Janice Ahn, Yilun Zhao, Xiaoxin Lu, Nan Zhang, Yusen Zhang, Ranran Haoran Zhang, Sujeeth Reddy Vummanthala, et al. 2024. Evaluating llms at detecting errors in llm responses. *arXiv preprint arXiv:2404.03602*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

Xue Li and Till Döhmen. 2024. Towards efficient data wrangling with llms using code generation. In *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning*, pages 62–66.

Zhenwen Li and Tao Xie. 2024. Using llm to select the right sql query from candidates. *Preprint*, arXiv:2401.02115.

Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. 2025. Nl2sql-bugs: A benchmark for detecting semantic errors in nl2sql translation. *arXiv preprint arXiv:2503.11984*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Jeff Z. Pan, Simon Razniewski, Jan-Christoph Kalo, Sneha Singhania, Jiaoyan Chen, Stefan Dietze, Hajira Jabeen, Janna Omeliyanenko, Wen Zhang, Matteo Lissandrini, Russa Biswas, Gerard de Melo, Angela Bonifati, Edlira Vakaj, Mauro Dragoni, and Damien Graux. 2023. Large Language Models and Knowledge Graphs: Opportunities and Challenges. *Transactions on Graph Data and Knowledge*, pages 1–38.

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. *arXiv preprint arXiv:2410.01943*.

Mohammadreza Pourreza and Davood Rafiei. 2024. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation. *arXiv preprint arXiv:2405.15307*.

Zhili Shen, Pavlos Vougiouklis, Chenxin Diao, Kaustubh Vyas, Yuanyi Ji, and Jeff Z. Pan. 2024. Improving Retrieval-augmented Text-to-SQL with AST-based Ranking and Schema Pruning. In *In Proc. of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP 2024)*, pages 7865–7879.

Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*.

James WA Strachan, Dalila Albergo, Giulia Borghini, Oriana Pansardi, Eugenio Scaliti, Saurabh Gupta, Krati Saxena, Alessandro Rufo, Stefano Panzeri, Guido Manzi, et al. 2024. Testing theory of mind in large language models and humans. *Nature Human Behaviour*, pages 1–11.

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

Zhen Tan, Dawei Li, Song Wang, Alimohammad Beigi, Bohan Jiang, Amrita Bhattacharjee, Mansooreh Karami, Jundong Li, Lu Cheng, and Huan Liu. 2024. Large language models for data annotation: A survey. *arXiv preprint arXiv:2402.13446*.

Gladys Tyen, Hassan Mansoor, Victor Cărbune, Yuanzhu Peter Chen, and Tony Mak. 2024. Llms cannot find reasoning errors, but can correct them given the error location. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 13894–13908.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2023. Mac-sql: Multi-agent collaboration for text-to-sql. *arXiv preprint arXiv:2312.11242*.

Hongru Wang, Deng Cai, Wanjun Zhong, Shijue Huang, Jeff Z. Pan, Zeming Liu, and Kam-Fai Wong. 2025. Self-Reasoning Language Models: Unfold Hidden Reasoning Chains with Few Reasoning Catalyst. In *In the Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL 2025)*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *Proceedings of the ACM on Management of Data*, 3(3):1–24.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh Murthy, Tian Lan, Lei Li, Renze Lou, Jiacheng Xu, et al. 2024. Diversity empowers intelligence: Integrating expertise of software engineering agents. *arXiv preprint arXiv:2408.07060*.

Danna Zheng, Mirella Lapata, and Jeff Z. Pan. 2024. Archer: A Human-Labeled Text-to-SQL Dataset with Arithmetic, Commonsense and Hypothetical Reasoning. In *In Proc. of the 18th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2024)*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024. Large language model enhanced text-to-sql generation: A survey. *arXiv preprint arXiv:2410.06011*.

A Prompts

A.1 SQL generation prompt

```
As an SQL expert, given the database, the natural language questions, please using
valid SQLite and understading External Knowledge, answer the question for the tables
 provided in database. Please keep the result sql in a sql code block wrapped by ```
sql\n ```, and is the last sql in the response.

Database: CREATE TABLE ...

Question:
{question}

External Knowledge:
{evidence}
```

A.2 SQL revision prompt

```
As an SQL expert, given the database, the natural language query problem, the SQL
generated by the model corresponding to the natural language problem, as well as the
 feedback obtained by querying the database: the expected result data generated by
Pandas code and the generated SQL's query result.
Please analyze the issues with the generated SQL, then try to revise or regenerate
an SQL statement according to the logic in the Pandas code provided to answer the
natural language query problem. Please keep the result sql in a sql code block
wrapped by ```sql\n ```, and is the last sql in the response.

Database: CREATE TABLE ...

Question:
{question}

External Knowledge:
{evidence}

Pandas code:
```python
{test_code}
```

Generated SQL:
{sql_generated}

Expected result data generated by Pandas code:
{expected_result}

Generated SQL execute result data:
{current_result}

Generated SQL test result:
{Error message}

Let's think step by step.
```

A.3 Test data generation prompt

```
Generate Python code for building DataFrames with data.

---
Follow the following format.

Instruction: Instructions, assumptions and Requirements

Example: An example of how to complete this task

Dataframes: DataFrames, with name, column name and data value examples

Foreign Keys: Foreign keys of DataFrames, used for merge
```

```
Samples Of Dataframes: Samples of DataFrames

Question: Natural language qestion

Knowledge: external knowledge evidence required to map the natural language
instructions into counterpart database values.

Reasoning: Let's think step by step in order to produce the python_code. We ...

Python Code: Python code for building DataFrames
---

Instruction:
**Role**: As a tester, your task is to design synthetic test data for the following
natural language question on the given dataframe and External Knowledge(if any).
- You should ...
- ...

Example:
{one example}

Dataframes:
{Original dataframes}

Foreign Keys:
{Foreign Keys}

Samples Of Dataframes:
{Original data samples}

Question:
{question}

Knowledge:
{evidence}

Reasoning: Let's think step by step in order to
```

A.4 Test code generation prompt

```
Transform a natural language query into a Pandas query.

---
Follow the following format.

Instruction: Instructions, assumptions and Requirements

Example: An example of how to complete this task

Dataframes: DataFrames, with name, column name and data value examples

Foreign Keys: Foreign keys of DataFrames, used for merge

Db Id: Database name used to indicate the domain

Question: Natural language qestion

Knowledge: external knowledge evidence required to map the natural language
instructions into counterpart database values.

Reasoning: Let's think step by step in order to produce the pandas_code. We ...

Pandas Code: Pandas code for query data
---

Instruction:
You are a data science expert. Your task is to understand external knowledge and
generate valid pandas code to query data from existing dataframes based on the needs
```

```
 of the problem. Before generating the final code, think step by step on how to
write the code.

Guideline:
1. The Pandas library has been imported as `pd`. You can reference it directly.
2. The DataFrames are loaded and available for use.
...

Tip:
1. When you need to find the highest or lowest values based on a certain condition,
using `sort_values()` followed by `head()` or `idxmax()` is prefered over finding
the data exact matches the value found by using `max(*)`/`min(*)`.
2. If the code needs `sort_values()` to sort the results, you should only include
the column(s) used for sorting in the `result` DataFrame if the question
specifically asks for them. Otherwise, omit these columns from the the `result`
DataFrame.
3. ...

Example:
{one example}

Dataframes:
{Dataframes with data samples}

Foreign Keys:
{Foreign Keys}

Question:
{question}

Knowledge:
{evidence}

Reasoning: Let's think step by step in order to
```

A.5 Test code inspection prompt

```
Refine and correct the generated Pandas code based on error messages or query result

---
Follow the following format.

Instruction: Instructions, assumptions and Requirements

Dataframes: DataFrames, with name, column name and data value examples

Foreign Keys: Foreign keys of DataFrames, used for merge

Question: Natural language qestion

Knowledge: external knowledge evidence required to map the natural language
instructions into counterpart database values.

Previous Code: Previously Generated Code

Query Result: Query result obtained from executing previously generated code (None
if any errors occur)

Error Messasge: Error message (if any)

Reasoning: Let's think step by step in order to produce the revised_pandas_code. We
...

Revised Pandas Code: Revised Pandas code for query data
---

Instruction:
You are a professional Python programming assistant. Understanding External
Knowledge(if any), revise `Previously Generated Code` based on `Error message` or `
```

```
Query result`. If `Error message` is provided, it means that the `Previously
Generated Code` has a problem and cannot be executed, and you need to modify the
previous code. If `Query result` is provided, you need to check whether the `
Previously Generated Code` and its `Query result` meet the requirements of the
problem.  Revise the previous code as needed, ensuring to the following **Standards
for revision** and modify the previous code as needed.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The DataFrames are loaded and available for use.
- Requirements for pandas code:
  - Use only Pandas operations for the solution.
  - Store the answer in a DataFrame named `result`.
  - ...
- Standards for revision
1. **Knowledge Utilization**:
...

Dataframes:
{Dataframes with data samples}

Foreign Keys:
{Foreign Keys}

Question:
{question}

Knowledge:
{evidence}

Previous Code:
{Previous test code}

Query Result:
{query_result}

Error Messasge:
{Error message}

Reasoning: Let's think step by step in order to
```

A.6 Test code selection prompt

```
system_prompt:

Instruction:
Given the Dataframes, Knowledge and Question, there are two candidate pandas queries
 along with their Exceution result. There is correct one and incorrect one, compare
the two candidate answers, analyze the differences of the query and the result.
Based on the Question and the provided Dataframes info, choose the correct one.


user_prompt:

**************************
Dataframes:
{DF_STR}

Foreign keys:
{FK_STR}
**************************
Question:
{QUESTION}

Knowledge:
{HINT}
**************************
Candidate A:
```python
{CANDIDATE_A_QUERY}
```
```

```
```

```
Execution result:
{CANDIDATE_A_RESULT}
**************************
Candidate B:
```python
{CANDIDATE_B_QUERY}
```

Execution result:
{CANDIDATE_B_RESULT}

Just output the correct candidate `<answer> A </answer>` or `<answer> B </answer>`.
```

B Experiment Settings

B.1 Datasets

Spider (Yu et al., 2018) is a large-scale, complex, and cross-domain semantic parsing and Text-to-SQL dataset annotated by college students. It consists of 10,181 questions and 5,693 unique complex SQL queries on 200 databases with multiple tables, covering 138 different domains.

BIRD (Li et al., 2024) is a pioneering, cross-domain dataset that examines the impact of extensive database contents on text-to-SQL parsing. The BIRD dataset consists of 12,751 text-to-SQL pairs and 95 databases, with a total size of 33.4 GB, covering 37 professional domains. It introduces new challenges, such as dirty and noisy database values, external knowledge grounding between natural language questions and database values, and SQL efficiency, particularly when dealing with massive databases.

B.2 Evaluation metrics

We use the official metric, **EX**ecution Accuracy(**EX**), as the primary measure to assess the effectiveness of our method. EX evaluates the correctness of the predicted SQL queries by comparing their execution results with those of the ground-truth SQLs.

B.3 Models

SQL Generation and Revision: Unless otherwise specified, we utilize Gemini 1.5 Pro for both the generation and revision of SQL queries.

Test case Generation: For test data generation, we used the text-embedding-3-small model to get the relevant value in the database and the text-embedding-3-large model to calculate embedding similarity and retrieve relevant context. Additionally, Gemini 1.5 Pro, GPT-4o, and Claude 3.5 Sonnet are employed for the generation and inspection of test cases.

B.4 Hyperparameters

SQL Generation and Revision: The temperature is set to 0, no few-shots demonstrations are provided, and the maximum revision number is set to 5.

Test Case Generation: The temperature is set to 0, a 1-shot is provided, and the maximum number of refinements is set to 5. In addition, as the number of code inspection rounds increases, the temperature will gradually increase by 0.01.

C Incorrect and Modified Gold SQL for each Error Type of Spider-dev Set

Figure 7 illustrates the proportional distribution of different gold SQL errors within the original Spider-dev set. There are 6 types of errors, where wrong values (32.5%), wrong semantic logic (21.7%), and data type errors (18.3%) rank the top three.

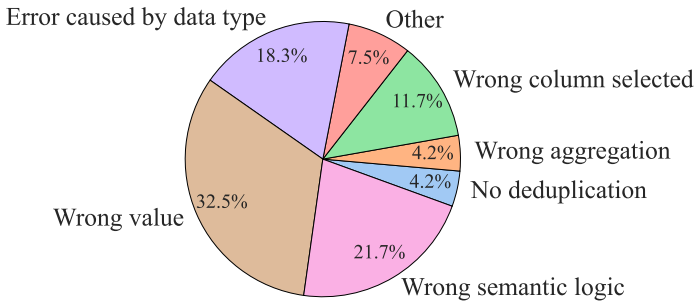

Figure 7: Detected gold SQL error type of Spider-dev set.

1. **Wrong value.** The data value used in the SQL filter condition does not match the data stored in the database. An example is shown in Figure 8.

Question	What are airport names at City 'Aberdeen'?
Original	`SELECT AirportName FROM AIRPORTS WHERE City = "Aberdeen"`
	Query result (✗): Empty
Test code	`result = airports_df[airports_df['City'].str.strip() == 'Aberdeen'][['AirportName']]`
	Query result (✓): [2 rows x 1 column]
Modified	`SELECT AirportName FROM airports WHERE TRIM(City) = 'Aberdeen';`
	Query result (✓): [2 rows x 1 column]
Explanation	In this database, the values for the "City" field are inconsistently formatted, often with extra spaces on either side. For example, 'Aberdeen' is stored as 'Aberdeen ' in the database.
DIN-SQL	`SELECT AirportName FROM airports WHERE City = 'Aberdeen'`
	Query result (✗): Empty
MAC-SQL	`SELECT` `AirportName` `FROM airports WHERE` `City` `= 'Aberdeen';`
	Query result (✗): Empty

Figure 8: Wrong value example. Both DIN-SQL and MAC-SQL fail to correctly handle the data value in the filtering condition, resulting in empty query outputs.

2. **Wrong semantic logic.** The SQL appears to understand the semantics of the question, but the logic used to write the SQL is incorrect. An example is shown in Figure 9.

<table>
<tr><td>Question</td><td>find id of the tv channels that from the countries where have more than two tv channels.</td></tr>
<tr><td>Original</td><td>
<pre>SELECT id FROM tv_channel
GROUP BY country HAVING count(*) > 2</pre>
Query result (✗):
<table><tr><th>id</th></tr><tr><td>700</td></tr></table>
</td></tr>
<tr><td>Test code</td><td>
<pre># Count TV channels per country
country_counts = TV_Channel_df['Country'].value_counts()

# Filter countries with more than two channels
countries_with_more_than_two = country_counts[country_counts
> 2].index

Select IDs of TV channels from these countries
result =
TV_Channel_df[TV_Channel_df['Country'].isin(countries_with_mo
re_than_two)][['id']]</pre>
</td></tr>
<tr><td>Modified</td><td>
<pre>SELECT t.id FROM TV_Channel t WHERE
t.Country IN (
 SELECT Country
 FROM TV_Channel
 GROUP BY Country
 HAVING COUNT(id) > 2
);</pre>
Query result (✓):
12 rows x 1 col
<table><tr><th>id</th></tr><tr><td>700</td></tr><tr><td>...</td></tr></table>
</td></tr>
<tr><td>Explanation</td><td>The original query selects the id column but aggregates by country, returning only one TV channel per country, which is a logical error. The correct approach is to first identify the qualifying countries and then retrieve all TV channels for those countries.</td></tr>
</table>

Figure 9: Wrong semantic logic.

3. **Errors caused by data type issues.** SQL does not take into account the special data types of some fields in the database, which causes problems. An example is shown in Figure 10.

Question

What is the model of the car with the smallest amount of horsepower?

Original

```
SELECT T1.Model
FROM CAR_NAMES AS T1 JOIN CARS_DATA AS T2
ON T1.MakeId = T2.Id
ORDER BY T2.horsepower ASC LIMIT 1;
```

Query result (✗):

Model
amc

Test code

```
# Convert 'Horsepower' to numeric, coercing errors to NaN
cars_data_df['Horsepower'] =
pd.to_numeric(cars_data_df['Horsepower'], errors='coerce')

# Merge cars_data_df with car_names_df
merged_df = cars_data_df.merge(car_names_df, left_on='Id',
right_on='MakeId')

# Find the index of the car with the smallest horsepower
min_hp_index = merged_df['Horsepower'].idxmin()

# Select the model of the car with the smallest horsepower
result = merged_df.loc[[min_hp_index], ['Model']]
```

Modified

```
SELECT ml.Model FROM cars_data cd
JOIN car_names cn ON cd.Id = cn.MakeId
JOIN model_list ml ON cn.Model = ml.Model
WHERE CAST(NULLIF(cd.Horsepower, '') AS
INTEGER) = (
 SELECT MIN(CAST(NULLIF(Horsepower, '') AS
INTEGER))
 FROM cars_data
 WHERE Horsepower GLOB '*[0-9]*')
LIMIT 1;
```

Query result (✓):

Model
volkswagen

Explanation

1. The original query returns a "minimum" horsepower of 100(amc), while the modified query returns 46(volkswagen), indicating that the original query is indeed incorrect.
2. Horsepower is stored as TEXT, with some values being non-numeric like "null".
3. The original query incorrectly sorts horsepower as TEXT, while the modified query filters and converts values before sorting to find the true minimum.

DIN-SQL

```
SELECT T1.Model
FROM car_names AS T1 JOIN cars_data AS T2
ON T1.MakeId = T2.Id
ORDER BY T2.Horsepower LIMIT 1
```

Query result (✗):

Model
amc

MAC-SQL

```
SELECT cn.`Model`
FROM cars_data AS cd JOIN car_names AS cn
ON cd.`Id` = cn.`MakeId`
ORDER BY cd.`Horsepower` ASC LIMIT 1;
```

Query result (✗):

Model
amc

Figure 10: Errors caused by data type example. Both DIN-SQL and MAC-SQL fail to handle the data types stored in the database correctly, resulting in a query result where the model found does not have the minimum horsepower.

4. **Wrong column selected.** The SQL query selects too many, too few, or incorrect columns. An example is shown in Figure 11.

Question	List the name, date and result of each battle.
Original	`SELECT name , date FROM battle` — Query result (✗): 8 rows x **2** cols
Test code	`result = battle_df[['name', 'date', 'result']]`
Modified	`SELECT name, date, result FROM battle;` — Query result (✓): 8 rows x 3 cols
Explanation	The original query omits the result column required in the question.

Figure 11: Wrong column selected.

5. **Wrong aggregation.** The error in the SQL lies in the `GROUP BY` statement. An example is shown in Figure 12.

Question: Find the average ranking for each player and their first name.

Original:

```
SELECT avg(ranking) , T1.first_name
FROM players AS T1 JOIN rankings AS T2
ON T1.player_id = T2.player_id
GROUP BY T1.first_name
```

Query result (✗): **1580** rows x 2 cols

avg(ranking)	first_name
...	...

Test code:

```
# Calculate average ranking for each player
avg_rankings =
rankings_df.groupby('player_id')['ranking'].mean().reset_inde
x()

# Merge with players_df to get first names
result = pd.merge(avg_rankings, players_df,
left_on='player_id', right_on='player_id')

# Select required columns and rename for clarity
result = result[['first_name', 'ranking']]
result = result.rename(columns={'ranking': 'average_ranking'})
```

Modified:

```
SELECT p.first_name, AVG(r.ranking) AS
average_ranking
FROM players p JOIN rankings r
ON p.player_id = r.player_id
GROUP BY p.player_id, p.first_name;
```

Query result (✓): 2775 rows x 2 cols

first_name	AVG(r.ranking)
...	...

Explanation: The first names of players may be duplicated. Aggregating by first name in the original query would result in multiple players with the same first name being grouped together, leading to errors in the calculation. The correct query should include the primary key in the aggregate fields.

Figure 12: Wrong aggregation.

6. **No deduplication.** The SQL query does not remove duplicate data. An example is shown in Figure 13.

Question

What are the names of the singers who performed in a concert in 2014?

Original

```
SELECT T2.name
FROM singer_in_concert AS T1
JOIN singer AS T2
ON T1.singer_id = T2.singer_id
JOIN concert AS T3
ON T1.concert_id = T3.concert_id
WHERE T3.year = 2014
```

Query result (✗): 6 rows x 1 col

Name
Timbaland
Justin Brown
John Nizinik
Justin Brown
.....

Test code

```
# Step 1: Filter concerts held in 2014
concerts_2014 = concert_df[concert_df['Year'] == '2014']

# Step 2: Get concert IDs for 2014
concert_ids_2014 = concerts_2014['concert_ID']

# Step 3: Find singers who performed in these concerts
singers_in_2014_concerts =
singer_in_concert_df[singer_in_concert_df['concert_ID'].isin(
concert_ids_2014)]

# Step 4: Convert Singer_ID to int64 for merging
singers_in_2014_concerts['Singer_ID'] =
singers_in_2014_concerts['Singer_ID'].astype(int)

# Step 5: Merge to get singer names
singers_with_names = singers_in_2014_concerts.merge(singer_df,
on='Singer_ID')

# Step 6: Select the required column
result = singers_with_names[['Name']]
```

Modified

```
SELECT DISTINCT s.Name
FROM concert c
JOIN singer_in_concert sic
ON c.concert_ID = sic.concert_ID
JOIN singer s
ON sic.Singer_ID = s.Singer_ID
WHERE c.Year = '2014';
```

Query result (✓): 5 rows x 1 col

Name
Timbaland
Justin Brown
.....

Explanation

The original query lacks "DISTINCT," causing "Justin Brown" to appear multiple times in the results.

Figure 13: No deduplication.

D Analysis of EX Decrease on the Modified Spider-dev Set for DIN-SQL and MAC-SQL

On our modified Spider dev dataset, we observe a decrease in the EX metric for both MAC-SQL and DIN-SQL. We analyze the number and distribution of cases where these methods are marked as correct on the original dataset but incorrect on the modified dataset. As shown in Table 7, the two main factors causing this decrease are "Wrong Value" and "Error Caused by Data Type." We provide specific examples for each of these causes to illustrate the issues, as shown in Figure 8 and Figure 10.

Method	Total number of cases	Wrong value	Data type error	Others
DIN-SQL	57	52.6%	31.6%	15.8%
MAC-SQL	53	54.7%	28.3%	17.0%

Table 7: Performance variation of DIN-SQL and MAC-SQL between the ordinary and modified Spider-dev set.

E Incorrect and Modified Gold SQL for each Error Type of the First Three Databases on the BIRD-dev Set

Figure 14 illustrates the proportional distribution of different gold SQL errors within the first three databases of the original BIRD development set. In the 'california_schools' database, the dominant error type is "wrong filter condition," accounting for 42.9% of the errors, followed by "others" at 28.6%. Both "formula calculation error" and "wrong column selected" contribute equally at 14.3%. In the 'financial' database, errors are more evenly distributed. "Table join error," "no deduplication," "wrong semantic logic," and "wrong filter condition" each make up 17.6% of the errors, while "wrong column selected" and "others" are slightly lower at 11.8%. "Formula calculation error" is the least common at 5.9%. In the 'toxicology' database, the two leading error types are "table join error" and "wrong semantic logic," each representing 31.8% of the total errors. "Wrong column selected" follows with 22.7%, while "wrong filter condition" accounts for 9.1%, and "others" for only 4.5%.

These distributions highlight different error tendencies across databases, suggesting that database characteristics influence the types of SQL errors most likely to occur.

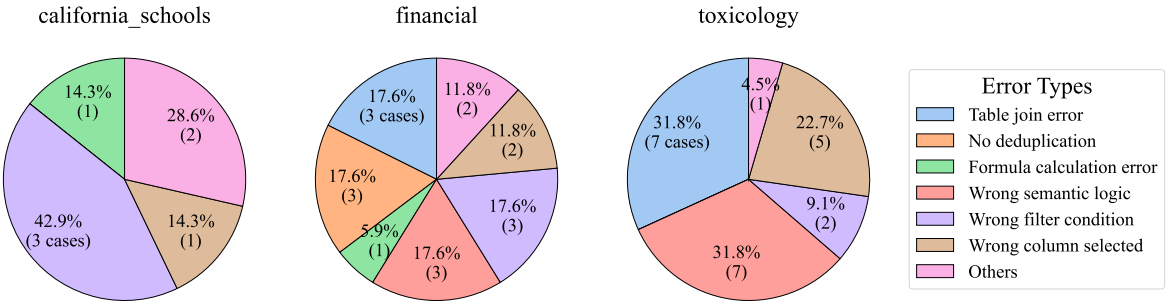

Figure 14: Detected gold SQL error type of the first 3 databases on the BIRD-dev set.

1. **Formula calculation error.** Errors using the calculation formula. An example is shown in Figure 15.

Question (Evidence)	What was the difference in the number of crimes committed in East and North Bohemia in 1996? (Difference in no. of committed crimes between 2 regions = **Total no. of committed crimes in 1996 in north Bohemia - Total no. of committed crimes in 1996 in east Bohemia**. A3 refers to region. Data about no. of committed crimes 1996 appears in A16)
Original	`SELECT SUM(IIF(A3 = 'east Bohemia', A16, 0)) - SUM(IIF(A3 = 'north Bohemia', A16, 0)) FROM district` Query result (✗): SUM(...) = -17734

Test code	`north_bohemia_crimes = district_df[district_df['A3'] == 'north Bohemia']['A16'].sum()`
	`east_bohemia_crimes = district_df[district_df['A3'] == 'east Bohemia']['A16'].sum()`
	`difference = north_bohemia_crimes - east_bohemia_crimes`
	`result = pd.DataFrame({'difference': [difference]})`

Modified	`SELECT SUM(CASE WHEN A3 = 'north Bohemia' THEN A16 ELSE 0 END) - SUM(CASE WHEN A3 = 'east Bohemia' THEN A16 ELSE 0 END) AS difference FROM district;` Query result (✓): SUM(...) = 17734
Explanation	The SQL error is a formula calculation error: the subtraction order is inverted, causing the result to be the negative of what's expected. The fix is to align the SQL with the problem's definition by placing the North Bohemia sum first in the subtraction.

Figure 15: Formula calculation error.

2. **Table join error.** Errors in selecting the appropriate fields or tables for joining. An example is shown in Figure 16.

Question (Evidence)

List all the elements with double bond, consisted in molecule TR024. (double bond refers to bond_type = '='; element = 'cl' means Chlorine; element = 'c' means …)

Original

```
SELECT T1.element
FROM atom AS T1 INNER JOIN bond AS T2
ON T1.molecule_id = T2.molecule_id
WHERE T1.molecule_id = 'TR024'
AND T2.bond_type = '='
```

Query result (✗): 76 rows x 1 col

element
c
…

Test code

```
filtered_bond = bond_df[(bond_df['molecule_id'] == 'TR024') &
  (bond_df['bond_type'] == '=')]
merged_df = pd.merge(
  filtered_bond, connected_df, on='bond_id')

element1 = pd.merge(merged_df, atom_df, left_on='atom_id',
right_on='atom_id', suffixes=('_bond', '_atom'))[['element']]

element2 = pd.merge(merged_df, atom_df, left_on='atom_id2',
right_on='atom_id', suffixes=('_bond', '_atom'))[['element']]

result = pd.concat([element1, element2]).drop_duplicates()
```

Modified

```
SELECT T1.element
FROM atom AS T1 INNER JOIN connected AS T3
ON T1.atom_id = T3.atom_id
INNER JOIN bond AS T2 ON T3.bond_id =
T2.bond_id
WHERE T2.molecule_id = 'TR024'
AND T2.bond_type = '=';
```

Query result (✓): 4 rows x 1 col

element
c
…

Explanation

The **element** and **atom** tables are linked through the **connected** table. However, the original gold SQL, by joining on **molecule_id**, inadvertently retrieves all atoms of the TR024 molecule instead of those specifically connected by double bonds.

Figure 16: Table join error.

3. **No deduplication.** (Figure 13)
4. **Wrong semantic logic.** (Figure 9)
5. **Wrong filter condition.** (Figure 17)
6. **Wrong column selected.** (Figure 11)

F Full Results for each of the First Three Databases on the Original and Modified BIRD-dev Set

Method	California_schools		Financial		Toxicology		Sum	
	Original	Modified	Original	Modified	Original	Modified	Original	Modified
GPT-4o	35.96	38.20	46.23	48.11	57.93	64.83	48.53	52.65
MAC-SQL	51.69	55.06	59.43	68.87	55.86	63.45	55.88	62.94
DIN-SQL	50.56	56.18	63.21	67.92	59.31	68.28	58.24	65.00
TS-SQL	**59.55**	**65.17**	**69.81**	**84.91**	**68.97**	**83.45**	**66.76**	**79.12**

Table 8: Performance of the first three databases on the original and modified BIRD-dev set.

G Comparison of LLM's Token Consumption

To gain deeper insights into the token consumption of TS-SQL compared to other methods, we conduct a token cost analysis against CHESS and E-SQL, which explicitly present their prompt templates in their papers, allowing for a direct and transparent comparison. Table 9 shows the total number of tokens per query for each method. The results indicate that TS-SQL demonstrates higher efficiency in token usage, with an average of 2,369 tokens per query, which is significantly lower than E-SQL's 3,237 tokens and

CHESS's 2,672 tokens. This suggests that TS-SQL achieves SQL generation and correction with lower computational overhead, offering practical advantages in terms of cost efficiency while maintaining strong performance.

Method	Token Cost per SQL Query	Growth Rate
TS-SQL	2,369	-
CHESS	2,672	↑12.8%
E-SQL	3,237	↑36.6%

Table 9: Token consumption between TS-SQL and other methods with similar performance.

H Typical Examples of Successful Error Correction on the BIRD-dev Set

1. **Incorrect filter condition.** (Figure 17)

Question: Which active district has the highest average score in Reading?

SQL generated:

```
SELECT dname FROM satscores
WHERE rtype = 'D'
ORDER BY AvgScrRead DESC LIMIT 1;
```

Query result (✗):

dname
Santa Cruz ...

Test code:

```
merged_df = pd.merge(satscores_df, schools_df, left_on='cds',
right_on='CDSCode', how='inner')
active_schools = merged_df[merged_df['StatusType'] ==
'Active']
highest_avg_read = active_schools['AvgScrRead'].max()
result = active_schools[active_schools['AvgScrRead'] ==
highest_avg_read][['dname']].head(1)
```

Final SQL:

```
SELECT T1.dname FROM satscores AS T1
INNER JOIN schools AS T2
ON T1.cds = T2.CDSCode
WHERE T2.StatusType = 'Active'
ORDER BY T1.AvgScrRead DESC LIMIT 1;
```

Query result (✓):

dname
Palo Alto

Figure 17: An example of correcting the error of **incorrect filter condition**. In this case, the question asks for `active` districts, but the generated SQL does not reflect this condition.

2. **Incorrect column selection.** (Figure 18)

Question: What atoms comprise TR186?

SQL generated:

```
SELECT element FROM atom
WHERE molecule_id = 'TR186';
```

Query result (✗): 41 rows x 1 col

element
c
...

Test code:

```
result = atom_df[atom_df['molecule_id'] ==
'TR186'][['atom_id']]
```

Final SQL:

```
SELECT atom_id FROM atom
WHERE molecule_id = 'TR186';
```

Query result (✓): 41 rows x 1 col

atom_id
TR186_1
...

Figure 18: An example of correcting the error of **incorrect column selection**. In this case, the question asks about atoms, but the generated SQL chooses the element column.

3. **Incorrect table selection.** (Figure 19)

Question

Lists all patients by ID who were diagnosed with Behcet's and had their exams between 01/01/197 and 12/31/1997.

SQL generated

```
SELECT ID FROM Patient
WHERE Diagnosis = 'Behcet'
AND Description BETWEEN '1997-01-01' AND
'1997-12-31';
```

Query result (✗): Empty

Test code

```
filtered_exams = Examination_df[
 (Examination_df['Diagnosis'] == 'Behcet') &
 (Examination_df['Examination Date'] >= '1997-01-01') &
 (Examination_df['Examination Date'] <= '1997-12-31')]
result = filtered_exams[['ID']].drop_duplicates()
```

Final SQL

```
SELECT ID FROM Examination
WHERE Diagnosis = 'Behcet'
AND `Examination Date` BETWEEN '1997-01-01'
AND '1997-12-31';
```

Query result (✓): 5 rows x 1 col

ID
3390870
...

Figure 19: An example of correcting the error of **incorrect table selection**. In this case, the query should filter for examination and diagnosis information in the `Examination` table, not the `Patient` table.

4. **Deduplication error.** (Figure 20)

Question

For the year that concluded on December 31, 1998, how many male patients on average were tested in the lab each month?

SQL generated

```
SELECT CAST(COUNT(DISTINCT l.ID) AS REAL) /
12
FROM Laboratory AS l
INNER JOIN Patient AS p ON l.ID = p.ID
WHERE strftime('%Y', l.Date) = '1998' AND
p.SEX = 'M';
```

Query result (✗):

CAST(... AS REAL) / 12
181.398648

Test code

```
laboratory_1998 = Laboratory_df[ (Laboratory_df['Date'] >=
'1998-01-01') & (Laboratory_df['Date'] <= '1998-12-31') ]
merged_df = pd.merge(laboratory_1998, Patient_df, on='ID',
how='inner')
male_patients_1998 = merged_df[merged_df['SEX'] == 'M']
average_male_patients_per_month = len(male_patients_1998) / 12
result = pd.DataFrame({'AverageMalePatientsPerMonth':
[average_male_patients_per_month]})
```

Final SQL

```
SELECT CAST(COUNT(l.ID) AS REAL) / 12
FROM Laboratory AS l
INNER JOIN Patient AS p ON l.ID = p.ID
WHERE l.Date BETWEEN '1998-01-01' AND '1998-
12-31' AND p.SEX = 'M';
```

Query result (✓):

CAST(... AS REAL) / 12
181.400601

Figure 20: An example of correcting the **deduplication error**. In this case, the incorrectly deduplicated results lead to an undercount of the average number of male patients tested per month, as some patients may have been tested multiple times within different months.

5. **String processing error.** (Figure 21)

Question: What is the average height of players born between 1990 and 1995?

SQL generated:

```
SELECT AVG(height) FROM Player
WHERE birthday BETWEEN '1990-01-01' AND
'1995-12-31';
```

Query result (✗):

AVG(height)
181.398648

Test code:

```
merged_df = pd.merge(satscores_df, schools_df, left_on='cds',
right_on='CDSCode', how='inner')
active_schools = merged_df[merged_df['StatusType'] == 'Active']
highest_avg_read = active_schools['AvgScrRead'].max()
result = active_schools[active_schools['AvgScrRead'] ==
highest_avg_read][['dname']].head(1)
```

Final SQL:

```
SELECT CAST(SUM(height) AS REAL) / COUNT(id)
AS average_height FROM Player
WHERE birthday >= '1990-01-01 00:00:00' AND
birthday < '1996-01-01 00:00:00';
```

Query result (✓):

average_height
181.400601

Figure 21: An example of correcting the **string processing error**. In this case, the string processing of the generated SQL misses a record with a birthday of `"1995-12-31 00:00:00"`, leading to an incorrect calculation result.

6. **Syntax error.** (Figure 22)

Question: Among the players born before the year 1986, how many of them would remain in his position and defense while the team attacked?

SQL generated:

```
SELECT COUNT(DISTINCT player_api_id)
FROM Player AS p
INNER JOIN Player_Attributes AS pa
ON p.player_api_id = pa.player_api_id
WHERE STRFTIME('%Y', p.birthday) < '1986'
AND pa.defensive_work_rate = 'high';
```

Parse error: ambiguous column name: player_api_id

Test code:

```
merged_df = pd.merge(satscores_df, schools_df, left_on='cds',
right_on='CDSCode', how='inner')
active_schools = merged_df[merged_df['StatusType'] == 'Active']
highest_avg_read = active_schools['AvgScrRead'].max()
result = active_schools[active_schools['AvgScrRead'] ==
highest_avg_read][['dname']].head(1)
```

Final SQL:

```
SELECT COUNT(DISTINCT p.player_api_id)
FROM Player AS p
INNER JOIN Player_Attributes AS pa
ON p.player_api_id = pa.player_api_id
WHERE STRFTIME('%Y', p.birthday) < '1986'
AND pa.defensive_work_rate = 'high';
```

Query result (✓):

COUNT(...)
892

Figure 22: An example of correcting the **syntax error**. As both the `Player_Attributes` and `Player` tables contain a column named `player_api_id`, the generated SQL is ambiguous about which table the column refers to, leading to a syntax error during execution.