ProgramTab: Boosting Table Reasoning of LLMs via Programmatic Paradigm

Anonymous ACL submission

Abstract

Table-based reasoning with large language models (LLMs), which requires reasoning based on natural language questions and structured tabular data, has gained widespread attention. However, a series of issues still constrain the application of this task. The previous approaches suffered from significant performance degradation when faced with large tables due to the difficulty of long text modeling and the limitation of input length for LLMs. The text-to-SQL approach is used to efficiently extract key information from tables and generate smaller sub-tables. However, tabular data, especially web tables, often lack the necessary structure and consistency, making them unsuitable for performing mathematical logic operations using SQL queries. We propose the ProgramTab framework, which guides LLMs employing in-context learning to perform tabular data preprocessing with Python code, as well as the momentous contents extraction with row and column extraction and SQL generation. Data preprocessing includes defining the data format and type based on the different questions. The experiment results on WikiTQ and TabFact datasets demonstrate that the ProgramTab framework effectively deals with table-based reasoning tasks and outperforms all LLM-based baselines.

1 Introduction

Tables, as a popular form of data representation, play a significant role in everyday work and life. Analysis and reasoning based on tabular data have emerged as a hot topic in natural language processing, attracting wide attention from academia and industry. The main downstream tasks of tabular reasoning include table-based fact verification (Chen et al., 2020; Aly et al., 2021) and table-based question answering (Panupong and Percy, 2015; Cho et al., 2019). The challenges of these tasks lie in how to enable language models to comprehend table data content, including text, numbers, etc., establish their connection with user queries, and execute efficient logical reasoning and computations.

Title: 1981 Houston Oilers season

date	opponent	result
september 6, 1981	at los angeles rams	w 27–20
september 13, 1981	at cleveland browns	w 9–3
september 27, 1981	miami dolphins	l 10–16
…	…	…
december 20, 1981	pittsburgh steelers	w 21–20

Figure 1: An example of a table in WikiTQ dataset.

Recently, LLMs (Brown et al., 2020; Hoffmann et al., 2022; OpenAI, 2022; Touvron et al., 2023) have significantly transformed the landscape of natural language processing tasks with their impressive understanding and generation capabilities. Instead of fine-tuning the pre-trained models, sufficiently making use of the in-context learning of LLMs to solve complex tabular data reasoning has been a mainstream direction (Chen, 2023; Cheng et al., 2023; Ye et al., 2023; Wang et al., 2024). However, current methods still face several limitations. Firstly, most of the work (Cheng et al., 2023; Ye et al., 2023; Wang et al., 2024) treats the entire table as an input, which is unsuitable for tables containing large amounts of data. When the total number of tokens in a table exceeds the maximum input limitation of LLMs, the content of the table will be truncated, leading to information loss and consequently affecting the performance of LLMs. This has been verified in the work of (Chen, 2023). To mitigate the length constraint of inputs, the common approach is to utilize a programmatic language, such as generating SQL queries to retrieve the most relevant rows and column data (Ye et al., 2023; Nahid and Rafiei, 2024b; Zhang et al., 2024c,a). However, **table data, especially the web table is usually provided as strings and often lacks the necessary structure and consistency, requiring conversion to the appropriate format and data types for mathematical logic operations to avoid calculation errors.** It will require SQL to preprocess the data while extracting the relevant information, which increases the complexity of generating SQL for LLMs. For example, for the table shown in Figure 1, when the question is about the number of games the Houston Oilers won in the 1981 season, the 'w' and 'l' symbols from the result cell aren't provided as a single column and need to be extracted, which is defined as the "lack of necessary structure". Regarding the absence of consistency, we can find that the structure at the "year" column in Figure 2 is inconsistent, such as "1931" and "spring 1932".

To address the above challenges, with the help of in-context learning in LLMs, we introduce the ProgramTab framework, which executes with program languages (Python and SQL) to flexibly handle the table contents based on the questions. Specifically, as shown in Figure 2, (1) we utilize the embedding model to compute the relevant scores of each line of tables with the questions and resort the lines in descending order. In the following steps, the top K lines with higher relevant scores are extracted as instances to replace the complete tables. With the most relevant lines as input, (2) LLMs are prompted to select the columns related to the questions, (3) generate the Python code to preprocess the table data, including unifying the data format and defining the data type for each column. After that, (4) SQL queries are generated using chain-of-thought (CoT) (Wei et al., 2023) and executed to obtain the most valuable information. Finally, (5) LLMs process this information and produce the final answers.

We validate our ProgramTab framework by conducting experiments on two challenging table reasoning datasets: WikiTQ (Panupong and Percy, 2015) and TabFact (Chen et al., 2020). With three LLM backbones, our evaluation demonstrated that ProgramTab achieves excellent performance on table-based reasoning benchmarks, and outperforms all the other baselines with different LLM backbones. Besides, ProgramTab is not limited by the input length of table data, which obtains a significant efficiency and effectiveness improvement compared with other strong baselines.

2 Related Work

In this section, we introduce the related approaches of table-based reasoning and divide them into two categories: fine-tuning-based and prompting-based table reasoning.

2.1 Fine-tuning-based Table reasoning

Table-based understanding and reasoning tasks are significant in data analysis systems. Many approaches focus on constructing pre-trained language models and fine-tuning them to address these tasks (Zhang et al., 2020; Patnaik et al., 2024). Among them, mask language models (MLM) are widely adopted. For example, TaPas (Herzig et al., 2020) acquires BERT (Devlin et al., 2019) to parse table information via pre-training. PASTA (Gu et al., 2022) pre-trains DeBERTaV3 (He et al., 2021) to perform six types of common sentence–table cloze tasks. Besides, TAPEX (Liu et al., 2022) employs the BART (Lewis et al., 2020) model to learn the neural SQL executors over a synthetic corpus. OmniTab (Jiang et al., 2022) leverages retrieval to pair relevant natural sentences with mask-based pre-training and synthesizes natural language questions by converting sampled SQL from tables. Inner Table Retrieval (ITR) (Lin et al., 2023) extracts sub-tables to preserve the most relevant information for the questions.

2.2 Prompting-based Table Reasoning

Recently, LLMs (Hoffmann et al., 2022; OpenAI, 2023; Touvron et al., 2023) have gained widespread attention due to their powerful understanding and generation capabilities. Given a few augmenting few-shot examples relevant to the tasks, the LLMs can tackle various reasoning tasks (Fu et al., 2023; Zhang et al., 2023). A few approaches also employ LLMs to tackle table reasoning tasks with few-shot prompts. TableCoT (Tai et al., 2023) systematically explores the performance of LLMs on table reasoning tasks and finds that LLMs are excellent at solving such tasks, especially combined with CoT approach. Besides, rather than generating general text, additional programmatic text, such as Python programs (Chen et al., 2022; Gao et al., 2023), and Text-to-SQL (Rajkumar et al., 2022) approaches are employed to improve the performance further. LEVER (Ni et al., 2023) improves the performance of code LLMs on language-to-code tasks by training separate verifiers to validate the programs generated by LLMs and their exe-

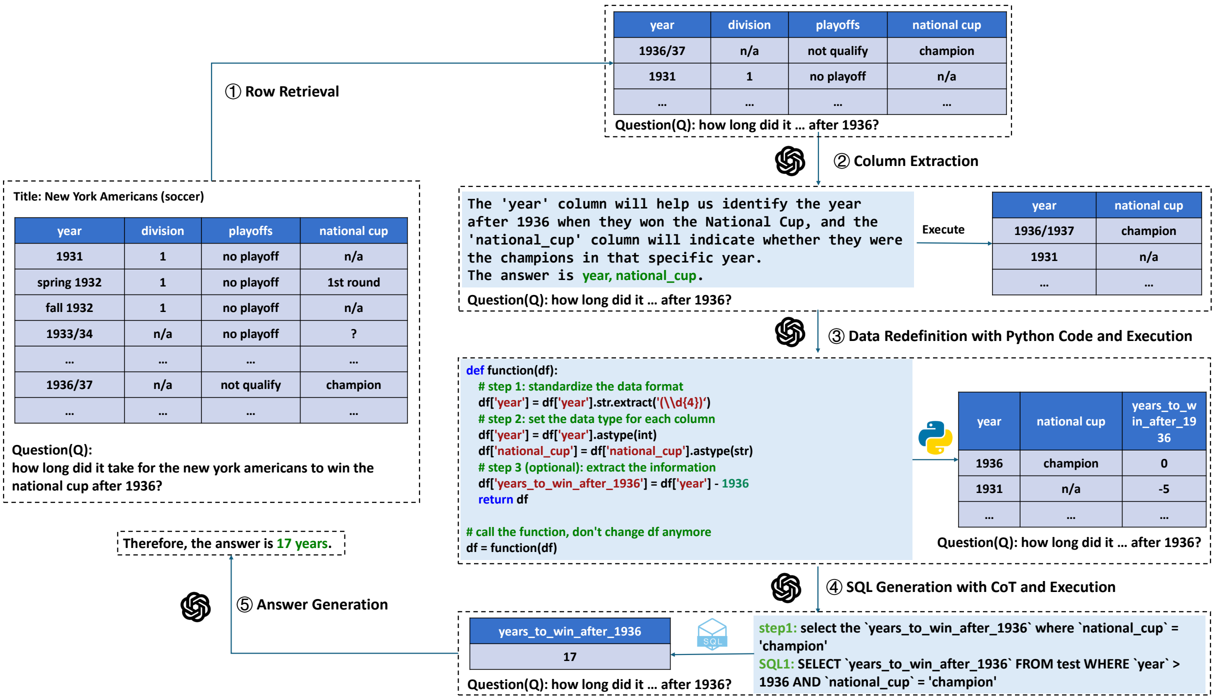

Figure 2: The overview of ProgramTab for table-based reasoning.

cution results. Binder (Cheng et al., 2023) maps the task input to a program that allows generating SQL or Python programs and extending their functions by calling LLMs in the program. ReAcTable (Zhang et al., 2024b) breaks down the problem into multiple steps and uses LLMs to generate code programs that are executed through external tools for each step. Finally, it leverages majority voting to improve overall accuracy. Wang et al. (2024) proposes a Chain-of-Table framework that designs a series of table operations and dynamically plans an operation chain based on the inputs. It's difficult for LLMs to perform reasoning when confronted with large tables with multiple rows. Dater (Ye et al., 2023), TabSQLify (Nahid and Rafiei, 2024b) and H-STAR (Nikhil et al., 2024) decompose the original table into the sub-table by selecting the relevant rows and columns. After that, Dater and Alter (Zhang et al., 2024a) also propose the parsing-execution-filling and query augmentation strategy respectively to decompose a complex question into simpler step-by-step sub-questions by generating an intermediate SQL. E^5 (Zhang et al., 2024c) presents an algorithm to condense large tables while maintaining useful information. The most similar work is NormTab (Nahid and Rafiei, 2024a), which utilizes LLMs to regularize table content, making it conducive to SQL query generation. Unlike previous works, which typically extract information directly using SQL queries to obtain answers—thus increasing the difficulty of SQL generation, we propose an innovative approach that leverages LLMs to generate code for data preprocessing. It effectively reduces the difficulty of SQL generation and improves efficiency compared to NormTab. Additionally, we present innovative optimization methods for SQL generation process.

3 ProgramTab Reasoning

As shown in Figure 2, ProgramTab consists of five procedures: 1) **row retrieval**, 2) **column extraction**, 3) **data definition with code**, 4) **SQL generation** and 5) **answer generation**. In this section, we describe the above procedures in detail. The original table is denoted as T.

3.1 Row Retrieval

To alleviate the limitation of the input length of LLMs, we first execute row retrieval, extracting the most relevant rows to represent the entire table content. Specifically, for each row of data in the table, we concatenate the column name and value of the cells to form a text segment, and an embedding model GTE-base (Li et al., 2023) is utilized to calculate the relevance score between the row data and the question. Ultimately, the top K most relevant rows are selected as instances in the prompt templates of the following steps, which effectively

```
### Task description: Please select the relevant columns about the
question from ### Table headers. We also provide a few rows about
the value of different headers to assist to choose the columns.
Please follow the format that describe the reasons of choosing the
column firstly and give the conclusion as 'The answer is' finally.

###Example1:
Question: which team won previous to crettyard?
### Table header: team | county | wins | years won
### A few rows:
row1: Greystones | Wicklow | 1 | 2011
row2: Ballymore Eustace | Kildare | 1 | 2010
row3: Maynooth | Kildare | 1 | 2009
### Answer: To find out which team won before crettyard, we need to
look at the 'years won' column to determine the year crettyard won
and find the team that won the year before. Besides, the 'team'
column is also crucial because it has the names of the teams that
won in those years. The answer is team, years won.


###Example2:
Question: did february 2012 or july 2006 have more total votes?
### Table header: polling_firm | month | link | favor | oppose
### A few rows:
row1: utgers-eagleton | march 2014 | | 64 | 28
row2: quinnipiac | july 2013 | | 60 | 31
row3: rutgers-eagleton | june 2013 | | 59 | 30
### Answer: To answer the question about whether February 2012 or
July 2006 had more total votes, we need to look at the 'month'
column to find the data for these two specific months. In addition,
the 'favor' and 'oppose' columns are also important because they
contain the number of votes. By adding these two columns together,
we can get the total votes for each month. The answer is month,
favor, oppose.

### Problem to be solved:
Question: {}
### Table header: {}
### A few rows:{}
### Answer:
```

Figure 3: Prompt for LLMs to extract columns.

alleviates the whole table as the input context.

3.2 Column Extraction

To minimize the impact of irrelevant data, it is essential to extract the relevant columns and utilize them for LLMs to conduct reasoning (Zhang et al., 2024a). As shown in Figure 3, given the question, table header, and top K rows as input context, we prompt LLMs to follow the examples and extract the related columns with additional explanation. Based on the LLMs filter columns, we extract them from T and obtain T_{col}.

3.3 Data Redefinition with Code

Specifically, to maintain the flexibility of table data, the string type is adopted for the table data especially collected from the web. Besides, the format of data is not always consistent which causes a great challenge for SQL generation. For example, as shown in Figure 2, the values of column *year* in T are not rigorous, which conclude three different formats with string type: '1931', 'spring 1932', and '1933/34'. Therefore, it's necessary to redefine data, including unifying the format, defining the data type, and extracting additional information (the detailed discussion about data redefinition is presented in Section 5.1). The related prompt is shown in Figure 4, given the current data format code, we acquire LLMs to generate Python code

```
### Task description: Based on the questions and current data format
with python, define a new function to supplement subsequent processing
operations with python according to the following steps:
- step 1 (optional): standardize the data format for each column.
- step 2: set the data type for each column (the type must be one of int,
float, string and datetime).
- step 3 (optional): extract the useful information by adding new
columns if needed.
- Note: you just need to output the code, don't answer the questions and
don't redefine the df.
Here are some examples to help you understand this task.
###Example1:
Question: how many people stayed at least 3 years in office?
### Current code of data format:
data = {'took_office': ['march 4, 1803', 'march 4, 1809', 'march 4,
1815'], 'left_office': ['march 3, 1809', 'march 3, 1815', 'april 18,
1816']}
df = pd.DataFrame(data)
### Answer:
def function(df):
        # step 1: standardize the data format: the question only need the
years
        df['took_office'] = df['took_office'].map(lambda x: x.split(',
')[-1])
        df['left_office'] = df['left_office'].map(lambda x: x.split(',
')[-1])
        # step 2: set the data type: set the year to int type
        df['took_office'] = df['took_office'].astype(int)
        df['left_office'] = df['left_office'].astype(int)
        # step 3: extract the information
        df['stayed_years_in_office'] = df['left_office'] -
df['took_office']
        return df
# call the function, don't change df anymore
df = function(df)

###Example2:
Question: what's the number of parishes founded in the 1800s?
### Current code of data format:
data = {'parish': ['st mary', 'the immaculate conception', 'st james the
less'], 'founded': ['1852', '1854', '1828']}
df = pd.DataFrame(data)
### Answer:
def function(df):
        # step 1: all data format are the same, there is no need to
change
        # step 2: set the data type
        df['parish'] = df['parish'].astype(str)
        df['founded'] = df['founded'].astype(int)
        # step 3: there is no data need to extract
        return df
# call the function, don't change df anymore
df = function(df)

### Problem to be solved:
```

Figure 4: The prompt for LLMs to perform data redefinition with code.

with the following steps. Firstly, if there is column data with inconsistent formats, standardize it to form a unified format. Besides, the data type for each column must be set to make it suitable for performing mathematical logical operations, such as defining the column to the integer type. We require that the data type must be one of integer, float, string, and datetime types. Finally, additional information could be extracted by adding new columns. Among them, annotations are added for each step to benefit LLMs to follow the above steps more effectively. Besides, steps 1 and 3 are optional, depending on the specific cases. For instance, the formats of each column in Figure 4 example 2 are consistent, there is no need to extract extra information. As a result, steps 1 and 3 are unnecessary.

3.4 SQL Generation

The table data after redefining is unified and meets the requirements for SQL execution. In this step, we make use of few-shot learning to prompt LLMs

```
### Task description:
You are a data scientist specializing in text-to-SQL tasks. Given the
Question and Database schema, you should decompose the question into
multiple steps and generate their SQL. Please add the `` for every
column you select in the SQL. Note: Just output the step and sql. Here
are some examples to help you understand this task.

### Example1:
Database title: The table about 2000 Olympic Games.
Database scheme:
CREATE TABLE test(
"race_name" TEXT; VALUES: [vuelta a guatemala, vuelta a colombia],
"winner_country" TEXT; VALUES: [usa, aus])
### Question: who won more races, the usa or australia?
### Sub-step and their SQL:
step1: obtain the count of usa win.
SQL1: SELECT COUNT(`race_name`) FROM test WHERE `winner_country` = 'usa'
step2: obtain the count of aus win.
SQL2: SELECT COUNT(`race_name`) FROM test WHERE `winner_country` = 'aus'
step3: Compare the count of usa and aus, and select the most
SQL3: SELECT `winner_country`, won_count FROM (SELECT `winner_country`,
COUNT(`race_name`) AS won_count FROM test WHERE `winner_country` in
('usa', 'aus')) ORDER BY won_count DESC LIMIT 1\n\n

### Example2:
Database title: The table about members of Third Incarnation of Lachlan.
Database scheme:
CREATE TABLE test(
"row_id" INTEGER; VALUES: [0, 1],
"member" TEXT; VALUES: [john ryan, james martin],
"term" TEXT; VALUES: [1859-1864, 1864-1869])
### Question: of the members of the third incarnation of the lachlan,
who served the longest?
### Sub-step and their SQL:
step1: obtain the all `member` and sorted them based on `term` with DESC
SQL1: SELECT `member` FROM test ORDER BY `term` DESC
step2: select the longest `member` by LIMIT 1
SQL2: SELECT `member` FROM test ORDER BY `term` DESC LIMIT 1

### Problem to be solved:\n
```

Figure 5: Prompt for LLMs to generate SQL with CoT.

to perform SQL generation. Specifically, as presented in Figure 5, the essential information is provided, such as the database title, schema, and top K relevant rows. With these contexts, LLMs are prompted to decompose the question into multiple steps and generate their sub-SQL with the CoT method. We find that the CoT style is beneficial for LLMs to generate the final SQL queries, and the specific analysis is described in Section 5.1.

3.5 Answer Generation

After executing the SQL query obtained from the previous step, the most relevant information is gained from the table. As presented in Figure 6, during this step, based on the results from executing the SQL query and the question, we utilize LLMs to reason with the additional explanation and finally make a conclusion. Consequently, we can conveniently extract the results from the conclusions as final answers. This approach helps LLMs concentrate on the relevant parts to understand the context and answer the questions.

4 Experiments

4.1 Datasets

We design relevant prompts and utilize the powerful in-context learning ability of LLMs to directly reason on the test set. We evaluate the proposed ProgramTab on three public table

```
### Task description: Based on the table title, question and execution
result of the sql query bellow, find the answer to the given question
correctly. If there are multiple answers, please split them by ' | '.
Note: Only choose the answers from SQL Answer.

Table_title: piotr kędzia
Question: in what city did piotr's last 1st place finish occur?
SQL: select `year`, `venus` from test where `position` = '1st' order by
`year` DESC LIMIT 1
SQL Answer:
Table Schema: year | venus
Values: 2007 | bangkok, Thailand
A: The SQL answer contain the year and venus about piotr's last 1st
place, and the question ask about the city which means the venus, and
the city where Piotr's last 1st place finish occurred is Bangkok,
Thailand.
Therefore, the answer is bangkok, thailand.

Table_title: playa de oro international airport
Question: how many more passengers flew to los angeles than to saskatoon
from manzanillo airport in 2013?
SQL: select `city`, `passengers` from test where `city` in ('united
states, los angeles', 'canada, saskatoon');
SQL Answer:
Table Schema: city | passengers
Values: united states, los angeles | 14,749
canada, saskatoon | 2,282
A: The SQL answer contains the number of passengers who flew to los
angeles and saskatoon from manzanillo airport 14,749, 2,282. So, the
difference in the number of passengers between los angeles and saskatoon
is 14,749 - 2,282 = 12,467.
Therefore, the answer is 12,467.

### Problem to be solved:\n
```

Figure 6: Prompt for LLMs to generate final answers.

reasoning benchmarks: TabFact (Chen et al., 2020), WikiTQ (Panupong and Percy, 2015) and HiTab (Cheng et al., 2022). Among them, TabFact is a table-based binary fact verification benchmark. Given a statement, we need to ascertain the truthfulness of it based on the table. We report the accuracy of the test set, which contains 2,024 statements and 298 tables. Besides, WikiTQ is one of the most commonly used and highly complex datasets, collected and annotated based on Wikipedia tables. The WikiTQ comprises 4,344 question-answer pairs in the test set. HiTab is the dataset that contains hierarchical tables with complex hierarchical indexing.

4.2 Baselines

We divide the baselines into two categories: (1) approaches that spend additional computing resources to train proprietary models with custom training data, such as TaPas (Herzig et al., 2020), GraPPa (Yu et al., 2021), TAPEX (Liu et al., 2022), PASTA (Gu et al., 2022), TaCube (Zhou et al., 2022), OmniTab (Jiang et al., 2022), ITR (Lin et al., 2023) and CABINET (Patnaik et al., 2024). (2) without training, approaches that design few shot prompts and employ the in-context ability of LLMs, such as TableCoT (Tai et al., 2023), ReAcTable (Zhang et al., 2024b), Binder (Cheng et al., 2023), Dater (Ye et al., 2023), Chain-of-Table (Wang et al., 2024), Alter (Zhang et al., 2024a), E^5 (Zhang et al., 2024c), NormTab (Nahid and Rafiei, 2024a), TabSQLify (Nahid and Rafiei,

Methods	Backbone	Accuracy
	Previous Work with Training	
TaPas	BERT	83.9
Tapex	BART	86.7
PASTA	DeBERTaV3	**90.8**
	Previous Work without Training	
E[5]	GPT-4	88.7
ReAcTable		73.1
TableCoT		73.1
Binder		79.1
Dater		78.0
Alter	GPT-3.5-Turbo	84.3
NormTab		68.9
TabSQLify		79.5
H-STAR		85.0
ProgramTab (Ours)		**85.9**
Binder		78.1
Dater		81.6
Chain-of-Table	Llama-3.1-70B-Instruct	85.6
TabSQLify		70.7
ProgramTab (Ours)		**86.8**
Binder		84.6
Dater		80.9
Chain-of-Table	GPT-4o-mini	84.2
TabSQLify		78.7
H-STAR		89.4
ProgramTab (Ours)		**89.6**

Table 1: Accuracy of ProgramTab compared to the baselines on TabFact test set.

2024b) and H-STAR (Nikhil et al., 2024).

4.3 Implementation Details

In our settings, we conduct experiments by utilizing closed-source LLMs (GPT-3.5-Turbo and GPT-4o-mini[1]) and the open-source LLM Llama-3.1-70B-Instruct[2] as the backbones. The prompt templates for each procedure are described in Section 3. Besides, the details of hyper-parameters are presented in Appendix A.2. Notably, syntax errors occasionally occurred during data redefinition and SQL generation, resulting in non-executable code. To address this issue, we adopted a retry mechanism. Specifically, when a runtime error occurred during both processes, we attempted to rerun the process, with a maximum of five attempts. If all five attempts failed, it was concluded that LLMs were unable to handle the given table, and no further steps were executed. About the evaluation metrics, we follow Nahid and Rafiei (2024b) to use the official denotation accuracy and employ the binary classification accuracy for WikiTQ and TabFact datasets evaluation respectively.

4.4 Results

As presented in Table 1 and Table 2 (the additional results on HiTab in Appendix A.4.), (1) the previous work, training with specific tasks perform well.

[1] https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[2] https://ai.meta.com/blog/meta-llama-3-1/

Methods	Backbone	Accuracy
	Previous Work with Training	
TaPas	BERT	48.8
GraPPa	RoBERTa	52.7
Tapex		57.5
TaCube		60.8
OmniTab	BART	62.8
ITR		63.4
CABINET		69.1
	Previous Work without Training	
TableCoT		48.8
Binder	Codex	61.9
ReAcTable		65.8
Dater		65.9
E[5]	GPT-4	65.5
ReAcTable		52.5
TableCoT		52.4
Binder		55.4
Dater		52.8
Alter	GPT-3.5-Turbo	67.4
TabSQLify		64.7
NormTab		61.2
H-STAR		69.6
ProgramTab (Ours)		**70.3**
Binder		50.5
Dater		43.5
Chain-of-Table	Llama3.1-70B-Instruct	62.2
TabSQLify		55.8
ProgramTab (Ours)		**75.5**
Binder		58.8
Dater		58.3
Chain-of-Table	GPT-4o-mini	55.6
TabSQLify		57.0
H-STAR		74.9
ProgramTab (Ours)		**76.0**

Table 2: Performance of ProgramTab compared to the baselines on WikiTQ test set.

Specifically, PASTA (Gu et al., 2022) achieves 90.8% accuracy on TabFact, while CABINET (Patnaik et al., 2024) obtains 69.1% on WikiTQ. Using GPT-4o-mini as the backbone, ProgramTab achieved performance comparable to PASTA on the TabFact dataset. Furthermore, on the WikiTQ dataset, ProgramTab outperformed CABINET regardless of the large model used as its backbone. Due to unnecessary additional fine-tuning, the generalization of ProgramTab is better. (2) Compared to previous work without training, ProgramTab with different LLM backbones outperforms the other baselines on all evaluation benchmarks. In addition, our framework with GPT-4o-mini achieves better performance compared to E^5 with GPT-4. (3) With stronger coding and reasoning abilities, ProgramTab with Llama-3.1-70B-Instruct and GPT-4o-mini achieve better performance.

5 Analysis

5.1 Ablation Study Results

To evaluate the effectiveness of each procedure in the ProgramTab framework, we pay attention to two important steps: **d**ata **r**edefinition (DR) and **S**QL **g**eneration (SG). Specifically, we remove the

Methods	TabFact	WikiTQ
Binder	79.1	55.4
Dater	78.0	52.8
TabSQLify	79.5	64.7
ProgramTab	**85.9**	**70.3**
w/o DR	81.6 (↓ 4.3)	59.4 (↓ 10.9)
w/o CoT SG	84.1 (↓ 1.8)	65.0 (↓ 5.3)

Table 3: Ablation results of GPT-3.5-Turbo-based ProgramTab with and without data redefinition and CoT SQL generation.

Methods	Backbone	TabFact	WikiTQ
Binder	Codex	-	29.6
Chain-of-Table	GPT-3.5-Turbo-16k-0613	-	44.8
Binder	GPT-3.5-Turbo	-	0.0
Dater	GPT-3.5-Turbo	-	34.6
TableCoT	GPT-3.5-Turbo	55.5	35.1
TabSQLify	GPT-3.5-Turbo	72.8	52.3
ProgramTab	GPT-3.5-Turbo	**86.6**	**68.0**

Table 4: Performance of ProgramTab and strong baselines on large tables from TabFact and WikiTQ.

Methods	Datasets	Cut-off(%)			
		0-10%	10-25%	25-50%	50%+
TabSQLify	WikiTQ	64.6	**60.6**	**66.3**	56.2
ProgramTab	WikiTQ	**70.8**	**62.4**	62.6	**68.0**
TabSQLify	TabFact	79.1	80.8	70.0	72.8
ProgramTab	TabFact	**89.0**	**86.5**	**77.5**	**86.4**

Table 5: Performance of ProgramTab on the different cutoff thresholds categories.

DR procedure described in Section 3.3 and keep the other steps unchanged. The result in Table 3 shows that without the DR step to preprocess the tabular data, it will require SQL to preprocess the data and extract the relevant information, which increases the complexity of generating SQL for LLMs. Therefore, the performance significantly decreases on both datasets, especially on WikiTQ which is more complex. This conclusion is also verified by Wang et al. (2024). Besides, we replace the procedure described in Section 3.4 with the SQL generation without CoT (denotes as w/o CoT SG). The special prompt is shown in Appendix A.1. Table 3 presents that the performance of ProgramTab w/o SG CoT drops when discarding question decomposition. It verifies that compared with direct SQL generation, decomposing the questions into multiple steps and generating their sub-SQL is effective in reducing the difficulty of SQL generation.

5.2 Performance Analysis under Large Tables

As described in Section 1, Chen (2023) and Ye et al. (2023) have presented that LLMs suffer from significant performance degeneration when dealing with large tables. To evaluate the effectiveness of ProgramTab, we extract the large tables from WikiTQ and TabFact datasets. Specifically, we define the large tables for WikiTQ when the token counts are larger than 4000 because 4000 tokens are the maximum token limitation for GPT-3.5-Turbo. Besides, We follow Nahid and Rafiei (2024b) to choose 1200 tokens for TabFact because the tables almost contain few data. We then compare ProgramTab with Binder, Dater, Chain-of-Table, TableCoT, and TabSQLify. As shown in Table 4, we observe that all strong baselines suffer from a significant decline in performance on two datasets. For example, Binder with Codex merely achieves 29.6% accuracy on the WikiTQ dataset and even can't be applied when utilizing GPT-3.5-Turbo as the backbone. Besides, TabSQLify obtains suboptimal performance thanks to its effective extraction of columns and rows employing the text-to-SQL method. In contrast, ProgramTab significantly outperforms all baselines and even improves compared with performance on the full TabFact dataset. It could be clarified that the row retrieval and column extraction procedures in our framework are effective in providing the relevant rows as the context, which is beneficial for SQL generation to extract the final information from the large tables.

5.3 Robustness Analysis

Following Nahid and Rafiei (2024b), we verify the robustness of ProgramTab based on the different cutoff thresholds. Specifically, the cutoff thresholds are established to discard tabular tokens exceeding these limits. For example, if the original table has 800 tokens and the maximum threshold is set to 600, it means that 200 tokens of the original table are truncated, and the percentage is 200/800 = 25.0%. In our experiment, we set the cutoff threshold at 2000 and 600 for WikiTQ and TabFact respectively. Table 5 shows four categories based on the above thresholds and presents that ProgramTab with GPT-3.5-Turbo outperforms TabSQLify except on the 25%-50% cutoff on WikiTQ. The results further demonstrate that the ProgramTab can extract the relevant information under limited token boundary conditions and is not sensitive to input length limitations for LLMs.

Methods	# of samples / step	Total # of samples
Binder	Neural SQL: 50	50
Dater	Decompose Table: 40 Generate Cloze: 20 Generate SQL: 20 Query: 20	100
Chain-of-Table	Dynamic Plan ≤ 5 Generate Args ≤ 19 Query: 1	≤ 25
TabSQLify	Decompose Table: 1 Query: 1	2
ProgramTab	Column Extraction: 1 Data Redefinition with Code: 1 Generate SQL: 1 Query: 1	4

Table 6: The number of samples generated by different methods adopting LLMs.

5.4 Efficiency Analysis

Following Wang et al. (2024), we analyze the efficiency of ProgramTab by evaluating the number of samples generated by LLMs. For each reasoning step, compared to the approaches that apply the self-consistency (Binder and Dater) strategy to generate multiple samples or adopt the iterative sample creation process (Chain-of-Table), ProgramTab adopts a greedy search strategy to produce a single response. Specifically, Table 6 shows the number of samples generated by LLMs for a single question in different methods on the WikiTQ dataset. We can find that LLMs are required to generate multiple samples for Binder and Dater, while Chain-of-Table adopts a more efficient approach to reduce the number of samples. TabSQLify achieves the minimum number of samples. Our approach adopts a greedy search strategy to obtain one response for each step, for a total of only four samples. Consequently, ProgramTab efficiently reduces computation time and resource costs and performs better.

5.5 Error Analysis

To systemically analyze the shortcomings of programTab with GPT-3.5-Turbo, we select two test sets (i.e., TabFact, and WikiTQ), and randomly choose 100 error samples from each dataset. Then, we manually examine these failures and they are classified into four error categories: 1) Missing Columns Error: LLMs don't select the relevant columns. 2) SQL Error: the generated SQL queries incorrectly filter the relevant information or contain syntax rule errors. 3) Code Error: the generated Python codes fail to unify the format and type of data, or introduce irrelevant information. 4) Reasoning Error: LLMs fail to generate the correct answers given the extracted relevant information. As shown in Figure 7, we can observe that the missing column and reasoning errors respectively account for a small portion of TabFact and WikiTQ. The main source of errors focuses on the code and SQL errors, especially on the WikiTQ. We replaced GPT-3.5-Turbo with GPT-4o-mini for code and SQL generation, and found that GPT-4o-mini effectively avoids the errors encountered with GPT-3.5-Tubo. The performance of these two LLMs in Table 1 and 2 can also be verified. Consequently, enhancing the capacity of code generation is effective in improving the performance further. We provide two suggestions for further exploration: (1) applying some training strategies, such as pre-training, supervised fine-tuning, reinforcement learning from human feedback, and so on. (2) Based on the specific questions, dynamically selecting the few-shot examples by employing the retrieval-augmented generation approach is also effective in decreasing the error ratio of the above problems.

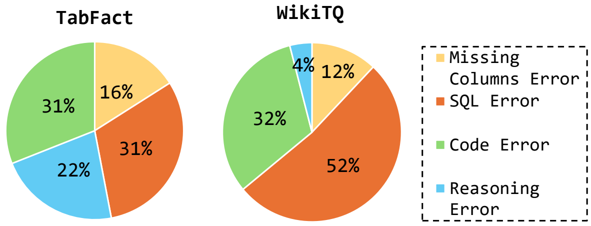

Figure 7: Statistic of different error types on TabFact and WikiTQ datasets.

6 Conclusion

In this paper, we illustrate the limitations of current table-based reasoning with LLMs approaches, including suffering from significant performance degradation when faced with large tables, and the inconsistent table data structure increases the difficulty of SQL generation. Consequently, we propose the ProgramTab framework, which sufficiently implements the strong in-context learning ability of LLMs to perform tabular data preprocessing with Python code and key information extraction with SQL generation. It achieves the best performance compared with the baselines and is not limited by the input length of table data. Hoping this flexible table-based reasoning framework can shed new light on the understanding of prompting LLMs for table understanding.

7 Limitations

In this section, we present several of the limitations of our approach - ProgramTab. Firstly, the data redefinition with code can preprocess the table data well, but more preprocessing for more complex table structures should be explored further. What's more, how to perform row retrieval more efficiently from tables with large amounts of rows is another optimization direction.

References

Rami Aly, Zhijiang Guo, Michael Schlichtkrull, James Thorne, Andreas Vlachos, Christos Christodoulopoulos, Oana Cocarascu, and Arpit Mittal. 2021. Feverous: Fact extraction and verification over unstructured and structured information. Preprint arXiv:2106.05707.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and et al. 2020. Language models are few-shot learners. In Advances in neural information processing systems, pages 33:1877–1901.

Wenhu Chen. 2023. Large language models are few(1)-shot table reasoners. In Findings of the Association for Computational Linguistics: EACL 2023, pages 1120–1130. Association for Computational Linguistics.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. Preprint arXiv:2211.12588.

Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2020. Tabfact: A large-scale dataset for table-based fact verification. In International Conference on Learning Representations (ICLR).

Zhoujun Cheng, Haoyu Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. 2022. HiTab: A hierarchical table dataset for question answering and natural language generation. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1094–1110. Association for Computational Linguistics.

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 International Conference on Learning Representations (ICLR).

Minseok Cho, Gyeongbok Lee, and Seung won Hwang. 2019. Explanatory and actionable debugging for machine learning: A tableqa demonstration. In Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, page 1333–1336. ACM.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 4171–4186. Association for Computational Linguistics.

Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. 2023. Complexity-based prompting for multi-step reasoning. In Advances in International Conference on Learning Representations.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In In International Conference on Machine Learning, pages 10764–10799.

Zihui Gu, Ju Fan, Nan Tang, Preslav Nakov, Xiaoman Zhao, and Xiaoyong Du. 2022. PASTA: Table-operations aware fact verification via sentence-table cloze pre-training. In Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 4971–4983. Association for Computational Linguistics.

Pengcheng He, Jianfeng Gao, and Weizhu Chen. 2021. DeBERTav3: Improving deBERTa using ELECTRA-style pre-training with gradient-disentangled embedding sharing. Preprint arXiv:2111.09543.

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. Association for Computational Linguistics.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego De, Las Casas, Lisa Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, GeorgeVanDen Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training compute-optimal large language models. Preprint arXiv:2203.15556.

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, pages 932–942. Association for Computational Linguistics.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 7871–7880. Association for Computational Linguistics.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. Preprint arXiv:2308.03281.

Weizhe Lin, Rexhina Blloshmi, Bill Byrne, Adria de Gispert, and Gonzalo Iglesias. 2023. An inner table retriever for robust table question answering. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 9909–9926. Association for Computational Linguistics.

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 In International Conference on Learning Representations.

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

Md Mahadi Hasan Nahid and Davood Rafiei. 2024b. TabSQLify: Enhancing reasoning capabilities of LLMs through table decomposition. In 2024 Annual Conference of the North American Chapter of the Association for Computational Linguistics.

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).

Abhyankar Nikhil, Gupta Vivek, Roth Dan, and Reddy Chandan K. 2024. H-star: Llm-driven hybrid sql-text adaptive reasoning on tables. Preprint arXiv:2407.05952.

OpenAI. 2022. Gpt-3.5-turbo. Technical Report.

OpenAI. 2023. Gpt-4. Technical Report.

Pasupat Panupong and Liang Percy. 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. Association for Computational Linguistics.

Sohan Patnaik, Heril Changwal, Milan Aggarwal, Sumit Bhatia, Yaman Kumar, and Balaji Krishnamurthy. 2024. Cabinet: Content relevance-based noise reduction for table question answering. In The Twelfth International Conference on Learning Representations.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. Preprint arXiv:2204.00498.

Chang-Yu Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. Exploring chain of thought style prompting for text-to-SQL. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 5376–5393. Association for Computational Linguistics.

Touvron, Hugo, Lavril, Thibaut, Izacard, Gautier, Martinet, Xavier, Lachaux, Marie-Anne, Lacroix, Timoth'ee, Rozi'ere, Baptiste, Goyal, Naman, Hambro, Eric, Azhar, Faisal, Rodriguez, Aurelien, Joulin, Armand, Grave, Edouard, Lample, and Guillaume. 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 International Conference on Learning Representations (ICLR).

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. In Proceedings of the 36th International Conference on Neural Information Processing Systems, pages 24824–24837.

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: Decompose evidence and questions for table-based reasoning. In Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 174–184.

Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, bailin wang, Yi Chern Tan, Xinyi Yang, Dragomir Radev, richard socher, and Caiming Xiong. 2021. Gra{pp}a: Grammar-augmented pre-training for table semantic parsing. In International Conference on Learning Representations.

Han Zhang, Yuheng Ma, and Hanfang Yang. 2024a. Alter: Augmentation for large-table-based reasoning. Preprint arXiv:2407.03061.

Hongzhi Zhang, Yingyao Wang, Sirui Wang, Xuezhi Cao, Fuzheng Zhang, and Zhongyuan Wang. 2020. Table fact verification with structure-aware transformer. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1624–1629. Association for Computational Linguistics.

Yunjia Zhang, Jordan Henkel, Avrilia Floratou, and Joyce Cahoon. 2024b. Reactable: Enhancing react for table question answering. In Proceedings of the VLDB Endowment 17(8), pages 1981–1994.

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. Association for Computational Linguistics.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2023. Automatic chain of thought prompting in large language models. In Advances in International Conference on Learning Representations.

Fan Zhou, Mengkang Hu, Haoyu Dong, Zhoujun Cheng, Fan Cheng, Shi Han, and Dongmei Zhang. 2022. TaCube: Pre-computing data cubes for answering numerical-reasoning questions over tabular data. In Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 2278–2291. Association for Computational Linguistics.

A Appendix

A.1 Prompts for SQL Generation without CoT

```
### Task description: You are a data scientist specializing
in text-to-SQL tasks. You should write a valid SQLite to
solve the following question based on the database scheme and
hint. Please add the `` for every column you select in the
SQL. Note: Just output the sql. Here are some examples to
help you understand this task.

### Example1:
Database title: The table about 2000 Olympic Games.
Database scheme:\nCREATE TABLE test(\n\"race_name\" TEXT;
VALUES: [vuelta a guatemala, vuelta a
colombia],\n\"winner_country\" TEXT; VALUES: [usa, aus])\n\n
### Question:\nwho won more races, the usa or australia?\n\n
### SQL: SELECT `winner_country`, won_count FROM (SELECT
`winner_country`, COUNT(`race_name`) AS won_count FROM test
WHERE `winner_country` in ('usa', 'aus')) ORDER BY won_count
DESC LIMIT 1\n\n

### Example2:
Database title: The table about members of Third Incarnation
of Lachlan.
Database scheme:\nCREATE TABLE test(\n\"row_id\" INTEGER;
VALUES: [0, 1],\n\"member\" TEXT; VALUES: [john ryan, james
martin],\n\"term\" TEXT; VALUES: [1859-1864, 1864-1869])\n\n
### Question:\nof the members of the third incarnation of the
lachlan, who served the longest?\n\n
### SQL: SELECT `member` FROM test ORDER BY `term` DESC LIMIT
1\n\n

### Example3:
Database title: The table about different season of giant
slalom and super g.
Database scheme:\nCREATE TABLE test(\n\"season\" TEXT; VALUES:
[1986, 1987],\n\"slalom\" TEXT; VALUES: [39,
24],\n\"giant_slalom\" TEXT; VALUES: [23, 9],\n\"super_g\"
TEXT; VALUES: [19, 18])\n\n
### Question:\nwhich super g had a slalom of less than 5 when
the giant slalom was 1?\n\n
### SQL: SELECT `super_g` FROM test WHERE `slalom` < 5 AND
`giant_slalom` = 1\n\n

### Problem to be solved:\n
```

Figure 8: Prompt for LLMs to generate SQL without CoT.

A.2 LLM Hyper-parameters

For all the procedures described in Section 3, we set the same hyper-parameters for LLMs. Specifically, the temperature is set to 0.6 while both top_p and the sample number are 1.

A.3 Table Size Reduction

We analyze the efficiency of ProgramTab in filtering irrelevant information and extracting the key tabular data from tables. To accomplish this, we count the average number of table cells that feed LLMs to generate the final answers. As presented in Figure 9, the average number of full table cells (original) is 183 and 101 respectively. There is a significant reduction after employing the TabSQLify approach. Our framework ProgramTab employs SQL generation procedure to effectively filter much irrelevant information and extract the most related data. It respectively reduces the average number of table cells to 4 and 15 for WikiTQ and TabFact datasets. These results also verify that ProgramTab can perform critical information extraction from amounts of tabular cells, and validly deal with large tables.

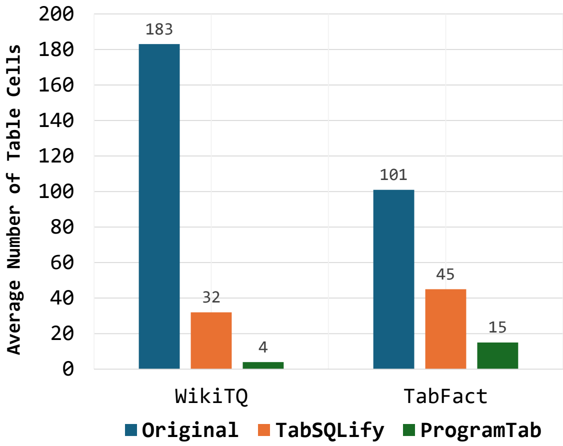

Figure 9: Comparison of the average number of table cells on two datasets.

Methods	Backbone	HiTab
ReAct (Yao et al., 2023)	GPT-4	81.87
E^5 (Zhang et al., 2024c)	GPT-4	85.08
ProgramTab	GPT-4o-mini	**83.57**

Table 7: Performance of ProgramTab on HiTab dataset.

A.4 Experiments on HiTab

To present the effectiveness of ProgramTab when applied to more complex tabular structures, we supplemented ProgramTab's experiments on the HiTab dataset, which contains hierarchical tables. To achieve this, we first reconstructed the hierarchical tables by merging certain column header information using a ":" delimiter, making them more suitable for processing by ProgramTab. The final experimental results are as follows: ProgramTab with GPT-4o-mini demonstrates promising performance, while adopting the E^5 method to process hierarchical tables yields even better performance.