ReAcTable: Enhancing ReAct for Table Question Answering

Yunjia Zhang
University of Wisconsin-Madison
yunjia@cs.wisc.edu

Jordan Henkel
Microsoft
jordan.henkel@microsoft.com

Avrilia Floratou
Microsoft
avflor@microsoft.com

Joyce Cahoon
Microsoft
jcahoon@microsoft.com

Shaleen Deep
Microsoft
shaleen.deep@microsoft.com

Jignesh M. Patel
Carnegie Mellon University
jignesh@cmu.edu

ABSTRACT

Table Question Answering (TQA) presents a substantial challenge at the intersection of natural language processing and data analytics. This task involves answering natural language (NL) questions on top of tabular data, demanding proficiency in logical reasoning, understanding of data semantics, and fundamental analytical capabilities. Due to its significance, a substantial volume of research has been dedicated to exploring a wide range of strategies aimed at tackling this challenge including approaches that leverage Large Language Models (LLMs) through in-context learning or Chain-of-Thought (CoT) prompting as well as approaches that train and fine-tune custom models.

Nonetheless, a conspicuous gap exists in the research landscape, where there is limited exploration of how innovative foundational research, which integrates incremental reasoning with external tools in the context of LLMs, as exemplified by the ReAct paradigm, could potentially bring advantages to the TQA task. In this paper, we aim to fill this gap, by introducing ReAcTable (**ReAc**t for **Table** Question Answering tasks), a framework inspired by the ReAct paradigm that is carefully enhanced to address the challenges uniquely appearing in TQA tasks such as interpreting complex data semantics, dealing with errors generated by inconsistent data and generating intricate data transformations. ReAcTable relies on external tools such as SQL and Python code executors, to progressively enhance the data by generating intermediate data representations, ultimately transforming it into a more accessible format for answering the user's questions with greater ease. Through extensive empirical evaluations using three popular TQA benchmarks, we demonstrate that ReAcTable achieves remarkable performance even when compared to fine-tuned approaches. In particular, it outperforms the best prior result on the WikiTQ benchmark by 2.1%, achieving an accuracy of 68.0% without requiring training a new model or fine-tuning.

PVLDB Reference Format:
Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. ReAcTable: Enhancing ReAct for Table Question Answering. PVLDB, 17(8): 1981 - 1994, 2024.
doi:10.14778/3659437.3659452

Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659452

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yunjiazhang/ReAcTable.git.

1 INTRODUCTION

Table question answering (TQA) [15] is a subfield of natural language processing (NLP) and information retrieval that focuses on answering natural language (NL) questions over tabular data such as Wikipedia tables, spreadsheets or relational tables. It constitutes a complex task that demands a fusion of contextual understanding, logical reasoning and analytical skills. TQA allows users without expertise in querying languages and data analytics to interact with their data using plain language and gain valuable insights. It is a vital tool that can enhance data accessibility, usability, and decision support across various domains, ultimately leading to more efficient and informed decision-making processes.

The significance of Table Question Answering (TQA) has spurred extensive research efforts, leading to the development of various strategies falling into two primary categories. In the first category, approaches like Tapas [11], Tapex [25], Tacube [66], and OmniTab [14] involve training or fine-tuning specialized models tailored specifically for the TQA task. The second category capitalizes on recent advancements in Large Language Models (LLMs). Works such as [5, 29, 56] within this category utilize LLMs to generate code capable of manipulating tabular data. However, in the context of industrial applications, the adoption of models that necessitate fine-tuning or training can introduce additional challenges. For instance, the process of training or fine-tuning a model can be resource-intensive, and collecting the required training data may pose challenges due to privacy considerations [63]. Moreover, incorporating models with distinct architectures can raise deployment complexities compared to utilizing a universal foundation model. Such specialized models can also prove challenging to maintain, especially as workloads or customer requirements evolve. Therefore, due to these practical constraints, we directly build on existing LLMs, focusing on the design of a high-performing, easy-to-implement, and easy-to-maintain TQA framework.

The emergence of Chain-of-Thought (CoT) prompting, which encourages a model to engage in step-by-step reasoning, has brought about a significant transformation in the utilization of Large Language Models (LLMs) for intricate multi-step tasks. Expanding the CoT ideas, the ReAct paradigm [55] has been introduced, enabling interactions between the model and external tools in an interleaved manner. This allows for greater synergy between reasoning and acting and facilitates real-time guidance and corrections during

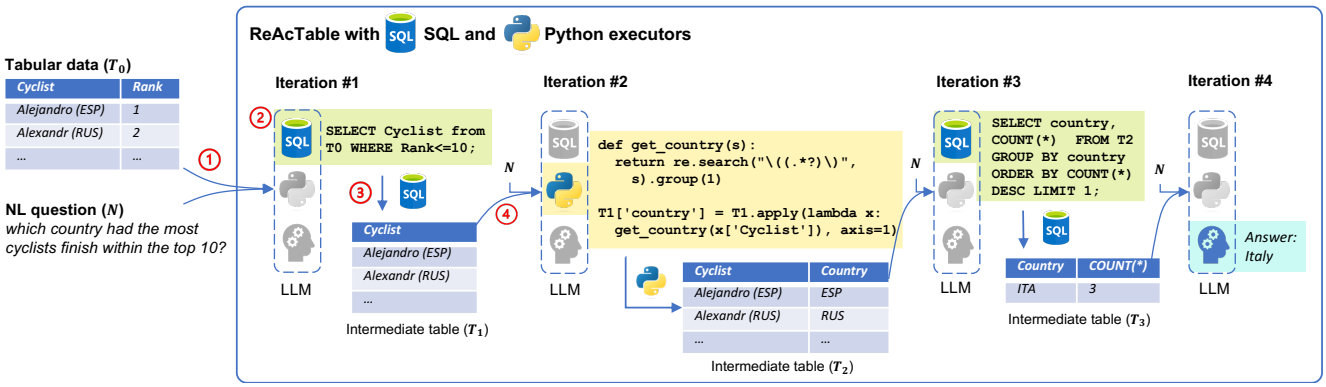


Figure 1: Overview of the ReAcTable framework with SQL and Python code executors.

task execution. These innovative strategies aim to address the limitations of traditional few-shot prompting methods [1]. Despite the promising results demonstrated by combining reasoning with external tools, to the best of our knowledge, the ReAct paradigm has not yet been applied to the TQA task.

In this paper, we bridge the gap by investigating how the principles behind the ReAct framework, i.e. CoT and availability of external tools, can be applied to the TQA task. Beyond the anticipated difficulty of accurately comprehending the user's natural language query, the TQA task poses a series of distinct challenges, including: (i) interpreting potentially intricate data semantics, (ii) the presence of noisy or inconsistent data, and (iii) the necessity for complex data transformations to derive the correct response. Let us take the example table in Figure 1 drawn from the WikiTQ [32] dataset. In this example, the user wants to know *"Which country had the most cyclists finish in the top-10?"*. As we see in the corresponding table, there is no column containing explicitly information about countries. Instead, the `Cyclist` column contains values that encompass both the cyclist's name and an accompanying abbreviation in parentheses signifying their affiliated country, constituting a column with data values that encapsulate complex semantics, all condensed into a single string entry. For a developer to answer this question, they would have to first understand where the country information is represented in the data, come up with a transformation to extract it from `Cyclist` column and then write a query/program to filter the data by rank and then group them by the corresponding country by applying the transformation on each row of the table. From the above example, the complexity inherent in both the questions (involving multiple intricate reasoning steps) and the tables (containing complex semantic representations), renders the table question answering problem a highly challenging task.

To tackle the highlighted complexities, we present ReAcTable – a novel TQA framework that draws inspiration from the ReAct framework. ReAcTable is designed to simplify the question-answering process through a methodical approach that includes step-by-step reasoning, executing code via external tools, and most importantly, generating intermediate tables. This innovative strategy allows ReAcTable to decompose complex reasoning tasks into smaller, manageable segments, iteratively refining the data through the creation of intermediate data representations. Our workflow facilitates a smoother transformation of data into a format that is more readily answerable, enhancing the accessibility and accuracy of responses to user queries.

Figure 1 presents an overview of the ReAcTable framework. The framework employs two external tools: one for executing SQL queries and another for running Python code. Although the framework is adaptable to a range of code execution tools capable of manipulating tabular data, we have selected these two, as SQL serves as the standard language for querying structured data, and Python stands as the predominant choice among data scientists for data cleaning and transformation tasks [34]. The input for the ReAcTable framework comprises two key elements: (i) a tabular data set (T_0) and (ii) a natural language question (N) regarding T_0. ReAcTable automatically forms a prompt that is sent to a LLM (see Section 3.2 for details on the prompt). The LLM is equipped with three possible actions: (i) generating SQL code, (ii) generating Python code, or (iii) directly providing an answer. If the LLM opts to generate code, ReAcTable automatically uses the appropriate code executor to process the code and generate an intermediate table (e.g., T_1 in Figure 1). The process is repeated until the LLM produces a direct answer to the initial question (more details can be found in Section 3). It is crucial to note that this iterative process progressively refines the data, generating increasingly reliable context in the form of intermediate tables for subsequent reasoning iterations. As an example, by the end of the second iteration, the intermediate table contains a distinct `Country` column extracted from the original data, that allows the next iteration to easily group the data by country. Another intriguing observation is that the LLM employed SQL code for querying tabular data while opting for the Python executor when handling string manipulation tasks, a pattern that closely aligns with human preferences. Finally, as we discuss in Section 3.4, ReAcTable additionally leverages majority voting to improve the predictive quality of LLMs and also handle errors/code execution exceptions that might result from noisy/inconsistent data values.

The key contributions of the paper are:

- We introduce a novel framework, ReAcTable, designed for Table Question Answering (TQA) tasks, taking inspiration from

prior work on CoT [51] and harnessing external tools such as ReAct [55]. ReAcTable makes effective use of Large Language Models (LLMs) to break down the TQA problem into multiple steps, generating logical operations in the form of code for tabular data processing as needed. Notably, ReAcTable dynamically integrates intermediate code execution results into the LLM's input for the TQA task, further allowing these results to enhance subsequent reasoning steps.

- We perform a thorough empirical evaluation of the ReAcTable framework using three popular TQA benchmarks (WikiTQ [32], TabFact [4], and FetaQA [28] and compare it with various state-of-the-art approaches (LLM and non-LLM-based). We find that ReAcTable achieves remarkable performance across all the data sets. In particular, the framework achieves a test accuracy of 68.0% on WikiTQ [32], the most commonly used benchmark for TQA tasks, outperforming the state-of-the-art approaches by 2.1%, even when comparing with approaches that require fine-tuning.
- We delve deep into the behavior of ReAcTable and analyze the contributions of each component (intermediate tables, availability of code executors, iterative processing) using an ablation study. We empirically demonstrate that the most significant improvement comes from the iterative generation of intermediate tables by the two code executors which aligns with our intuition that progressive refinement of the data can improve the LLMs predictive capabilities. We also carefully examine the effects of using different LLMs and majority voting mechanisms. More detailed results and analysis are shown in Section 4.3.

2 BACKGROUND AND RELATED WORK

In the context of question-answering tasks, LLMs have become pivotal due to their ability to generate coherent and contextually relevant responses to users' questions. Despite advancements in LLMs, "out-of-the-box" LLMs still struggle with complex questions [5, 51, 55]. To address these limitations, various methodologies have been proposed to enhance the performance of "out-of-the-box" LLMs. Since ReAcTable is built upon these foundational works, we provide related background and terminology in this section.

2.1 Large Language Models

Large Language Models (LLMs) represent a breakthrough in natural language processing, transforming the landscape of human-computer interaction and information processing. These models, often based on transformer architectures [50], such as GPT-3 [1], Codex [2], T5 [37], and Code Llama [39], are pre-trained on vast amounts of text data. By capturing context, these LLMs perform well in generating coherent and contextually relevant text, exhibiting capabilities ranging from text completion and language translation to question answering and code generation. The underlying mechanism involves attention mechanisms that allow models to weigh the significance of different words within a given context[50]. LLMs have demonstrated remarkable performance across numerous tasks, including machine translation [1, 67], question answering [1, 38], schema matching [63], and database tuning [19, 47].

Few-shot prompting plays a pivotal role in harnessing the power of LLMs for a wide range of natural language understanding and generation tasks. By providing LLMs with a limited set of examples and a carefully crafted prompt, researchers have demonstrated their ability to generalize knowledge and perform specific tasks with minimal supervision [1]. The effectiveness of few-shot prompting is also related to prompt engineering and example selection strategies. Prompt tuning techniques, such as gradient-based optimization of prompts [21, 62], enable fine-grained control over model behavior by automatically tuning the prompt for a given task. These methods collectively enable practitioners to tailor LLMs to diverse tasks and domains, effectively leveraging the few-shot capabilities of these models to tackle real-world language understanding and generation challenges.

2.2 Majority Voting Mechanisms

LLMs may produce responses that are uncertain or influenced by biases. Majority voting mechanisms in LLMs serve as a widely adopted approach to improve the response quality and mitigate potential biases. With majority voting mechanisms, multiple responses are sampled from the LLM's output distribution and one of the responses is selected based on some criteria. One common majority voting method is simple majority voting, which simply selects the response that occurs most frequently among the multiple generated outputs. Other criteria can be applied in majority voting methods, tailored to specific applications. For example, in code generation tasks, majority voting methods can also consider the execution results of generated code as a basis for selection [18, 29]. The choice of voting method depends on the context and requirements of the application.

2.3 Chain-of-Thought and ReAct Framework

In addition to simply prompting the LLMs, multiple prompting paradigms are designed to enhance the quality of reasoning and responses generated by these models. One specific example is Chain-of-Thought (CoT) prompting [51], which goes beyond traditional prompting and introduces a structured approach to guide the model's reasoning process. In the CoT paradigm, the reasoning process is organized into multiple intermediate steps, allowing the model to solve one simpler subproblem at a time and progressively build a coherent response. This structured approach helps LLMs tackle complex tasks effectively.

The ReAct framework [55] has expanded upon the foundational ideas of the CoT paradigm, introducing the concept that interactions with external components can substantially enhance the capabilities of LLMs. By enabling LLMs to engage with external components, ReAct broadens the scope of tasks and applications that these models can handle, making them more versatile and adaptable. A similar idea has also been introduced into modern LLMs, including GPT-4 [30], enabling them to provide users with better results via external plugins.

2.4 Natural Language to Code

Natural language to code refers to the challenging task of automatically translating human-readable natural language descriptions into executable programming code. Traditionally, this problem has been tackled using rule-based approaches, which involve designing

handcrafted grammar and syntactic rules to parse and interpret natural language queries [10, 13, 35].

While these rule-based approaches have shown promise, they often struggle with handling the nuances and variations of natural language. In recent years, there has been a significant shift towards model-based methods for natural language to code conversion. These methods employ machine learning models, often based on neural networks, to learn the mapping between natural language and code from large data sets [7, 8, 43].

Furthermore, the advent of large language models (LLMs) has opened up new possibilities for natural language to code tasks. Researchers have explored the use of LLMs with few-shot prompting to generate code snippets from natural language descriptions, effectively treating the model as a powerful code generator [1, 2]. To ensure the generated code is free of syntax errors, additional techniques such as controlled decoding and post-processing checks are employed [16, 18, 29, 40, 41]. These advancements highlight the potential of LLMs in automating the natural language to code conversion process, making it more robust for various applications.

The availability of high-quality natural language to code data sets, including Spider [58], CoSQL [57], SParC [59], WikiSQL [64], and BIRD [20], has been instrumental in advancing research in the domain of natural language to code conversion. These datasets provide rich and diverse examples of natural language queries paired with corresponding code snippets, covering a wide range of programming languages and database domains.

2.5 Table Question Answering

Table Question Answering (TQA) is a task that resides at the intersection of natural language processing and data analytics. In this paper, our focus is on solving the single-table TQA problem. Specifically, when presented with a natural language question N about a relational table T_0, our objective is to provide the correct answer to N. The output answer may take the form of a tuple list or a sentence in natural language. For instance, the WikiTQ [32] benchmark presents answers in tuple lists (e.g., "2001|2002|2003"), while the FeTaQA [28] benchmarks uses natural language as the answer format (e.g., "Harvey beat Royds by 1,463 votes"). We assume that both schema-level information and data content of the tabular data T_0 are available to answer the given question N.

3 ReAcTable: ReAct FOR TQA TASKS

ReAcTable (**ReAc**t for **Table** Question Answering tasks) is an instantiation of the ReAct framework [55], designed to tackle complex, multi-step Table Question Answering (TQA) problems (as defined in Section 2.5). By incorporating concepts from ReAct, along with majority voting mechanisms and specialized code executors, ReAcTable is able to break down complex TQA problems into smaller, simpler sequenced tasks. For each of these sequenced tasks, it employs generated code to manipulate the target table. Thus, ReAcTable significantly boosts the performance of pre-trained Large Language Models (LLMs) without additional fine-tuning.

3.1 Overview

Figure 1 provides an overview of the ReAcTable framework. The inputs to ReAcTable consist of (i) a relational data table T_0, and (ii) a natural language question N. The ultimate goal of ReAcTable is to produce the correct answer to the given question.

ReAcTable iteratively separates the complex TQA task into smaller tasks. At each iteration, the tabular data (T_0) and the natural language question (N) are first input into the LLM (① in Figure 1). The LLM can then perform one of three distinct operations (② in Figure 1): (i) generating a SQL query, (ii) generating Python code, or (iii) directly answering the question.

The ReAcTable framework is designed to be adaptable, allowing for the integration of other code executors in addition to SQL and Python. In our work, we use these two specific executors as they are commonly used by data scientists to manipulate tabular data. If the LLM generates SQL or Python code, ReAcTable activates the corresponding executor. The execution results are in an intermediate data table (③ in Figure 1). This table is derived from the initial tabular data and is tailored to address the question more directly. The intermediate table, along with the original natural language question (N), is then fed back into the LLM for subsequent iterations (④ in Figure 1). The iterative process continues until the LLM provides a direct answer to the question (instead of generating code for an executor). To enable the LLM to "learn" how to answer the question, we employ the in-context learning paradigm [1], using (static) few-shot examples in our prompts. The formulation of these prompts is elaborated in Section 3.2.

For the example question illustrated in Figure 1 ("*which country had the most cyclists finish within the top 10*"), ReAcTable employs four iterations to generate the answer. This TQA example is sourced from the WikiTQ data set [32]. In the initial iteration, ReAcTable generates SQL code to select the cyclists who finished the race within the top 10 ranks. This SQL code is further executed, producing an intermediate table T_1. Subsequently, using the intermediate table T_1 as input, ReAcTable's LLM generates Python code to extract the country code from the `Cyclist` column for each row. The python code is executed producing table T_2. For the third iteration, ReAcTable generates SQL to count how many times each country appears in the intermediate table T_2, producing table T_3. Finally, in the last iteration, ReAcTable leverages the LLM to generate an answer based on the country code found in T_3. Through these executors, ReAcTable improves the problem-solving capabilities of the LLM by effectively using intermediate tables (T_1, T_2, and T_3).

3.2 Prompting Large Language Models

Prompt template. ReAcTable uses a prompt template that is instantiated on every iteration based on the output of the LLM at the previous iteration. The very first prompt contains the original table T_0 (the schema, values and data format) and the user's NL question. It also instructs the LLM to think step-by-step (CoT) and utilize the two available code executors as needed. The top part of Figure 2 reflects this prompt. Additionally, it contains few-shot demonstrations (not included in the figure because of space requirements) whose format we describe at the end of this section.

After each iteration, ReAcTable would have generated either the answer to the question or an intermediate table. In case an intermediate table is generated, it is leveraged to instantiate the prompt template at the follow-up iteration. Specifically, in the ongoing iteration, ReAcTable enhances the prompt used in the previous round

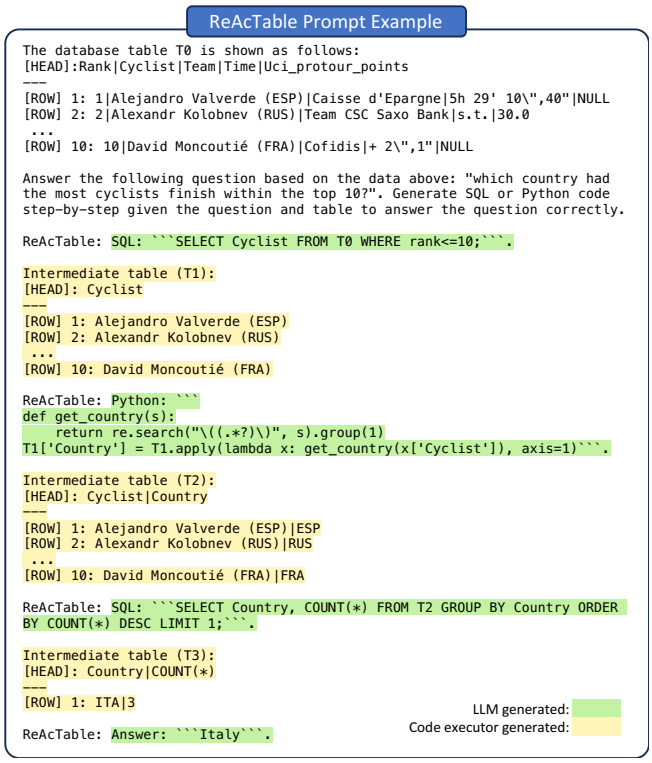


Figure 2: ReAcTable prompt after the 4^{th} iteration for our running example.

by incorporating the SQL or Python code generated in the preceding step, alongside the intermediate table formed by executing this code. This updated prompt functions as the input for the present iteration, guaranteeing that the LLM possesses a comprehensive view of the related context at the current stage.

Figure 2 displays the prompt created by ReAcTable at the end of the fourth iteration shown in Figure 1. As shown in the figure this prompt contains all the information used in the very first prompt (top part of the figure) along with the LLM output (green) and intermediate table (yellow) produced at each subsequent iteration.

Note that ReAcTable is not restricted to using the "SQL-Python-SQL-Answer" pattern shown in Figure 1. It can opt for any combination of executors and the prompt template will be initiated accordingly. For straightforward questions with intuitive answers, ReAcTable may not generate any intermediate code, while for complex questions, ReAcTable may employ five or more iterations to break down the complexity into smaller sub-questions. This flexibility allows ReAcTable to adapt to questions of varying complexity.

Few-shot demonstrations. In-context learning through few-shot demonstrations is a common technique for calibrating LLMs to specific tasks [1]. By utilizing a small set of examples that closely represent the problem structure, the LLM can be adapted to the task at hand at *inference time*. In ReAcTable, we also use few-shot demonstrations to guide LLMs. These demonstrations are inserted at the beginning of the prompt template along with the original table and NL question. These examples are omitted from Figure 2 due to the space limit. We provide a brief overview of how these examples are selected in Section 4.1.

3.3 Interacting with External Code Executors

Code execution. Once the LLM used by ReAcTable generates code, the system initializes corresponding code executors to execute the code using the provided table in the current context. In the example shown in Figure 1, the table is represented as a SQLite [12] table when running SQL queries, while a Pandas DataFrame [26] is used to run Python code.

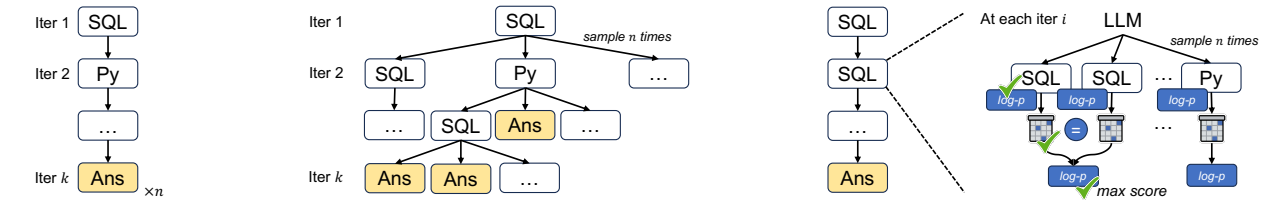
Handling exceptions. Given that the code is generated by the LLM, there remains a possibility of encountering exceptions during code execution. Effectively managing such exceptions is crucial for ensuring the "last-mile" performance of ReAcTable. Common exceptions are addressed as follows:

- **SQL exceptions**: While the tables in the prompt are numbered, there remains a possibility that the SQL predicted by the LLM should be executed on a different table, especially when randomness is amplified under a high-temperature setup (detailed in Section 3.4). To mitigate this issue, we implement a retry mechanism. This mechanism enables SQL queries on previous intermediate tables to be attempted in reverse order. If a query successfully executes on a table, the resulting table is then utilized as the next intermediate table in the ReAcTable framework.
- **Python exceptions**: In our Python running environment, we take precautions to prevent "module not found" exceptions by importing common packages, such as regular expressions (re) and date-time (datetime). In cases where a new module is required in the generated Python code, we dynamically install the necessary package during runtime and rerun the Python code. Additionally, for handling other potential Python code exceptions, we employ "try-except" to encapsulate the Python functions. By default, we return zero as the result value in such cases.
- **Other exceptions**: In the case of other exceptions, a common scenario arises where the model generates semantically correct code, yet the code cannot be successfully executed due to noisy data. In such instances, we leverage the LLM's comprehension of the noisy data and "force" it to produce an answer. To accomplish this, we append the lead word "Answer" to the end of the prompt and utilize the LLM for completion.

Taken together, the three exception handling strategies described above and the integration of voting mechanisms (detailed in the following section) help ReAcTable achieve best-in-class performance across *three distinct TQA benchmarks* in the category of techniques that do not leverage fine-tuning or training of custom models.

3.4 Voting Mechanisms

Majority voting mechanisms are frequently used to improve the predictive quality of LLMs, as they facilitate the exploration of various responses and more effectively address ambiguity [18]. Since the core of ReAcTable is an LLM, it can integrate with different voting mechanisms to increase the accuracy of its predictions. In this section, we describe three different voting strategies with which the ReAcTable framework can effectively operate: (i) simple majority



(a) Simple majority voting (b) Tree-exploration voting (c) Execution-based voting

Figure 3: Overview of voting mechanisms

voting, (ii) tree-exploration voting, and (iii) execution-based voting. Figure 3 shows an overview of these voting mechanisms.

3.4.1 Simple majority voting.
Figure 3a shows the simple majority voting mechanism used in ReAcTable. We use chains to represent the problem-solving steps in ReAcTable, with nodes representing the programs or answers predicted by the LLM. In simple majority voting, we apply a high-temperature setting to the LLM [44] and perform ReAcTable's step-by-step solving iterations for a total of n times. As a result, we obtain n predicted answers. The majority answer among these predictions is selected as ReAcTable's final prediction.

3.4.2 Tree-exploration voting.
Figure 3b demonstrates the tree-exploration voting mechanism. The idea behind tree-exploration voting is to enable LLMs to explore multiple intermediate steps before arriving at the final answer. Unlike simple majority voting, which repeats the entire chain multiple times, tree-exploration voting allows the LLM to sample n times at each prediction, resulting in a fanout of n in the reasoning tree. In this voting mechanism, ReAcTable traverses all branches of the tree until each branch reaches an answer. Finally, ReAcTable selects the majority of the answers as the final prediction. The algorithm for tree-exploration voting is shown in Algorithm 1.

Algorithm 1 ReAcTable with tree-exploration voting

Input: Table T_0, Question N, Temperature t, Sample time n
Output: Predicted answer P
$tabs = [T_0]$, $answers = []$
$treeBranches = initQueue()$ *# a queue to store branches*
$treeBranches.append(tabs)$
while $treeBranches$ is not empty:
  $tabs' = treeBranches.popLeft()$
  $prompt = preparePrompt(tabs', N_0)$
  $all\_preds = LLM(prompt, t, n)$
  for $pred$ in all_preds:
    if $pred$ is code:
      $T' = Execute(pred, tabs')$
      $treeBranches.append(tabs' + [T'])$
    else:
      $answers.append(pred)$
return $getMajority(answers)$

3.4.3 Execution-based voting.
In the two voting methods above, although they recognize the data context from earlier stages (as stated in the prompt), they do not consider the output of the code (intermediary result table) when making decisions on which prediction to take. To utilize the data context for selecting the next step, we introduce an execution-based voting mechanism [18, 29].

In execution-based voting, ReAcTable allows the LLM to sample n predictions at each reasoning step. Instead of exhaustively exploring the tree like tree-exploration voting, execution-based voting selects only one of the predictions as the intermediate step. For each of the n predictions, if the prediction is a program, ReAcTable executes the program and retrieves the resulting table. If equivalent tables are produced, the log probabilities are merged by selecting the maximum log probability (log-p in Figure 3c). Finally, ReAcTable selects the code or answer with the highest score as the next step in the reasoning chain. This decision-making process is repeated for each step in the question-answering procedure. Algorithm 2 shows a step-by-step explanation of execution-based voting.

Algorithm 2 ReAcTable with execution-based voting

Input: Table T_0, Question N, Temperature t, Sample time n
Output: Predicted answer P
$tabs = [T_0]$
while true:
  $prompt = preparePrompt(tabs, N_0)$
  $all\_preds = LLM(prompt, t, n)$
  $resultLog = initResultLog()$ *# initialize the result log*
  for $pred$, log_prob in all_preds:
    if $pred$ is code:
      $resultLog.update(Execute(pred, tabs), log\_prob)$
    else:
      $resultLog.update(pred, log\_prob)$
  # get prediction with max score
  $pred = resultLog.get\_prediction()$
  if $pred$ is code:
    $T' = Execute(pred, tabs)$ *# execute the code*
    $tabs.append(T')$ *# log the intermediate table*
  else:
    return $pred$

3.5 Discussion

Comparing ReAcTable with CoT and ReAct. ReAcTable can be viewed as an advanced and specialized version of the CoT and ReAct paradigms, specifically tailored to address the unique challenges and requirements of TQA tasks. The primary distinction between

these frameworks and ReAcTable lies in ReAcTable's utilization of code executors, which are also essential tools for data scientists, to generate intermediate tables. When working with these intermediate tables which are progressively refined, the LLM produces more coherent and semantically correct code to solve the user question in a step-by-step fashion. Furthermore, ReAcTable incorporates specific design elements crucial for solving TQA problems, such as handling execution exceptions and integrating with majority voting methods. ReAcTable demonstrates that the CoT paradigm is also performance-critical in the TQA scenario, providing data scientists with an easily implementable and high-performing framework.

Comparing different majority voting methods. In ReAcTable, simple majority voting and tree-exploration majority voting can explore a broader range of possible solution paths. By generating multiple code segments and finally selecting the answers that occur the most frequently, they enable the exploration of various solutions to the given question. This naturally aligns with the observation that there could be multiple solutions that lead to the same correct answer. In contrast, execution-based majority voting prioritizes the selection of code segments that are more likely to produce semantically correct results when executed. Instead of exploring the diverse solutions to the same question, this approach emphasizes the practicality and correctness of the generated code.

The selection of the majority voting methods in ReAcTable depends on multiple factors, including the complexity of the TQA task and the capabilities of the underlying LLM. In this paper, we empirically compare these majority voting methods in Section 4.2 and demonstrate that all majority voting methods result in improved performance compared to ReAcTable without majority voting. Additionally, we demonstrate that the selection of the optimal voting method is also a non-trivial task as it depends on the capabilities of the underlying LLM (see Section 4.4). We discuss this aspect in our directions for future work.

4 EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of ReAcTable. Our findings show that ReAcTable surpasses state-of-the-art approaches across multiple commonly used TQA datasets (often beating more complex approaches that fine-tune custom models). To dive deeper into understanding why ReAcTable achieves better performance, we conduct a comprehensive ablation study and analyze the interplay between various voting mechanisms and other parameters (like model choice).

4.1 Experimental Setup

Benchmarks. We adopt three commonly used benchmarks – WikiTQ [32], TabFact [4], and FeTaQA [28] – to evaluate ReAcTable against state-of-the-art baseline approaches. For all data sets, we use the training sets to create static few-shot examples for ReAcTable and use the same sets of few-shot examples throughout the paper.

- ***WikiTQ:*** WikiTableQuestions (WikiTQ) is a data set designed to facilitate research in the field of question answering over structured tabular data [32]. The WikiTQ data set comprises 14,149 question-answer pairs in the training set and 4,344 in the test set. The answers in the WikiTQ data set can take three forms: (i) natural language answers derived from a single tuple, (ii) lists of values extracted from multiple tuples, or (iii) analytical answers that do not exist in the tables.
- ***TabFact:*** the Table-Fact-Checking (TabFact) data set provides a diverse collection of tables sourced from various domains, accompanied by a set of fact-checking queries [4]. The answers to the given queries in TabFact are binary ("yes" or "no"), indicating whether the given queries state facts or not based on the tabular data. To reduce the experimental cost without losing generality, we chose to use the small test set provided [4] to evaluate the performance of ReAcTable and all baseline approaches [5]. The small test set contains 1,998 question-answer pairs.
- ***FeTaQA:*** Free-form Table Question Answering (FeTaQA) is a free-form question-answering data set built upon Wikipedia and presents a different TQA scenario: one in which answers are free-form natural language [28]. FeTaQA contains 7,326 question-answer pairs in the training set and 2,006 in the test set.

Baselines. There are two categories of baseline approaches: (i) approaches that require training, and (ii) approaches that do not require training. Because they differ in their training data requirements (and in their industrial applicability), we report these two categories separately. For the approaches that require training, we include MAPO [23], IterativeSearch [6], MeRL [17], TableFormer [53], Table-BERT [3], ProgVGAT [54], LogicFactChecker [65], SAT [60], Tapex [25], TaCube [66], OmniTab [14], TaPas [11], SaMoE [61], PASTA [9], Lever [29], and the T5 series of models (T5-Small, T5-Base, and T5-Large) [36]. In terms of approaches that do not require training, we report Binder [5] and Dater [56] due to their strong performance. Because the best-performing baseline varies across different benchmarks, we report the best-performing baselines for each benchmark. It is also worth noting that these LLM-based baseline approaches also incorporate various majority voting methods as described in [5, 56]. For all the baseline approaches, we report the best results observed.

ReAcTable Configurations. In our experiments, we report the performance of ReAcTable with and without the three majority voting methods. We denote ReAcTable without majority voting as *ReAcTable*, ReAcTable with simple majority voting as *ReAcTable with s-vote*, ReAcTable with tree-exploration majority voting as *ReAcTable with t-vote*, and ReAcTable with execution-based majority voting as *ReAcTable with e-vote*. Regarding the temperature parameter in LLMs [44], we employ two settings: We set the LLM temperature to zero for *ReAcTable*, while for all experiments utilizing majority voting, we consistently set the temperature to 0.6, which is the common setting of previous works [18, 29]. Additionally, for the code execution environment, we use SQL and Python executors in ReAcTable, unless otherwise specified (Section 4.3.3). For the underlying LLM of ReAcTable, we use Codex [2, 45] as the default LLM. To evaluate the versatility of ReAcTable, we also evaluate ReAcTable with other GPT-series LLMs in Section 4.4. We implement ReAcTable as a Python project and evaluate using Microsoft Fabric Notebooks [27] and Azure OpenAI.

Few-shot Demonstrations. In our experiments, we maintain a consistent approach to selecting few-shot examples from the training set across all benchmark datasets. Specifically, for each of the three

Table 1: Performance of ReAcTable on WikiTQ data set.

Methods	Accuracy
Approaches require training	
MAPO	43.8%
IterativeSearch	44.7%
MeRL	44.1%
T5-3B	50.3%
TableFormer	52.6%
Tapex	57.5%
TaCube	60.8%
OmniTab	62.8%
Lever	**62.9%**
Approaches without training	
Binder	61.9%
Dater	65.9%
ReAcTable	65.8%
with s-vote	**68.0%**
with t-vote	66.4%
with e-vote	67.2%

Table 2: Performance of ReAcTable on TabFact data set.

Methods	Accuracy
Approaches require training	
Table-BERT	68.1%
ProgVGAT	72.6%
LogicFactChecker	74.3%
SAT	75.5%
TaPas	83.9%
Tapex	86.7%
SaMoE	86.7%
PASTA	**90.8%**
Approaches without training	
Binder	85.1%
Dater	85.6%
ReAcTable	83.1%
with s-vote	**86.1%**
with t-vote	84.2%
with e-vote	84.9%

Table 3: Performance of ReAcTable on FeTaQA data set.

Methods	ROUGE-1	ROUGE-2	ROUGE-L
Approaches require training			
T5-Small	0.55	0.33	0.47
T5-Base	0.61	0.39	0.53
T5-Large	0.63	0.41	0.53
Approaches without training			
Dater	0.66	0.45	0.56
ReAcTable	**0.71**	**0.46**	**0.61**

datasets - WikiTQ, TabFact, and FeTaQA, we opt for five examples from the training set and manually create problem-solving demonstrations. Among these five examples, four are addressed using SQL queries, while one necessitates the incorporation of Python code in addition to SQL queries. This selection aligns with our observation that the majority of TQA tasks can be effectively handled using SQL queries, with a smaller subset of questions requiring Python for more complex processing. The choice of five examples is motivated by the necessity to keep the prompt length within the constraints of the context window limit [1]. Notably, from our observation, increasing the number of examples does not necessarily result in improved performance. The process of selecting the most effective few-shot examples remains a longstanding challenge when working with prompting LLMs [42]. Therefore, we regard this aspect as a potential area for future research.

Metrics. We mainly use accuracy to compare the response quality of ReAcTable with the baseline approaches. Since the table question answering task can give multiple tuples as output answers, we use set-based comparison to evaluate the output answer against the given gold answer. To evaluate the quality of the WikiTQ data set, we use the official Python-based WikiTQ evaluator [52]. We simply use string matching for TabFact. As for FeTaQA, since the gold answers are free-form natural language-based answers, we use the commonly adopted ROUGE-N and ROUGE-L metrics [24] to evaluate the "similarity" of answers.

4.2 Performance of ReAcTable

In this section, we compare the performance of ReAcTable with state-of-the-art approaches using three commonly used TQA data sets: WikiTQ, TabFact, and FetaQA.

Results. Table 1, 2, and 3 show the performance of ReAcTable on WikiTQ, TabFact, and FetaQA, respectively.

As shown in Table 1, *ReAcTable with s-vote* achieves an accuracy of 68.0%, outperforming all baseline approaches (including both fine-tuned approaches and approaches without fine-tuning). It is worth noting that the results of the baseline approaches also incorporate various majority voting methods [5, 29]. For *ReAcTable* (without majority voting), it still maintains an accuracy of 65.8%. Among the three majority voting methods, *ReAcTable with s-vote* performs the best (68.0%), while *ReAcTable with t-vote* exhibits a relatively lower accuracy (66.4%). However, all three majority voting methods (*ReAcTable with s-vote, t-vote, and e-vote*) improve the performance of the original *ReAcTable*.

Turning our attention to the TabFact data set, as shown in Table 2, we also observe that *ReAcTable with s-vote* outperforms all state-of-the-art approaches without fine-tuning. Regarding the approaches with fine-tuning, *ReAcTable with s-vote* is still 4.7% lower than the best-performing baseline.

Regarding FeTaQA, as shown in Table 3, given that the gold answers to the questions are free-form natural language sentences, we use the commonly adopted ROUGE-1, ROUGE-2, and ROUGE-L as evaluation metrics, where higher values represent better results.

ReAcTable achieves the highest ROUGE score compared to any other reported baseline.

Takeaways. As shown in the above results, ReAcTable consistently outperforms the state-of-the-art approaches on commonly used benchmarks. It is also worth noting that, in contrast to fine-tuning-based approaches, ReAcTable does not require any training steps, making ReAcTable easier to implement and deploy.

Regarding various voting mechanisms, we observe that ReAcTable with simple majority voting outperforms the other two voting methods. Beyond this experiment, we also observe that these majority voting methods may perform differently when using different LLMs (see Section 4.4). We also present a detailed analysis of the voting methods in Section 3.4.

4.3 Analyzing ReAcTable

In this section, we aim to analyze ReAcTable to address a fundamental question: *Why does ReAcTable outperform most other approaches*? Considering that the major difference between ReAcTable and basic CoT/ReAct prompting is using intermediate tables to improve subsequent reasoning steps, we first analyze how the intermediate tables affect performance. Then, we investigate whether controlling the maximum number of iterations has an impact on the performance of ReAcTable. Finally, we analyze how the choice of different code executors (SQL and Python) affects the results.

4.3.1 *Effect of intermediate tables.*

To evaluate the impact of intermediate tables on ReAcTable's performance, we conduct an ablation study of *ReAcTable* by removing the intermediate tables and creating a new method named *Codex-CoT*. This method produces code sequences for tabular data through a single LLM completion step, directly executing this code for the final answer. The key difference between *Codex-CoT* and *ReAcTable* is the use of intermediate tables in code generation, allowing for a direct comparison to assess the influence of intermediate results on overall performance.

As shown in Section 4.2, simple majority voting (*ReAcTable with s-vote*) stands out as the best-performing configuration among various majority voting methods. Therefore, in this experiment, we exclusively focus on the simple majority voting setup, specifically comparing *Codex-CoT with s-vote* with *ReAcTable with s-vote*.

Results. Table 4 presents the results of our ablation study on the WikiTQ and TabFact data sets.

We observe that *Codex-CoT* achieves an accuracy of only 49.4%, while *ReAcTable* performs notably better with an accuracy of 65.8%, which is a 16.4% improvement. Interestingly, when applying simple majority voting to *Codex-CoT*, the accuracy drops to 47.7%. One possible reason for this decline is that when the LLM is uncertain about the answer, using a high-temperature setup (0.6 in *Codex-CoT with s-vote*) can further increase uncertainty, leading to worse results compared to the low-temperature setup [18].

Regarding the TabFact data set, we make a similar observation from Table 4, where *Codex-CoT* exhibits a 12.0% lower accuracy compared to *ReAcTable*. Furthermore, even after applying simple majority voting, *Codex-CoT with s-vote* achieves an accuracy of 72.3%, which is still 13.8% behind the accuracy of *ReAcTable with s-vote*.

Table 4: Comparison of ReAcTable vs. Codex with simple chain-of-thought on WikiTQ and TabFact data sets.

WikiTQ		TabFact	
Methods	Accuracy	Methods	Accuracy
Codex-CoT	49.4%	Codex-CoT	71.1%
with s-vote	47.7%	*with s-vote*	72.3%
ReAcTable	65.8%	ReAcTable	83.1%
with s-vote	**68.0%**	*with s-vote*	**86.1%**

Takeaways. Our ablation study demonstrates that the inclusion of intermediate tables significantly enhances the performance of ReAcTable. Moreover, it is worth noting that majority voting mechanisms with a high temperature do not consistently yield better accuracy, especially when the LLM is uncertain about the answer.

4.3.2 *Number of iterations.*

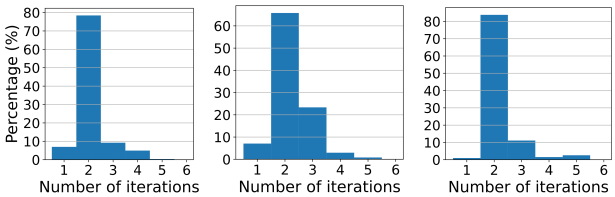
ReAcTable uses multiple rounds of interactions with LLMs and code executors to handle complex question-answering tasks over tabular data. In ReAcTable, the number of iterations can affect the performance: more iterations may enable ReAcTable to reason through difficult questions. To gain insights into how the number of iterations affects ReAcTable 's performance, we aim to answer two questions: 1) *How many iterations does ReAcTable utilize when we do not control the number of iterations*, and 2) *How does controlling the maximum iteration number impact the results*.

Number of iterations. We first analyze the number of iterations when we allow an *unlimited number of iterations* for ReAcTable. In this experiment, we use *ReAcTable with s-vote* on three data sets (WikiTQ, TabFact, and FeTaQA) to perform the study. Figure 4 illustrates the distribution of iteration numbers. As shown in Figure 4, across the three data sets, all questions are resolved within five iterations, with over 70% of the questions being answered within two iterations.

Furthermore, we break down the accuracy of *ReAcTable with s-vote* on the WikiTQ data set (shown in Figure 4a) to reveal the detailed accuracy corresponding to the different numbers of iterations chosen by ReAcTable (w.r.t. different bars in Figure 4a). As demonstrated in Table 5, ReAcTable achieves its highest performance when it opts for two iterations (72.3%). However, as the number of iterations increases, the performance of ReAcTable gradually declines. This observation suggests that questions that require more iteration steps might be inherently challenging for ReAcTable to handle, resulting in lower accuracy.

Limiting the number of iterations. Next, we investigate whether imposing a limit on the number of iterations in ReAcTable impacts its performance. To establish a maximum iteration limit at k, we terminate the reasoning process at iteration $k-1$ if ReAcTable does not opt to directly answer the question. When reaching iteration k, we force ReAcTable to directly answer the question by appending the leading word "Answer" to the prompt.

Table 6 presents the results of ReAcTable when various maximum iteration limits are imposed. As the maximum iteration limit



(a) WikiTQ (b) TabFact (c) FeTaQA

Figure 4: Distribution of the number of iterations.

Table 5: Accuracy breakdown of ReAcTable on WikiTQ data set. We show the accuracy when ReAcTable uses different numbers of iterations to solve the questions, i.e. the accuracy for each bar in Figure 4a.

Iteration # used by ReAcTable	Accuracy
Iteration # = 1 (# of data points: 233)	62.8%
Iteration # = 2 (# of data points: 3,426)	72.3%
Iteration # = 3 (# of data points: 364)	60.3%
Iteration # = 4 (# of data points: 264)	59.3%
Iteration # = 5 (# of data points: 19)	46.2%

Table 6: Performance of ReAcTable on WikiTQ data set.

	limit = 1	*limit* = 2	*limit* = 3	*unlimited*
Accuracy	49.2%	65.1%	67.3%	**68.0%**

is raised, the accuracy of ReAcTable also rises. An interesting observation is that with a maximum limit set at two iterations, the accuracy reaches 65.1% (from 49.2%). When the maximum limit exceeds two, the increase in accuracy becomes less pronounced. This observation aligns with the finding that a substantial portion of questions can be effectively answered within two iterations.

Takeaways. Based on the aforementioned results, the majority of questions in the three commonly used data sets can be answered in just two iterations. Moreover, for complex question-answering tasks where ReAcTable employs more than two iterations to generate an answer, restricting the number of iterations does not lead to improved accuracy. This shows the adaptability of ReAcTable to the complexity of questions and highlights the importance of allowing ReAcTable's flexibility in the iteration process.

4.3.3 Effect of Using Different Code Executors.
In the default configuration of ReAcTable, two external executors are utilized: the SQL executor and the Python executor. The SQL code generated by ReAcTable primarily handles data selection, while the Python code generated by ReAcTable deals with data formatting. In this experiment, our objective is to assess the individual performance contributions of each executor.

For our experiments, we adjusted the ReAcTable configurations to create a variant named ReAcTable (SQL), which uses only SQL for tabular data manipulation. This change means ReAcTable (SQL) depends entirely on SQL, potentially complicating data reformatting and relying more on the LLM's understanding. We limited our ablation study to the Python executor because our version of ReAcTable primarily uses the SQL executor for essential tabular data tasks.

Results. Table 9 presents the results on the WikiTQ and TabFact data sets. When using *ReAcTable (SQL)*, the accuracy on WikiTQ drops from 65.8% to 62.5%. After applying simple majority voting (*ReAcTable with s-vote*), removing the Python executor results in a decrease of 3.5%. Similarly, on TabFact, removing the Python executor leads to a significant drop in accuracy, up to 9.9%. This highlights the importance of the Python executor in enhancing the performance on these data sets.

Takeaways. From our results, the inclusion of the Python executor indeed improves accuracy, particularly by allowing complex data reformatting via Python. It is also important to note that ReAcTable exhibits flexibility and can be adapted to work with various code executors beyond the default SQL and Python executors. We further discuss the choice of code executors in Section 5.2.

4.3.4 Effect of Using Different Majority Voting Methods.
In the results showcased in Tables 1 and 2, our findings indicate that among the three majority voting methods, the simple majority voting approach consistently outperforms the others when employed in conjunction with our ReAcTable configuration. Furthermore, the integration of voting methods significantly enhances the overall performance of ReAcTable, as thoroughly discussed in Section 4.2.

In addition to variations in performance, the majority voting methods also exhibit disparities in the number of code predictions (LLM inferences). This aspect is of paramount importance as a higher number of code predictions can lead to increased resource utilization and time consumption [46]. To provide a comprehensive overview of these prediction costs, we present the average number of code predictions, along with the end-to-end TQA time, for the WikiTQ dataset in Table 7. As anticipated, ReAcTable (without majority voting) requires a minimal 2.4 predictions per question. In contrast, the tree-exploration voting method necessitates a larger number of predictions due to its exponential relationship with the number of steps However, given that questions are answered within five steps (see Figure 4), the overall volume of LLM inferences remains manageable, averaging 30.2 per question. It is important to note that the reported latencies are measured without leveraging parallelization. Implementing parallel execution of independent LLM inferences could significantly diminish these overheads, offering a pathway to enhanced efficiency.

In summary, simple majority voting exhibits strong performance while retaining simplicity of implementation. Execution-based voting generates semantically correct code, while tree-exploration-based voting explores a broad spectrum of answering approaches but incurs significant resource costs and requires further optimization. The selection of the most suitable voting method for specific scenarios remains a non-trivial research challenge, and we highlight it as a promising avenue for future exploration.

4.3.5 Analyzing the Computational Efficiency of ReAcTable.
In this section, we analyze ReAcTable 's computational efficiency. We report four runtime statistics for ReAcTable: (i) average code

Table 7: The average number of LLM inferences performed by ReAcTable on the WikiTQ data set.

Methods	Avg. # of LLM inferences	Avg. end-to-end latency (seconds)
ReAcTable	2.4	5.3
with s-vote	12.1	27.6
with t-vote	30.2	62.2
with e-vote	15.3	31.5

Table 8: Analyzing the runtime of ReAcTable on WikiTQ.

	Time (seconds)
Avg. code prediction time	2.171
Avg. code prediction time per token	0.045
Avg. code execution time	0.187
Avg. code execution time per result row	0.003
Avg. end-to-end question answering time	5.323

prediction time, (ii) average code prediction time per token, (iii) average code execution time, (iv) average code execution time per result row, and (v) average end-to-end question answering time. Among these statistics, average code prediction time and average code prediction time per token pertain to code prediction efficiency (i.e., the inference time of LLMs), while average code execution time and average code execution time per result row relate to the efficiency of executing Python or SQL code over tabular data. We derive these statistics by running ReAcTable (without majority voting methods) on the WikiTQ data set, and similar results can be expected with other data sets.

Table 8 provides a summary of the runtime statistics for ReAcTable. Among all the reported times, code prediction consumes the most time in ReAcTable. In comparison, code execution is relatively less time consuming, as the tables in all the TQA data sets are not of a large scale. Additionally, even with large-scale datasets, since ReAcTable only uses a small number of rows in its prompts (see Section 3.2), ReAcTable can be extended to execute the code over a small set of sampled data rows and/or return only the top-k rows, enhancing the overall code execution efficiency. As there are currently limited TQA data sets with industry-scale tables available, we consider the preparation of such data sets and the exploration of efficient large-scale TQA as a future direction.

In comparison to baseline approaches, ReAcTable stands out for its efficiency, as it eliminates the need for additional training or fine-tuning steps within the TQA pipeline. Regarding baselines that do not involve training, ReAcTable exhibits latency similar to *Dater*, with the primary latency stemming from the LLM inference process. In contrast, when compared to *Binder*, which utilizes LLMs to process data values, ReAcTable notably reduces the number of LLM inferences, resulting in reduced latency.

Table 9: Performance of ReAcTable on WikiTQ and TabFact data sets with only the SQL query executor.

WikiTQ		**TabFact**	
Methods	Accuracy	Methods	Accuracy
ReAcTable	65.8%	ReAcTable	83.1%
with s-vote	**68.0%**	*with s-vote*	**86.1%**
with t-vote	66.4%	*with t-vote*	84.2%
with e-vote	67.2%	*with e-vote*	84.9%
ReAcTable (*SQL*)	62.5%	ReAcTable (*SQL*)	75.4%
with s-vote	64.5%	*with s-vote*	76.2%
with t-vote	64.1%	*with t-vote*	77.1%
with e-vote	63.6%	*with e-vote*	75.8%

Table 10: Performance of ReAcTable with various GPT-series models on WikiTQ data set.

Methods	Accuracy
ReAcTable (code-davinci-002)	65.8%
with s-vote	**68.0%**
with t-vote	66.4%
with e-vote	67.2%
ReAcTable (text-davinci-003)	63.3%
with s-vote	64.1%
with t-vote	64.5%
with e-vote	65.0%
ReAcTable (gpt3.5-turbo)	52.4%
with s-vote	51.8%
with t-vote	52.5%
with e-vote	N.A.[1]

4.4 ReAcTable with Various Language Models

Currently, the landscape of LLMs is diverse and quickly evolving. Likewise, ReAcTable is versatile in its ability to improve the prediction quality of various LLMs for the TQA task. To illustrate the effectiveness of ReAcTable with different LLMs, we evaluate ReAcTable using various GPT series language models.

From the large number of models offered by OpenAI [31], we opted for two commonly used models in addition to the default *code-davinci-002* model: *text-davinci-003* and *gpt3.5-turbo* [49].

Results. The results of ReAcTable with various LLMs are shown in Table 10 for the WikiTQ data set. In the table, we do not report the performance of *ReAcTable with e-vote* using *gpt3.5-turbo* since *gpt3.5-turbo* does not provide probability scores [49].

From the table, the accuracy achieved with the additional two models (*text-davinci-003* and *gpt3.5-turbo*) is lower than when using the default Codex model (*code-davinci-002*). Specifically, with *text-davinci-003*, ReAcTable achieves a maximum accuracy of 65.0% on WikiTQ (*ReAcTable with e-vote*).

[1]Since *gpt3.5-turbo* does not provide probability scores [49], *e-vote* is not applicable on *gpt3.5-turbo*.

Our results reveal that chat-based models tend to generate answers in a more natural language format. For example, in the WikiTQ dataset, when the gold answer is "Francisco Bravo Medical Magnet High School|2007", *ReAcTable with gpt3.5-turbo* predicts "the first school to reach 800 API is Francisco Bravo Medical Magnet High School in the year 2007". While we still use the official evaluator provided with the WikiTQ dataset [52] to ensure consistency, it is worth noting that this official evaluator does not recognize synonyms or paraphrases of the correct answer as valid. We also consider the development of an evaluator that accounts for synonyms and paraphrases as a potential future direction.

Furthermore, it is worth noting that for all ReAcTable configurations with *text-davinci-003*, execution-based voting methods (*ReAcTable with s-vote*) consistently yield the best results. This observation suggests that execution-based voting is effective in selecting semantically correct code for a model that is not primarily designed for code generation, like *text-davinci-003*.

Takeaways. From the results, ReAcTable exhibits the capability to effectively utilize various language models. However, the choice of model can also impact ReAcTable's end-to-end performance.

5 LESSONS LEARNED

In this section, we summarize the lessons we learned during our exploration of LLMs in TQA tasks. We provide discussions on the design spaces that we considered and the limitations of ReAcTable.

5.1 Design Spaces of LLM for TQA tasks

Prior research in TQA has mainly explored two strategies: (i) training or fine-tuning LLMs, and (ii) using pre-trained LLMs without modification. The former involves customizing LLMs for TQA, which can enhance performance but faces challenges like data acquisition and high costs. The latter strategy employs pre-trained LLMs directly, requiring careful prompting and design to be effective despite their initial lack of task-specific optimization. We focused on the latter, leveraging pre-trained LLMs due to their simplicity and flexibility. By adapting LLM enhancement frameworks, CoT and ReAct, for TQA, ReAcTable outperforms many standard methods, maintaining its ease of use and adaptability.

5.2 Choosing External Code Executors

In our experiments, we employ SQL and Python as the two external code executors. We have chosen these two executors because they are commonly utilized by data scientists for tabular data manipulation tasks. In our configuration, SQL primarily handles data selection, while Python is employed for complex data reformatting operations. Furthermore, incorporating these code executors also facilitates the possibility for data scientists to easily assess the correctness of (intermediate) code. In addition, using these code executors does not introduce additional implementation complexities, unlike approaches like Binder [5], which necessitate complex re-implementation of the SQL executor.

5.3 Majority Voting Mechanisms

In this paper, we investigate the three majority voting methods in ReAcTable: (i) simple majority voting, (ii) tree-exploration voting, and (iii) execution-based voting. As elaborated in Section 3.5, simple majority voting and tree-exploration voting involve exploring multiple solution paths for the same question, whereas execution-based voting prioritizes the generation of semantically correct code.

It is important to emphasize that the application of majority voting may potentially lead to a performance decline, particularly when the model exhibits uncertainty regarding the answer. This could be attributed to the fact that the high-temperature setup in majority voting might further accentuate the model's uncertainty. Furthermore, majority voting introduces additional prompting costs. Such cost can also be crucial, as prompting LLMs consumes substantial GPU resources. Therefore, one should exercise caution when considering the use of majority voting methods.

5.4 Limitations and Future Works

In the context of ReAcTable, we employ few-shot prompting methods, which involve the manual creation of examples based on the training set. This process can be intricate, and fine-tuning prompts falls beyond the scope of this paper. While there are existing methods proposed for searching few-shot examples from a corpus [5, 33] and tuning model prefixes [22], ReAcTable differs from traditional table question answering methods. It utilizes the CoT and ReAct paradigm, which requires the creation of few-shot examples to follow the "chain-of-code" style. This presents a challenge for automation, as it often necessitates human involvement in the process. We recognize automatic prompt tuning and the selection of few-shot examples as potential directions for future research.

While our current work primarily focuses on the foundational aspect of answering questions over tabular data using LLMs with multi-step code execution, specifically addressing single-table question answering, an enticing direction for future research lies in extending our approach to tackle large-scale multi-table scenarios. Within the realm of multi-table scenarios, the intricate multi-table merging semantics (such as inner joins, outer joins, union, intersection, etc.) presents a challenge for LLMs to effectively capture. Moreover, due to the limited availability of large-scale multi-table question answering datasets, we recognize the formulation of such datasets and research aimed at addressing these complexities as a novel direction. Additionally, we also acknowledge the importance of automating the selection process for the most effective majority voting method, as discussed in Sections 3.5 and 4.3.

Finally, we note that recent work in the space of *multi-modal* query planning [48] presents yet another intriguing area for future research. In such works, questions operate not just over a single table, or even multiple tables, but, instead, over entirely different modalities of data such as images.

6 CONCLUSION

In this paper, we investigate how the TQA problem can be effectively addressed using foundational advances in LLMs. We introduce ReAcTable, a framework that employs LLMs to reason step-by-step and iteratively generates intermediate tables using external code executors. Our experimental results demonstrate that ReAcTable outperforms existing state-of-the-art approaches. Our findings illustrate that a simple, yet carefully adapted LLM-based framework can still surpass many state-of-the-art approaches tailored to the table question answering task.

REFERENCES

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[3] Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2019. TabFact: A Large-scale Dataset for Table-based Fact Verification. *CoRR* abs/1909.02164 (2019). arXiv:1909.02164 http://arxiv.org/abs/1909.02164

[4] 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. *International Conference on Learning Representations* (2020). https://openreview.net/forum?id=rkeJRhNYDH

[5] 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. arXiv:2210.02875 [cs.CL]

[6] Pradeep Dasigi, Matt Gardner, Shikhar Murty, Luke Zettlemoyer, and Eduard Hovy. 2019. Iterative Search for Weakly Supervised Semantic Parsing. 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)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 2669–2680. https://doi.org/10.18653/v1/N19-1273

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[8] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. 933–944.

[9] 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. arXiv:2211.02816 [cs.CL]

[10] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java expressions from free-form queries. In *Proceedings of the 2015 acm sigplan international conference on object-oriented programming, systems, languages, and applications*. 416–432.

[11] 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*. Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.acl-main.398

[12] Richard D Hipp. 2020. SQLite. https://www.sqlite.org/index.html

[13] Reid Holmes and Gail Murphy. 2005. Using structural context to recommend source code examples. *Proceedings - 27th International Conference on Software Engineering, ICSE05*, 117–125. https://doi.org/10.1109/ICSE.2005.1553554

[14] 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. arXiv:2207.03637 [cs.CL]

[15] Nengzheng Jin, Joanna Siebert, Dongfang Li, and Qingcai Chen. 2022. A Survey on Table Question Answering: Recent Advances. arXiv:2207.05270 [cs.CL]

[16] Rogers Jeffrey Leo John, Dylan Bacon, Junda Chen, Ushmal Ramesh, Jiatong Li, Deepan Das, Robert V. Claus, Amos Kendall, and Jignesh M. Patel. 2023. DataChat: An Intuitive and Collaborative Data Analytics Platform. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia (Eds.). ACM, 203–215. https://doi.org/10.1145/3555041.3589678

[17] Shauharda Khadka, Somdeb Majumdar, and Kagan Tumer. 2019. Evolutionary Reinforcement Learning for Sample-Efficient Multiagent Coordination. *CoRR* abs/1906.07315 (2019). arXiv:1906.07315 http://arxiv.org/abs/1906.07315

[18] Anirudh Khatry, Joyce Cahoon, Jordan Henkel, Shaleen Deep, Venkatesh Emani, Avrilia Floratou, Sumit Gulwani, Vu Le, Mohammad Raza, Sherry Shi, Mukul Singh, and Ashish Tiwari. 2023. From Words to Code: Harnessing Data for Program Synthesis from Natural Language. arXiv:2305.01598 [cs.DB]

[19] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2023. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. arXiv:2311.03157 [cs.DB]

[20] Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, et al. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint arXiv:2305.03111* (2023).

[21] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).

[22] Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing Continuous Prompts for Generation. 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)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, Online, 4582–4597. https://doi.org/10.18653/v1/2021.acl-long.353

[23] Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V. Le, and Ni Lao. 2018. Memory Augmented Policy Optimization for Program Synthesis with Generalization. *CoRR* abs/1807.02322 (2018). arXiv:1807.02322 http://arxiv.org/abs/1807.02322

[24] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[25] 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. arXiv:2107.07653 [cs.CL]

[26] Wes McKinney et al. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, Vol. 445. Austin, TX, 51–56.

[27] Microsoft Power BI 2024. *Microsoft Power BI.* Retrieved Jan 27, 2023 from https://msit.powerbi.com/home?experience=power-bi

[28] Linyong Nan, Chiachun Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, Rui Zhang, Wojciech Kryściński, Hailey Schoelkopf, Riley Kong, Xiangru Tang, Mutethia Mutuma, Ben Rosand, Isabel Trindade, Renusree Bandaru, Jacob Cunningham, Caiming Xiong, and Dragomir Radev. 2022. FeTaQA: Free-form Table Question Answering. *Transactions of the Association for Computational Linguistics* 10 (2022), 35–49.

[29] 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. arXiv:2302.08468 [cs.LG]

[30] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[31] OpenAI models 2023. *OpenAI models.* Retrieved Sep 17, 2023 from https://platform.openai.com/docs/models

[32] Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. arXiv:1508.00305 [cs.CL]

[33] Ethan Perez, Douwe Kiela, and Kyunghyun Cho. 2021. True few-shot learning with language models. *Advances in neural information processing systems* 34 (2021), 11054–11070.

[34] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, Markus Weimer, Avrilia Floratou, Carlo Curino, and Konstantinos Karanasos. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *SIGMOD Rec.* 51, 2 (jul 2022), 30–37. https://doi.org/10.1145/3552490.3552496

[35] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. 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)*. 878–888.

[36] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[37] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

[38] Joshua Robinson, Christopher Michael Rytting, and David Wingate. 2022. Leveraging large language models for multiple choice question answering. *arXiv preprint arXiv:2210.12353* (2022).

[39] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[40] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 9895–9901. https://aclanthology.org/2021.emnlp-main.779

[41] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454* (2022).

[42] Hongjin Su, Jungo Kasai, Chen Henry Wu, Weijia Shi, Tianlu Wang, Jiayi Xin, Rui Zhang, Mari Ostendorf, Luke Zettlemoyer, Noah A Smith, et al. 2022. Selective annotation makes language models better few-shot learners. *arXiv preprint arXiv:2209.01975* (2022).

[43] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[44] Temperature setup of GPT models 2023. *Temperature setup of GPT models.* Retrieved Sep 28, 2023 from https://platform.openai.com/docs/api-reference/completions

[45] The Codex model of OpenAI 2023. *The Codex model of OpenAI.* Retrieved Sep 28, 2023 from https://openai.com/blog/openai-codex

[46] The prompting cost of OpenAI 2023. *The prompting cost of OpenAI.* Retrieved Dec 22, 2023 from https://openai.com/pricing

[47] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the 2022 international conference on management of data*. 190–203.

[48] Matthias Urban and Carsten Binnig. 2024. CAESURA: Language Models as Multi-Modal Query Planners. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. https://www.cidrdb.org/cidr2024/papers/p14-urban.pdf

[49] Usage of GPT-3.5-Turbo and GPT-4 Models 2023. *Usage of GPT-3.5-Turbo and GPT-4 Models.* Retrieved Sep 17, 2023 from https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/chatgpt?pivots=programming-language-chat-completions

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[51] 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. arXiv:2201.11903 [cs.CL]

[52] WikiTableQuestions data set 2023. *WikiTableQuestions data set.* Retrieved Sep 28, 2023 from https://github.com/ppasupat/WikiTableQuestions

[53] Jingfeng Yang, Aditya Gupta, Shyam Upadhyay, Luheng He, Rahul Goel, and Shachi Paul. 2022. TableFormer: Robust Transformer Modeling for Table-Text Encoding. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 528–537. https://doi.org/10.18653/v1/2022.acl-long.40

[54] Xiaoyu Yang, Feng Nie, Yufei Feng, Quan Liu, Zhigang Chen, and Xiaodan Zhu. 2020. Program Enhanced Fact Verification with Verbalization and Graph Attention Network. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 7810–7825. https://doi.org/10.18653/v1/2020.emnlp-main.628

[55] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL]

[56] 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. arXiv:2301.13808 [cs.CL]

[57] Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, et al. 2019. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. *arXiv preprint arXiv:1909.05378* (2019).

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

[59] Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, et al. 2019. Sparc: Cross-domain semantic parsing in context. *arXiv preprint arXiv:1906.02285* (2019).

[60] 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)*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1624–1629. https://doi.org/10.18653/v1/2020.emnlp-main.126

[61] Minjia Zhang, Conglong Li, Xiaoxia Wu, Zhewei Yao, and Yuxiong He. 2022. SaMoE: Parameter Efficient MoE Language Models via Self-Adaptive Expert Combination. (2022).

[62] Ningyu Zhang, Luoqiu Li, Xiang Chen, Shumin Deng, Zhen Bi, Chuanqi Tan, Fei Huang, and Huajun Chen. 2021. Differentiable prompt makes pre-trained language models better few-shot learners. *arXiv preprint arXiv:2108.13161* (2021).

[63] Yunjia Zhang, Avrilia Floratou, Joyce Cahoon, Subru Krishnan, Andreas C. Müller, Dalitso Banda, Fotis Psallidas, and Jignesh M. Patel. 2023. Schema Matching using Pre-Trained Language Models. In *ICDE*. IEEE. https://www.microsoft.com/en-us/research/publication/schema-matching-using-pre-trained-language-models/

[64] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).

[65] Wanjun Zhong, Duyu Tang, Zhangyin Feng, Nan Duan, Ming Zhou, Ming Gong, Linjun Shou, Daxin Jiang, Jiahai Wang, and Jian Yin. 2020. LogicalFactChecker: Leveraging Logical Operations for Fact Checking with Graph Module Network. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 6053–6065. https://doi.org/10.18653/v1/2020.acl-main.539

[66] Fan Zhou, Mengkang Hu, Haoyu Dong, Zhoujun Cheng, Shi Han, and Dongmei Zhang. 2022. TaCube: Pre-computing Data Cubes for Answering Numerical-Reasoning Questions over Tabular Data. arXiv:2205.12682 [cs.IR]

[67] Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. 2023. Multilingual Machine Translation with Large Language Models: Empirical Results and Analysis. arXiv:2304.04675 [cs.CL]