PI-SQL: Enhancing Text-to-SQL with Fine-Grained Guidance from Pivot Programming Languages

Yongdong Chi[1*], **Hanqing Wang**[1*], **Yun Chen**[1], **Yan Yang**[1]
Jian Yang[3], **Zonghan Yang**[2], **Xiao Yan**[4], **Guanhua Chen**[5†]
[1]Shanghai University of Finance and Economics, [2]Tsinghua University,
[3]Beihang University, [4]Wuhan University, [5]Southern University of Science and Technology

Abstract

Text-to-SQL transforms the user queries from natural language to executable SQL programs, enabling non-experts to interact with complex databases. Existing prompt-based methods craft meticulous text guidelines and examples to facilitate SQL generation, but their accuracy is hindered by the large semantic gap between the texts and the low-resource SQL programs. In this work, we propose PI-SQL, which incorporates the high-resource Python program as a pivot to bridge between the natural language query and SQL program. In particular, PI-SQL first generates Python programs that provide fine-grained step-by-step guidelines in their code blocks or comments, and then produces an SQL program following the guidance of each Python program. The final SQL program matches the reference Python program's query results and, through selection from candidates generated by different strategies, achieves superior execution speed, with a reward-based valid efficiency score up to 4.55 higher than the best-performing baseline. Extensive experiments demonstrate the effectiveness of PI-SQL, which improves the execution accuracy of the best-performing baseline by up to 3.20.

1 Introduction

SQL is a standard programming language designed for managing and manipulating relational databases (Website, 2023). Although popular and general, SQL programs can be challenging for non-experts to write, particularly when it comes to complex data querying tasks. Text-to-SQL models convert natural language queries into executable SQL programs (Androutsopoulos et al., 1995; Li and Jagadish, 2014; Li et al., 2024c; Yu et al., 2018), enabling non-experts to interact with complex databases and extract insights from big data (Cai et al., 2018; Wang et al., 2020; Cao et al., 2021).

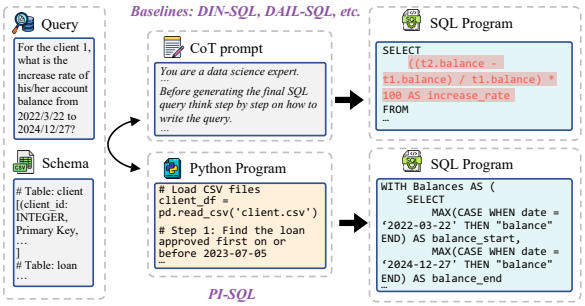

Figure 1: Given the database schema and a user query, text-to-SQL models generate an executable SQL program. Different from the text guidance produced by chain-of-thought, PI-SQL resorts to the granular guidance from a pivot programming language.

Recently, many text-to-SQL models have been proposed based on large language models (LLMs), using either prompt engineering (Pourreza and Rafiei, 2024; Qu et al., 2024; Dong et al., 2023; Gao et al., 2024a; Talaei et al., 2024; Pourreza et al., 2025) or supervised fine-tuning (Gao et al., 2024a; Li et al., 2023a, 2024b; Gao et al., 2024b). The prompt-based methods require meticulously crafted guidelines (Pourreza and Rafiei, 2024; Qu et al., 2024; Dong et al., 2023; Gao et al., 2024a; Talaei et al., 2024; Pourreza et al., 2025) as well as curated few-shot in-domain examples (Pourreza and Rafiei, 2024; Gao et al., 2024a; Talaei et al., 2024). The fine-tuning-based methods rely on high-quality training data, which is expensive to obtain. Moreover, tailored to the training data domain, they may not generalize in other domains (Dong et al., 2023; Hong et al., 2024).

In this paper, we propose PI-SQL, a novel prompt-based method that enhances text-to-SQL by incorporating a high-resource PIvot programming language to provide fine-grained guidance. Motivated by multilingual pretraining (Xue, 2020; Lample, 2019; Huang et al., 2019) and triangular machine translation (Kim et al., 2019; Zhang et al., 2022), PI-SQL adopts a Python program as a

[*] Equal Contribution.
[†] Corresponding Author.

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 25120–25144

pivot to bridge natural language and SQL programs. As shown in Figure 1, compared with text-based guidance generated using chain-of-thought reasoning (Pourreza and Rafiei, 2024; Dong et al., 2023; Gao et al., 2024a; Talaei et al., 2024), PI-SQL utilizes Python programs to provide more detailed step-by-step reasoning through code blocks, comments. Existing program-of-thought-based text-to-SQL methods (Xia et al., 2024; Xu et al., 2024) overlooked the considerable structure gap between the Python and SQL programs, while PI-SQL contains approaches to mitigate the intrinsic difference between Python and SQL.

Specifically, PI-SQL consists of an intermediate guidance preparation stage and an SQL generation stage guided by the generated Python. In the first stage, to enhance subsequent SQL generation guided by Python, PI-SQL employs three strategies to generate Python programs tailored for diverse SQL application scenarios, and also steers this Python generation by incorporating SQL adaptation instructions into the prompt. In the second stage, each Python program serves as guidance for generating SQL programs. SQL programs that produce results consistent with the majority output of the Python programs are retained, and the one with the best execution efficiency is chosen. As a result, PI-SQL fully leverages high-resource programming languages like Python to generate highly accurate and efficient SQL programs, without requiring few-shot examples or supervised fine-tuning with labeled data.

We compare PI-SQL with ten state-of-the-art (SOTA) baselines on the famous BIRD (Li et al., 2024d) and Archer (Zheng et al., 2024) benchmarks. The results show that PI-SQL outperforms the baselines in both execution accuracy (EX) and reward-based valid efficiency score (R-VES), which are two popular metrics for text-to-SQL. Compared with the best-performing baseline, the EX improvement of PI-SQL is 3.20 on the BIRD dev set.[1]

2 Related Work

Prompt-Based Text-to-SQL Methods Given the strong generalization ability of LLMs, recent mainstream research has shifted towards leveraging their powerful few-shot and zero-shot capabilities through prompt-based approaches. Numerous research efforts have focused on enhancing text-to-SQL performance from various aspects, including improved schema linking (Pourreza and Rafiei, 2024; Gao et al., 2024a; Dong et al., 2023), the selection of more effective few-shot demonstrations (Pourreza and Rafiei, 2024; Sun et al., 2023; Chen et al., 2023; Gao et al., 2024a; Qu et al., 2024; Talaei et al., 2024; Pourreza et al., 2025), incorporating natural language chain-of-thought reasoning (Pourreza and Rafiei, 2024; Gao et al., 2024a; Talaei et al., 2024; Pourreza et al., 2025), and using the self-consistency method to boost performance with additional test-time computing (Sun et al., 2023; Gao et al., 2024a; Lee et al., 2024; Talaei et al., 2024; Maamari et al., 2024; Pourreza et al., 2025). Different from these works, PI-SQL achieves better text-to-SQL performance by leveraging the fine-grained guidance from high-resource programming languages in a zero-shot setting.

Program of Thoughts Program of Thoughts (PoT) is an extension of the Chain of Thought (CoT) prompting strategy, aiming to mitigate errors in intermediate reasoning by executing intermediate steps as Python programs. PoT has been widely adopted in various tasks (Payoungkhamdee et al., 2025; Sahu et al., 2024; Sarch et al., 2024; Zhang et al., 2024), due to its improved reliability of numerical and logical inference. However, its application to text-to-SQL remains limited. R^3 (Xia et al., 2024) directly employs PoT to generate Python code before SQL to guide the SQL generation, while Xu et al. (2024) incorporates PoT into the text-to-SQL training process. These approaches overlook the semantic and structural gap between Python and SQL. In contrast, PI-SQL's design provides better SQL generation guidance with Python, which mitigates the intrinsic difference between Python and SQL.

3 Method

3.1 Motivation and Insights

Previous works (Pourreza et al., 2025; Talaei et al., 2024; Pourreza and Rafiei, 2024) apply text-based reasoning as the guidance to generate SQL responses. However, they still struggle with various hard-level queries and database schemas, primarily due to the limited SQL corpus encountered during pretraining. Unlike these approaches, we propose to leverage fine-grained guidance from high-resource programming languages (PI-SQL). High-resource programming languages such as Python

[1] Our code is publicly available at https://github.com/sustech-nlp/Pi-SQL.

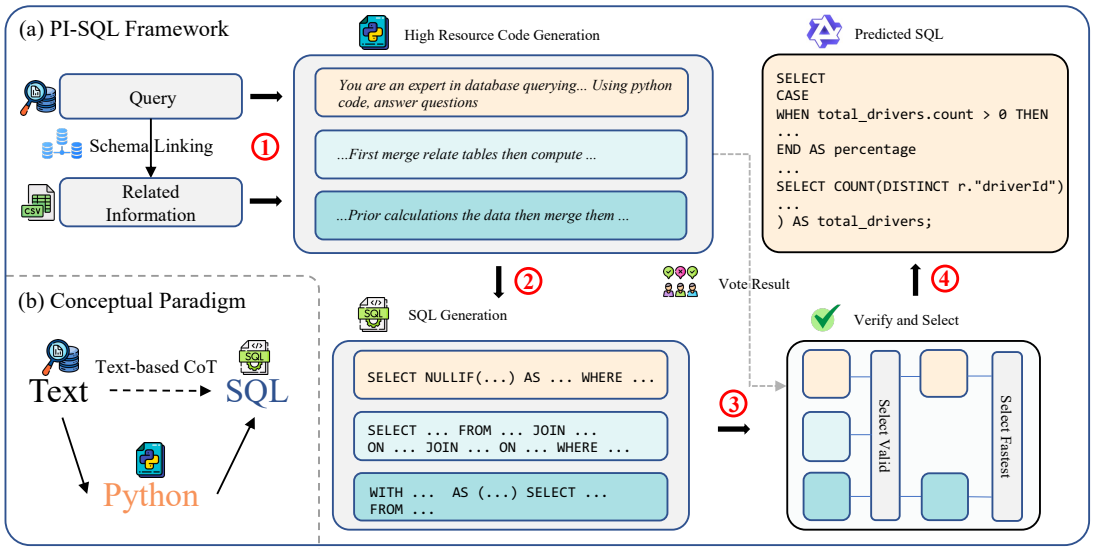

Figure 2: Overview of our PI-SQL method. (a) The workflow of PI-SQL. It incorporates high-resource programming languages like Python to provide step-by-step fine-grained guidance and verification to enhance LLM-based text-to-SQL. (b) The difference between PI-SQL and existing text-based CoT approaches.

serve as pivot languages, bridging the gap between SQL and natural language, akin to the triangular neural machine translation model (Zhang et al., 2022).

The PI-SQL framework is motivated by the advancements achieved through multilingual pretraining (Xue, 2020; Lample, 2019; Huang et al., 2019) and triangular neural machine translation (Kim et al., 2019; Zhang et al., 2022). (1) The low-resource languages (LRL) share similar syntaxes and lexemes with the high-resource languages from the same language family. During multilingual pretraining, the shared information serves as anchor points to better align the representation space of these languages, improving the performance on LRL via cross-lingual transfer. (2) In the case of triangular machine translation, a high-resource pivot language is incorporated to improve the translation from the source language to the target language. For example, instead of directly translating from English to Estonian, Finnish is used as a pivot language, as it belongs to the same language family as Estonian, and the translation from Finnish to Estonian is easier. The English text is first translated into Finnish, and then the Finnish translation is further translated into Estonian. The triangular translation improves the performance of low-resource language pairs.

The situation is similar for SQL and its corresponding high-resource programming language, Python. Both programs are widely applied in the field of data analysis and share similar logic and keywords. During code pretraining, SQL aligns with the representation space of Python and benefits from the large-scale Python data, similar to the case in multilingual pretraining. Meanwhile, Python and natural language are well aligned in a shared representation space, as both are high-resource and are jointly pretrained with shared anchor points like comments in Python code.

In this way, the PI-SQL invites Python as a pivot language to bridge user queries in natural language and the low-resource SQL programs for better text-to-SQL performance, as shown in Figure 2.

When compared with direct SQL generation, guidance with the corresponding Python program has several advantages, illustrated as follows.

- **Proficient**. Large-scale Python data contributes to the proficiency and accuracy of data analysis with Python for LLMs.
- **Fine-Grained**. Different from the nested operations in SQL, Python programs decompose complex data query tasks into verifiable code blocks as well as comments to form step-by-step reasoning trajectories. The execution results, such as exceptions or errors, also serve as fine-grained feedback and self-reminders for SQL generation.
- **Modular**. Various Python packages abstract the reasoning process and facilitate the generation of corresponding code blocks.

Specifically, given a complex user query q_u, the LLM first responds with a high-resource programming language C_i^p with different data analysis

strategies G_i. The LLM then generates the SQL program C_i^s with the guidance of the Python program C_i^p and its execution results E_i^p. PI-SQL consists of two stages: Intermediate Guidance Preparation (Section 3.2) and SQL Generation (Section 3.3).

3.2 Intermediate Guidance Preparation

The Intermediate Guidance Preparation stage aims to generate diverse fine-grained Python programs as guidance for text-to-SQL tasks. Following previous works (Talaei et al., 2024; Caferoğlu and Özgür Ulusoy, 2024; Cao et al., 2024), we incorporate a schema-linking module to retrieve relevant tables and columns from a schema with the user query. The retrieved data is further converted to `csv` files for interaction with the Python program.[2] However, the transformation from Python to SQL cannot be directly applied due to the fundamental differences between Python and SQL. While Python is a procedural language, SQL is a declarative, structured query language. This distinction may pose challenges in using Python to guide SQL generation. The result of R^3 in Table 1 demonstrates this. To bridge this gap, we introduce SQL-specific strategies as shown in Algorithm 1 along with a set of Python-to-SQL adaptation rules. These components help mitigate the mismatch between the two languages and allow us to more effectively leverage Python's execution signals in the SQL selection process.

Diverse Python Generation Strategies. The PI-SQL framework crafts three different strategies to guide the reasoning trajectories of Python program generation. As a high-level, interpreted scripting programming language, Python has different reasoning paths compared to SQL in the data analysis task. The Python program analyzes data sequentially, where the relevant data is first filtered and then combined for further analysis. The SQL program benefits from the efficient data analysis engine that SQL users are accustomed to first combining all relevant data and then further analyzing. To better generate Python programs that can guide SQL generation across different application scenarios, we design three distinct strategies: merge, filter, and direct.

These strategies are illustrated below, encouraging intermediate guidance to incorporate diverse reasoning trajectories:

- **Merge-First Strategy**. This strategy asks the LLM to merge and join the relevant columns first based on the input information. Then filter and extract the required data. This strategy aligns with the design philosophy of relational databases. As different data are decomposed and stored separately, they are first reconstructed with the foreign keys and then analyzed.
- **Filter-First Strategy**. This prompt guideline suggests the LLM filter and prepare the relevant columns first based on the input information. Then the model directly generates further analysis code based on the filtered data. This strategy follows the vanilla practice of Python programs in data analysis tasks.
- **Vanilla Direct Generation**. This strategy does not impose any suggestion on the LLM. The model generates the Python program in a freestyle learned during pretraining.

With the guidance of different strategies, the LLM is expected to generate Python programs with different reasoning trajectories. Moreover, Python offers a richer set of libraries and functions for data analysis that are not supported by SQL engines. To enhance Python-to-SQL adaptation and improve the quality of guidance, we also explicitly prompt LLMs to use APIs and functions that closely resemble valid SQL operations.

Verification of Python Program. The diverse generations of Python programs are verified with execution with the `csv` files.[3] The self-consistency method (Wang et al., 2023) is employed to determine the reference query result for the user's query from all Python execution outcomes. This reference result is then used to select the final SQL response in a subsequent stage. We contend that selecting the SQL based on Python results effectively serves as a double-check mechanism, further ensuring the faithfulness of the chosen SQL.

3.3 SQL Generation with Python Guidance

In the second stage, the SQL responses are generated with guidance from the corresponding fine-grained Python program, as shown in Algorithm 1. Subsequently, the PI-SQL framework verifies these generated SQL programs by executing them

[2] We measured that constructing the `csv` files takes only about 0.024 seconds per question on average, which is negligible compared to the inference time.

[3] We discuss the execution time in Appendix B.6.

Method	Zero-shot	Few-shot		Archer		BIRD		B-Simple		B-Moderate		B-Challenging	
		Fixed	Dynamic	EX	R-VES	EX	R-VES	EX	R-VES	EX	R-VES	EX	R-VES
Vanilla	✓			10.58	10.19	54.30	55.94	61.84	64.96	43.10	42.16	42.07	42.55
C3	✓			16.35	23.04	57.37	53.65	65.51	61.40	46.98	43.66	38.62	36.19
DIN-SQL			✓	8.65	13.56	50.07	46.80	58.16	54.29	39.44	37.08	32.41	30.10
DAIL-SQL			✓	16.35	18.07	55.02	51.02	62.16	57.67	46.98	43.40	35.17	33.04
TA-SQL		✓		8.65	10.06	55.15	52.06	63.35	59.79	44.18	41.33	37.93	37.08
R^3			✓	20.19	21.27	52.67	47.72	57.95	53.23	44.83	39.87	44.14	37.71
CHESS		✓		21.15	25.81	61.02	56.91	68.54	64.20	49.78	46.15	48.97	44.81
CHASE-SQL			✓	**25.96**	28.70	61.34	59.16	68.54	64.35	52.80	52.11	48.97	48.69
E-SQL		✓		16.35	17.13	58.47	54.80	65.08	61.05	51.29	47.89	39.31	37.00
RSL-SQL			✓	13.46	14.87	61.34	56.76	67.89	63.13	52.80	48.38	44.18	42.96
PI-SQL	✓			25.00	**30.10**	**64.54**	**63.71**	**70.92**	**70.06**	**56.47**	**55.63**	**49.66**	**49.06**
Δ	-	-	-	-0.96	+1.40	+3.20	+4.55	+2.38	+5.10	+3.67	+3.52	+0.69	+0.37

Table 1: Execution Accuracy (EX) and Valid Efficiency Score (VES) on the Archer and BIRD datasets. BIRD provides the results for different query difficulty levels. We indicate whether a method is zero-shot or few-shot, and if few-shot, whether it uses fixed or dynamic shots. The best and runner-up results for each case are marked with **bold** and underline, respectively. 'Δ' is the performance gain of PI-SQL over the best-performing baseline.

in the database. An SQL program is deemed a valid candidate if its execution result matches the reference query result. Finally, among all valid candidates, the highest execution efficiency is selected as the final SQL response, R_f^s. This entire stage, including the SQL generation process and the final selection, is detailed in Algorithm 1, and the specific prompt template used for SQL generation can be found in Appendix C.2.

Algorithm 1 Algorithm for PI-SQL framework

Data: LLM θ, user query q_u, relational database D associated with the query, and strategy of Python program generation C.
Result: Model predicted SQL program R_f^s

$\mathcal{R}_p = \text{GeneratePython}(\theta, C, q_u, D)$
$\mathcal{R}_s = \emptyset$
for *each Python program* R_i^p *in* $\mathcal{R}_p$ **do**
  $E_i^p = \text{Execute}(R_i^p)$ `// Execute the code and get results`
  $R_i^s = \text{GenerateSQL}(\theta, q_u, D, R_i^p)$ `// Generate SQL for the code`
  $\mathcal{R}_s = \mathcal{R}_s \cup R_i^s$
end
$\mathcal{R}_{se} = \emptyset$
for *each SQL program* R_i^s *in* $\mathcal{R}_s$ **do**
  $E_i^s = \text{ExecuteSQL}(R_i^s, D)$ `// Execute SQL and get database results`
  $\mathcal{R}_{se} = \mathcal{R}_{se} \cup \{(R_i^s, E_i^s)\}$
end
$\text{MajorityResult} = \text{FindMostFrequent}(\mathcal{R}_p)$ `// Find the most frequent execution result among Python codes`
$\text{ValidSQL} = \text{SelectValidSQL}(\mathcal{R}_{se}, \text{MajorityResult}, D)$ `// Select valid SQL that matches majority result`
$R_f^s = \arg\min_{r \in \text{ValidSQL}} \text{ExecutionTime}(r)$ `// Select SQL with the least execution time`
return R_f^s

4 Experiments

4.1 Experiment Settings

Benchmarks We conduct experiments on two widely recognized text-to-SQL datasets: BIRD (Li et al., 2024d) and Archer (Zheng et al., 2024). They are designed to encompass various real-world scenarios, featuring simple and complex query structures. Spider (Yu et al., 2018) is not selected for evaluation for the following reasons: 1) Our method focuses on more challenging text-to-SQL queries, whereas the queries in Spider are relatively easy; 2) We have found that the ground-truth SQL programs in Spider are noisy (Zhong et al., 2023), which makes the evaluation results unreliable. More details about the benchmarks are available in Appendix A.1.

Metrics We utilize Execution Accuracy (EX) (Yu et al., 2018) and Reward-based Valid Efficiency Score (R-VES) (Li et al., 2024d) as evaluation metrics to assess the methods' performance.

• **Execution Accuracy (EX):** EX measures the ratio of correctly predicted SQL programs by comparing their execution results with those of the ground-truth SQL programs on the same database instance.

• **Reward-based Valid Efficiency Score (R-VES):** R-VES evaluates the performance of models that generate SQL queries, considering their accuracy and runtime performance. As an improvement over the previous Valid Efficiency Score (VES), R-VES incorporates the execution time of correct queries into the evaluation while mitigating the influence of abnormal or outlier execution times.

Implementation Details To ensure a fair comparison in a unified setting, we use the same LLM backbone with a temperature of 0 and a maximum token limit of 4096 for PI-SQL and all baselines. We choose GPT-4o-mini (OpenAI, 2024) as the backbone for the computational constraints.[4]

[4]For example, performing CHESS on the BIRD dev set with GPT-4o (Hurst et al., 2024) costs approximately $ 800.

And to compare with advanced performance, we also perform PI-SQL with Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) in section 5.1. We adopt the schema linking module from RSL-SQL (Cao et al., 2024), as schema linking is not the primary focus of this work. Furthermore, we evaluate PI-SQL using various backbone models in Section 5.3 and on models of varying scales in Appendix B.8.

Baselines We compare PI-SQL with ten baselines: **Vanilla**, **C3** (Dong et al., 2023), **DIN-SQL** (Pourreza and Rafiei, 2024), **DAIL-SQL** (Gao et al., 2024a), **TA-SQL** (Qu et al., 2024), **R^3** (Xia et al., 2024) **CHESS** (Talaei et al., 2024), **CHASE-SQL** (Pourreza et al., 2025), **E-SQL** (Caferoğlu and Özgür Ulusoy, 2024), and **RSL-SQL** (Cao et al., 2024). The vanilla method is the same as PI-SQL, except for the absence of Python guidance. We introduce each baseline in detail in the Appendix A.2.

4.2 Main Results

Table 1 shows the comparison results of PI-SQL against the baselines. Overall, PI-SQL outperforms all other baselines on both BIRD and Archer in terms of execution accuracy and efficiency. This is impressive because our zero-shot approach outperforms the zero-shot baseline vanilla, C3, and the few-shot baselines, specifically DIN-SQL, DAIL-SQL, TA-SQL, and CHESS.

When comparing PI-SQL with baselines across different query difficulty levels on BIRD, we find that PI-SQL shows consistent improvements over the baselines on different difficulty levels. Specifically, it improves over baselines by 2.38 to 12.97 EX on simple-level queries, 3.67 to 17.03 EX on moderate queries, and 0.69 to 17.25 EX on challenging queries. This may be attributed to the rich data processing capabilities of Python, which enable large language models (LLMs) to handle a wide range of queries more effectively. PI-SQL also achieves consistent improvements on the R-VES benchmark over the baselines by 4.55 to 16.91, further validating its effectiveness. These results highlight the strong potential of our method in real-world scenarios, as it: (1) can handle queries of varying difficulty across diverse contexts, and (2) generates SQL queries that are both accurate and efficient.

Table 8 and 9 in Appendix B.1 compare the inference token usage and inference cost of PI-SQL with the baselines. PI-SQL has a lower inference cost than the best-performing baseline CHESS, a higher cost than vanilla and C3, and a comparable cost to the other baselines. We believe this test-time cost is justifiable for two reasons: 1) It substantially enhances performance over vanilla and C3, especially for challenging queries; 2) The generated SQL programs can be executed by users multiple times in practice, making the significant R-VES improvement achieved by PI-SQL particularly valuable. In Section 4.3, we provide additional evidence that the improvement of PI-SQL over the vanilla method is not solely due to the increased test time computing.

Setup	Overall		Simple		Moderate		Challenging	
	EX	R-VES	EX	R-VES	EX	R-VES	EX	R-VES
Vanilla	54.30	55.94	61.84	64.96	43.10	42.16	42.07	42.55
Vanilla+SC	59.71	57.77	66.27	64.19	52.37	50.99	41.38	38.57
Ablation on code generation mode								
Merge	61.93	59.83	68.11	66.08	53.45	51.34	49.66	47.13
Filter	61.99	60.19	67.89	66.01	54.09	52.48	49.66	47.74
Direct	62.58	61.09	68.97	67.41	53.88	52.69	49.66	47.67
Ours(Mixed)	**64.54**	**63.71**	**70.92**	**70.06**	**56.47**	**55.63**	**49.66**	**49.06**
Ablation on SQL selection method								
Mixed+SC	63.62	61.53	70.05	67.88	54.74	52.77	51.03	49.09
Ours(Mixed+CV)	**64.54**	**63.71**	**70.92**	**70.06**	**56.47**	**55.63**	**49.66**	**49.06**
Ablation on Python-SQL adaptation								
W/O adaptation	63.36	61.97	70.16	68.83	54.31	52.86	48.97	47.34
Ours	**64.54**	**63.71**	**70.92**	**70.06**	**56.47**	**55.63**	**49.66**	**49.06**

Table 2: Ablation study on BIRD dataset. We perform ablation for the Python generation strategy and SQL selection method. 'SC' means self-consistency, while 'CV' means cross verification. The best result for each case is highlighted in **bold**.

4.3 Ablation Study

In this section, we conduct an ablation study on 3 key components of PI-SQL using the BIRD benchmark: the Python generation strategy, the SQL selection method, and the Python-SQL adaptation. For the Python generation strategy, we evaluate four variants: using merge, filter, or direct individually, or using a combination of all three. For the SQL selection method, we either select by referring to the Python execution result (cross-verification) or by taking a majority vote from the SQL execution results. Regarding the Python-SQL adaptation, we compare the performance of PI-SQL with and without this adaptation. We also add a vanilla+self-consistency baseline, for which we directly generate N SQLs for each query and select the final SQL program using self-consistency of the SQL execution results. We set the value of N to 11 to ensure that the token cost of this baseline matches that of PI-SQL, and the temperature to 0.5 is determined on a validation set.

Table 2 shows the ablation results. When us-

ing the same inference token cost, PI-SQL outperforms the vanilla method by 4.83 EX and 5.94 R-VES. This indicates that our performance improvement is not solely attributable to the test time scaling law but rather to the effective guidance provided by high-resource programming languages.

For ablation on Python generation strategies, we observe that using any single Python generation strategy substantially improves upon vanilla GPT-4o-mini, achieving an improvement of over 7.63 EX and 3.89 R-VES. However, mixing all strategies yields the best performance across all difficulty levels, surpassing the best single method by 1.96 EX and 2.62 R-VES. This indicates that different strategies are complementary, highlighting the importance of using a mixed approach.

Our cross-verification approach for SQL selection consistently outperforms the self-consistency method across all difficulty levels and evaluation metrics. This could be attributed to the diversity of errors made by Python and SQL, which contrasts with the more similar errors produced by different SQLs.

Python-SQL adaptation consistently improves the performance of PI-SQL, achieving an overall gain of 1.18 in EX score and 1.74 in R-VES, with particularly notable improvements on moderate and challenging queries. This may be attributed to the fact that more complex problems benefit from clearer and more relevant guidance.

Consequently, the mixed generation strategies, cross verification method, and Python-SQL adaptation collectively enhance the generation quality of PI-SQL, distinguishing it from previous, direct PoT-based methods such as R^3.

5 Analyses

5.1 Comparison with SOTA Results

Due to computational constraints, We could not perform a direct comparison with other methods using SOTA LLMs such as GPT-4o under the same experimental setting. Instead, we evaluate PI-SQL with Qwen2.5-Coder-32B-Instruct and compare its performance against the SOTA methods such as Distillery (Maamari et al., 2024), OpenSearch-SQL (Xie et al., 2025), and XiYan-SQL (Gao et al., 2024b) reported on the BIRD leaderboard[5] in Table 3. The results show that PI-SQL, as a zero-shot approach, can achieve performance comparable to or even surpassing SOTA methods that rely on fine-tuning, refinement, or powerful proprietary models, using only a 32B open-source model. As shown in Appendix B.7, a naive refinement method enhances PI-SQL by 0.58 in EX score and 3.14 in R-VES score, underscoring its compatibility with refinement strategies.

Method	Backbone	Finetuned	With Refinement	EX
R^3	GPT-4		✓	61.80
CHESS†	GPT-4o		✓	65.00
E-SQL†	GPT-4o		✓	65.58
Distillery†	GPT-4o	✓		67.21
OpenSearch-SQL†	GPT-4o		✓	69.30
CHASE-SQL†	Gemini 1.5 pro	✓	✓	73.01
XiYan-SQL†	UNK	✓	✓	**73.34**
XiYan-SQL†	Qwen2.5-Coder-32B	✓	✓	67.01
Ours	Qwen2.5-Coder-32B			**67.40**

Table 3: Comparison with state-of-the-art (SOTA) methods on the BIRD dev set. Due to computational cost, PI-SQL was evaluated using a 32B open-source model. †Results are taken from the BIRD leaderboard. R^3 results are from Xia et al. (2024).

5.2 Comparison with Zero-shot Methods

To ensure fair comparison with the zero-shot PI-SQL, all baselines were evaluated in a zero-shot setting. Although methods like DIN-SQL are not zero-shot, we include them to assess performance changes without in-context examples, thereby revealing the dependency of such methods on few-shot demonstrations. Table 4 shows PI-SQL significantly outperforms baselines under these conditions, achieving 5.09-27.97 higher EX and 3.97-29.26 higher R-VES scores. This underscores PI-SQL's advantages: no need for complex shot design and superior generalization. While most few-shot baselines decline without examples, DIN-SQL, DAIL-SQL, RSL-SQL, and CHASE-SQL show relative stability. In contrast, TA-SQL, CHESS, R^3, and E-SQL experience substantial drops (EX score reductions of 18.58, 24.00, 4.10, and 4.23, respectively), as their rule-based SQL generation heavily relies on few-shot examples for effective LLM rule interpretation and application.

5.3 Using Different LLM Backbones

In this section, we further investigate the performance of PI-SQL on open-sourced LLM backbones, including Qwen2.5-Coder-32B-Instruct, QwQ-32B (Team, 2025b), and Gemma-3-27B-IT (Team, 2025a).

As shown in Table 5, PI-SQL consistently enhances the performance of vanilla methods across

[5] https://bird-bench.github.io

Method	Overall				Simple		Moderate		Challenging	
	EX	R-VES	ΔEX	ΔR-VEX	EX	R-VES	EX	R-VES	EX	R-VES
Vanilla	54.30	55.94	-	-	61.84	64.96	43.10	42.16	42.07	42.55
C3	57.37	53.65	-	-	65.51	61.40	46.98	43.66	38.62	36.19
DIN-SQL	50.85	47.51	+0.78	+0.71	58.70	55.04	41.16	38.44	31.72	28.45
DAIL-SQL	53.45	48.80	-1.57	-2.22	59.56	54.88	47.41	44.86	33.79	33.49
TA-SQL	36.57	34.45	-18.58	-17.61	42.92	40.43	28.02	26.23	23.45	22.60
CHESS	37.02	37.46	-24.00	-19.45	44.75	45.19	26.07	26.33	22.75	23.79
R^3	48.57	49.13	-4.10	+1.41	54.05	54.33	42.67	43.89	32.41	32.77
RSL-SQL	59.45	58.95	-1.89	+2.19	66.27	66.52	51.08	49.76	42.76	40.11
E-SQL	54.24	49.13	-4.23	-5.67	60.91	61.26	50.00	50.49	34.78	33.64
CHASE-SQL	59.25	59.74	-0.79	+0.58	63.67	64.72	53.66	53.29	48.96	48.62
PI-SQL	**64.54**	**63.71**	-	-	**70.92**	**70.06**	**56.47**	**55.63**	**49.66**	**49.06**

Table 4: Comparing PI-SQL with zero-shot baselines on the BIRD dataset. ΔEX/VES denotes the zero-shot EX/R-VES minus the few-shot EX/R-VES. The best result for each case is highlighted in **bold**.

Model	Overall		Simple		Moderate		Challenging	
	EX	R-VES	EX	R-VES	EX	R-VES	EX	R-VES
Vanilla method								
Qwen-Coder	59.97	58.61	64.76	63.64	55.17	53.40	44.83	43.12
QwQ	55.08	54.06	62.49	61.78	46.34	44.98	35.86	33.87
Gemma3	58.87	57.51	64.32	62.96	51.72	50.71	46.90	44.49
GPT-4o-mini	54.30	55.94	61.84	64.96	43.10	42.16	42.07	42.55
With PI-SQL								
Qwen-Coder	**67.40**	65.54	**72.86**	71.14	**59.91**	**57.89**	56.55	54.29
QwQ	64.34	62.79	71.14	69.41	56.90	55.52	44.83	43.80
Gemma3	66.42	**65.62**	**72.86**	**72.08**	56.25	55.63	**57.93**	**56.41**
GPT-4o-mini	64.54	63.71	70.92	70.06	56.47	55.63	49.66	49.06

Table 5: Performance of PI-SQL with different LLM backbones on the BIRD dev set. The best result for each case is highlighted in **bold**.

all backbones, irrespective of their architectures and model types. This demonstrates the versatility and robustness of our method, which requires no additional training and can be applied to a wide range of backbones. To further demonstrate the effectiveness of PI-SQL, we compare it with fine-tuned methods (Appendix B.5), evaluate its scalability on the Qwen-Coder-Instruct series (Appendix B.8), provide case studies (Section 5.5 and Appendix B.2), and conduct experiments on a more advanced LLM, DeepSeek-V3 (Liu et al., 2024) (Appendix B.9).

5.4 Analyzing Python Generation Results

To better understand how the pivot program improves SQL generation, we analyze the intermediate results from Python. Table 6 shows the EX of the final Python result, selected with self-consistency after executing all Python candidate programs. We can conclude that: 1) PIS could significantly outperform VanS, demonstrating the effectiveness of Python guidance. 2) Second, we observe that PIP shows advantages over PIS on the challenging subsets. This suggests that there is still room for improvement in our method, and using Python as guidance for SQL generation remains a promising direction worth exploring. On the other hand, Python is often not supported for data queries in numerous real-world database applications. In such instances, SQL code is essential, rendering the PIP approach inapplicable. 3) As detailed in Table 11, the R-VES results underscore the superior efficiency of PIS compared to PIP, indicating its significant potential for practical application. 4) By also considering the complementary results from Table 10, we note that on simple and moderate tasks, PIS outperforms PIP. This indicates that SQL may have certain advantages over Python in handling straightforward data query tasks. We believe that combining the respective strengths of SQL and Python is a promising future research direction.

Model	Overall			Challenging		
	VanS	PIS	PIP	VanS	PIS	PIP
Qwen-Coder	59.97	67.40	66.42	44.82	56.55	61.37
QwQ	55.08	64.34	65.58	35.86	44.82	62.06
Gemma3	58.86	66.42	62.12	46.89	57.93	54.48
GPT-4o-mini	54.30	64.54	65.44	42.07	49.66	59.31

Table 6: Performance (EX) of Python program on the BIRD dev set. VanS and PIS denote SQL performance without and with Python guidance, while PIP denotes Python performance.

Model	Overall			Challenging		
	Merge	Filter	Direct	Merge	Filter	Direct
Qwen-Coder	34.49	35.27	30.25	31.72	33.10	35.17
QwQ	34.49	31.42	34.09	33.10	22.07	44.83
Gemma3	31.29	33.64	35.07	29.66	33.79	36.55
GPT-4o-mini	31.94	34.15	33.89	35.17	31.03	33.79

Table 7: Distribution of Python selection rates across different Python generation strategies.

To analyze the contribution of different Python generation strategies to the final SQL program, we present the distribution of Python strategy selection rates in Table 7. A strategy is considered selected when the final SQL program is guided by the Python program generated in that specific strategy. We find that all three strategies contribute to the final SQL program, with the merge strategy accounting for the largest share. Additionally, higher-performing backbones tend to exhibit a more balanced contribution across various strategies, possibly because they can generate effectively across various strategies, whereas weaker LLMs rely more on appropriate prompt guidance.

5.5 Case Study

We conduct a case study in Figure 3, comparing the SQL program generated by PI-SQL with both the gold SQL program and the one generated by

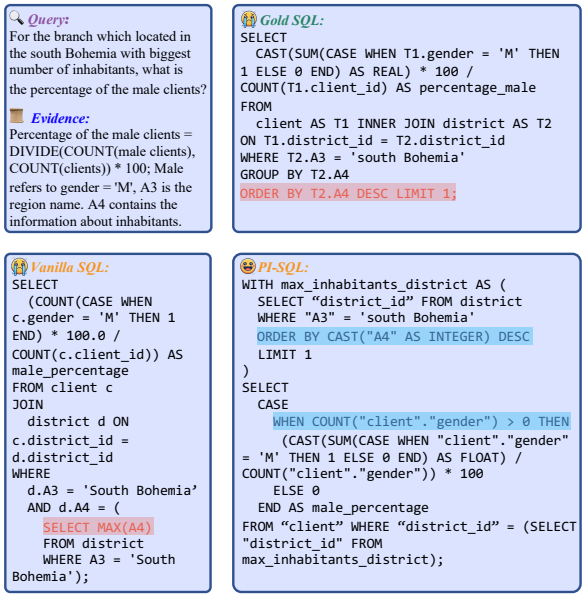

Figure 3: Case study of a specific query and its corresponding evidence, showcasing the gold SQL and the SQL generated by the vanilla method and PI-SQL.

the vanilla method. Both the gold SQL and vanilla return incorrect SQL programs as they directly sort by `A4`, which is of text type. In contrast, PI-SQL generates the correct SQL program by casting `A4` to an integer type before sorting. The SQL program generated by PI-SQL is also superior to the other two in terms of robustness, execution efficiency, and readability. PI-SQL is more robust as it incorporates comprehensive considerations in case statements, such as avoiding division-by-zero errors by checking `COUNT("client"."gender") > 0`. It is more efficient and readable as it structurally decouples the identification of the target district (the most populous district in South Bohemia) from the subsequent calculation of male client percentages. In contrast, the gold SQL and the vanilla SQL combine filtering, table joins, grouping, and calculations into a single monolithic block, resulting in low execution efficiency and poor readability.

5.6 Discussion for Scope of Supported SQL Features

Despite our method's strong performance on various benchmarks, a key consideration is its underlying assumption that Python execution over CSV files can fully simulate a database environment. This assumption may not hold for more advanced SQL constructs like window functions or recursive queries. Therefore, it is necessary to discuss the scope of SQL features our method supports.

We have investigated this issue and found that while Python's representation of certain advanced SQL constructs may differ in style and native efficiency, it is fully capable of expressing the necessary logic:

- **Window Functions:** Pandas offers a rich set of analogous APIs, such as groupby().rank() and shift(), to achieve similar results.
- **Recursive Queries:** These can be expressed procedurally using native Python loops.
- **Complex Joins and Subqueries:** The logic can be replicated by first computing intermediate results and then combining them.

Crucially, we argue that the goal of the Python generation step is not a direct, one-to-one translation into SQL. Instead, it serves as a high-level logical blueprint to formulate a sound overall strategy in PI-SQL. For this purpose, Python's expressive power is entirely sufficient.

To demonstrate this empirically, we designed a new test case for recursive queries, which are arguably the most challenging for Python to represent. The full details of this case study are provided in Appendix B.10, including the schema descriptions, the natural language question, the intermediate Python program, and the final SQL generated by both the vanilla method and PI-SQL.

We compared the SQL generated by our method against the vanilla approach. Our findings show that while both methods produced correct results, the SQL generated by PI-SQL is superior. It executes significantly faster on our machine (280ms vs. 406ms) and is more robust and readable due to its dynamic parsing logic and ordered output. This validates that our Python-guided approach can produce higher-quality SQL even for complex constructs.

6 Conclusion

In this paper, we present PI-SQL, a high-resource programming language-guided SQL generation system with two key stages: intermediate guidance preparation and guided SQL generation. Experiments across various benchmarks and difficulty levels prove that our zero-shot method, PI-SQL, significantly outperforms all baselines, including those with few-shot examples or requiring fine-tuning. The success of PI-SQL underscores the potential of leveraging programming languages as an intermediate step in guiding code generation, offering new insights for future text-to-SQL research.

Limitations

PI-SQL depends on multiple generated Python codes to guide the LLM in generating SQL programs. This process introduces additional inference tokens, leading to higher computational costs during test time. One potential solution to alleviate this issue is to introduce a router that selectively schedules text queries for either direct generation or Python-guided generation. We plan to explore this approach in future work.

Acknowledgements

This project was supported by National Natural Science Foundation of China (No. 62306132), Guangdong Basic and Applied Basic Research Foundation (No. 2025A1515011564), Natural Science Foundation of Shanghai (No. 25ZR1402136) and the Fundamental Research Funds for the Central Universities (CXJJ-2024-463). This work was done during Yongdong's internship at Southern University of Science and Technology. We thank the anonymous reviewers for their insightful feedback on this work.

References

Ion Androutsopoulos, Graeme D Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases–an introduction. *Natural language engineering*, 1(1):29–81.

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

Ruichu Cai, Boyan Xu, Zhenjie Zhang, Xiaoyan Yang, Zijian Li, and Zhihao Liang. 2018. An encoder-decoder framework translating natural language to database queries. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*.

Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. 2021. Lgesql: line graph enhanced text-to-sql model with mixed local and non-local relations. *arXiv preprint arXiv:2106.01093*.

Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. 2024. Rsl-sql: Robust schema linking in text-to-sql generation. *arXiv preprint arXiv:2411.00073*.

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

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, et al. 2023. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024a. Text-to-sql empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145.

Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. 2024b. Xiyan-sql: A multi-generator ensemble framework for text-to-sql. *Preprint*, arXiv:2411.08599.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-generation database interfaces: A survey of llm-based text-to-sql. *arXiv preprint arXiv:2406.08426*.

Haoyang Huang, Yaobo Liang, Nan Duan, Ming Gong, Linjun Shou, Daxin Jiang, and M. Zhou. 2019. Unicoder: A universal language encoder by pre-training with multiple cross-lingual tasks. In *Conference on Empirical Methods in Natural Language Processing*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Yunsu Kim, Petre Petrov, Pavel Petrushkov, Shahram Khadivi, and Hermann Ney. 2019. Pivot-based transfer learning for neural machine translation between non-English languages. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 866–876, Hong Kong, China.

G Lample. 2019. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2024. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. *arXiv preprint arXiv:2405.07467*.

Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024a. The dawn of natural language to sql: Are we fully ready? *arXiv preprint arXiv:2406.01265*.

Fei Li and Hosagrahar V Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. *Preprint*, arXiv:2302.05965.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023b. RESDSQL: decoupling schema linking and skeleton parsing for text-to-sql. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 13067–13075.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024b. CODES: towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024c. 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.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024d. 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.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-SQL in the age of well-reasoned language models. In *NeurIPS 2024 Third Table Representation Learning Workshop*.

OpenAI. 2024. Models Documentation: GPT-4o mini. `https://platform.openai.com/docs/models#gpt-4o-mini`. Accessed: 2025-02-10.

Patomporn Payoungkhamdee, Pume Tuchinda, Jinheon Baek, Samuel Cahyawijaya, Can Udomcharoenchaikit, Potsawee Manakul, Peerat Limkonchotiwat, Ekapol Chuangsuwanich, and Sarana Nutanong. 2025. Towards better understanding of program-of-thought reasoning in cross-lingual and multilingual environments. *arXiv preprint arXiv:2502.17956*.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2025. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *The Thirteenth International Conference on Learning Representations*.

Mohammadreza Pourreza and Davood Rafiei. 2024. Din-SQL: decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems*, volume 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. In *ACL (Findings)*, pages 5456–5471.

Pritish Sahu, Karan Sikka, and Ajay Divakaran. 2024. Pelican: Correcting hallucination in vision-LLMs via claim decomposition and program of thought verification. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 8228–8248, Miami, Florida, USA. Association for Computational Linguistics.

Gabriel Sarch, Lawrence Jang, Michael Tarr, William W Cohen, Kenneth Marino, and Katerina Fragkiadaki. 2024. Vlm agents generate their own memories: Distilling experience into embodied programs of thought. *Advances in Neural Information Processing Systems*, 37:75942–75985.

Ruoxi Sun, Sercan Ö Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, et al. 2023. Sql-palm: Improved large language model adaptation for text-to-sql (extended). *arXiv preprint arXiv:2306.00739*.

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

Gemma Team. 2025a. Gemma 3.

Qwen Team. 2025b. Qwq-32b: Embracing the power of reinforcement learning.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: relation-aware schema encoding and linking for text-to-sql parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Website. 2023. Iso/iec 9075-1:2023 information technology — database languages sql.

Hanchen Xia, Feng Jiang, Naihao Deng, Cunxiang Wang, Guojiang Zhao, Rada Mihalcea, and Yue Zhang. 2024. r^3: "this is my sql, are you with me?" a consensus-based multi-agent system for text-to-sql tasks. *arXiv preprint arXiv:2402.14851*.

Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *arXiv preprint arXiv:2502.14913*.

Bo Xu, Shufei Li, Yifei Wu, Shouang Wei, Ming Du, Hongya Wang, and Hui Song. 2024. Chain-of-program prompting with open-source large language models for text-to-sql. In *2024 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

L Xue. 2020. mt5: A massively multilingual pre-trained text-to-text transformer. *arXiv preprint arXiv:2010.11934*.

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*.

Liang Zhang, Anwen Hu, Haiyang Xu, Ming Yan, Yichen Xu, Qin Jin, Ji Zhang, and Fei Huang. 2024. TinyChart: Efficient chart understanding with program-of-thoughts learning and visual token merging. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1882–1898, Miami, Florida, USA. Association for Computational Linguistics.

Meng Zhang, Liangyou Li, and Qun Liu. 2022. Triangular transfer: Freezing the pivot for triangular machine translation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 644–650, Dublin, Ireland.

Danna Zheng, Mirella Lapata, and Jeff Pan. 2024. Archer: A human-labeled text-to-SQL dataset with arithmetic, commonsense and hypothetical reasoning. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 94–111, St. Julian's, Malta. Association for Computational Linguistics.

Ruiqi Zhong, Charlie Snell, Dan Klein, and Jason Eisner. 2023. Non-programmers can label programs indirectly via active examples: A case study with text-to-SQL. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5126–5152, Singapore.

A More Experiment Details

A.1 Benchmark Details

We conduct experiments on two widely recognized open-sourced text-to-SQL datasets: BIRD (Li et al., 2024d) and Archer (Zheng et al., 2024). BIRD contains over 12,751 unique question-SQL pairs derived from 95 large-scale databases spanning over 37 professional domains. The databases are designed to mimic real-world scenarios, featuring messy data rows and complex schemas. Archer is a human-labeled dataset focused on text-to-SQL queries involving Arithmetic, Commonsense, and Hypothetical Reasoning. We use its English subset with 1,042 question-SQL pairs, spanning 20 English-language databases across 20 distinct domains.

A.2 Baselines

We compare PI-SQL with ten baselines.

• **Vanilla**: This method is the same as PI-SQL, except without Python guidance.

• **C3** (Dong et al., 2023): C3 is a zero-shot text-to-SQL method that incorporates Clear Prompting, Calibration with Hints, and Consistent Output to optimize model input, mitigate biases, and maintain output consistency, respectively.

• **DIN-SQL** (Pourreza and Rafiei, 2024): DIN-SQL tackles the text-to-SQL task by decomposing it into smaller, manageable sub-tasks, solving them in an adaptive in-context learning framework that adjusts based on the task at hand.

• **DAIL-SQL** (Gao et al., 2024a): DAIL-SQL uses code prompts to represent the query and selects in-context learning examples based on the query and its pre-generated SQL.

• **TA-SQL** (Qu et al., 2024): TA-SQL leverages schema linking and logical synthesis alignment modules, in conjunction with in-context learning, to mitigate hallucinations.

• **R^3** (Xia et al., 2024): R^3 establishes a framework with direct Python guidance and consensus-based refinement for text-to-SQL tasks.

• **CHESS** (Talaei et al., 2024): CHESS is a multi-agent framework for text-to-SQL using in-context learning, consisting of agents such as the Information Retriever, Schema Selector, Candidate Generator, and Unit Tester. For a fair comparison, we exclude the Unit Tester agent.

• **CHASE-SQL** (Pourreza et al., 2025): CHASE-SQL enhances text-to-SQL performance by utilizing a divide-and-conquer generation approach, chain-of-thought reasoning for refinement, and instance-aware synthetic few-shot example generation. Additionally, it trains a candidate selection model using the BIRD training set.

• **E-SQL** (Caferoğlu and Özgür Ulusoy, 2024): E-SQL leverages direct schema linking via question enrichment and incorporates candidate predicates to address key challenges in text-to-SQL, including

complex schemas, query ambiguity, and intricate SQL generation.

• **RSL-SQL** (Cao et al., 2024): RSL-SQL combines techniques including bidirectional schema linking, contextual information augmentation, binary selection strategy, and multi-turn self-correction, achieving robust schema linking and thus improving text-to-SQL performance.

A.3 Data License and Usage

The BIRD dataset is licensed under CC BY-SA 4.0, and the Archer dataset is licensed under CC BY 4.0. Both datasets are used exclusively for academic research purposes following their respective licenses.

B More Experiment Results

B.1 Inference Cost Comparision

Table 8 and Table 9 quantify the average inference cost per query on the BIRD dev set across different methods. While PI-SQL incurs higher computational costs on the BIRD dev set overall than vanilla (by 6.136$), C3 (by 6.485$), DAIL-SQL (by 3.300$), TA-SQL (by 4.178$), and RSL-SQL (by 2.025$), it remains less than DIN-SQL (by 4.421$), CHESS (by 10.608$), R^3 (by 2.966$), E-SQL (by 2.854$), and CHASE-SQL (by 1.108$). Critically, this modest cost increase is justified by PI-SQL's significant performance gains over all baselines (see Table 1), demonstrating its practical efficiency for real-world deployment.

B.2 Case Study for Code Generation

We present a comparative analysis in Figure 4, Figure 5, and Figure 6 to further demonstrate the necessity of incorporating these code generation methods. These figures illustrate query execution strategies and code generation outputs across three methods: direct, merge, and filter. Direct code employs sequential task execution without table merging, instead explicitly linking data through foreign keys. This method prioritizes simplicity and transparency. Merge code first consolidates tables into a joined dataset before computing metrics, favoring holistic data integration. Filter code optimizes efficiency by eliminating irrelevant data at the early stages of processing. These approaches exhibit distinct characteristics, with each proving optimal under specific problem constraints. For instance, direct code excels in straightforward scenarios requiring traceability, while merge code suits complex multi-table analyses, and filter code benefits resource-intensive tasks. This observation aligns with the conclusions in Section 5.3, where we analyze method selection patterns across varying difficulty levels and LLM backbones.

B.3 Complementary Python Programs performance

We provide additional performance results of the Python programs on the BIRD dev set at various difficulty levels in Table 10.

To provide a deeper insight into the performance of PIP and PIS, Table 11 presents the R-VES scores corresponding to the results in Table 6. The results reveal that executing the logic directly in Python is less efficient than executing the final generated SQL. This finding is crucial: it underscores the importance of our Python-to-SQL approach. While Python serves as an excellent intermediate language for logical reasoning and planning, generating native SQL is vital for achieving the high performance required in real-world data processing tasks.

B.4 Complementary Contribution of Different Python Generation Strategies

We present the distribution of Python strategy selections on the BIRD dev set across various difficulty levels in Table 12.

B.5 Comparison with Fine-tuned Methods

In this section, we compare PI-SQL to methods that are explicitly fine-tuned on the text-to-SQL task. Specifically, we compare with CodeS (Li et al., 2024b) and RESD-SQL (Li et al., 2023b). CodeS are state-of-the-art, fully open-source language models (1B–15B parameters) designed for the text-to-SQL task. It employs incremental pre-training on a curated SQL-centric corpus, enhancing SQL generation, schema linking, and domain adaptation via strategic prompt construction and bi-directional data augmentation. RESD-SQL is fine-tuned from T5 series models on the Spider training set, with an explicitly designed model structure to decouple schema linking from skeleton parsing.

We find in Table 13 that PI-SQL consistently outperforms the fine-tuned baselines by 6.04 to 31.44 EX and 19.59 to 47.08 VES overall, demonstrating its superiority in terms of both accuracy and efficiency. The performance gap is more pronounced in the moderate and challenging level subsets, indicating that guiding with a high-resource

Method	Overall		Simple		Moderate		Challenging	
	Input Avg.	Output Avg.	Input Avg.	Output Avg.	Input Avg.	Output Avg.	Input Avg.	Output Avg.
Vanilla	1005	272	950	244	1085	306	1094	341
C3	529	138	524	111	536	161	537	235
DIN-SQL	28185	1124	23555	905	37073	1422	29275	1569
DAIL-SQL	3153	1789	2988	1686	3364	1939	3530	1968
TA-SQL	6917	212	7446	195	6192	221	5859	288
CHESS	42573	2008	41256	1683	44623	2397	44416	2837
R^3	14578	3472	12970	3656	16687	3109	18092	3465
E-SQL	25335	702	25996	816	24003	886	25409	754
CHASE-SQL	12878	2551	12080	1225	13745	1244	15202	1752
RSL-SQL	12207	450	12518	454	11849	434	11461	470
PI-SQL	8117	2938	7666	2734	8744	3191	8987	3431

Table 8: Inference token usage of different methods in average on the BIRD dev set.

Method	Overall		Simple		Moderate		Challenging	
	Input ($)	Output ($)	Input ($)	Output ($)	Input ($)	Output ($)	Input ($)	Output ($)
Vanilla	0.347	0.376	0.198	0.204	0.113	0.128	0.036	0.045
C3	0.183	0.191	0.109	0.093	0.056	0.067	0.018	0.031
DIN-SQL	9.728	1.552	4.902	0.754	3.870	0.594	0.955	0.205
DAIL-SQL	1.088	2.471	0.622	1.404	0.351	0.810	0.115	0.257
TA-SQL	2.388	0.293	1.550	0.163	0.647	0.093	0.191	0.038
CHESS	14.694	2.773	8.586	1.401	4.659	1.001	1.449	0.370
R^3	5.032	4.793	2.699	3.044	1.742	1.298	0.590	0.452
E-SQL	8.744	0.969	5.410	0.679	2.506	0.370	0.829	0.098
CHASE-SQL	4.445	3.522	2.514	1.020	1.435	0.519	0.496	0.229
RSL-SQL	4.213	0.621	2.605	0.378	1.237	0.181	0.374	0.061
PI-SQL	2.802	4.057	1.596	2.277	0.913	1.333	0.293	0.448

Table 9: Inference cost (GPT-4o-mini) of different methods for solving all queries of the BIRD dev set ($).

```
# Merge Approach

# Merge tables based on foreign key relationships
merged_df = district_df.add_prefix('district_')\
    .merge(client_df.add_prefix('client_'), left_on='district_district_id',
right_on='client_district_id')\
    .merge(disp_df.add_prefix('disp_'), left_on='client_client_id', right_on='disp_client_id')

# Filter for districts in south Bohemia
south_bohemia_df = merged_df[merged_df['district_A3'] == 'south Bohemia']

# Find the district with the maximum number of inhabitants
south_bohemia_df['district_A4'] = south_bohemia_df['district_A4'].astype(int)  # Ensure A4
(inhabitants) is treated as integer
max_inhabitants_district = south_bohemia_df.loc[south_bohemia_df['district_A4'].idxmax()]

# Get client data from the district with maximum inhabitants
filtered_clients = client_df[client_df['district_id'] ==
max_inhabitants_district['district_district_id']]

# Calculate the percentage of male clients
male_clients_count = filtered_clients[filtered_clients['gender'] == 'M'].shape[0]
total_clients_count = filtered_clients.shape[0]

# To avoid division by zero if there are no clients, set result to 0 if total_clients_count is zero
if total_clients_count > 0:
    male_percentage = (male_clients_count / total_clients_count) * 100
else:
    male_percentage = 0  # No clients means no percentage

# Store the result
result = male_percentage
```

Figure 4: A merge code generation case. The merge code merges all tables first, filters districts in "South Bohemia," and calculates metrics from the joined dataset.

```
# Filter Approach

# Filter the district for 'south Bohemia' and find the max inhabitants
south_bohemia_districts = district_df[district_df['A3'] == 'south Bohemia'].copy()
south_bohemia_districts['A4'] = south_bohemia_districts['A4'].astype(int)  # Ensure A4 is integer
for comparison
max_inhabitants_district = south_bohemia_districts.loc[south_bohemia_districts['A4'].idxmax()]

# Retrieve the district_id of the district with the most inhabitants
district_id_max = max_inhabitants_district['district_id']

# Now, filter the client data for this district_id
clients_in_district = client_df[client_df['district_id'] == district_id_max]

# Calculate the percentage of male clients (gender = 'M')
male_client_count = clients_in_district[clients_in_district['gender'] == 'M'].shape[0]
total_client_count = clients_in_district.shape[0]

# Handling division by zero
if total_client_count > 0:
    male_percentage = (male_client_count / total_client_count) * 100
else:
    male_percentage = 0  # If there are no clients, percentage is 0

# Store the result
result = male_percentage
```

Figure 5: A filter code generation case. The filter code filters districts and isolates the target district, and then filters clients, emphasizing modularity and memory efficiency.

```
# Direct Approach

# Filter for districts in the 'south Bohemia' region
south_bohemia_df = district_df[district_df['A3'] == 'south Bohemia']

# Convert the A4 column (inhabitants) to integers for calculation
south_bohemia_df['A4'] = south_bohemia_df['A4'].astype(int)

# Find the district with the maximum inhabitants
max_inhabitants_district = south_bohemia_df.loc[south_bohemia_df['A4'].idxmax()]

# Get the district_id for the district with the biggest number of inhabitants
district_id_max = max_inhabitants_district['district_id']

# Now filter clients in the identified district
clients_in_district = client_df[client_df['district_id'] == district_id_max]

# Count total clients and male clients
total_clients = len(clients_in_district)
male_clients_count = len(clients_in_district[clients_in_district['gender'] == 'M'])

# Calculate the percentage of male clients
percentage_male_clients = (male_clients_count / total_clients) * 100 if total_clients > 0 else 0

# Store the result
result = percentage_male_clients
```

Figure 6: A direct code generation case. The direct code filters districts in "South Bohemia," identifies the most populous district, and then queries client data using the retrieved district_id.

Model	Overall			Simple			Moderate			Challenging		
	VanS	PiS	PiP	VanS	PiS	PiP	VanS	PiS	PiP	VanS	PiS	PiP
Qwen-Coder	59.97	67.40	66.42	64.75	72.86	70.70	55.17	59.91	59.48	44.82	56.55	61.37
QwQ	55.08	64.34	65.58	62.48	71.13	69.08	46.33	56.89	55.38	35.86	44.82	62.06
Gemma3	58.86	66.55	62.12	64.32	72.75	67.24	51.72	57.11	54.31	46.89	57.24	54.48
GPT-4o-mini	55.35	64.54	65.44	60.86	70.92	70.27	48.28	56.47	57.75	42.76	49.66	59.31

Table 10: Performance of Python program on the BIRD dev set. VanS and PiS denote SQL performance without and with Python guidance, while PiP denotes Python performance.

Model	Overall			Challenging		
	VanS	PIS	PIP	VanS	PIS	PIP
Qwen-Coder	58.61	65.54	45.39	43.12	54.29	41.68
QwQ	54.06	62.79	52.18	33.87	43.80	46.30
Gemma3	57.51	65.62	51.96	44.49	56.41	44.48
GPT-4o-mini	55.94	63.71	57.83	42.55	49.06	47.80

Table 11: Efficiency (R-VES) of Python program on the BIRD dev set. VanS and PIS denote SQL performance without and with Python guidance, while PIP denotes Python performance.

programming language is more effective than fine-tuning for solving difficult text-to-SQL problems. The performance of fine-tuned models heavily depends on the quality and quantity of the fine-tuning data. However, creating a large-scale dataset with challenging text-to-SQL pairs is costly and difficult. Conversely, PI-SQL leverages Python to guide LLMs in handling challenging SQL programs without the need for a high-quality fine-tuning dataset.

B.6 Execution Time of Generated Code

This section details the average execution time for the Python and SQL code generated by PI-SQL (as shown in Table 14). At approximately 1 second per query, this execution time is negligible compared to the inference cost.

B.7 Combination with Refinement

To demonstrate that PI-SQL can be readily integrated with refinement techniques, we present its performance in Table 15 using a simple refinement strategy. Specifically, if no generated SQL query produces results that align with the Python-voted outcome, the SQL query is regenerated.

B.8 Scaling Experiments

To evaluate the generality and robustness of PI-SQL across models of different scales, we also conducted experiments using the Qwen2.5-Coder-Instruct series across different model sizes in Table 16. We observe that PI-SQL consistently improves both EX and R-VES scores across models. Overall, the performance gains become more pronounced with larger models, likely due to their stronger Python reasoning and generalization capabilities. This reveals the potential of our method when applied to more powerful language models. These results further demonstrate the effectiveness of our approach.

B.9 Experiments on More Advanced Open-source Models

We conduct additional experiments using the advanced open-source model, DeepSeek-V3, on the BIRD development set in Table 17. Due to budget constraints, we benchmarked our method against the vanilla approach and CHASE-SQL, which was the top-performing baseline in our main experiments. The results clearly show our method consistently outperforming both baselines, achieving gains of over 3.13 in overall EX and 5.73 in R-VES. This confirms the effectiveness of our approach when applied to more powerful language models. These new findings, when considered alongside our existing results on various open-source backbones (Section 5.3), further underscore the practical significance of our approach. The consistent performance gains across models of different scales highlight our method's utility for a wide range of real-world scenarios, including local or private deployments where capable open-source LLMs are increasingly preferred.

B.10 Details of Recursive Query Case Study

The full details of this case study are presented below. Figure 7 shows the database schema and the natural language question. One of the intermediate Python programs generated by our method is detailed in Figure 8. Finally, Figure 9 and Figure 10 present the final SQL generated by both the vanilla method and PI-SQL for comparison.

C Prompt Templates

C.1 Prompt Templates for Intermediate Guidance Preparation

The prompt used for the generation of Python program guidance in three different strategies(merge, filter, and direct) is available in Figure 11, Figure 12, and Figure 13, respectively.

C.2 Prompt Template for SQL Generation

The prompt template used to generate the SQL response at the second stage is available in Figure 14.

D Broader Impacts

Our PI-SQL method allows non-technical users to generate SQL queries using natural language, improving productivity and making data more accessible. It can benefit fields like healthcare, finance, and education by enabling faster, data-driven decisions without requiring SQL expertise. However,

Model	Overall			Simple			Moderate			Challenging		
	Merge	Filter	Direct	Merge	Filter	Direct	Merge	Filter	Direct	Merge	Filter	Direct
Qwen-Coder	34.49	35.27	30.25	35.35	34.49	30.16	33.62	37.50	28.88	31.72	33.10	35.17
QwQ	34.49	31.42	34.09	35.03	32.32	32.65	33.84	32.54	33.62	33.10	22.07	44.83
Gemma3	31.68	33.90	34.42	31.57	34.92	33.51	32.54	31.47	35.99	29.66	35.17	35.17
GPT-4o-mini	31.94	34.16	33.90	31.14	35.46	33.41	32.54	32.54	34.91	35.17	31.03	33.79

Table 12: Distribution of Python selection rates across different Python generation strategies. The percentages represent the proportion of SQL outputs generated by each strategy (merge, filter, or direct) that are selected as the final answer, summing to 100% for each model across all strategies.

```
Schema:
# Table: atom
[
(molecule_id:TEXT, ColumnComment: #The molecule id column in the atom table is used to identify the
molecule to which the atom belongs. Commonsense evidence: TRXXX_i represents ith atom of molecule
TRXXX. Example values include TR186, TR447, and TR040., Example: [TR000, TR001, TR002]),
(atom_id:TEXT, Primary Key, ColumnComment: #The atom id column in the atom table is used to store
the unique id of atoms. Example values include TR238_12, TR296_49, and TR007_21., Example: [TR000_1,
TR000_2, TR000_3])
]
# Table: bond
[
(bond_id:TEXT, Primary Key, ColumnComment: #bond_id is a unique id representing bonds within the
bond table. The value of bond_id is a string of the format TRxxx_A1_A2, where TRXXX refers to which
molecule, A1 and A2 refers to which atom. For example, bond_id can be TR301_28_32, TR047_2_4,
TR181_18_34., Example: [TR000_1_2, TR000_2_3, TR000_2_4]),
(molecule_id:TEXT, ColumnComment: #This column identifies the molecule in which the bond appears.
For example, TR094, TR475, and TR438 are some of the values that appear in this column., Example:
[TR000, TR001, TR002]),
(bond_type:TEXT, ColumnComment: #bond_type is a text column in the bond table. It represents the
type of the bond. The possible values are ['#', '=', '-']. The commonsense evidence is:
#'-': single bond
#'=': double bond
#'#': triple bond, Example: [-, =, #])
]
# Table: molecule
[
(molecule_id:TEXT, Primary Key, ColumnComment: # 'molecule id is a unique identifier of a molecule
in the molecule table. The values in this column are text. The possible values in this column are
TR398, TR158, and TR134. The value description for this column is as follows: '+' indicates that
the molecule is carcinogenic, while '-' indicates that the molecule is not carcinogenic. For
example, TR398 is carcinogenic, while TR158 and TR134 are not carcinogenic.', Example: [TR000,
TR001, TR002]),
(label:TEXT, ColumnComment: #This column is called label and is of type text. It is a column in the
molecule table of the toxicology database. The column describes whether this molecule is
carcinogenic or not. The possible values for this column are + and -., Example: [+])
]

Foreign keys:
atom.molecule_id=molecule.molecule_id
bond.molecule_id=molecule.molecule_id

Question:
For molecule_id = 'TR000' and start_atom_id = 'TR000_1', which atoms should be included in the
output?
```

Figure 7: Schema descriptions and the natural language question of the recursive query case study.

An intermediate Python program:

```
import pandas as pd
from collections import defaultdict, deque

# Read CSV files
atoms_df = pd.read_csv('atom.csv')
bonds_df = pd.read_csv('bond.csv')

def get_connected_atoms(molecule_id, start_atom_id):
    # Step 1: Filter bonds that belong to the current molecule
    mol_bonds = bonds_df[bonds_df['molecule_id'] == molecule_id]

    # Step 2: Build the graph (adjacency list)
    graph = defaultdict(list)

    for _, row in mol_bonds.iterrows():
        bond_id = row['bond_id']
        # Format like TR000_1_2 → split out the atom indices
        parts = bond_id.split('_')
        atom1 = f"{molecule_id}_{parts[1]}"
        atom2 = f"{molecule_id}_{parts[2]}"
        graph[atom1].append(atom2)
        graph[atom2].append(atom1)  # Undirected edge

    # Step 3: Breadth-First Search (BFS) traversal
    visited = set()
    queue = deque()

    queue.append(start_atom_id)
    visited.add(start_atom_id)

    while queue:
        current = queue.popleft()
        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return list(visited)

# Test the function
molecule_id = 'TR000'
start_atom_id = 'TR000_1'

result = get_connected_atoms(molecule_id, start_atom_id)
print("Connected atoms:", result)
```

Figure 8: An intermediate Python program of the recursive query case study.

```
Vanilla SQL:
WITH RECURSIVE BondAtoms AS (
    SELECT
        bond_id,
        molecule_id,

        -- Extract atom1 and atom2; e.g., from bond_id = 'TR000_1_2' build TR000_1 and TR000_2
        molecule_id || '_' ||
        SUBSTR(bond_id, LENGTH(molecule_id) + 2,
            INSTR(SUBSTR(bond_id, LENGTH(molecule_id) + 2), '_') - 1
        ) AS atom1,

        molecule_id || '_' ||
        SUBSTR(
            bond_id,
            LENGTH(molecule_id) + 2 + INSTR(SUBSTR(bond_id, LENGTH(molecule_id) + 2), '_')
        ) AS atom2

    FROM bond
    WHERE molecule_id = 'TR000'
),

AtomGraph AS (
    -- Starting atom (root)
    SELECT atom_id FROM atom
    WHERE atom_id = 'TR000_1' AND molecule_id = 'TR000'

    UNION

    -- Recursively find other atoms connected through bonds
    SELECT
        CASE
            WHEN ag.atom_id = ba.atom1 THEN ba.atom2
            ELSE ba.atom1
        END AS atom_id
    FROM AtomGraph ag
    JOIN BondAtoms ba
      ON ag.atom_id = ba.atom1 OR ag.atom_id = ba.atom2
)

-- Final output: all connected atom_id values
SELECT DISTINCT atom_id FROM AtomGraph;
```

Figure 9: The final SQL generated by the vanilla method of the recursive query case study.

```
PI-SQL:
WITH RECURSIVE BondAtoms AS (
    SELECT
        bond_id,
        molecule_id,

        -- Replace bond_id with TR000_1 and TR000_2 (the two atoms)
        molecule_id || '_' ||
            substr(
                replace(bond_id, molecule_id || '_', ''),
                1,
                instr(replace(bond_id, molecule_id || '_', ''), '_') - 1
            ) AS atom1,

        molecule_id || '_' ||
            substr(
                replace(bond_id, molecule_id || '_', ''),
                instr(replace(bond_id, molecule_id || '_', '') , '_') + 1
            ) AS atom2

    FROM bond
    WHERE molecule_id = 'TR000'
),

AtomGraph AS (
    -- Starting atom
    SELECT 'TR000_1' AS atom_id

    UNION

    -- Recursively find connected atoms
    SELECT
        CASE
            WHEN ag.atom_id = ba.atom1 THEN ba.atom2
            ELSE ba.atom1
        END AS atom_id
    FROM AtomGraph ag
    JOIN BondAtoms ba
      ON ag.atom_id = ba.atom1 OR ag.atom_id = ba.atom2
)

SELECT DISTINCT atom_id
FROM AtomGraph
ORDER BY atom_id;
```

Figure 10: The final SQL generated by PI-SQL of the recursive query case study.

Method	Overall		Simple		Moderate		Challenging	
	EX	VES	EX	VES	EX	VES	EX	VES
SFT CodeS-1B	50.30	52.45	58.70	61.11	37.60	39.89	36.80	37.38
SFT CodeS-3B	54.90	58.28	62.80	64.96	44.30	50.98	38.20	38.99
SFT CodeS-7B	57.00	60.83	64.60	66.88	46.90	49.53	40.30	58.42
SFT CodeS-15B	58.50	61.54	65.80	67.87	48.80	51.69	42.40	52.71
RESDSQL-Base	33.10	34.05	42.30	42.75	20.20	22.16	16.00	16.54
RESDSQL-Large	38.60	40.81	46.50	47.21	27.70	30.00	22.90	34.67
RESDSQL-3B	43.90	45.64	53.50	53.35	33.30	35.49	16.70	28.84
PI-SQL	**64.54**	**81.13**	**70.92**	**88.88**	**56.47**	**71.06**	**49.66**	**63.92**

Table 13: Comparison with fine-tuned models on the BIRD dev set. The baseline results are cited from Li et al. (2024a). Since the baselines exclusively report VES, this table presents VES scores rather than R-VES. The best result for each case is highlighted in **bold**.

Method	Overall		Simple		Moderate		Challenging	
	Python	SQL	Python	SQL	Python	SQL	Python	SQL
PI-SQL	0.961	0.166	0.905	0.170	1.036	0.170	1.082	0.124

Table 14: Execution time (s) of the generated code (average per query).

the method could also be misused to query leaked or unauthorized databases, risking privacy breaches. To address this, robust access controls and privacy safeguards must be implemented to ensure responsible use.

Method	Overall		Simple		Moderate		Challenging	
	EX	R-VES	EX	R-VES	EX	R-VES	EX	R-VES
PI-SQL	64.54	63.71	70.92	70.06	56.47	55.63	49.66	49.06
+refinement	**65.12**	**66.85**	**71.03**	**73.01**	**56.90**	**58.17**	**53.79**	**55.40**

Table 15: The results of PI-SQL with a simple refinement strategy.

Model	Overall		Simple		Moderate		Challenging	
	EX	R-VES	EX	R-VES	EX	R-VES	EX	R-VES
Vanilla method								
Qwen2.5-Coder-7B	51.63	49.76	60.97	59.10	39.66	37.55	30.34	29.27
Qwen2.5-Coder-14B	61.21	58.20	68.43	64.95	51.94	49.72	44.83	42.31
Qwen2.5-Coder-32B	59.97	58.61	64.76	63.64	55.17	53.40	44.83	43.12
With PI-SQL								
Qwen2.5-Coder-7B	54.24	56.02	62.81	64.93	42.24	43.46	37.93	39.33
Qwen2.5-Coder-14B	65.45	67.77	71.78	74.41	56.47	58.19	53.79	56.07
Qwen2.5-Coder-32B	67.40	65.54	72.86	71.14	59.91	57.89	56.55	54.29

Table 16: Performance of PI-SQL with LLMs of different scales on the BIRD dev set.

Model	Overall		Simple		Moderate		Challenging	
	EX	R-VES	EX	R-VES	EX	R-VES	EX	R-VES
Vanilla	59.58	55.74	66.92	63.00	50.65	46.94	41.38	37.66
CHASE-SQL	62.91	62.65	67.57	67.56	57.11	56.41	**51.72**	**51.29**
PI-SQL	**66.04**	**68.38**	**71.35**	**74.22**	**60.99**	**62.47**	48.28	50.08

Table 17: Performance of PI-SQL with DeepSeek-V3 on the BIRD dev set. The best results for each case are highlighted in bold.

```
You are an expert in database querying and proficient in handling data stored in CSV files, treating them as database
tables. The current database has been exported to corresponding CSV files. Your task is to answer user queries by
writing Python code that queries these CSV files. You will be provided with examples of past queries in the database
and their SQL solutions. Use the SQL logic as a reference to formulate Python code solutions for the CSV files.
Focus on providing accurate and efficient Python code that directly queries the CSV files and produces the desired result.


you should answer what csv file to use and what columns to use, and what messages you will answer before you write the
code. When answering, carefully distinguish the following four elements:
The content mentioned in the question.
The content mentioned in the evidence.
The description provided in the ValueComment.
The data format provided in the Example.

Before proceeding with any computations, you need to first merge the relevant tables.
Before merge data, use .add_prefix('tablename') to avoid the name conflicts.
Pay attention to the foreign key relationship between tables and you should merge the tables based on the foreign key
relationship like SQL does before you answer the question.
You should only use the columns in the schema and the foreign key given you. Don't use other columns.

When filtering data, pay close attention to the format of the data in the Example field, as it represents the actual
data stored in the database. Be especially cautious with strings and dates to ensure their format matches exactly,
including time zones and separators.
Given a question and evidence, extract relevant information or make a judgment. However, please note that case
sensitivity may exist between the question and evidence. To avoid issues caused by mismatched casing, always refer to
the examples provided in Example: []. Use them as guidance to handle case sensitivity properly.

When answering the question, ensure the result directly addresses the query. Specifically:
1. If the question asks for a field name, return the corresponding column values.
2. If the question asks for an ID or code, ensure the correct ID or code column is used.
3. For questions about details, identity codes, or similar information, prioritize inspecting the column names to
locate the relevant field.
4. For comparison questions, directly state the answer (e.g., the item or option) without restating the context or
providing explanations.
5. The result order should be same as the question. For example, "what is the name and id of the one with the most
students registered?" you should return [(name), (id)] not [(id), (name)].

When you process data, you should follow:
1. Unless the question requires it, directly return the relevant data without any additional processing, such as
string concatenation, truncation, etc.
2. The result should match question, exactly without other info.
3. x between a and b means a <= x <= b, or equivalently, x >= a and x <= b.
4. You should follow evidence.

Please answer the following question based on the given data structure. The CSV file contains the following columns:
{csv_schema}
Using python code, answer the following questions for the csv file provided above.
Question: {question}
Extral knowledge: {evidence}
Use the the following csv path to read the csv file: {csv_path}
Here is some tables, columns you should pay attention: {tentative_schema}

After filtering and processing the data, use print(result) in the end. The final answer should be stored in the
variable result, and it should be either:
1. An array (list).
2. A number (integer or float).
3. A string.

Your code should be \\\python and \\\and the code should be valid python code.
```

Figure 11: Template for CSV-based Code Generation

```
You are an expert in database querying and proficient in handling data stored in CSV files, treating them as database
tables. The current database has been exported to corresponding CSV files. Your task is to answer user queries by
writing Python code that queries these CSV files. You will be provided with examples of past queries in the database
and their SQL solutions. Use the SQL logic as a reference to formulate Python code solutions for the CSV files.
Focus on providing accurate and efficient Python code that directly queries the CSV files and produces the desired result.


you should answer what csv file to use and what columns to use, and what messages you will answer before you write the
code. When answering, carefully distinguish the following four elements:
The content mentioned in the question.
The content mentioned in the evidence.
The description provided in the ValueComment.
The data format provided in the Example.

Prior to performing any calculations, avoid aggregating tables unless absolutely necessary. Focus first on filtering
out the relevant data, then use that filtered data to derive the correct answer. This approach ensures that you're
working with only the necessary information before performing any complex operations.
Before merge data, use .add_prefix('tablename') to avoid the name conflicts.
Pay attention to the foreign key relationship between tables and you should merge the tables based on the foreign key
relationship like SQL does before you answer the question.
You should only use the columns in the schema and the foreign key given you. Don't use other columns.

When filtering data, pay close attention to the format of the data in the Example field, as it represents the actual
data stored in the database. Be especially cautious with strings and dates to ensure their format matches exactly,
including time zones and separators.
Given a question and evidence, extract relevant information or make a judgment. However, please note that case
sensitivity may exist between the question and evidence. To avoid issues caused by mismatched casing, always refer to
the examples provided in Example: []. Use them as guidance to handle case sensitivity properly.

When answering the question, ensure the result directly addresses the query. Specifically:
1. If the question asks for a field name, return the corresponding column values.
2. If the question asks for an ID or code, ensure the correct ID or code column is used.
3. For questions about details, identity codes, or similar information, prioritize inspecting the column names to
locate the relevant field.
4. For comparison questions, directly state the answer (e.g., the item or option) without restating the context or
providing explanations.
5. The result order should be same as the question. For example, "what is the name and id of the one with the most
students registered?" you should return [(name), (id)] not [(id), (name)].

When you process data, you should follow:
1. Unless the question requires it, directly return the relevant data without any additional processing, such as
string concatenation, truncation, etc.
2. The result should match question, exactly without other info.
3. x between a and b means a <= x <= b, or equivalently, x >= a and x <= b.
4. You should follow evidence.

Please answer the following question based on the given data structure. The CSV file contains the following columns:
{csv_schema}
Using python code, answer the following questions for the csv file provided above.
Question: {question}
Extral knowledge: {evidence}
Use the the following csv path to read the csv file: {csv_path}
Here is some tables, columns you should pay attention: {tentative_schema}

After filtering and processing the data, use print(result) in the end. The final answer should be stored in the
variable result, and it should be either:
1. An array (list).
2. A number (integer or float).
3. A string.

Your code should be \\\python and \\\and the code should be valid python code.
```

Figure 12: Template for code generate merge

```
You are an expert in database querying and proficient in handling data stored in CSV files, treating them as database
tables. The current database has been exported to corresponding CSV files. Your task is to answer user queries by
writing Python code that queries these CSV files. You will be provided with examples of past queries in the database
and their SQL solutions. Use the SQL logic as a reference to formulate Python code solutions for the CSV files.
Focus on providing accurate and efficient Python code that directly queries the CSV files and produces the desired result.


you should answer what csv file to use and what columns to use, and what messages you will answer before you write the
code. When answering, carefully distinguish the following four elements:
The content mentioned in the question.
The content mentioned in the evidence.
The description provided in the ValueComment.
The data format provided in the Example.

Before merge data, use .add_prefix('tablename') to avoid the name conflicts.
Pay attention to the foreign key relationship between tables and you should merge the tables based on the foreign key
relationship like SQL does before you answer the question.
You should only use the columns in the schema and the foreign key given you. Don't use other columns.

When filtering data, pay close attention to the format of the data in the Example field, as it represents the actual
data stored in the database. Be especially cautious with strings and dates to ensure their format matches exactly,
including time zones and separators.
Given a question and evidence, extract relevant information or make a judgment. However, please note that case
sensitivity may exist between the question and evidence. To avoid issues caused by mismatched casing, always refer to
the examples provided in Example: []. Use them as guidance to handle case sensitivity properly.

When answering the question, ensure the result directly addresses the query. Specifically:
1. If the question asks for a field name, return the corresponding column values.
2. If the question asks for an ID or code, ensure the correct ID or code column is used.
3. For questions about details, identity codes, or similar information, prioritize inspecting the column names to
locate the relevant field.
4. For comparison questions, directly state the answer (e.g., the item or option) without restating the context or
providing explanations.
5. The result order should be same as the question. For example, "what is the name and id of the one with the most
students registered?" you should return [(name), (id)] not [(id), (name)].

When you process data, you should follow:
1. Unless the question requires it, directly return the relevant data without any additional processing, such as
string concatenation, truncation, etc.
2. The result should match question, exactly without other info.
3. x between a and b means a <= x <= b, or equivalently, x >= a and x <= b.
4. You should follow evidence.

Please answer the following question based on the given data structure. The CSV file contains the following columns:
{csv_schema}
Using python code, answer the following questions for the csv file provided above.
Question: {question}
Extral knowledge: {evidence}
Use the the following csv path to read the csv file: {csv_path}
Here is some tables, columns you should pay attention: {tentative_schema}

After filtering and processing the data, use print(result) in the end. The final answer should be stored in the
variable result, and it should be either:
1. An array (list).
2. A number (integer or float).
3. A string.

Your code should be \\\python and \\\and the code should be valid python code.
```

Figure 13: Template for code generate direct

You are an expert Python developer specializing in data analysis and database management. Your role is to assist users in transforming data queries originally written for CSV files into SQL queries for SQLite databases. Your solutions should be precise, efficient, and easy to understand.

1. **Understanding the Original Code Logic:**
* Analyze the provided Python code or logic written for CSV files.
* Determine the user's intent and interpret the expected functionality based on the original question.

2. **Writing Precise SQL Queries:**
* Write SQL queries that replicate the same functionality or results when executed on the SQLite database.
* Pay attention to:
- Spaces in the column name: use double quotes to quote the column name.
- Calculations on columns: ensure column types are correct, using the `CAST` function for type conversion.
* Strictly follow the user's requirements:
- Select only the columns explicitly mentioned in the user's question. Avoid including unnecessary columns or values.
* Use SQLite functions only: for example, use `STRFTIME()` for date manipulation (e.g., `STRFTIME('%Y', SOMETIME)` to extract the year).For non-standard formats (e.g., 'YYYYMM'), use `SUBSTR()` to manually extract parts (e.g., `SUBSTR(date_column, 1, 4)` to extract the year).

3. **Ensuring Accuracy of Results:**
* Validate that the logic in the query matches the user's intent.
* Pay attention to the output to ensure it fulfills the question's requirements. Even if the original code logic is correct, ensure the result aligns precisely with the user's request.

4. **Optimizing for SQLite:**
* Translate Python or CSV-based operations into efficient SQL syntax suitable for SQLite.
* Use `GROUP BY`, `DISTINCT`, and other SQLite-specific functionalities where applicable.

Question: {question}
Extral knowledge: {evidence}
Database structure: {db_schema}
Original Python code to solve the question: {python_code}
Return: "'`{sql}`"' and the code should be valid SQL code.

Figure 14: Template for code guide SQL