TeleAI at SemEval-2025 Task 8: Advancing Table Reasoning Framework with Large Language Models

Sishi Xiong, Mengxiang Li, Dakai Wang, Yu Zhao, Jie Zhang, Changzai Pan, Haowei He, Xiangyu Li, Wenhan Chang, Zhongjiang He*, Shuangyong Song*, Yongxiang Li

Institute of Artificial Intelligence (TeleAI), China Telecom Corp Ltd

Correspondence: {xiongsishi, hezj, songshy}@chinatelecom.cn

Abstract

The paper presents our system developed for SemEval-2025 Task 8, which focuses on table question answering (TQA). TQA tasks face challenges due to the characteristics of real-world tabular data, such as large size, incomplete column semantics, and entity ambiguity. To address these issues, we propose a large language model (LLM)-powered and programming-based table reasoning framework, named TableReasoner. It models a table using the schema that combines structural and semantic representations, enabling holistic understanding and efficient processing of large tables. We design a multi-step schema linking plan to derive a focused table schema that retains only query-relevant information, eliminating ambiguity and alleviating hallucinations. This focused table schema provides precise and sufficient table details for query refinement and programming. Furthermore, we integrate the reasoning workflow into an iterative thinking architecture, allowing incremental cycles of thinking, reasoning and reflection. Our system achieves first place on both subtasks[1].

1 Introduction

Table Question Answering is a spin-off Question Answering (QA) task, aiming at responding to natural language questions based on table data that feature heterogeneous and complex two-dimensional structure (Lu et al., 2025). Compared to tasks involving unstructured or plain text, TQA faces greater challenges and emphasizes the model's reasoning abilities. The primary challenges encompass large size, incomplete semantics, and entity ambiguity. To advance research on table understanding by applying language models, SemEval-2025 introduces Task 8: Question Answering on Tabular Data (Osés-Grijalba et al., 2025).

In this paper, we present TableReasoner, a systematic LLM-powered and programming-based framework for TQA. We introduce the table schema as the table representation, instead of entire or truncated table text in conventional format (such as CSV or Markdown), enabling our framework capable of processing large tables. TableReasoner employs a programming module to solve questions using the table schema to understand the table content from a holistic perspective. To ensure precise scheduling for query decomposition and programming, we design a "parsing-linking-refinement" action flow. It refines the global table schema into a focused one only associated with the original query, to some extent alleviating model hallucinations. In addition, inspired by the ReAct (Yao et al., 2023) paradigm, we incorporate the reasoning workflow into a "thought-action-observation" architecture, facilitating iterative cycles of thinking, reasoning and reflection within our system.

TableReasoner is flexible to any advanced language model. It delivers excellent results even without fine-tuning, while further performance enhancements can be achieved through fine-tuning and majority voting. Our system wins first place on both subtasks, validating the superiority and scalability of our framework.

2 Background

2.1 DataBench Dataset

DataBench (Osés Grijalba et al., 2024) is an English benchmark comprising 80 real-world tables, with splits of 49/16/15 for training, development, and testing for TQA task. It contains five types of question-answer pair: boolean, category, number, list[category], and list[number]. The test set contains 522 human-annotated question-answer pairs. We categorize the tables of test set into large, medium, and small sets based on the number of cells. The distribution of QA types and the divi-

[1]Codes: https://github.com/ccx06/TableReasoner.

* Corresponding authors.

Proceedings of the The 19th International Workshop on Semantic Evaluation (SemEval-2025), pages 1828–1841

sion of size are detailed in Appendix A. DataBench offers a reduced version called DataBench Lite, where each table retains the first 20 rows of data. Correspondingly, the competition includes two subtasks: subtask A on DataBench and subtask B on DataBench Lite.

2.2 Related works

TableLlama (Zhang et al., 2024a), TableGPT (Zha et al., 2023), and StructLM (Zhuang et al., 2024) continue to pre-train models with the decoder-only architecture like Llama(Touvron et al., 2023) towards various table-related tasks. However, table tuning demands a substantial amount of high-quality labeled data, which may be difficult to obtain in certain domains.

With the progress of LLMs, the paradigm has increasingly shifted from traditional models pre-trained from scratch or task-specific modules designed for narrow applications (Liu et al., 2023; Xiong et al., 2024) to leveraging LLMs' strong reasoning and emergent abilities for adapting to downstream tasks (Ruan et al., 2024; Wu et al.; Li et al., 2024b; Shao and Li, 2025). LEVER (Ni et al., 2023), Binder (Cheng et al., 2023), PoTable (Mao et al., 2024) and OpenTab (Kong et al., 2024) are programming-based methods, solving questions with the assistance of SQL or Python codes. However, these methods exhibit limitations in fully comprehending the relationships between questions and table data. Dater (Ye et al., 2023), Chain-of-Table (Wang et al., 2024) and ReActTable (Zhang et al., 2024b) prompt LLMs iteratively to locate critical rows and columns. These methods feed the entire table text into the model, which are usually not applicable for larger tables due to inherent context length constraint. TableRAG (Chen et al., 2024) introduces a million-token table understanding framework, while the encoding and matching operations result in accuracy loss and additional budget.

3 System Overview

Our system is implemented using TableReasoner, an LLM-powered and programming-based framework, as shown in Figure 1. It's designed to integrate a reasoning workflow into an iterative thinking process.

3.1 Reasoning Workflow

Table Schema Generation. We utilize the table schema to describe and provide table information to LLMs. Firstly, we use Python to read the spreadsheet file. Each column is considered as a distinct feature, characterized by data type and meta statistical attributes (e.g., the maximum, minimum, mean and median values of numerical data; unique categories and the most frequently occurring items of categorical data). Additionally, K random rows are selected as example values. Then, to disambiguate specific column names (such as abbreviations), we incorporate the latent semantics of column names into the table schema. In detail, we treat the above table metadata as a preliminary schema to prompt the LLM to generate descriptions of the table and each column. The global table schema is structured as JSON format, an illustrative example is provided in Appendix B.

This table representation method expands the capacity of TableReasoner in processing large-sized tables. The token complexity of a table schema is approximately $O(N)$, which is significantly lower than the complexity $O(M \times N)$ of the entire table when M is very large, where M and N denote the number of rows and columns respectively.

Table Schema Linking. We propose a "parsing-linking-refinement" action flow to refine the global table schema into a focused schema strongly associated with the query. (i) Parsing. We prompt LLM to parse the query based on the table schema, decoupling it into specific sequential sub-queries. (ii) Linking. During parsing, the LLM is also prompted to extract relevant columns for each sub-query, a process referred to as *column linking*. To align the entities mentioned in the query with the values in the table, an *entity linking* step is employed. Specifically, the LLM is used to identify whether the query contains entities, extract them, and suggest belonging table columns. Next, Python is utilized to read all elements of the corresponding columns, and the Longest Common Subsequence algorithm is applied to retrieve elements from column entries whose overlap rate with the query entity exceeds 0.6. Finally, we use the LLM to precisely select the aligned value from these recalled elements. For instance, "*Mr Harari*" in the query is linked to "*Yuval Noah Harari*" in the table. (iii) Refinement. At last, we prune irrelevant columns from the global table schema and integrate entity alignment information as needed, yielding a focused table schema. This module effectively reduces token consumption and noise input.

Query Refinement. To enhance comprehension of complex reasoning tasks, we further decompose

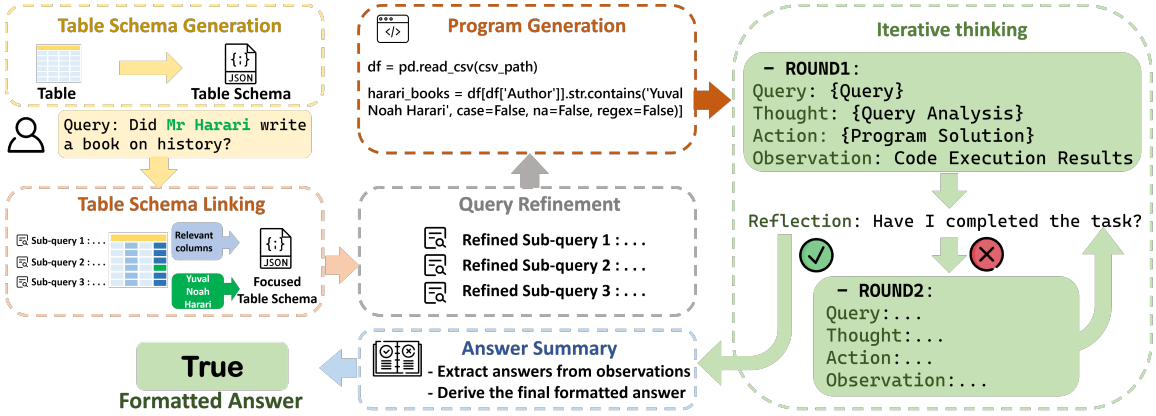

Figure 1: Architecture of TableReasoner framework. The table is firstly converted into a table schema containing global features and example values. The table schema and query go through a reasoning workflow which comprising 5 core sequential modules to obtain observations. The reasoning process and observations are then fed into an iterative thinking framework. It's repeated until expectations are met.

the query into S progressive sub-queries along with associated column names using Chain-of-Thought (CoT) prompting (Wei et al., 2022). The sole distinction from query parsing in table schema linking stage lies in employing the focused table schema which is more information-dense and de-noised.

Program-assisted Solution Generation. The TableReasoner framework centers on programming to derive precise results. Specifically, we guide the LLM to generate a Program-of-Thoughts (PoT) (Chen et al., 2023) solution, utilizing the focused table schema and refined queries from prior steps. The generated codes are subsequently executed in isolated environments (e.g., Python interpreter) to produce verifiable results. Compared to only textual reasoning approach, the program-assisted solution can effectively mitigate numerical hallucinations in multi-step data acquisition and processing.

Answer Summary. Due to the strict answer format requirements of DataBench, we introduce an answer summary module at the end of the workflow. This module generates the final formatted answer by summarizing intermediate thoughts and observations derived from Python executions during the iterative reasoning process.

3.2 Iterative Thinking Paradigm

Inspired by the ReAct, we design the iterative thinking paradigm, integrating the reasoning workflow into a "*thought-action-observation*" architecture, with the goal of bringing incremental self-reflection and decision-making mechanisms into the framework. "*Thought*" corresponds to the decomposed sub-queries from the Query Refinement stage; "*action*" represents the program ideas and codes generated during the Program-assisted Solution Generation stage; and "*observation*" is the feedback from code execution. This creates a reasoning cycle in our workflow. Upon completion of each cycle, the system evaluates whether the query can be answered with the current reasoning state. If yes, it proceeds to the Answer Summary stage; otherwise, it generates a new follow-up query and repeats the process.

3.3 Supervised Fine-tuning

For the purpose of boosting the QA ability on DataBench dataset, we fine-tune the LLMs deployed in Query Refinement and Program-assisted Solution Generation stages. We adopt rejection sampling (Yuan et al., 2023) method on DataBench train and development datasets to synthetize training data. Please refer to Appendix C for details.

4 Experimental setup

Model[2]. We conduct extensive experiments on popular LLMs, including the open-sourced Qwen2.5 series (Yang et al., 2024; Hui et al., 2024), Llama3 series (Dubey et al., 2024), TeleChat2-35B (He et al., 2024; Li et al., 2024a), Mistral-Large[3] and close-sourced GPT-4o (OpenAI, 2023). GPT-4o is invoked via the official API interface, and other models are deployed and invoked locally.

[2]Unless stated otherwise, all models employed in the experiments are the Instruct versions.

[3]https://huggingface.co/mistralai/Mistral-Large-Instruct-2407

Model	Avg	boolean	category	number	list[category]	list[number]
Code-based						
Qwen2.5-7B	69.15 / 69.92	73.64 / 76.74	63.74 / 61.54	70.51 / 69.87	59.72 / 65.28	72.97 / 74.32
Qwen2.5-32B	77.39 / 77.59	85.27 / 89.15	68.13 / 72.53	70.51 / 72.44	83.33 / 75.00	82.43 / 78.38
Qwen2.5-32B-Coder	81.03 / 81.03	93.02 / 89.15	78.02 / 74.73	76.28 / 79.49	73.61 / 77.78	79.73 / 82.43
Qwen2.5-72B	81.03 / 81.22	86.05 / 89.92	80.22 / 71.43	82.69 / 82.05	77.78 / 76.39	71.62 / 81.08
Llama3.1-8B	51.15 / 51.72	55.81 / 50.39	36.26 / 39.56	66.03 / 66.67	43.06 / 37.50	36.49 / 52.70
Llama3.3-70B	74.14 / 77.39	79.84 / 88.37	76.92 / 68.13	73.72 / 78.21	72.22 / 69.44	62.16 / 77.03
TeleChat2-35B	71.07 / 78.54	86.82 / 86.82	73.63 / 73.63	62.82 / 76.92	68.06 / 73.61	59.46 / 79.73
Mistral-Large	85.63 / 83.91	95.35 / 93.02	78.02 / 81.32	83.97 / 83.33	83.33 / 73.61	82.43 / 83.78
GPT-4o	85.44 / 83.72	**96.90** / 93.02	76.92 / 76.92	83.97 / 83.33	79.17 / 75.00	83.78 / 86.49
TableReasoner without SFT						
Qwen2.5-7B	81.61 / 82.95	87.6 / 90.6	74.73 / 76.92	79.49 / 82.05	84.72 / 80.41	81.08 / 86.49
Qwen2.5-32B	89.85 / 89.66	95.35 / **95.35**	86.81 / **87.91**	86.54 / 85.26	**89.27** / **87.44**	89.19 / 94.59
Qwen2.5-72B	87.55 / 88.31	92.25 / **95.35**	80.22 / 84.62	86.54 / 85.90	86.11 / 84.67	90.54 / 89.19
Mistral-Large	90.23 / **90.04**	95.35 / 93.02	**90.11** / **87.91**	**88.46** / **89.74**	84.72 / 80.50	90.54 / **97.30**
Combination△	**90.61** / **90.04**	95.35 / 94.57	**90.11** / 86.81	86.54 / 88.46	86.11 / 84.67	**94.59** / 94.59

Table 1: Performance comparison of Accuracy(%) on DataBench / DataBench Lite test sets. △ means the combination of Mistral-Large for Program-assisted Solution Generation and Qwen2.5-32B for other modules.

Implementation. We fine-tune the Qwen2.5-32B-Instruct and Mistral-Large models for the Query Refinement and Program-assisted Solution Generation stages respectively, applying Low-Rank Adaptation method (Hu et al., 2022). Each model is trained for 5 epochs, with learning rate of 5e-6, lora rank of 8 and total batch size of 32. We set the temperature to 0 to ensure stable output during inference. If majority-voting strategy is adopted, the temperature is configured according to the default values. Within the iterative thinking framework, the maximum round of reasoning cycle is set to 5. We design delicate prompts combining few-shot learning, structured output and CoT strategies to mitigate model bias and address intricate details. Some prompts can be found in Appendix E.

Baseline. We compare 2 common prompting approaches with our TableReasoner. **(i) Zero-shot In-Context Learning** (Z-ICL), which provides the task description and table data in the prompt for textual reasoning. **(ii) Code-based**, which directly answers queries by generating PoT solution leveraging a single LLM and executing codes. For the sake of fairness, we format program execution results through the Answer Summary module to produce final answers. In these cases, we represent tables in Markdown format, which is one of the most common verbalized types of tabular data.

5 Results and Analysis

5.1 Main Results

The main results are shown in Table 1. For the code-based approach, Mistral-Large and GPT-4o show superior performance. A consistent trend of performance enhancement is observed as the model parameters increased in Qwen model series. The results of the Z-ICL method are provided in Appendix D.

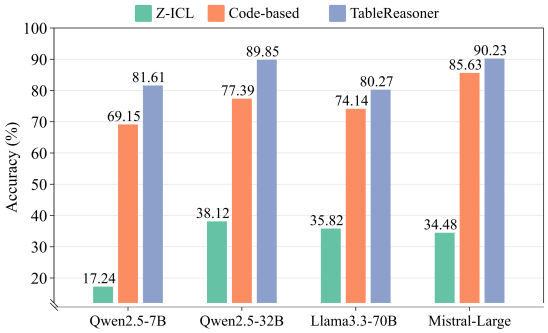

Figure 2: Accuracy comparison with baselines on DataBench test set.

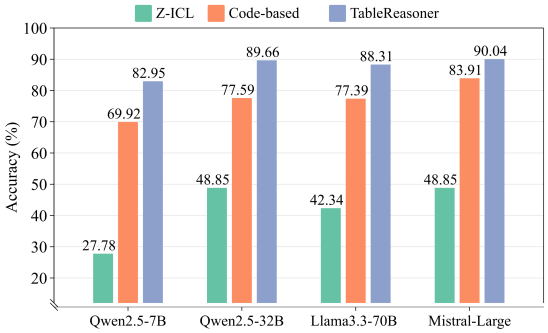

Figure 3: Accuracy comparison with baselines on DataBench Lite test set.

Improvements brought by TableReasoner. TableReasoner consistently enhances the performance across both DataBench and DataBench Lite without fine-tuning. As evidenced in Figure 2 and

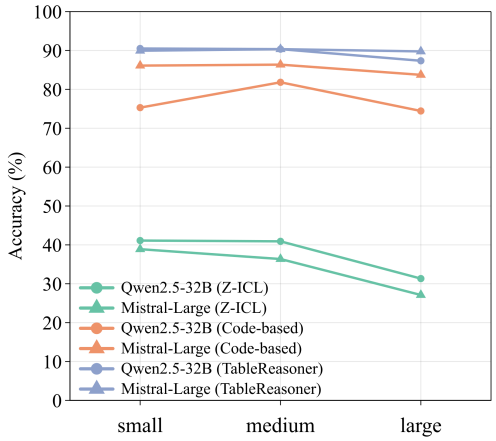

Figure 4: Accuracy comparison between tables of different sizes on DataBench test set.

3, it brings remarkable improvements in accuracy, with absolute increases of 40%+ under the configurations of various backbone models, when compared to the Z-ICL approach. It also realizes substantial improvements to the code-based approach. Moreover, TableReasoner narrows the performance gap between models of small and large parameter size. Notably, integrating Qwen2.5-7B into TableReasoner achieves high accuracy scores, surpassing the top performer in the small-scale model competition rankings (fewer than 8B parameters).

Analysis on different QA types. Table 1 shows that models perform well on Boolean but struggle with list[category] QA types. These complex queries require the model to have a profound understanding of user intent and possess a deep comprehension of tabular data, including deduplication and multi-column correlation analysis. The proposed TableReasoner exhibits more balanced performance advantages across various QA types.

Analysis on table size. We explore the performance on small, medium and large size of tables. As depicted in Figure 4, the Z-ICL approach experiences a sharp performance decline as the table size increases. The code-based approach consistently outperforms Z-ICL and maintain relatively stable performance on tables of different sizes. It not only validates the efficacy of the PoT method but also highlights its great potential for handling large-scale tabular data. Remarkably, TableReasoner exhibits outstanding scalability and robustness, showing minimal degradation as the table size expands.

5.2 Ablation Study

We perform ablation study to verify the effectiveness of each component of TableReasoner workflow. The experiments are conducted on the framework that embeds Qwen2.5-32B model, and results are shown in Table 2. The accuracy shows a drastic deterioration when removing table schema generation and schema linking modules and replacing table schema with the markdown-format text in other remaining stages. The accuracy decreases by 5.37% on the test set and 1.92% on the Lite test set, strongly demonstrating the superiority of applying table schema as the representation, especially for large-sized tables. It is suggested that the specific data in each row is of lesser importance compared to comprehending table's overall structure and column features for programming-based solutions. Removing the schema linking or query refinement module leads to a drop in accuracy, highlighting the importance of the focused table schema and refined sub-queries.

Method	Test	Lite Test
TableReasoner	89.85	89.66
- table schema	84.48 (↓ 5.37)	87.74 (↓ 1.92)
- schema linking	87.55 (↓ 2.30)	88.31 (↓ 1.35)
- query refinement	88.51 (↓ 1.34)	89.27 (↓ 0.39)

Table 2: Ablation results on DataBench and DataBench Lite test sets.

Method	Test	Lite Test
TableReasoner	90.61	90.04
+ Fine-tuning	92.53	91.19
+ Majority-voting (k=5)	93.87	91.76

Table 3: Results of useful strategies.

5.3 Effects of Strategies

LLM Combination. We observe Mistral performs better in code generation tasks. Building upon this finding, we design a hybrid architecture: employing Mistral-Large in Program-assisted Solution Generation stage, while retaining Qwen2.5-32B for other functional modules. The experimental results shown in Table 1 indicate that this hybrid architecture slightly improves accuracy by 0.76% on DataBench test set compared to the architecture using solely Qwen2.5-32B.

Fine-tuning & Majority-voting. To further improve the accuracy on DataBench, we employ the fine-tuned LLMs described in Section 4 within the hybrid architecture. As demonstrated in Table 3, coupled with majority voting based on the self-consistency principle, our system achieves state-of-the-art accuracy of 93.87% and 91.76% on test and Lite test sets, respectively.

6 Conclusion

In this paper, we introduce an LLM-powered and programming-based table easoning framework — TableReasoner, which achieves first place in both subtasks of SemEval-2025 Task 8. We utilize the table schema as a representation, understanding the table from a holistic perspective and addressing the context length constraint. A comprehensive schema linking is implemented in TableReasoner, which provides a focused and precise table schema for query refinement and programming. Besides, we propose an iterative thinking paradigm, incorporating the reasoning workflow to facilitate incremental thinking and reflection. We further enhance performance through fine-tuning and majority voting. Extensive experiments indicate our system is of high scalability and performance across real-world table datasets and has a greater advantage on large-sized tables.

Limitations

For Z-ICL and Code-based methods using LLMs without fine-tuning, prompt design plays a critical role in influencing model performance. For instance, representing tabular data in different formats, such as JSON, CSV or Markdown, can lead to varying results, as different LLMs may exhibit format preferences. Due to time constraints, we were unable to comprehensively evaluate the effects of prompt variations on the comparative experimental outcomes. We will further investigate the impact of different prompt designs.

In this study, we focus on the domain of reasoning over tabular data. Our proposed TableReasoner framework achieves high accuracy by encouraging deliberate reasoning steps. However, it requires numerous inference iterations which are time-consuming. In future work, We will explore adaptive action flow to balance inference times with accuracy.

References

Si-An Chen, Lesly Miculicich, Julian Eisenschlos, Zifeng Wang, Zilong Wang, Yanfei Chen, YASUHISA FUJII, Hsuan-Tien Lin, Chen-Yu Lee, and Tomas Pfister. 2024. Tablerag: Million-token table understanding with language models. In *Advances in Neural Information Processing Systems*, volume 37, pages 74899–74921. Curran Associates, Inc.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Binding language models in symbolic languages. In *The Eleventh International Conference on Learning Representations*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, and et al. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Zhongjiang He, Zihan Wang, Xinzhang Liu, Shixuan Liu, Yitong Yao, Yuyao Huang, Xuelong Li, Yongxiang Li, Zhonghao Che, Zhaoxi Zhang, Yan Wang, Xin Wang, Luwen Pu, Huinan Xu, Ruiyu Fang, Yu Zhao, Jie Zhang, Xiaomeng Huang, Zhilong Lu, Jiaxin Peng, Wenjun Zheng, Shiquan Wang, Bingkai Yang, Xuewei he, Zhuoru Jiang, Qiyi Xie, Yanhan Zhang, Zhongqiu Li, Lingling Shi, Weiwei Fu, Yin Zhang, Zilu Huang, Sishi Xiong, Yuxiang Zhang, Chao Wang, and Shuangyong Song. 2024. Telechat technical report. *Preprint*, arXiv:2401.03804.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Kezhi Kong, Jiani Zhang, Zhengyuan Shen, Balasubramaniam Srinivasan, Chuan Lei, Christos Faloutsos, Huzefa Rangwala, and George Karypis. 2024.

Opentab: Advancing large language models as open-domain table reasoners. In *The Twelfth International Conference on Learning Representations*.

Xiang Li, Yiqun Yao, Xin Jiang, Xuezhi Fang, Chao Wang, Xinzhang Liu, Zihan Wang, Yu Zhao, Xin Wang, Yuyao Huang, Shuangyong Song, Yongxiang Li, Zheng Zhang, Bo Zhao, Aixin Sun, Yequan Wang, Zhongjiang He, Zhongyuan Wang, Xuelong Li, and Tiejun Huang. 2024a. Tele-flm technical report. *Preprint*, arXiv:2404.16645.

Zhongqiu Li, Zhenhe Wu, Mengxiang Li, Zhongjiang He, Ruiyu Fang, Jie Zhang, Yu Zhao, Yongxiang Li, Zhoujun Li, and Shuangyong Song. 2024b. Scalable database-driven kgs can help text-to-sql. In *Proceedings of the ISWC 2024 Posters, Demos and Industry Tracks: From Novel Ideas to Industrial Practice co-located with 23nd International Semantic Web Conference (ISWC 2024), Hanover, Maryland, USA, November 11-15, 2024*, volume 3828 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Shixuan Liu, Chen Peng, Chao Wang, Xiangyan Chen, and Shuangyong Song. 2023. icsberts: Optimizing pre-trained language models in intelligent customer service. *Procedia Computer Science*, 222:127–136. International Neural Network Society Workshop on Deep Learning Innovations and Applications (INNS DLIA 2023).

Weizheng Lu, Jing Zhang, Ju Fan, Zihao Fu, Yueguo Chen, and Xiaoyong Du. 2025. Large language model for table processing: A survey. *Frontiers of Computer Science*, 19(2):192350.

Qingyang Mao, Qi Liu, Zhi Li, Mingyue Cheng, Zheng Zhang, and Rui Li. 2024. Potable: Programming standardly on table-based reasoning like a human analyst. *Preprint*, arXiv:2412.04272.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jorge Osés Grijalba, L. Alfonso Ureña-López, Eugenio Martínez Cámara, and Jose Camacho-Collados. 2024. Question answering over tabular data with DataBench: A large-scale empirical evaluation of LLMs. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 13471–13488, Torino, Italia. ELRA and ICCL.

Jorge Osés-Grijalba, Luis Alfonso Ureña-López, Eugenio Martínez Cámara, and Jose Camacho-Collados. 2025. SemEval-2025 task 8: Question answering over tabular data. In *Proceedings of the 19th International Workshop on Semantic Evaluation (SemEval-2025)*, Vienna, Austria. Association for Computational Linguistics.

Yucheng Ruan, Xiang Lan, Jingying Ma, Yizhi Dong, Kai He, and Mengling Feng. 2024. Language modeling on tabular data: A survey of foundations, techniques and evolution. *arXiv preprint arXiv:2408.10548*.

Jiawei Shao and Xuelong Li. 2025. Ai flow at the network edge. *IEEE Network*, pages 1–1.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, and Tomas Pfister. 2024. Chain-of-table: Evolving tables in the reasoning chain for table understanding. In *The Twelfth International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Zhenhe Wu, Zhongqiu Li, Jie Zhang, Mengxiang Li, Yu Zhao, Ruiyu Fang, Zhongjiang He, Xuelong Li, Zhoujun Li, and Shuangyong Song. Mr-sql: Multi-level retrieval enhances inference for llm in text-to-sql. In *Database Systems for Advanced Applications - 30th International Conference, DASFAA 2025*.

Sishi Xiong, Yu Zhao, Jie Zhang, Li Mengxiang, Zhongjiang He, Xuelong Li, and Shuangyong Song. 2024. Dual prompt tuning based contrastive learning for hierarchical text classification. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12146–12158, Bangkok, Thailand. Association for Computational Linguistics.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '23, page 174–184, New York, NY, USA. Association for Computing Machinery.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *Preprint*, arXiv:2308.01825.

Liangyu Zha, Junlin Zhou, Liyao Li, Rui Wang, Qingyi Huang, Saisai Yang, Jing Yuan, Changbao Su, Xiang Li, Aofeng Su, Tao Zhang, Chen Zhou, Kaizhe Shou, Miao Wang, Wufang Zhu, Guoshan Lu, Chao Ye, Yali Ye, Wentao Ye, Yiming Zhang, Xinglong Deng, Jie Xu, Haobo Wang, Gang Chen, and Junbo Zhao. 2023. Tablegpt: Towards unifying tables, nature language and commands into one gpt. *Preprint*, arXiv:2307.08674.

Tianshu Zhang, Xiang Yue, Yifei Li, and Huan Sun. 2024a. TableLlama: Towards open large generalist models for tables. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6024–6044, Mexico City, Mexico. Association for Computational Linguistics.

Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. 2024b. Reactable: Enhancing react for table question answering. *Proc. VLDB Endow.*, 17(8):1981–1994.

Alex Zhuang, Ge Zhang, Tianyu Zheng, Xinrun Du, Junjie Wang, Weiming Ren, Stephen W. Huang, Jie Fu, Xiang Yue, and Wenhu Chen. 2024. Structlm: Towards building generalist models for structured knowledge grounding. *Preprint*, arXiv:2402.16671.

A Distribution of DataBench Test Set

The distribution of QA-pair types is depicted in Figure 5. We split tables in DataBench test set into large, medium, and small groups based on the number of cells, as shown in Table 4. The small tables include 069_Taxonomy, 071_COL, 072_Admissions, 075_Mortality and 080_Books, total 180 questions; the medium tables include 066_IBM_HR, 073_Med_Cost, 074_Lift, 077_Gestational and 078_Fires, total 176 questions; and the large tables include 067_TripAdvisor, 068_WorldBank_Awards, 070_OpenFoodFacts, 076_NBA and 079_Coffee, total 166 questions.

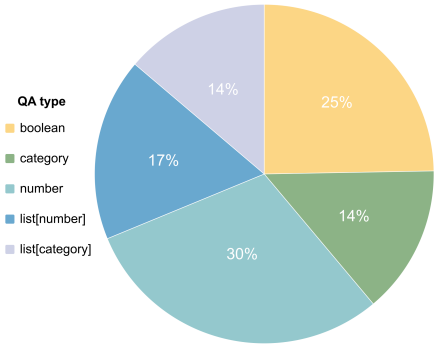

Figure 5: The distribution of Question-Answer pair types on test set.

Type	Dataset Number	Average Cells
Large	5	1519065
Medium	5	18131
Small	5	2882

Table 4: The split of table size on DataBench test set.

B Table Schema Example


```
{
    "file_path":"072_Admissions/all.csv",
    "table_name":"072_Admissions",
    "table_description": "The Admissions
    table contains information about
    applicants for graduate programs,
    including their test scores, academic
     performance, and the likelihood of
    being admitted. This data can be used
     to analyze factors that influence
    admission chances and to predict
    admission outcomes for new applicants
    .",
    "number_of_rows": 500,
    "column_list": [
        "Serial No.",
        "GRE Score",
        "TOEFL Score",
        "University Rating",
        "SOP",
        "LOR",
        "CGPA",
        "Research",
        "Chance of Admit"
    ],
    "column_description": [
            ...(omit)...
        {
```

Model Name	Avg	boolean	category	number	list[category]	list[number]
Qwen2.5-7B	17.24 / 27.78	41.86 / 53.49	3.30 / 13.19	9.62 / 21.79	5.56 / 11.11	17.57 / 31.08
Qwen2.5-32B	38.12 / 48.85	82.95 / 75.19	26.37 / 48.35	20.51 / 39.74	16.67 / 27.78	31.08 / 44.59
Qwen2.5-32B-Coder	32.37 / 41.95	76.74 / 73.64	16.48 / 37.36	17.95 / 32.05	13.89 / 23.61	21.62 / 32.43
Qwen2.5-72B	41.95 / 51.92	76.74 / 86.05	26.37 / 51.65	30.13 / 39.74	20.83 / 23.61	44.59 / 47.30
Llama3.1-8B	29.31 / 25.10	62.79 / 45.74	21.62 / 25.68	17.31 / 22.44	12.50 / 12.50	21.98 / 9.89
Llama3.3-70B	35.82 / 42.34	79.84 / 84.50	18.68 / 21.98	17.31 / 27.56	23.61 / 25.00	29.73 / 43.24
TeleChat2-35B	32.18 / 42.72	75.97 / 75.97	14.29 / 36.26	18.59 / 32.69	15.28 / 25.00	21.62 / 32.43
Mistral-Large	34.48 / 48.85	71.32 / 71.32	25.27 / 51.65	19.23 / 42.31	18.06 / 29.17	28.38 / 40.54
GPT-4o	41.57 / 56.32	84.50 / 83.72	28.57 / 59.34	26.92 / 52.56	20.83 / 25.00	32.43 / 44.59

Table 5: Performance of Z-ICL prompting approach on DataBench / DataBench Lite test sets. The metric is accuracy(%).


```
            "column_name": "SOP",
            "dtype": "float64",
            "example": {
                "minimum_value": 1.0,
                "maximum_value": 5.0,
                "median_value": 3.5,
                "average_value": 3.374
            },
            "specific_meaning": "The
    Statement of Purpose (SOP) score of
    the applicant, on a scale of 1 to 5."
        },
        ...(omit)...
    ]
    "cell_example": [
        {
            "Serial No.": 469,
            "GRE Score": 323,
            "TOEFL Score": 110,
            "University Rating": 4,
            "SOP": 4.0,
            "LOR": 5.0,
            "CGPA": 8.88,
            "Research": 1,
            "Chance of Admit": 0.81
        },
        ...(omit)...
        ]
}
```


C Preparation of Fine-tuning Dataset

The preparation of our fine-tuning dataset involves the following steps: (1) **Reasoning Path Collection**. Within our pipeline, we utilize multiple advanced LLMs—such as GPT-4o, Qwen2.5-72B-Instruct, and Mistral-Large—to perform five rounds of inference on both the training and development sets of DataBench (excluding the DataBench Lite subset), with the temperature parameter set to its default value (not 0). The inputs and outputs of the Query Refinement and Program-assisted Solution Generation modules are recorded for each inference round. Consequently, for each question q, we obtain two sets of reasoning paths: $G[q, (u_i, u_o), (c_i, c_o)]$, where u_i and u_o denote the input and output of Query Refinement step, c_i and c_o denote the input and output of Program-assisted Solution Generation step. (2) **Filtering Incorrect Reasoning Paths**. We verify the correctness of the final reasoning results against ground truth answers and discard any reasoning paths that lead to incorrect results.(3) **Reasoning Path Selection**. We employ a rule-based reward mechanism to select the optimal reasoning paths. For the Query Refinement model, we prioritize paths (u_i, u_o) that correctly associate column names with more subqueries. For the Program-assisted Solution Generation model, we select paths (c_i, c_o) with greater code length. If multiple candidates remain, one is chosen randomly.

For questions that have never been answered correctly, we manually remove ambiguous ones and annotate the others with hints. Specifically, we provide one to three key clues to guide the LLMs toward correctly interpreting and answering the question. The question and its corresponding hints are then concatenated and reprocessed following the aforementioned steps.

After filtering and annotating as described above, the fine-tuning dataset consists of 1184 out of 1308 training samples,

D Performance of Z-ICL approach

As shown in Table 5, the performance of Z-ICL approach falls short on both DataBench test and Lite test sets, with the highest scores being 41.38/56.70, achieved by leveraging GPT-4o. This Z-ICL paradigm confronts several intractable challenges, including text truncation due to length limitations, potential hallucination phenomena, and the absence of explicit reasoning processes.

E Prompts

E.1 Table Description Prompt

Given a database schema regarding "{table_name}", your task is to analyse all columns in the database and add detailed explanations for database and each column.

Requirements:
1.Response should include column names and the specific meanings of each column to help users better understand the data content.
2.Response format example:
{
"Table_Description":...,
"Column_Description": [
{"column_name": "Age", "specific_meaning": "Represents User's Age."},
{"column_name": "Joined Date", "specific_meaning": "The date on which the user joined."},
{"column_name": "Gender", "specific_meaning": "User's Gender, with 2 categories."},
{"column_name": "City", "specific_meaning": "City where the user resides, with 32 categories and only one category example displayed."}]
}

Definition of fields:
Table_Description: Explain the main content and possible uses of the table.
Column_Description: Explain the meaning of each column.
Ensure that the response format is a compact and valid JSON format without any additional explanations, escape characters, line breaks, or backslashes.

Database Schema
{table_schema}

Please response in **JSON** format complying with the above requirements.

E.2 Query Refinement Prompt

As an experienced and professional data analysis assistant, your goal is to analyze a user's question and identify the relevant columns that might contain the necessary data to answer user's question based on the table schema. The table schema consists of table descriptions and multiple column descriptions.

Specifically, you need to complete two sub tasks:
`[task1]`
Thoroughly understand and analyze user's question. You should orient your approach towards resolving user query by referencing the information provided in the table schema, and break down the original query into more specific, complete and executable sub-queries.

`[task2]`
For each query to be answered, identify and extract the relevant columns from the 'column_list' field in the table schema that are necessary to answer the query.

Instruction
`[task1 Instruction]`
- You should attempt to decompose the original query into more specific, progressively detailed, step-by-step sub-queries. Ensure the sub-queries maintain high relevance to the original query and executability to table retrieval, and confirm that no critical information is omitted.
- You can recognize key entities, intentions, special reminder, and specific objects from user's question, which can help you accurately analyze user issues.
- Ensure that each query can be answered by retrieving relevant values from the table.
- Pay attention to the expression of the maximum value (maximum/top/highest/most/lowest/smallest/last, etc) in user's query.

`[task2 Instruction]`
- Identify one or more relevant columns from the 'column_list' field in the table schema that are necessary to answer each query.
- Distinguish between easily confused column names, and refer to column descriptions and example values if necessary, to ensure the accuracy of the relevant columns extracted.
- The user's terminology may have multiple meanings or their expression might be ambiguous. In such cases, try to infer the most likely intent from the user's query and provide all potentially relevant columns.
- When queries are vague or ambiguous, attempt to infer the most likely intent based on the user's question and the table description, and provide all potentially relevant columns as comprehensively as possible.

- Ensure no necessary columns are omitted.
- Please reflect and ensure that the extracted column names must exist in the table schema(‘Field: column_list‘). Prohibit modification and avoid any illusions to ensure that the relevant values can be read from the table.

Output format
Please answer with a list of sub_queries in JSON format **without any additional explanation**.

Examples:
Question: What are the average sales, cost, and profit per order for children’s food?
Response:[
{"Query1": "Filter the data to include only orders related to children’s food.", "relevant_column_list": ["product_category"]},
{"Query2": "Calculate the average sales per order for children’s food.", "relevant_column_list": ["sales"]},
{"Query3": "Calculate the average cost per order for children’s food.", "relevant_column_list": ["cost"]},
{"Query4": "Calculate the average profit per order for children’s food.", "relevant_column_list": ["profit"]}
]
Question: What is the average concentration of PM2.5 in Sichuan Province in January 2015?
Response: [
{"Query1": "Select data from January 2015.", "relevant_column_list": ["date<the Gregorian calendar>"]},
{"Query2": "Further filter the data of Sichuan Province from the results of Query1.", "relevant_column_list": ["province"]},
{"Query3": "Calculate the average concentration of PM2.5.", "relevant_column_list": ["PM2.5"]}]
Let’s begin!
Table Schema
{table_schema}

Response the user’s question ‘{query}‘ strictly follow the above guidelines.
Question: {query}
Response:

E.3 Answer Summary Prompt

Based on the following thought process records, generate the Final Answer of the user query "{query}" to the table.
Rules
1. Thoroughly analyze the connection between the query and the thought process, and extract the correct Final Answer.
2. Determine the data type of Final Answer based on the understanding of user question. The data type of Final Answer must be one of the following:
- Boolean: Valid answers include "True" or "False"(must be string).
- Category: A catgory value (e.g., "Bryin", "try your best!").
- Number: A numerical value, which may represent a computed statistic (e.g., average, maximum).
- List: A list containing number or categories. The expected format example is: [’real estate’, ’investments’, ’pharmaceuticals’, ’software’].
3. Output the Final Answer directly without any prefix words or explanations. Your Final Answer’s data type must be a number, a category, or a list. Answer with a complete sentences in Final Answer is strictly prohibited.

Attention! The data type is just for reference to help you provide the correct format of the Final Answer. The Final Answer content should be derived from the information in the thought process records. Here are your thought process records:
{thought_process}

=========

User Query

Query: {query}

Final Answer:

E.4 Iterative Thinking Prompt

As an intelligent assistant for table analysis, your primary task is to analyze the table schema and assist in answering questions based on the data. To perform this, follow these guidelines:
1.You cannot view the table directly. However, you are provided with schema details and some sample cell values.
2.Use these schema details to frame relevant Python queries that progressively solve the user’s question.
3.Strictly adhere to the structured format below to document your thought process, actions, observations, and responses.

Provided Information:
Schema Retrieval Results:
{table_schema}

Thinking Format:
- Query: Input question that need to be answered.
- Thought: You should always think about what to do and clearly state that.
- Action: Generate concrete Python-based ideas based on table schema retrieval results to get the observation or answer.
- Observation: Provide observations or results from the action. If unavailable, note the missing information or ambiguities.
(Repeat the Thought/Action/Observation steps as needed)
- Thought: After sufficient observations, decide if the original input question can be answered. If so, articulate the response based on the findings.
- Response: Present a concise and accurate answer to the original input question.

Task:
Given the table schema retrieval results above, analyze the input question and generate the thought or response in the structured format.

Input Question:
{query}

Thinking Process Records:
{history_thinking}

(Remember! Make sure your brief output always adheres to one of the following two formats:
A. If the answer to the question can be obtained or inferred from thinking process records, indicating you have completed the task, please output:
Thought: 'I have completed the task'
Response:

B. Otherwise, please further rewrite and generate an **improved and clearer query** of the user's target question '{query}' based on previous thinking without explanation, and point out potential considerations and error prone points that neeed to be noted, making it easier for LLMs to uderstand and analyse, please output:

Query:
)

E.5 Z-ICL Prompt

You are an assistant tasked to response the question asked of a given Table in markdown format. Before providing your response, you need to fully understand and utilize the information contained in the Table. You must response in a single JSON with your answer to the question and your explanation:
* "answer": answer using information from the provided Table only.
* "explanation": A short explanation on why you gave that answer.

Answer Requirements:
1. Determine the data type of answer based on the understanding of user question. The data type of answer must be one of the following:
- Boolean: Valid answers include "True" or "False"(must be string).
- Category: A catgory value.
- Number: A numerical value, which may represent a computed statistic (e.g., average, maximum).
- List: A list containing number or categories. The expected format example is: ['real estate', 'investments', 'pharmaceuticals', 'software'].
2. Output the response directly without any prefix words or explanations.
3. The answer value in response must be derived from the values extracted from the provided data, and any unnecessary rewriting, expansion or format conversion is not allowed.

Response Format:
Question: What is the name of the richest passenger?
Table:
```
"""
| passenger | wealth($) |
| ————— | ————— |
| value1 | value2 |
"""
```
Response: {
"answer": "value1",

```
    "explanation": ""
    }

Now let's start!

Question: {question}
Table:
"""
{table_data}
"""
Response:
```

E.6 Code-based Prompt

```
You are a professional programming assistant designed to utilize the Python package 'pandas' to analyze the table and Response efficient and robust Python code for answering user's Question. The code will read the file from the given 'Table_path' and perform data extraction.
You should act in accordance with the following requirements:
1. Generate chain-of-thought execution ideas based on the understanding of the table content and the user's Question. Describe in detail the algorithm steps as much as possible, including Question analysis, table data format parsing method and code logic description.
2. Then write Python codes according to your approach to solve the question. The codes need to be concise and easy to understand, and if necessary, add comments for clarification.
3. Note that your analysis must be based entirely on the Table data, with special attention to the content and format of the table cells.

You should deliberately go through the user's Question, Table_path and Table and strictly follow the guidelines to appropriately answer the user's Question. You can only output a standardized JSON object, including "code_thought" and "code", and you are prohibited from outputting any other unnecessary thought processes. Ensure that your Response can be read by json.loads().

### Guidelines:
**Thought generation**: With the goal of addressing the user's Question, refer to table_data to generate step-by-step code writing ideas.

**File Reading**: Depending on the table file format and size, efficiently read data from the given 'Table_path' (supporting formats such as CSV, Excel) and load it into a Pandas DataFrame. For larger datasets, choose an appropriate method to ensure performance.

**String Matching**:
- When performing string matching, it is best to use the '.contains()' method instead of a completely strict equal match ('=='). When using '.contains()' function, set the 'regex=False'. Usage Example: filtered_df = df[df['Publication'].str.contains('Harpercollins Publishers (India)', case=False, na=False, regex=False)]

**Sorting and Ranking**:
- If the query involves rankings, top/bottom N, max/min, higher/lower than, etc, please sort the data using 'sort_values()'. If one or more columns of data to be sorted may have the same value, they should be sorted twice in index order. Usage Example: 'df.sort_values(by='value', ascending=False, kind='mesort')'.
- Ensure that the DataFrame is sorted by index even if the values are the same.
- When sorting string type numbers, first convert the data type of the numbers from string to float. Usage Example: 'sorted_unique_ids = sorted([float(u) for u in unique_supplier_ids if not pd.isna(u)])\n earliest_5_suppliers = sorted_unique_ids[:5]'
- Use unique operation with caution when sorting and ranking.

**Special reminders**:
- The generated code should be robust, including error handling and file format compatibility. It should strictly match the column names mentioned in the user's Question, avoiding irrelevant or mismatched columns.
- Unless otherwise specified, please ignore null or empty values.
- Pay attention to the wording of the question to determine if uniqueness is required or if repeated values are allowed. Unless otherwise specified, the unique operation ('.unique()') is not necessary when sorting or finding the maximum/top/highest/-most/lowest/smallest/last (etc) N values in most cases.
- Pay attention to **the format of example values** before you manipulate the data in a certain column. Deeply think about how to correctly parse and extract ill-formed data, Not JUST anomaly capture.
- For Boolean problems, it is not necessary to output all elements, only obtain True or False answers, or obtain the first few elements to avoid too much unnecessary output.
- The results of mathematical operations must be specific number values, and Scientific notation cannot be used.
```

Code Instruction:
Your code must be like:
"import pandas as pd\n def parse_labels(s):\n if s == '[]':\n return []\n return [label.strip() for label in s.strip('[]').split(',')]\n df = pd.read_csv('all.csv')\n # Explode the labels into individual rows\n labels = df['labels_en'].apply(parse_labels).explode()\n # Count occurrences of each label\n label_counts = labels.value_counts()\n # Find the label with the highest number\n most_common_label = label_counts.idxmax()\n print('the label with the highest number of products', most_common_label)"

- Ensure the final answer is the last line in python code.
- Note that "Answer" is just the placeholder in the code. You should replce it with a entity name or a specific description derived from the user's input, as short as possible. Note that when single quotes are included in the answer description, please use double slashes: 'print('Alice's score')'

Response Format:
User's Question: Which label has the highest number of products?
Response:
{
"code_thought":" To find the single label with the highest number of associated products, we'll: 1. Parse the <labels_en> column to extract individual labels; 2. Handle empty lists and string formatting issues; 3. Count occurrences of each label; 4. Identify the label with the highest count.",
"code": "import pandas as pd\n def parse_labels(s):\n if s == '[]':\n return []\n return [label.strip() for label in s.strip('[]').split(',')]\n df = pd.read_csv('all.csv')\n # Explode the labels into individual rows\n labels = df['labels_en'].apply(parse_labels).explode()\n # Count occurrences of each label\n label_counts = labels.value_counts()\n # Find the label with the highest number\n most_common_label = label_counts.idxmax()\n print('the label with the highest number of products', most_common_label)"
}

Let's begin!
Now please deliberately go through the following user's Question, Table_path and Table word by word and strictly follow the above guidelines to appropriately answer the question. You can only output a standardized JSON object, including "code_thought" and "code", and you are prohibited from responsing without any prefix words or explanations. Ensure that your Response can be read by json.loads().

User's Question: {question}
Table File Path: {table_path}
Table:
"""
{table_data}
"""
Response: