TABDSR: Decompose, Sanitize, and Reason for Complex Numerical Reasoning in Tabular Data

Changjiang Jiang[1], Fengchang Yu[1], Haihua Chen[2], Wei Lu[1], Jin Zeng[1,3]
[1]Wuhan University,
[2]University of North Texas,
[3]Hubei University of Economics
Correspondence: yufc2002@whu.edu.cn

Abstract

Complex reasoning over tabular data is crucial in real-world data analysis, yet large language models (LLMs) often underperform due to complex queries, noisy data, and limited numerical capabilities. To address these issues, we propose TABDSR, a framework consisting of: (1) a query decomposer that breaks down complex questions, (2) a table sanitizer that cleans and filters noisy tables, and (3) a program-of-thoughts (PoT)-based reasoner that generates executable code to derive the final answer from the sanitized table. To ensure unbiased evaluation and mitigate data leakage, we introduce a new dataset, CalTab151, specifically designed for complex numerical reasoning over tables. Experimental results demonstrate that TABDSR consistently outperforms existing methods, achieving state-of-the-art (SOTA) performance with 8.79%, 6.08%, and 19.87% accuracy improvement on TAT-QA, TableBench, and TABDSR, respectively. Moreover, our framework integrates seamlessly with mainstream LLMs, providing a robust solution for complex tabular numerical reasoning. These findings highlight the effectiveness of our framework in enhancing LLM performance for complex tabular numerical reasoning. Data and code are available upon request.

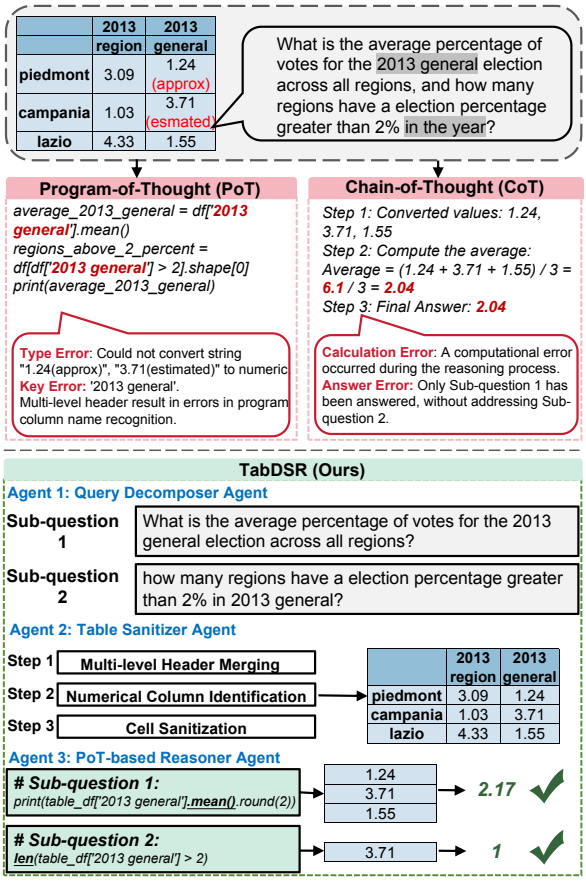

Figure 1: Illustration of the comparison between PoT, CoT, and the proposed TABDSR.

1 Introduction

Table Question Answering (TQA) requires extracting and reasoning over numerical information from tabular data to produce correct answers. It has found broad application in financial TQA (Chen et al., 2021; Zhu et al., 2021) and mathematical reasoning with tabular data (Lu et al., 2023). Although large language models (LLMs) exhibit strong general reasoning capabilities, real-world TQA remains challenging. TQA questions often demand multi-hop reasoning (Biran et al., 2024), involving multiple calculation steps, and visual table conversions can introduce noise, further degrading performance. Recently, agent-based approaches, such as Deep Research (Wentao Zhang, 2025) and GUI Agents (Yuan et al., 2025), have been explored to handle multi-step reasoning and complex multi-hop tasks.

Approaches to TQA numerical reasoning generally fall into three categories: pre-trained models, fine-tuning LLMs, and prompt-based LLMs. Pre-trained and fine-tuned models typically require large amounts of high-quality TQA data, and they struggle to generalize to new or unseen tasks. As a result, prompt-based methods have become the mainstream solution. Nonetheless, as illus-

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 3172–3196
November 4-9, 2025

trated in Figure 1, they still face three critical challenges: (1) **Multi-hop Complexity**: Even LLMs with strong reasoning abilities can fail to answer multi-hop questions accurately. (2) **Data Quality and Structure**: Program-of-Thought (PoT) approaches (Chen et al., 2023) rely on clean and consistently typed table columns; mixed-type columns (e.g., "1.24(approx)" and "1.55" in the same column) can trigger runtime errors. (3) **Limited Numerical Computation**: Current LLMs do not truly compute but rather mimic numerical procedures seen in training data (Mirzadeh et al., 2024).

Humans, by contrast, excel at table reasoning via a three-step process: (1) decompose complex questions into simpler sub-questions, (2) interpret the table's structure and semantics, and (3) extract relevant data and perform precise calculations. Inspired by this process, we introduce TABDSR, a prompt-based framework tailored for numerical reasoning with complex tables. As depicted in Figure 1, TABDSR employs three agents that mirror human reasoning: (1) A **Query Decomposer Agent** that splits the original query into tractable sub-questions, (2) A **Table Sanitizer Agent** that refines the table to ensure a clean, machine-readable format, (3) A **PoT-based Reasoner Agent** that generates and executes programs to derive the final answer.

Evaluating TQA methods is further complicated by data leakage in existing datasets (Deng et al., 2024), which can obscure the true capabilities of LLMs. To address this, we propose CALTAB151 , a new dataset specifically designed to assess complex numerical reasoning in TQA while minimizing data leakage risks. This provides a more reliable benchmark for evaluating prompt-based frameworks, ensuring a fair comparison of different LLM-based solutions for table-based numerical reasoning.

Our contributions are summarized below:

- We propose TABDSR, a framework that combines Query Decompose Agent, Table Sanitizer Agent and PoT-based Reasoner Agent, specifically designed to improve the performance of LLMs in numerical reasoning tasks within TQA.
- We conducted a comprehensive analysis of our approach in several open-source and closed-source LLMs. TABDSR significantly improves the performance of LLMs in numerical reasoning within TQA tasks. Experimental results across different LLMs demonstrate the transferability of our method.
- We construct CALTAB151 , a novel dataset in complex numerical reasoning TQA task, to avoid potential data leakage that could undermine the reliability of the reported metrics on the public datasets.

2 Related Work

Pre-trained models. Early work on Table Question Answering (TQA) often relied on large pre-trained models fine-tuned for specific downstream tasks. Given a table and a question, some methods generate executable SQL queries to extract the answer (Liu et al., 2022; Jiang et al., 2022), while others directly predict the answer text (Herzig et al., 2020). Although these approaches have shown promising results, they typically require large-scale, high-quality tabular datasets and significant computational resources for training. As a result, they face challenges in domain adaptation and often struggle to generalize to new or unseen tables.

Fine-tuning LLMs. Recent studies have shown that directly fine-tuning large language models (LLMs) can yield strong performance on table reasoning tasks (Badaro et al., 2023), as seen in systems like TableLlama (Zhang et al., 2024a), TableGPT2 (Li et al., 2024), and TableLLM (Zhang et al., 2024b). These fine-tuned models obviate the need for elaborate prompt design by framing the task as a single-round QA process. However, this approach compresses the reasoning steps into a single inference pass, limiting traceability for numerical calculations.

Prompt-based LLMs. For TQA with LLMs, direct prompting (DP) feeds the table and question into an LLM for a single-step answer without intermediate reasoning. In contrast, **Chain-of-Thought (CoT) prompting** compels LLMs to generate step-by-step reasoning before producing the final answer. Within CoT, two main CoT variants are textual chain-of-thought (TCoT) and symbolic chain-of-thought (SCoT). TCoT appends prompts like "Let's think step by step!" (Kojima et al., 2022), and SCoT uses symbolic commands to iteratively refine results (Wu et al., 2024). An emerging trend in TQA is to merge CoT prompting with multiple LLM calls, as demonstrated by MFORT-QA(Guan et al., 2024), which first employs few-

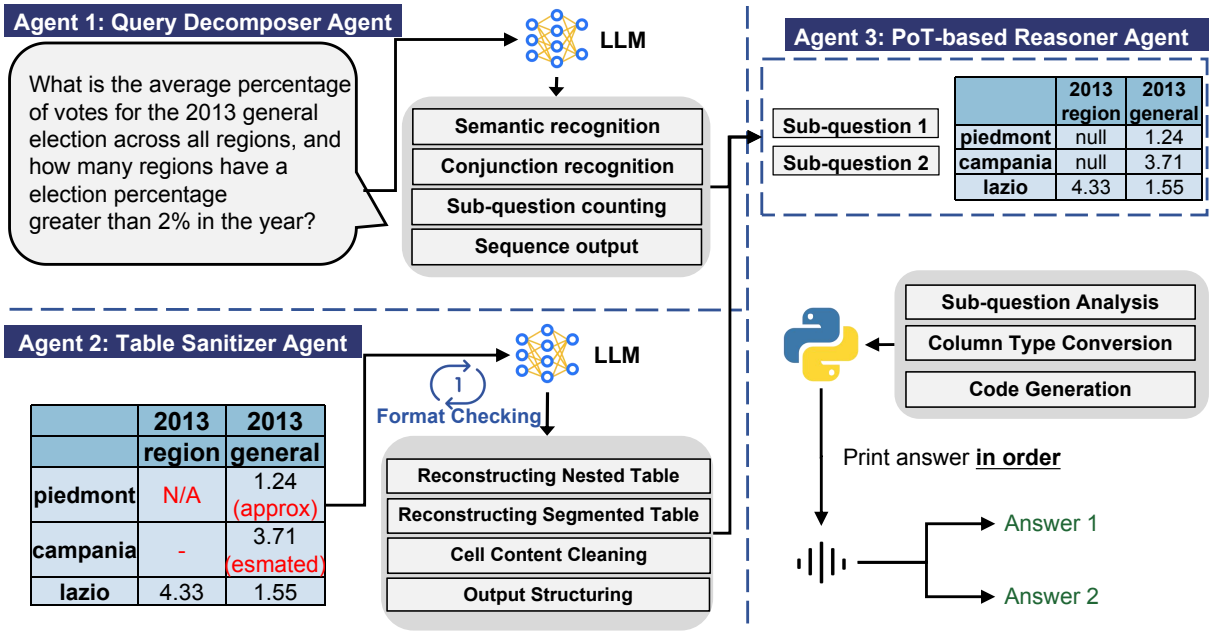

Figure 2: TABDSR's collaborative pipeline: Synchronous execution of query decomposer agent and table sanitizer agent for complex numerical reasoning via PoT.

shot prompts to retrieve relevant tables, then applies CoT prompts to guide the reasoning process. However, GSM-Symbolic(Mirzadeh et al., 2024) reveals limitations in LLM numerical reasoning: systematically altering numerical values in math problems degrades model performance. This finding highlights that CoT prompting alone cannot ensure accurate numerical calculations.

Another popular approach is **Program-of-Thought (PoT)** prompting. PoT-based methods employ LLMs to generate executable code, addressing the models' inherent computational limitations. However, as illustrated in Figure 1, these methods rely on clear questions, correct table structures, and consistent column types. Parallel lines of research on question decomposition often train sequence-to-sequence models to split a complex question into smaller sub-questions (Perez et al., 2020; Zhang et al., 2019), but this requires extensive manual labeling. In TQA, question decomposition is frequently combined with table-level preprocessing. For instance, TabSQLify(Nahid and Rafiei, 2024a) condenses large tables to reduce contextual overhead, and MIX-SC(Liu et al., 2024b) integrates a Python interpreter with ReAct (Yao et al., 2023) iterative reasoning to refine its outputs. Similarly, Chain-of-Table (Wang et al., 2024) dynamically plans operations based on sub-task selection.

Despite these advances, two challenges persist: (1) existing methods often fail to explicitly address multi-hop questions, risking incomplete or incorrect answers, and (2) they overlook column-type inconsistencies, which can cause LLM-generated code to misinterpret text noise as numerical data and trigger computation errors.

3 TABDSR

Task Formulation The goal of Table Question Answering (TQA) is to predict an answer A given two table strings: a **table** T and a **question** Q. As illustrated in Figure 2, our TABDSR framework addresses TQA's numerical reasoning challenges via three specialized agents, each responsible for a key component of the reasoning pipeline.

3.1 Query Decomposer Agent

Complex numerical reasoning often requires **multi-hop** interpretation of the question. Single-turn models frequently fail to capture this complexity, resulting in incomplete or inaccurate answers. Previous work underscores the importance of proper question understanding in TQA.

Existing prompt-based methods typically feed both the question and the table into an LLM. For example, DATER (Ye et al., 2023) uses the prompt "Decompose questions into sub-questions and transform them into a cloze-style format" However, these decomposition instructions can be vague, yielding inconsistent sub-question granularity. Moreover, given that tables are often much

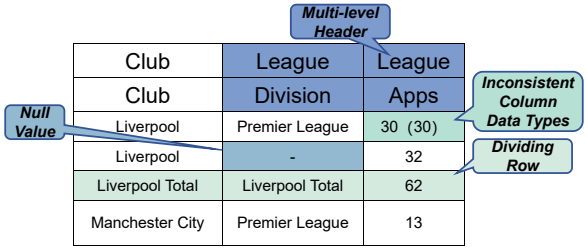

Figure 3: Example of a noisy tabular data; The original table is sourced from the TableBench (Wu et al., 2024); The table has multi-level headers, which seem to introduce errors and noise due to visual table conversions; For the sake of presentation, we manually removed certain rows and columns and modified a few cells.

larger than the question text, crucial details in the question may be overshadowed by the table content during decomposition.

To address these issues, our Query Decomposer Agent takes only the question as input and ignores the table entirely. By significantly reducing the prompt length, we ensure that pertinent details in the question are more likely to be retained. Concretely, we design the prompt to decompose the query based on textual cues such as conjunctions ("and", "or") and punctuation (commas), treating each segment as an independent sub-question.

We further mitigate potential LLM hallucinations by drawing inspiration from Chain-of-Thought (CoT) reasoning (Wei et al., 2024). Specifically, we instruct the model to output the number of sub-questions in a list format, accompanied by a carefully chosen example from a complex TQA query. This example demonstrates the desired level of decomposition and guides the model's reasoning process.

3.2 Table Sanitizer Agent

Beyond the complexity of the question text, TQA tasks are often complicated by textual tables that lose the visual cues crucial for understanding hierarchical or segmented data. These tables can be lengthy, contain redundant rows and columns, include null values, or introduce noise through the conversion process from a visual to a text-based format. To tackle these issues, our **Table Sanitizer Agent** optimizes both the structure and content of textual tables.

Structural Optimization, which consists of two scenarios: (1) Reconstructing Nested Tables (Multi-level Headers). In text format, headers from multi-level tables can be split into separate lines, obscuring their logical relationships. We prompt the model to detect these nested headers and merge them based on their semantic similarities. (2) Reconstructing Segmented Tables. Tables sometimes appear in sections separated by blank rows or dividing lines, and these visual cues are lost in plain text. We enhance the prompt to identify these segmentation rows and either remove or extract them, depending on the query requirements.

Content Optimization, mainly focuses on Cell Content Cleaning. Subsequent reasoning steps rely on valid cell entries. Accordingly, we instruct the model to remove extraneous symbols (e.g., "%," currency symbols, commas), explanatory notes, emojis, and other non-numeric characters. We then convert numerical data into integer or float formats and standardize blank cells (e.g., "–", "N/A") to a consistent "null" label.

Although some studies simplify tables to reduce complexity (Ye et al., 2023; Nahid and Rafiei, 2024a), modern LLMs can effectively handle long inputs. For instance, Qwen2.5:7b supports a 128K context window (Yang et al., 2024), making the full table content manageable without sacrificing critical information. Consequently, our Table Sanitizer Agent retains the complete table to preserve as much detail as possible for downstream TQA tasks.

To mitigate potential hallucinations from the LLM that could lead to table-cleaning errors, we incorporate a reflection mechanism. Specifically, the cleaned table string is first validated by a Python parser. If the parser fails to read the table, the resulting error message, along with the newly generated string, is fed back into the prompt as contextual guidance for the LLM to regenerate a corrected version. To prevent infinite loops, we limit this regeneration process to a single additional iteration in our current experiments. The overall workflow is illustrated in Figure 2.

3.3 PoT-based Reasoner Agent

Having decomposed the original question into sub-questions and sanitized the table, we now have well-defined queries and clean data ready for computation. However, studies such as Mirzadeh et al. (2024) indicate that LLMs alone often struggle with precise numerical reasoning. To circumvent this limitation, our PoT-based Reasoner Agent employs Program-of-Thought (PoT) techniques to generate Python code that performs the necessary calculations on the sanitized table.

While some approaches (Wang et al., 2024; Nahid and Rafiei, 2024a) rely on SQL for table computations, our method prioritizes simplicity and computational efficiency. Specifically, we load the sanitized data into a Pandas DataFrame and leverage its flexible APIs to handle tasks such as filtering, aggregation, and arithmetic operations. To ensure robustness, we (1) extract relevant data into the DataFrame based on each sub-question, (2) generate Python code that executes calculations specific to these sub-questions, (3) restrict certain Pandas methods to avoid inconsistencies across versions, (4) validate data formats through consistency checks, minimizing the risk of errors caused by unexpected input types.

After computing the results for each sub-question, we reassemble them in a logical sequence—considering any dependencies between questions—to produce the final TQA answer. This modular pipeline ensures that the correctness of PoT-based reasoning is maximized by the clarity of the sub-questions and the cleanliness of the tabular data, underscoring the importance of the first two agents in our framework.

4 Construction of CALTAB151

To ensure a fair evaluation that mitigates data leakage from existing public datasets, we propose an annotation framework combining LLM-generated queries with human-verified answers (details in Appendix B). This process culminates in a high-quality numerical reasoning dataset, CALTAB151 , composed of 151 table samples drawn from TableBench(Wu et al., 2024) (84), FinQA(Chen et al., 2021) (27), TAT-QA(Zhu et al., 2021) (32), and AitQa(Katsis et al., 2022) (8).

We construct CALTAB151 through the **six** steps: (1) **Numerical Perturbation**: To maintain realistic yet varied numeric values, we randomly perturb cell values by $\pm 3\%$–5% of their original values. The prompt can be seen in Figure 6. (2) **Cell Noise Addition**: To simulate natural table noise, we inject context-appropriate symbols (e.g., $, €, or %) into randomly chosen numeric columns, aligned with their semantic relevance. (3) **Structural Randomization**: We enhance structural diversity by shuffling rows or columns and randomly deleting a subset of them, thereby introducing a broader range of table configurations. (4) **Random Null Value Filling**: To model incomplete data, we replace 2–4 cells with labels such as "None", "Null", "N/A", "???", or "-". (5) **Multi-hop Question Generation**: To increase question complexity, we construct an agent for generating multi-hop questions, where each question consists of multiple sub-questions, with each subsequent question depending on the answer to the previous one. The agent guides the model to generate two sub-questions and then merge them into a coherent two-hop question based on natural semantics, ensuring the coherent question is contextually relevant. The Prompt can be seen in Figure 5. (6) **Answer Annotation**: Finally, to ensure data accuracy, human annotators manually calculate and verify answers to all generated multi-hop questions.

This multi-pronged approach produces a robust and realistic dataset that captures both the structural and semantic challenges of real-world TQA, providing a more reliable benchmark for evaluating LLM-based numerical reasoning.

5 Experiments and Results

5.1 Experimental Settings

Baselines. We implement the following baselines:

- **Pre-trained Models** We have selected TAPEX (Liu et al., 2022), and OmniTab (Jiang et al., 2022). The tapex-large-finetuned-wtq and omnitab-large-finetuned-wtq as backbone. Both are fine-tuned on the WikiTQ dataset (Pasupat and Liang, 2015), which is a dataset in the TQA task.

- **Fine-tuning LLMs** We evaluate several fine-tuning LLMs, including TableLlama (Zhang et al., 2024a), TableLLM (Zhang et al., 2024b), TableGPT2 (Su et al., 2024).

- **Prompt-based LLMs** We compare different only prompts methods with TABDSR, include DP, TCoT (Wei et al., 2024), PoT (Chen et al., 2023) and SCoT. In addition, we evaluate TABDSR against latest prompt-based methods that include Chain-of-Table (Wang et al., 2024), TabSQLify (Nahid and Rafiei, 2024a), E5 (Zhang et al., 2024c), NormTab (Nahid and Rafiei, 2024b), and ReAaTable (Zhang et al., 2023)
  footnoteThe prompts can be found in the Appendix F..

Datasets. We select three TQA datasets for experiments [1]:

[1]Examples from each dataset can be found in Appendix C.

Methods	TAT-QA		TableBench		CALTAB151	
	Acc	ROUGE-L	Acc	ROUGE-L	Acc	ROUGE-L
Pretrained-models						
TAPEX (Liu et al., 2022)	12.56	17.23	24.10	27.98	8.61	12.38
OmniTab (Jiang et al., 2022)	13.61	17.32	27.74	31.73	9.93	14.28
Fine-tuning LLMs						
TableGPT2-7B (Su et al., 2024)	9.58	9.71	44.57	45.95	14.24	15.11
TableLLM-13B@PoT (Zhang et al., 2024b)	3.67	3.85	40.16	41.95	8.61	10.07
TableLLM-13B@DP (Zhang et al., 2024b)	35.56	37.44	42.91	44.21	24.17	28.32
TableLlama-7B (Zhang et al., 2024a)	27.84	33.51	23.70	26.75	7.28	11.50
Prompt-based LLMs						
Chain-of-Table (Wang et al., 2024)	34.89	41.33	35.80	37.87	<u>25.83</u>	<u>31.60</u>
TabSQLify$_{\text{col+row}}$ (Nahid and Rafiei, 2024a)	<u>51.62</u>	<u>58.17</u>	43.04	46.97	23.51	29.62
MIX-SC (Liu et al., 2024b)	30.77	34.06	<u>46.67</u>	<u>49.41</u>	16.56	20.78
$E^5{}_{\text{code}}$ (Zhang et al., 2024c)	40.1	44.65	9.4	11.09	8.94	10.42
$E^5{}_{\text{zero-shot}}$ (Zhang et al., 2024c)	44.58	48.67	8.72	9.5	7.28	8.96
NormTab (Nahid and Rafiei, 2024b)	21.26	22.68	37.14	38.73	10.93	12.53
ReAcTable (Zhang et al., 2023)	28.72	31.27	25.27	26.31	11.59	13.38
TABDSR (Our Method)	**60.41** (+8.79)	**62.76** (+4.59)	**52.75** (+6.08)	**55.39** (+5.98)	**45.70** (+19.87)	**50.57** (+18.97)

Table 1: Table reasoning results on TAT-QA, TableBench and CALTAB151 ; **Bold** indicates the best performance; <u>Underline</u> indicates the second-best performance; Red indicates the improvement measured against the second-best performance method; TableLLM-13B@PoT refers to conducting code execution prompt to generate the final answer; TableLLM-13B@DP refers to conducting direct text answer generation prompt to generate the final answer.

- **TableBench** (Wu et al., 2024) covers complex TQA questions. The questions involve complex numerical reasoning, fact-checking, Data Analysis, and Visualization. For our experiments, we choose questions related to fact verification and numerical reasoning, containing 493 samples, as these tasks align closely with our research objectives.

- **TAT-QA** (Zhu et al., 2021) challenges models to perform numerical reasoning requiring arithmetic operations, comparisons, and compositional logic. The raw TAT-QA test set contains questions related to tables, table+text (relevant paragraphs), and pure text (relevant paragraphs). Since our task is only related to tables, we select a total of 736 examples where the "answer_from" field is "table".

- **CALTAB151** includes multi-hop questions annotated by professional annotators. These questions require reasoning over multiple table entries. It includes 151 table samples.

Experimental Details. We configure the LLMs with a maximum token length of 4096 and a temperature of 0.1. For fine-tuning approaches:

- **TableLlama** uses **Llama-2-7b-longlora-8k-ft** as its backbone (Touvron et al., 2023).

- **TableGPT2** leverages **Qwen/Qwen2.5-7B** (Yang et al., 2024).

- **TableLLM-13b** is based on **CodeLlama-13b-Instruct-hf** (Roziere et al., 2023).

We adopt **Qwen2.5-7B (Yang et al., 2024)** as the backbone for all prompt-based LLM configurations to ensure a fair comparison.

Evaluation Metrics. We employ accuracy and ROUGE-L (Lin, 2004) as our primary evaluation metrics. Although TableBench uses ROUGE-L as its main metric, we argue that measuring textual overlap alone may not fully capture a model's performance in complex numerical reasoning. Thus, we include accuracy to more robustly evaluate whether each predicted answer is correct, offering a comprehensive view of the models' capabilities.

5.2 Results and Analysis

As shown in Table 1, TABDSR achieves state-of-the-art performance across all three numerical reasoning TQA benchmarks. Notably, the 7B-parameter model even surpasses TableLLM-13B in both accuracy and ROUGE-L, highlighting the effectiveness of TABDSR regardless of model size.

Model	Method	TAT-QA		TableBench		CALTAB151	
		Acc	ROUGE-L	Acc	ROUGE-L	Acc	ROUGE-L
Qwen2.5-7B	DP	29.68	35.27	16.26	19.80	8.28	12.84
	PoT	9.34	9.70	36.02	37.57	11.59	13.15
	TCoT	55.02	62.41	29.45	34.40	19.87	27.00
	SCoT	43.65	49.96	24.48	29.12	16.23	24.60
	TABDSR $_{R}$	18.70	19.03	45.68	47.74	23.18	25.59
	TABDSR $_{D+R}$	17.69	17.91	45.14	46.86	17.88	19.30
	TABDSR $_{S+R}$	**60.50**	**62.86**	51.41	53.86	39.07	44.44
	TABDSR $_{D+S+R}$	60.41	62.76	**52.75**	**55.49**	**45.70**	**50.57**
Qwen2.5-Code-7B	DP	32.80	37.99	22.99	26.34	14.57	20.58
	PoT	20.31	20.50	40.84	42.14	5.96	6.53
	TCoT	59.61	**66.75**	42.78	46.62	22.85	31.69
	SCoT	46.78	53.71	28.05	33.19	18.54	26.17
	TABDSR $_{R}$	18.63	18.98	53.92	55.75	30.13	33.21
	TABDSR $_{D+R}$	18.09	18.45	51.37	52.71	28.15	29.51
	TABDSR $_{S+R}$	**60.96**	62.56	**55.98**	**58.18**	46.69	50.67
	TABDSR $_{D+S+R}$	60.21	61.52	54.43	56.30	**48.01**	**51.39**
Qwen2.5-72B	DP	57.77	64.55	38.47	42.89	25.50	35.97
	PoT	39.31	41.53	50.02	51.84	29.80	32.16
	TCoT	69.45	73.49	58.23	62.39	50.00	57.32
	SCoT	73.15	77.21	47.75	52.07	36.42	44.93
	TABDSR $_{R}$	59.19	60.50	56.95	59.28	50.66	53.97
	TABDSR $_{D+R}$	56.54	57.74	57.15	59.68	51.32	54.76
	TABDSR $_{S+R}$	82.62	84.95	60.56	63.22	**63.25**	**68.21**
	TABDSR $_{D+S+R}$	**83.19**	**85.39**	**61.44**	**64.28**	60.26	64.63

Table 2: Ablation Study on TAT-QA, TableBench, and CALTAB151; Query Decomposer (D), Table Sanitizer (S), and PoT-based Reasoner (R) are abbreviated as shown; **Bold** indicates the best performance under the same dataset and model with different Prompts. Underline indicates the second-best performance under the same conditions; Qwen2.5-7B (Yang et al., 2024), Qwen2.5-Code-7B (Hui et al., 2024), and Qwen2.5-72B (Yang et al., 2024) use their respective instruct-tuning models.

When comparing model categories, we make the following observations: (1) Pre-trained models exhibit the weakest performance. (2) Fine-tuned LLMs rank second overall but struggle significantly on unseen datasets due to biases arising from dataset dependency; for example, TableGPT2-7B and TableLLM-13B@PoT excel on TableBench but show sharp performance drops on the other two datasets. (3) Prompt-based LLMs generally outperform fine-tuned models, suggesting that large models already possess inherent numerical reasoning abilities.

Within the prompt-based category, TABDSR outperforms existing techniques for two key reasons: (1) **Effective Multi-hop Decomposition**. Competing methods often tackle multi-hop questions in a single pass, which can lead to errors or omissions. (2) **Robust Table Sanitization.** Many methods rely on SQL-based splitting to handle large tables but overlook unclean cell content, causing calculation errors. By contrast, TABDSR 's dedicated Decomposer and Sanitizer agents ensure higher reliability. Detailed case studies are provided in Appendix D.

Moreover, nearly every method achieves its lowest performance on CALTAB151 , which we attribute to the absence of data leakage—an advantage that may exist in publicly available datasets. Consequently, CALTAB151 offers a more stringent

Model	Method	TAT-QA		TableBench		CalTab151	
		Acc	ROUGE-L	Acc	ROUGE-L	Acc	ROUGE-L
GPT-4o	DP	55.31	61.95	47.99	52.90	37.09	45.98
	PoT	48.66	49.51	57.31	59.70	45.36	49.15
	TCoT	60.21	65.55	61.44	65.19	45.70	54.13
	SCoT	63.97	69.27	55.40	59.90	41.39	49.32
	TabDSR $_{D+S+R}$	**80.96**	**83.24**	**62.89**	**65.27**	**62.25**	**66.46**
DeepSeek-V3	DP	62.21	66.79	47.46	52.49	34.77	43.50
	PoT	57.53	58.54	49.14	51.70	37.42	40.44
	TCoT	78.95	81.64	59.11	62.77	52.65	59.61
	SCoT	79.66	83.62	52.43	56.85	38.74	47.69
	TabDSR $_{D+S+R}$	**83.67**	**86.13**	**63.84**	**66.48**	**61.92**	**67.09**

Table 3: Transferability of TabDSR: Performance Improvements on GPT-4o (Achiam et al., 2023) and DeepSeek-V3 (Liu et al., 2024a); **Bold** indicates the best performance under the same dataset and model with different Prompts. Underline indicates the second-best performance under the same conditions.

test of genuine numerical reasoning capabilities.

5.3 Ablation Study

Table 2 highlights how each agent contributes to final performance. Following Mirzadeh et al. (2024), which suggests that LLMs can accurately compute numerical values via **R (PoT)**, we include the reasoner (R) in all ablation settings.

Effect of the Sanitizer Agent (TabDSR S). Comparing TabDSR $_{S+R}$ with TabDSR $_{R}$ alone, we observe consistent performance gains—indicating that table sanitization improves data quality for downstream computations. Moreover, the combined TabDSR $_{D+S+R}$ setting achieves the highest accuracy overall, underscoring how integrating the Decomposer and Sanitizer agents yields further benefits.

Effect of the Decomposer Agent (TabDSR $_{D}$). In some model–dataset combinations, TabDSR $_{D+R}$ slightly underperforms TabDSR $_{R}$. We attribute this to the chaotic nature of certain tables (e.g., complex multi-level headers), which can dilute the value of decomposing the question first. Nonetheless, in most cases, adding the Decomposer (TabDSR $_{D+R}$) or both Decomposer and Sanitizer (TabDSR $_{D+S+R}$) improves performance over TabDSR $_{R}$ alone, confirming the overall positive impact of multi-hop question decomposition.

5.4 Transferability

Our optimization strategy is fundamentally prompt-based, raising a concern that it only benefits LLMs with limited reasoning capabilities. To test its transferability, we applied TabDSR to GPT-4o(Achiam et al., 2023) and DeepSeek-V3 (Liu et al., 2024a), two LLMs widely regarded for their strong reasoning abilities. As shown in Table 3, our method continues to enhance performance on these powerful models, suggesting that TabDSR is not merely compensating for weaker LLMs but provides genuine improvements in numerical reasoning.

Comparing Qwen2.5-72B in Table 2 with GPT-4o and DeepSeek-V3 in Table 3, we find that their results are closely aligned. Although GPT-4o and DeepSeek-V3 have more parameters than Qwen2.5-72B, they also demonstrate comparable outcomes on various public benchmarks (Hendrycks et al., 2020, 2021). These findings indicate that our method effectively boosts complex numerical reasoning performance in LLMs that already possess a robust baseline of numerical reasoning skills.

6 Conclusion

We introduced TabDSR, a three-agent, prompt-based framework that significantly elevates numerical reasoning in Table Question Answering (TQA). Our method consistently outperforms pre-trained models, fine-tuned LLMs, and other prompt-based solutions, demonstrating its effectiveness in handling complex tabular data. By decomposing multi-hop questions and sanitizing noisy table content, TabDSR fully harnesses the inherent numerical reasoning capabilities of LLMs, enhancing their performance regardless of parameter size.

This work also has practical implications for various real-world domains—such as finance, business intelligence, healthcare, and e-commerce—where robust analysis of complex, noisy tabular data is critical. The prompt-based nature of TABDSR lowers barriers to adoption by reducing the need for extensive data annotation or specialized training, enabling more cost-effective and scalable deployment of powerful TQA systems. Moreover, our framework's transferability across diverse model architectures highlights its potential for broader integration into enterprise workflows, data analytics platforms, and decision-support systems that require reliable, multi-step numerical calculations over large datasets. We hope these findings inspire further research and innovation in complex numerical reasoning for TQA, spurring the development of even more versatile and efficient solutions.

Limitations

Although the TABDSR performs well, it still falls far short of human performance in answering complex tabular numerical reasoning questions. Our method is prompt-only, and its performance is limited by the reasoning ability of the LLM.

CALTAB151 requires manual verification during the final validation stage, which makes the annotation process costly. As a result, the dataset size is relatively small, and the range of question types is limited. In future work, we plan to expand the dataset by increasing its size, enriching the diversity of question types, and covering a broader range of domains to enable more comprehensive evaluations.

We opted for a restricted Decomposer that operates solely on the question. While this design choice may slightly degrade performance in some isolated cases, our experiments show that it brings consistent and often mild improvements in the majority of scenarios, especially in terms of stability and generalizability across datasets. We acknowledge the trade-off here and consider it a pragmatic decision to ensure robustness and utility in real-world settings. In future work, we plan to explore hybrid approaches that can selectively incorporate table signals while preserving decomposition reliability.

The Sanitizer is constrained by the reflection capabilities of the underlying base model. In some cases, even after multiple iterations, it fails to correctly repair the tables, leading to excessive and redundant calls. Moving forward, we will implement call monitoring and fallback mechanisms to ensure that when the Sanitizer fails, a default resolution strategy can be applied efficiently.

Ethical Considerations

This work involves the use of AI systems in two aspects. First, AI tools were utilized to assist with the translation of this paper. Second, AI models were employed during the data construction process; however, all generated data strictly followed the guidelines described in Appendix A and was used solely for academic and research purposes.

We ensured that no personally identifiable information (PII) or sensitive data was included in CALTAB151 .

Acknowledgments

This research is supported by supported by the Young Scientists Fund of the National Natural Science Foundation of China (Grant No.72304215).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Gilbert Badaro, Mohammed Saeed, and Paolo Papotti. 2023. Transformers for tabular data representation: A survey of models and applications. *Transactions of the Association for Computational Linguistics*, 11:227–249.

Eden Biran, Daniela Gottesman, Sohee Yang, Mor Geva, and Amir Globerson. 2024. Hopping too late: Exploring the limitations of large language models on multi-hop queries. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14113–14130, Miami, Florida, USA. Association for Computational Linguistics.

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*.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan R Routledge, et al. 2021. Finqa: A dataset of numerical reasoning over financial data. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3697–3711.

Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. 2024. Investigating data contamination in modern benchmarks for large language models. 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 8706–8719, Mexico City, Mexico. Association for Computational Linguistics.

Che Guan, Mengyu Huang, and Peng Zhang. 2024. Mfort-qa: Multi-hop few-shot open rich table question answering. In *Proceedings of the 2024 10th International Conference on Computing and Artificial Intelligence*, pages 434–442.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4320–4333, Online. Association for Computational Linguistics.

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*.

Zhengbao Jiang, Yi Mao, Pengcheng He, Graham Neubig, and Weizhu Chen. 2022. OmniTab: Pretraining with natural and synthetic data for few-shot table-based question answering. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Yannis Katsis, Saneem Chemmengath, Vishwajeet Kumar, Samarth Bharadwaj, Mustafa Canim, Michael Glass, Alfio Gliozzo, Feifei Pan, Jaydeep Sen, Karthik Sankaranarayanan, and Soumen Chakrabarti. 2022. AIT-QA: Question answering dataset over complex tables in the airline industry. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Track*, pages 305–314, Hybrid: Seattle, Washington + Online. Association for Computational Linguistics.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2024. Table-gpt: Table fine-tuned gpt for diverse table tasks. *Proc. ACM Manag. Data*, 2(3).

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

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

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2022. TAPEX: Table pre-training via learning a neural SQL executor. In *International Conference on Learning Representations*.

Tianyang Liu, Fei Wang, and Muhao Chen. 2024b. Rethinking tabular data understanding with large language models. 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 450–482, Mexico City, Mexico. Association for Computational Linguistics.

Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. 2023. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. In *International Conference on Learning Representations (ICLR)*.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229*.

Md Nahid and Davood Rafiei. 2024a. TabSQLify: Enhancing reasoning capabilities of LLMs through table decomposition. 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 5725–5737, Mexico City, Mexico. Association for Computational Linguistics.

Md Mahadi Hasan Nahid and Davood Rafiei. 2024b. NormTab: Improving symbolic reasoning in LLMs through tabular data normalization. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 3569–3585, Miami, Florida, USA. Association for Computational Linguistics.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language*

Processing (Volume 1: Long Papers), pages 1470–1480, Beijing, China. Association for Computational Linguistics.

Ethan Perez, Patrick Lewis, Wen tau Yih, Kyunghyun Cho, and Douwe Kiela. 2020. Unsupervised question decomposition for question answering. In *EMNLP*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Aofeng Su, Aowen Wang, Chao Ye, Chen Zhou, Ga Zhang, Guangcheng Zhu, Haobo Wang, Haokai Xu, Hao Chen, Haoze Li, Haoxuan Lan, Jiaming Tian, Jing Yuan, Junbo Zhao, Junlin Zhou, Kaizhe Shou, Liangyu Zha, Lin Long, Liyao Li, Pengzuo Wu, Qi Zhang, Qingyi Huang, Saisai Yang, Tao Zhang, Wentao Ye, Wufang Zhu, Xiaomeng Hu, Xijun Gu, Xinjie Sun, Xiang Li, Yuhang Yang, and Zhiqing Xiao. 2024. Tablegpt2: A large multimodal model with tabular data integration. *Preprint*, arXiv:2411.02059.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

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 H. Chi, Quoc V. Le, and Denny Zhou. 2024. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Yuzhen Xiao Yongcong Li Ce Cui Yilei Zhao Rui Hu Yang Liu Yahui Zhou Bo An Wentao Zhang, Liang Zeng. 2025. Agentorchestra: A hierarchical multi-agent framework for general-purpose task solving. *Preprint*, arXiv:2506.12508.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xinrun Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, et al. 2024. Tablebench: A comprehensive and complex benchmark for table question answering. *arXiv preprint arXiv:2408.09174*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

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

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.

Xinbin Yuan, Jian Zhang, Kaixin Li, Zhuoxuan Cai, Lujian Yao, Jie Chen, Enguang Wang, Qibin Hou, Jinwei Chen, Peng-Tao Jiang, et al. 2025. Enhancing visual grounding for gui agents via self-evolutionary reinforcement learning. *arXiv preprint arXiv:2505.12370*.

Haoyu Zhang, Jingjing Cai, Jianjun Xu, and Ji Wang. 2019. Complex question decomposition for semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4477–4486, Florence, Italy. Association for Computational Linguistics.

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.

Xiaokang Zhang, Jing Zhang, Zeyao Ma, Yang Li, Bohan Zhang, Guanlin Li, Zijun Yao, Kangli Xu, Jinchang Zhou, Daniel Zhang-Li, Jifan Yu, Shu Zhao, Juanzi Li, and Jie Tang. 2024b. Tablellm: Enabling tabular data manipulation by llms in real office usage scenarios. *Preprint*, arXiv:2403.19318.

Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. 2023. Reactable: Enhancing react for table question answering. *CoRR*, abs/2310.00815.

Zhehao Zhang, Yan Gao, and Jian-Guang Lou. 2024c. e^5: Zero-shot hierarchical table analysis using augmented LLMs via explain, extract, execute, exhibit and extrapolate. 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 1244–1258, Mexico City, Mexico. Association for Computational Linguistics.

Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. 2021. TAT-QA: A question answering benchmark on a hybrid of tabular and textual content in finance. In *Proceedings of the 59th Annual*

Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 3277–3287, Online. Association for Computational Linguistics.

A License

For all datasets in our experiments, TableBench is under the license of MIT. The AIT-QA dataset is under the license of CDLA-Sharing-1.0. The TAT-QA dataset is under the license of Creative Commons (CC BY) Attribution 4.0 International. The FinQA dataset is under the license of Creative Commons Attribution 4.0 International. All of these licenses allow their data for academic use.

B Construction of CALTAB151

The pipeline of constructing the CALTAB151 dataset can refer to Figure 4. Figures 5 and 6 are Prompts used in the construction of CALTAB151 .

The datasets utilized in this study were sourced exclusively from pre-existing open-source repositories, with strict adherence to their respective licensing agreements throughout both data acquisition and implementation phases, and are solely for academic use.

Manually annotation is carried out by students with a specialized background in computer science. This entire process follows a strict data annotation workflow. We provide professional computing devices to support the annotators in completing the task.

C Examples of Datasets

Table 4, Table 5 and Table 6 demonstrate the examples of each dataset in our experiment. All tabular data are stored in the format of JSON strings.

D Case Study

We analyzed the error cases of the Qwen2.5-7B-Instruct on the CALTAB151 dataset. The detailed cases can be found in Figure 7.

In the case study, it is observed that the output of LLMs only addresses part of the question. For example, in CoT-based methods, not all sub-questions in complex multi-hop reasoning are answered. We propose that large models fail to recognize the sub-questions in such multi-hop scenarios, leading to incomplete answers.

In this case, MIX-SC, a PoT-based method, produces the error message: "could not convert string to float: '3,275'.". The error occurs due to the presence of the non-numeric character "," which prevents successful conversion to a float. This issue arises because PoT-based methods, during the code generation process, focus solely on generating the code to output the final answer, while neglecting the necessary preprocessing of table contents.

From the reasoning steps of TabSQLify and Chain-of-Table, it becomes clear that while some numbers are selected correctly, the subsequent calculations remain inaccurate. This further highlights the limitation of CoT-based methods in performing precise numerical computations.

E Prompts in TABDSR

In this subsection, we show the prompts used in our methods, as shown in Figures 7, 8, and 9.

F Other Prompts in Experiments

The prompts used in Table 1 and Table 2 are as follows:

Fine-tuning LLMs For TableGPT2 (Li et al., 2024), prompt for answer generation is Figure 10. For TableLLM (Zhang et al., 2024b), prompt for answer generation is Figure 11 and 12. For TableLlama (Zhang et al., 2024a), prompt for answer generation is Figure 13.

Prompt-based LLMs For Chain-of-Table (Wang et al., 2024), TabSQLify (Nahid and Rafiei, 2024a), and MIX-SC (Liu et al., 2024b), their codes are publicly available.

DP, PoT, TCoT and SCoT Prompts are from TableBench (Wu et al., 2024), as shown in Figure 14, Figure 15, Figure 16 and Figure 17.

G Appendix: Failure Analysis

To better understand the robustness and limitations of our TabDSR system, we conducted a comprehensive failure analysis covering all three agents: Decomposer (D), Sanitizer (S), and Executor (R). For each module, a failure leads to a fallback behavior, potentially degrading the final performance. The table 8 below reports the failure rates for each module (S in Table 9, R in Table 10) across three datasets:

Sanitizer (S) Error Types

The Sanitizer component encountered parsing issues when converting JSON outputs into struc-

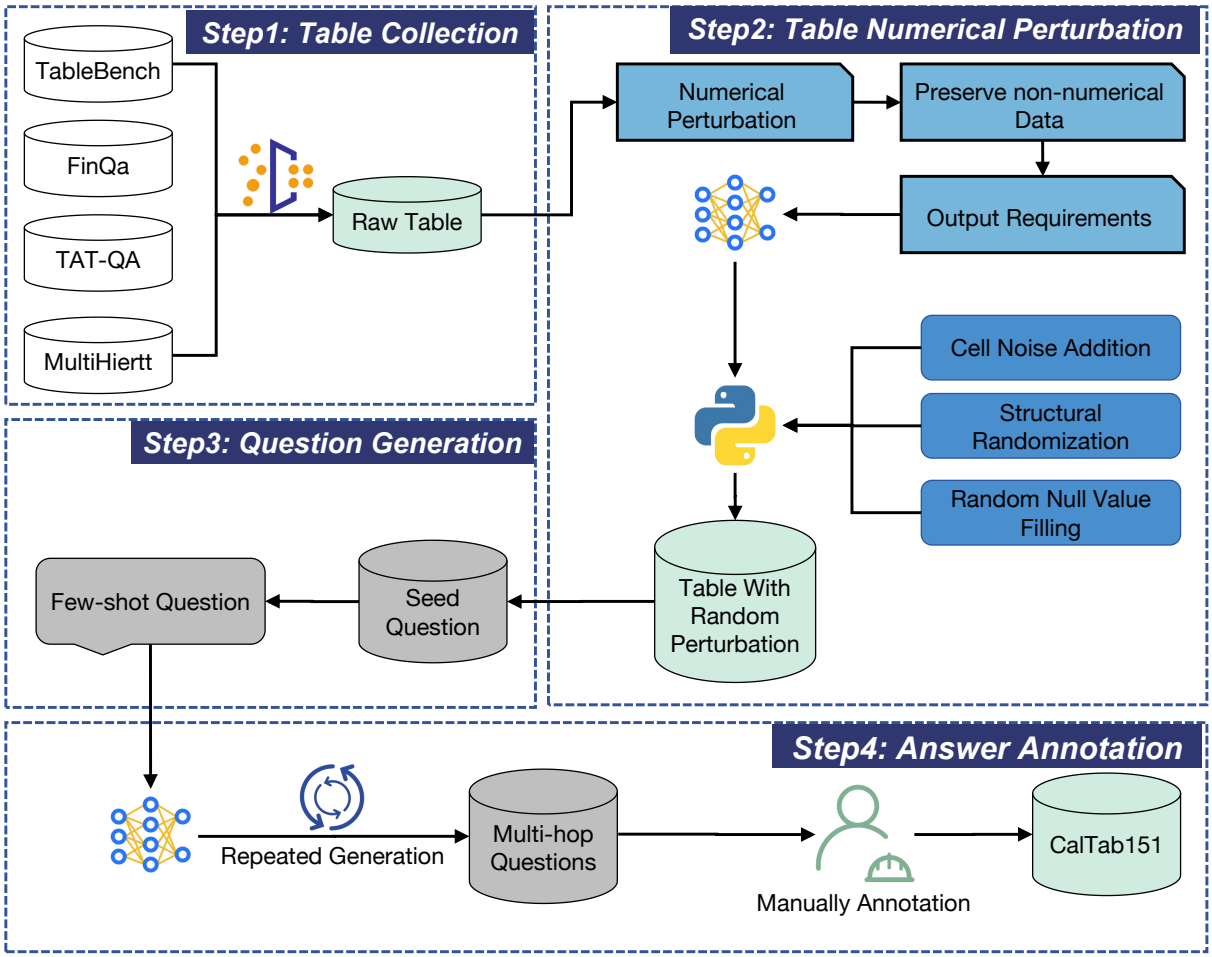

Figure 4: The construction pipeline of CALTAB151 .

tured tables. The most common errors were `JSONDecodeError` and `ValueError`.

Executor (R) Error Types

The Executor agent, responsible for code execution, had the highest failure rate. The frequent exceptions—such as `ValueError`, `KeyError`, and `TypeError`—indicate the challenges in generating robust code for table reasoning tasks.

Discussion

Our analysis reveals that the **Decomposer (D)** component was highly robust, with zero failures across all datasets, validating the structural soundness of our question decomposition pipeline.

The **Sanitizer (S)** exhibited low but non-negligible failure rates (1%–3%), mostly due to minor formatting or schema inconsistencies in the JSON output. These could potentially be mitigated by enhancing schema conformity or introducing lightweight JSON repair strategies.

The **Executor (R)** was the most failure-prone module, with 15%–17% of samples triggering code execution errors. This reflects the inherent difficulty of generating correct Python code under diverse and noisy table inputs. Addressing these issues remains an open challenge, potentially benefiting from more constrained decoding strategies or runtime feedback loops.

Overall, the failure analysis offers concrete insights for future improvement of each agent in the TabDSR pipeline.

Table	Question	Answer
{'columns': ['Remuneration key performance indicator', '2019 actual', '2019 threshold', '2019 target', '2019 maximum', 'Remuneration measure'], 'data': [['Group operating profit (ŏ0a3m)', '277.3', '256.7', '270.3', '283.8', 'Annual Incentive Plan'], ['Group cash generation (ŏ0a3m)', '296.4', '270.7', '285.0', '299.2', 'Annual Incentive Plan'], ['Group ROCE (%)', '54.5', '50.1', '52.7', '55.3', 'Annual Incentive Plan'], ['2017-2019 EPS (%)', '57.5', '27.6', 'N/A', '52.3', 'Performance Share Plan'], ['2017-2019 relative TSR (percentile TSR)', '94th', '50th', 'N/A', '75th', 'Performance Share Plan']]}	What was the maximum group operating profit in 2019?	283.8
{'columns': ['(In millions of dollars, except capital intensity)', 'Years ended December 31', '', ''], 'data': [['', '2019', '2018', '%Chg'], ['Capital expenditures 1', '', '', ''], ['Wireless', '1,320', '1,086', '22'], ['Cable', '1,153', '1,429', '(19)'], ['Media', '102', '90', '13'], ['Corporate', '232', '185', '25'], ['Capital expenditures 1', '2,807', '2,790', '1'], ['Capital intensity 2', '18.6%', '18.5%', '0.1 pts']]}	What was the increase / (decrease) in wireless capital expenditure from 2018 to 2019?	234
{'columns': [' ', 'June 30,', ' '],'data': [[' ', '2019', '2018'], ['Deferred tax assets', '', ''], ['Non-capital loss carryforwards', '$161,119', '$129,436'], ['Capital loss carryforwards', '155', '417'], ['Undeducted scientific research and development expenses', '137,253', '123,114'], ['Depreciation and amortization', '683,777', '829,369'], ['Restructuring costs and other reserves', '17,845', '17,202'], ['Deferred revenue', '53,254', '62,726'], ['Other', '59,584', '57,461'], ['Total deferred tax asset', '$1,112,987', '$1,219,725'], ['Valuation Allowance', '$(77,328)', '$(80,924)'], ['Deferred tax liabilities', '', ''], ['Scientific research and development tax credits', '$(14,482)', '$(13,342)'], ['Other', '(72,599)', '(82,668)'], ['Deferred tax liabilities', '$(87,081)', '$(96,010)'], ['Net deferred tax asset', '$948,578', '$1,042,791'], ['Comprised of:', '', ''], ['Long-term assets', '1,004,450', '1,122,729'], ['Long-term liabilities', '(55,872)', '(79,938)'], ['', '$948,578', '$1,042,791']]}	What is the total assets as of June 30, 2019?	948,578

Table 4: TAT-QA Dataset examples

Prompt Template of Query Generation

You are a data augmentation assistant. Your goal is to choose two most relevant SubQueries and integrate two of those SubQueries into a human readable and professional question. The Question should demonstrate a logical progression and incorporate nested relationships.
Reference SubQueries

What was the amount of unrecognized stock-based compensation expense related to unvested employee stock options in 2019?
What was the total stock-based compensation expense amount in 2018?
How long is it expected to take for the unrecognized stock-based compensation expense related to unvested RSUs to be recognized?
What is the total stock-based compensation expense and unrecognized stock-based compensation expense in 2019?
What was the change in the amount of stock options in 2019 from 2018?
What was the percentage change in the amount of RSUs in 2019 from 2018?

answer

The questions "What was the change in the amount of stock options in 2019 from 2018?" and "What was the percentage change in the amount of RSUs in 2019 from 2018?" both reference the period "in 2019 from 2018." Therefore, we can combine them into a single question about the same time period. The final question is formatted in JSON as follows:

```json
{
  "Question": "What was the change in the amount of stock options in 2019 from 2018? Additionally,
  what was the percentage change in the amount of RSUs during the same period?"
}
```

Reference SubQueries

```
{{ReferQuestion}}
```

answer

You must output in json format:

- Question: string, a human readable and professional question, consist of two most relevant SubQueries.

```json
{
  "Question": "string, a human readable and professional question."
}
```

Give a final question in json format in the end, Let's think step and step!

Figure 5: **Query Generation** generates multi-hop queries by few-shot prompt.

Prompt Template of Numerical Perturbation

You are a data augmentation assistant. Your task is to generate a new table by applying the following transformation rules exclusively to numeric data in the table, including numbers stored as strings. Specifically:
1. `Numerical Perturbation`:
   - Identify Numeric Data:
      - Include all numeric values (integers, floats, or numbers stored as strings), even if mixed with other characters (e.g., $100.00, 123.45kg).
      - Exclude data that clearly represents dates or times (e.g., YYYY-MM-DD, MM/DD/YYYY, or time formats like HH:MM:SS).
   - Apply Perturbation:
      - Randomly adjust numeric values (including those within strings) by up to ±3%-5% of their original value.
   - Maintain data realism:
      - If the original value is an integer, the perturbed value must remain an integer.
      - If the value is a float, retain its decimal format with appropriate precision.
      - For strings containing numeric values, only adjust the numeric portion, leaving non-numeric characters intact (e.g., $100.00 → $103.00).
   - Ensure that the perturbation keeps the values realistic within the context of the data.
2. `Preserve Other Data`:
   - Retain all non-numeric columns, values, and formats unchanged.
   - Date or time columns must not be perturbed or modified.
3. `Output Requirements`:
   - Directly output the augmented table in JSON format, maintaining the structure of the input table. The JSON must include:
      - "columns": An array of column names.
      - "data": A 2D array of table rows after transformation.
      - "index": The original index of each row.

- `Identification Criteria`:
   - Numeric Columns: Include numbers (int, float) and numbers stored as strings, even if they contain additional non-numeric characters.
   - Date Columns: Avoid perturbation for values matching common date/time formats (e.g., YYYY-MM-DD or HH:MM:SS).

Input

```json
{{Inputs}}
```

Output

Figure 6: **Numerical Perturbation** with zero-shot prompt, generate a new table by applying transformation rules exclusively to numeric data in the table, including numbers stored as strings, but excluding data of dates or times.

Table	Question	Answer
{'columns': ['season', 'tropical lows', 'tropical cyclones', 'severe tropical cyclones', 'strongest storm'], 'data': [['1990 - 91', 10, 10, 7, 'marian'], ['1991 - 92', 11, 10, 9, 'jane - irna'], ['1992 - 93', 6, 3, 1, 'oliver'], ['1993 - 94', 12, 11, 7, 'theodore'], ['1994 - 95', 19, 9, 6, 'chloe'], ['1995 - 96', 19, 14, 9, 'olivia'], ['1996 - 97', 15, 14, 3, 'pancho'], ['1997 - 98', 10, 9, 3, 'tiffany'], ['1998 - 99', 21, 14, 9, 'gwenda'], ['1999 - 00', 13, 12, 5, 'john / paul']]}	What is the average number of tropical cyclones per season?	10.6
{'columns': ['draw', 'artist', 'song', 'points', 'place'], 'data': [[1, 'niamh kavanagh', 'in your eyes', 118, 1], [2, 'suzanne bushnell', 'long gone', 54, 7], [3, 'patricia roe', 'if you changed your mind', 75, 3], [4, 'róisín ní haodha', 'mo mhúirnín óg', 34, 8], [5, 'champ', '2nd time around', 79, 2], [6, 'off the record', 'hold out', 61, 6], [7, 'dav mcnamara', 'stay', 67, 4], [8, 'perfect timing', 'why aren't we talking anyway', 62, 5]]}	What is the difference in points between the artist with the highest points and the average points of the top 3 artists?	35.67
{'columns': ['party', 'administrative panel', 'agricultural panel', 'cultural and educational panel', 'industrial and commercial panel', 'labour panel', 'national university of ireland', 'university of dublin', 'nominated by the taoiseach', 'total'], 'data': [['fianna fáil', 2, 4, 2, 3, 5, 0, 0, 9, 25], ['fine gael', 3, 4, 3, 3, 2, 1, 0, 0, 16], ['labour party', 1, 1, 0, 1, 2, 0, 0, 0, 5], ['clann na talmhan', 0, 1, 0, 0, 0, 0, 0, 0, 1], ['independent', 1, 0, 0, 1, 1, 2, 3, 1, 9], ['total', 7, 11, 5, 9, 11, 3, 3, 11, 60]]}	What is the total number of seats held by parties that have at least 2 seats in the agricultural panel, and what percentage of the total seats do they represent?	41, 68.33%

Table 5: TableBench Dataset examples

Table	Question	Answer
{'columns': ['player', 'average', '100s', 'matches', 'highest score', 'runs', '50s'], 'index': [0, 1, 2, 3, 4, 5, 6, 7], 'data': [['lionel palairet', '32.04', '1', '10.0', '103', '$575.00', '5'], ['herbie hewett', '18.98', '0', '12.0', '67', '$398.00', '2'], ['richard palairet', '19.52', '0', '10.0', '76', '$273.000', '1'], ['sammy woods', '18.82', '0', '11.0', '51', '$339.0', '1'], ['vernon hill', '12.58', '0', '9.0', '32', 'N/A', '0'], ['john challen', '25.98', '0', '9.0', '-', '$364.0', '2'], ['george nichols', '10.56', '0', '12.0', '38', '$222.00', '0'], ['ted tyler', '10.14', '0', '12.0', '63', '$172.0', '1']]}	What is the total number of matches played by all players in the table, and what is the average number of matches per player, given the total number of matches?	85, 10.63
{'columns': ['', '', 'Year Ended December 31,', 'Year Ended December 31,'], 'index': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24], 'data': [['', '', '2017 (a)%', '2016 (a)%'], ['Nonoperating income (expense):', 'Interest income', '59', '43'], ['Operating expense:', 'Depreciation and amortization', '2,213', '2,036'], ['Nonoperating income (expense):', 'Income tax expense', '923', '1,585'], ['Operating expense:', 'Other operating expenses', '5,717', '5,477'], ['Operating revenue:', 'Passenger revenue', '$35,494', '$34,432'], ['Operating expense:', 'Operating income', '3,781', '4,474'], ['Nonoperating income (expense):', 'Miscellaneous, net', '(104)', '(11)'], ['Nonoperating income (expense):', 'Interest expense', '(691)', '(694)'], ['Nonoperating income (expense):', 'Interest capitalized', '87', '74'], ['Operating expense:', 'Landing fees and other rent', '2,307', '2,230'], ['Operating expense:', 'Aircraft maintenance materials and outside repairs', '1,912', '1,801'], ['Nonoperating income (expense):', 'Total nonoperating expense, net', '(650)', '(588)'], ['Operating revenue:', 'Total operating revenue', '38,917', '37,654'], ['Operating expense:', 'Total operating expense', '35,136', '33,180'], ['Nonoperating income (expense):', 'Income before income taxes', '3,131', '3,886'], ['Operating expense:', 'Special charges', '181', '767'], ['Operating expense:', 'Regional capacity purchase', '2,299', '2,263'], ['Operating expense:', 'Salaries and related costs', '11,269', '10,481'], ['Operating expense:', 'Aircraft fuel', '7,120', '5,987'], ['Operating expense:', 'Distribution expenses', '1,478', '1,437'], ['Operating expense:', 'Aircraft rent', '640', '700'], ['Operating revenue:', 'Cargo', '1,147', '962'], ['Nonoperating income (expense):', 'Net income', '$2,208', '$2,301'], ['Nonoperating income (expense):', 'Earnings per share, diluted', '$7.27', '$6.96']]}	What is the total nonoperating expense for the year ended December 31, 2017, and What is the percentage change it the percentage change in total nonoperating income from 2016 to 2017? Express it as a percentage.	7860.27, -0.1446
{'columns': ['($ in millions)', '', ''], 'data': [['For the year ended December 31:', '2019', '2018'], ['Net cash provided by/(used in) continuing operations', '', ''], ['Operating activities', '15,200′,′15,700'], ['Investing activities', '(27,800)', '(5,050)'], ['Financing activities', '9,300', '(10,800)'], ['Effect of exchange rate changes on cash, cash equivalents and restricted cash', '(172)', '(510)'], ['Net change in cash, cash equivalents and restricted cash', '$(3,390)', '$(650)']], 'index': [0, 1, 2, 3, 4, 5, 6]}	What was the increase or decrease in cash from operating activities from 2018 to 2019, and What is the corresponding percentage change during the same period?	-477, -3.13

Table 6: CalTab151 Dataset examples

Error Type	Question	Error
Code Execution Error (MIX-SC)	What is the total number of Jamaicans granted British citizenship from 1997 to 2008, excluding years with null or missing data, and how many records are valid over the same period?	valueerror: could not convert string to float: '3,275'
Partial Answer (TabSQLify)	What is the total amount of estimated interest expense across all years, and how does it compare to the total amount of senior unsecured notes, higher or lower?	step 1: the estimated interest expense values are: - 2011: $273,200 - 2012: $252,911 - 2013: $227,796 - 2014: $203,042 - 2015: $182,892 - after 2015: $365,797 summing these values: 273200+252911+227796+203042+182892+365797=1407638. the total estimated interest expense is $1,407,638. step 2: calculate the total senior unsecured notes the senior unsecured notes values are: - 2011: $87,368 - 2012: $126,866 - 2013: $2014 (note: this is not a numerical value, so it should be treated as 0) - 2014: $101,535 - 2015: $677 after 2015: $817,115 summing these values: 87368 + 126866 + 0 + 101535 + 677 + 817115 = 1133661 so, the total senior unsecured notes is $1,133,661. step 3: compare the totals now, we compare the two totals: - total estimated interest expense: $1,407,638 - total senior unsecured notes: $1,133,661 $1,407,638 is greater than $1,133,661. answer: the total estimated interest expense is higher than the total senior unsecured notes
Partial Answer,Calculational error (Chain-of-Table)	What is the number of categories of aircraft listed in the table? and what is the total average seating capacity across the aircraft models? Do not calculate the unoccupied parts of the aircraft.	The number of categories of aircraft listed in the table is 13. The total average seating capacity across these aircraft models is 2040 (calculated as the sum of all values in the table: 131 + 154 + 182 + 265 + 299 + 164 + 183 + 216 + 270 + 318 + 232 + 292 + 101 + 143 = 2040).

Table 7: Case Studies for different errors in Qwen2.5-7B-Instruct; Green is the correct answer. Red is the incorrect answer of prompt-based methods, which have been used in the Table 1.

Component	TatQa	TableBench	CalTab151
Decomposer (D)	0.00	0.00	0.00
Sanitizer (S)	0.03	0.03	0.01
Executor (R)	0.15	0.15	0.17

Table 8: Failure rates of each component in TabDSR.

Error Type	TatQa	TableBench	CalTab151
JSONDecodeError	14	3	1
ValueError	10	9	0
AttributeError	0	1	0

Table 9: Sanitizer (S) error types across datasets.

Prompt Template of Query Decomposer

You are tasked with analyzing a user query to identify the number of sub-questions it contains. Your objectives are to:
1. `Identify Sub-Questions`: Split the query into sub-questions based solely on conjunctions (like "and", "or") or punctuation (such as commas). Treat each segment as a distinct sub-question boundary.
2. `Count Sub-Questions`: Provide the total number of sub-questions identified.
3. `List Sub-Questions`: List each sub-question in order as they appear in the original query.

Input Format

You will receive input in JSON format with the following keys:
- `Query`: User query string.

```json
{
  "Query": "Query String"
}
```

Example Input

```json
{
  "Query": "How much money was spent on product A and how much did product B sell in total in 2015?
           Finally, tell me the total sales of both products for the entire year."
}
```

Expected Output

Your output should be a JSON object containing:
1. `subQueryCount`: The total number of sub-questions.
2. `subQueries`: A list of sub-questions in their original form.

Example Output

```json
{
  "subQueryCount": 3,
  "subQueries": [
    "How much money was spent on product A",
    "how much did product B sell in total in 2015",
    "tell me the total sales of product A and product B for the entire year"
  ]
}
```

Important Notes
- `Strict JSON Format`: Ensure the output is valid JSON that can be parsed by json.loads.
- `No Complex Reasoning`: Do not attempt to infer meanings, just split based on conjunctions and punctuation.

Input

```json
{{Inputs}}
```

Figure 7: **Query Decomposer** divides a question into several sub-questions.

Prompt Template of Table Sanitizer

You are tasked with cleaning and processing a JSON-formatted table while preserving the original structure as much as possible. Your main objectives are to:
1. Ensure column names are unique.
2. Clean cell data.
3. Maintain consistent formatting within the table.

Input Structure

You are provided with a JSON-formatted table containing three fields: `columns`, `data`, and `index`.
- `columns`: An array of strings, each representing the name of a table column.
- `data`: A 2D array where each nested list represents a row in the table, with each element corresponding to the cells in that row under each column.
- `index`: An array of integers, each representing the index of a table row.
  - `Note`: Rows in the `data` field are numbered starting from 0 up to the total number of rows minus one. The header row, if present, is not included in these numbers; only rows with actual data are counted.

Task Requirements

You need to clean and process the table based on the following requirements:

1. **Columns Cleaning:**
   - Ensure all column names are unique.
   - If duplicate column names exist:
     - Check if the first row of data contains nested column headers.
     - If nested headers are present, remove this row from the data. Rename columns to ensure uniqueness while preserving the original context.

2. **Cell Data Cleaning:**
   - **Numerical Columns:**
     - Remove extraneous symbols (e.g., %, $, commas, etc.) and ensure consistent numerical formatting.
     - Convert these values into a numerical type (e.g., float or integer) in the output JSON format.
   - **Non-Numerical Columns:**
     - Replace invalid, empty, or missing cells (e.g., N/A, null, None) with `null`.

3. **Row Filtering:**
   - Identify and exclude rows containing summary information such as: “Total”, “Sum”, “Average”, or similar statistical descriptors.
   - Retain all other rows to preserve the integrity of the dataset.

4. **Output Structure:**
   - Ensure that the resulting table maintains the JSON format:
     - All cleaned and processed columns and rows must be included.
     - No essential data should be lost unless explicitly instructed to remove it (e.g., summary rows).

Output Structure

The output should be a JSON object with the same structure as the input, containing the cleaned and processed data:
- `columns`: An array of strings, each representing the name of a table column.
- `data`: A 2D array where each nested list represents a row in the table, with each element corresponding to the cells in that row under each column.
  - Ensure the output maintains the strict JSON format enclosed in `json`.
  - All numerical data columns must be converted to appropriate numerical types.

Input

```json
{{Inputs}}
```

Figure 8: **Table Sanitizer** preprocesses json-formatted table.

Prompt Template of PoT-based Reasoner

Input format
You will be provided with a valid python code containing a dict of **table_data** with the following keys.
- **columns**: An array of strings, each representing the name of a table column.
- **data**: A 2D array where each nested list represents a row in the table, with each element corresponding to the cells in that row under each column.
- **index**: An array of integer or string, each representing the index of a table row.
  - **Note**: The rows in the data are numbered starting from 0 up to the total number of rows minus one (for example, if there are 10 rows, they would be numbered from 0 to 9). The header row, if present, is not included in these numbers; only rows with actual data are counted.
- **Queries**: A array of string containing the user's sub-queries or request for specific information from the table.

Analyze the table's structure by recognizing the relation between columns and their respective cells in data. Use these associations to identify relevant information in each cell that pertains to the Query.

The input format is as follows:

```python
{{InputExample}}
```

Task instructions

You should follow these requirements below:

- **Analyze the Queries**:
  - For each sub-query, provide the following:
    - **Sub-query order**: Label each sub-query in order (e.g., "Sub-query 1:", "Sub-query 2:", etc.).
    - **Column and row indices**: Identify the relevant columns and rows in the table that are needed to answer the sub-query.
    - **Python code**: For each sub-query, write the corresponding Python code to extract the relevant data and compute the answer.
- **Code Quality**:
  - The Python code must be concise, easy to understand, and modular.
  - If necessary, add comments for clarity.
  - Follow best practices for code efficiency and readability.
- **Data Context**:
  - Base your analysis entirely on the provided table data. Do not use any external data or make assumptions.
  - If the Query is not related to the provided table data, politely refuse and provide a response explaining why.
- **Handling Multiple Sub-Queries**:
  - For multiple sub-queries, print each sub-query's order (e.g., "Sub-query 1:", "Sub-query 2:").
  - For each sub-query, identify the relevant column and row indices and extract the necessary information.
  - For each sub-query, generate Python code to retrieve the data from the table.
- **Data Type Casting**:
  - Identify every column in the DataFrame and cast columns to appropriate data types (e.g., int, float, object) if necessary to ensure the code executes correctly.
- **Output Formatting**:
  - Provide Python code that loads the table data using the pandas library, don't response any other description.
  - Ensure to load the table with command table_df=pd.DataFrame(table_data['data'], columns=table_data['columns']).
  - If the Query involves numerical calculations, perform them using DataFrame methods to get the final answers and print the final answers.
    - **Print the final answers**: Ensure that the final output includes the print() function to display answers. Do not print any other description information.
    - **Handle numerical outputs**: For any query involving calculations, format the final answer using Python's rounding function **round()** to ensure that results are output with exactly two decimal places.

Replace index_1, index_2, etc., with the actual indices based on the identified columns and rows. If no columns or rows are identified as relevant, return an empty array for that key.

User Input

```python
{{Inputs}}
```

Figure 9: **PoT-based Reasoner** integrates the sub-questions and sanitized tabular data into a unified reasoning framework.

Prompt Template of TableGPT2

Given access to several pandas dataframes, write the Python code to answer the user's question.
/*
"{var_name}.head(5).to_string(index=False)"as follows:
{df_info}
*/

Question: {user_question}

Figure 10: **TableGPT2's Prompt for Answer Generation**

Prompt Template of TableLLM

[INST]
Below are the first few lines of a CSV file. You need to write a Python program to solve the provided question.
Header and first few lines of CSV file:
{csv_data}
Question: {question}[/INST]

Figure 11: **TableLLM's Prompt for Code Solution (PoT)**

Prompt Template of TableLLM

[INST]
Offer a thorough and accurate solution that directly addresses the Question outlined in the [Question]. The answer should follow the format below:
[Answer Format]
Final Answer: AnswerName1, AnswerName2...
Ensure the final answer format is the last output line and can only be in the "Final Answer: AnswerName1, AnswerName2..."form, no other form. Ensure the "AnswerName"is a number or entity name, as short as possible, without any explanation.
[Table Text]
There is a table with no title.
[Table]
```
{table_in_csv}
```
[Question]
[question]
[Solution][INST/]

Figure 12: **TableLLM's Prompt for Text Answer (DP)**

Prompt Template of TableLlama

Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.
Instruction:
This is a table QA task. The goal of this task is to answer the question given the table.
Input:
{input}
Question:
question
Response:

Figure 13: **TableLlama's Prompt for Answer Generation**

Prompt Template of DP

Your task is to answer questions based on the table content.
The answer should follow the format below:
[Answer Format]
Final Answer: AnswerName1, AnswerName2...
Ensure the final answer format is the last output line and can only be in the "Final Answer: AnswerName1, AnswerName2..."form, no other form. Ensure the "AnswerName"is a number or entity name, as short as possible, without any explanation.
Give the final answer to the question directly without any explanation.
Read the table below in JSON format:
[TABLE]
{{tableString}}
Let's get start!
Question: {{questionString}}

Figure 14: **DP Prompt in TableBench**

Prompt Template of PoT

You are a data analyst proficient in Python. Your task is to write executable Python code to analyze the table and then answer questions.
[Guidelines]
You should act following requirements below:
1. based on the question, write out your analytical approach, and then write Python code according to this approach.
2. The code needs to be concise and easy to understand, and if necessary, add comments for clarification.
3. Code blocks need to strictly start with ```python and end with ```.
4. Your analysis must be based entirely on the above data. If the user's question is not related to data analysis, please politely refuse.
5. You need to generate executable code. If there are results to be presented, please use the print function; if there are charts, please use the matplotlib library to draw them.
6. Ensure to load the table with command ```df = pd.read_csv('table.csv')```.
The answer should follow the format below:
[Answer Format]
Final Answer: AnswerName1, AnswerName2...
Ensure the final answer format is the last output line and can only be in the "Final Answer: AnswerName1, AnswerName2..."form, no other form. Ensure the "AnswerName"is a number or entity name, as short as possible, without any explanation.
Let's think step by step and then generate python code to analyze table and present the final answer to the question.
Read the table below in JSON format:
[TABLE]
{{tableString}}
Let's get start!
Question: {{questionString}}

Figure 15: **PoT Prompt in TableBench**

Error Type	**TatQa**	**TableBench**	**CalTab151**
ValueError	39	17	1
KeyError	43	13	6
SyntaxError	0	1	2
NameError	3	10	5
TypeError	13	23	8
IndexError	7	8	3
AttributeError	3	0	0
UFuncTypeError	0	1	0
Total Errors	**108**	**74**	**25**
Error Rate (Errors / Dataset Size)	**0.15**	**0.15**	**0.17**

Table 10: Executor (R) error types and frequencies.

Prompt Template of TCoT

You are a table analyst. Your task is to answer questions based on the table content.
The answer should follow the format below:
[Answer Format]
Final Answer: AnswerName1, AnswerName2...
Ensure the final answer format is the last output line and can only be in the "Final Answer: AnswerName1, AnswerName2..."form, no other form. Ensure the "AnswerName"is a number or entity name, as short as possible, without any explanation.
Let's think step by step and then give the final answer to the question.
Read the table below in JSON format:
[TABLE]
{{tableString}}
Let's get start!
Question: {{questionString}}

Figure 16: **TCoT Prompt in TableBench**

Prompt Template of SCoT

You are a table analyst. Your task is to utilize the Python package 'pandas' to analyze the table and then answer questions.
[Guidelines]
You should act in following patterns step by step to analyze the table and then give the final answer:
Patterns]
Thought: You should always think about what to do to interact with Python code base on Result
Action: the action can **ONLY** be single line python code
Result: Simulate the result of the execution of the python code in Action, analyse that result and decide whether to continue or not
(This thought/Action/Result can repeat N times) answer should follow the format below:
[Answer Format]
Final Answer: AnswerName1, AnswerName2...
Ensure the final answer format is the last output line and can only be in the "Final Answer: AnswerName1, AnswerName2..."form, no other form. Ensure the "AnswerName"is a number or entity name, as short as possible, without any explanation.
Let's think step by step and then give the final answer to the question.
Ensure to have a concluding thought that verifies the table, observations and the question before giving the final answer.
Read the table below in JSON format:
[TABLE]
{tableString}
Let's get start!
Question: {questionString}

Figure 17: **SCoT Prompt in TableBench**