CHAIN-OF-QUERY: Unleashing the Power of LLMs in SQL-Aided Table Understanding via Multi-Agent Collaboration

Songyuan Sui[1], Hongyi Liu[1], Serena Liu[1], Li Li[2],
Soo-Hyun Choi[2], Rui Chen[2], Xia Hu[1]
[1]Rice University
{Songyuan.Sui, Hongyi.Liu, Serena.Liu, Xia.Hu}@rice.edu
[2]Samsung Electronics America
{li.li1, sh9.choi, rui.chen1}@samsung.com

Abstract

Table understanding requires structured, multi-step reasoning. Large Language Models (LLMs) struggle with it due to the structural complexity of tabular data. Recently, multi-agent frameworks for SQL generation have shown promise in tackling the challenges of understanding tabular data, but existing approaches often suffer from limitations such as the inability to comprehend table structure for reliable SQL generation, error propagation that results in invalid queries, and over-reliance on execution correctness. To address these issues, we propose CHAIN-OF-QUERY (CoQ), a novel multi-agent framework for SQL-aided table understanding. CoQ adopts natural-language-style representations of table schemas to abstract away structural noise and enhance understanding. It employs a clause-by-clause SQL generation strategy to improve query quality and introduces a hybrid reasoning division that separates SQL-based mechanical reasoning from LLM-based logical inference, thereby reducing reliance on execution outcomes. Experiments with four models (both closed- and open-source) across five widely used benchmarks show that CHAIN-OF-QUERY significantly improves accuracy from 61.11% to 74.77% and reduces the invalid SQL rate from 9.48% to 3.34%, demonstrating its superior effectiveness in table understanding. The code is available at https://github.com/SongyuanSui/ChainofQuery.

1 Introduction

Large Language Models (LLMs) have shown remarkable performance across a variety of natural language processing (NLP) tasks (Yang et al., 2023). However, they still struggle with understanding tabular data (Sui et al., 2023a). This challenge stems from two main factors. First, the structure of tabular data differs significantly from that of plain text. It organizes information via rows and columns, introducing hierarchical relationships, positional dependencies, and implicit semantics that are challenging for language models to capture. Second, table-based tasks often involve multi-step, structured reasoning operations, such as aggregation, comparison, and arithmetic computation (Lu et al., 2025). They go beyond surface-level language understanding.

To adapt LLMs for table understanding, prior work has explored many approaches. Many studies (Herzig et al., 2020; Zhang et al., 2023a,b; Li et al., 2024b; He et al., 2025) fine-tuned LLMs on table-specific datasets, but these methods treat the task as a single-turn generation problem, which results in shallow reasoning paths and overlooks intermediate data. Recent studies (Wang et al., 2024c; Ji et al., 2024; Zhou et al., 2025) built agent-based pipelines with table manipulation tools to conduct multi-step reasoning. Nevertheless, these tools rely on human-defined programs whose functionalities are restricted and cannot perform complex or adaptive operations beyond their predefined scope.

We argue that incorporating Structured Query Language (SQL) provides a more principled approach for understanding tabular data. SQL is inherently designed to access, filter, and aggregate information from relational tables, making it well-aligned with the needs of table understanding. Although existing work on agent-based SQL generation has made some progress in table understanding (Ye et al., 2023; Kong et al., 2024a), its performance remains suboptimal. **We identify three key challenges in SQL-aided table understanding with LLMs. 1.** Existing approaches typically feed the table and task instructions into an LLM, assuming the LLM can understand the table in order to produce meaningful SQL. However, if the LLM already struggles with table understanding, it is unlikely to generate accurate SQL in the first place. This workflow thus inherits LLMs' structural inability, as effective SQL generation

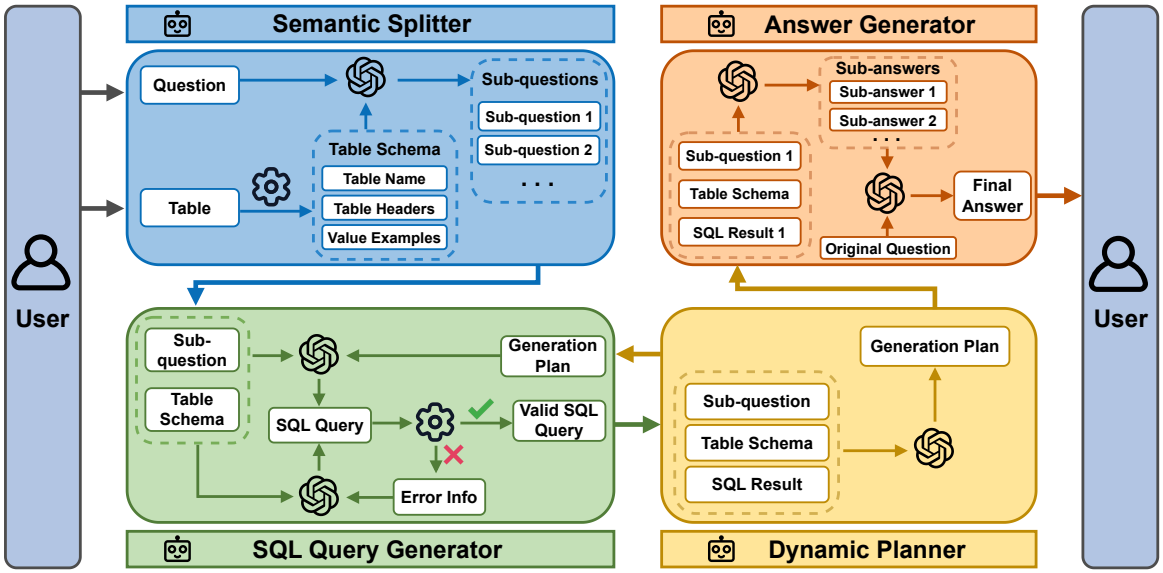

Figure 1: Overview of CHAIN-OF-QUERY Framework.

still requires a solid grasp of the table's layout and semantics—something LLMs inherently lack. **2.** Supporting complex, multi-step reasoning often requires SQL queries with multiple clauses and nested structures. Such queries are difficult to construct correctly in a single shot, and even small errors can cascade into query failure. **3.** Existing methods depend heavily on the correctness of SQL execution, as they directly use SQL results as final answers. This further leads LLMs to generate overly complicated queries, reducing reliability.

To tackle these challenges, we propose CHAIN-OF-QUERY (CoQ), a novel multi-agent framework for SQL-aided table understanding. To address the first challenge, CoQ replaces raw table inputs with **natural-language representations of table schemas**. Our insight is that SQL generation depends primarily on high-level schema, which can be easily and accurately expressed in natural language, bypassing the need for LLMs to interpret complex tabular structures. This abstraction allows LLMs to focus on semantics without being hindered by noisy or irregular layouts. To address the second challenge, we propose a **Clause-by-Clause SQL Generation Strategy**, which incrementally constructs SQL queries one clause at a time, forming a chain of progressively refined queries. To solve the third challenge, we design a **Hybrid Reasoning Division Strategy** that separates mechanical reasoning (executed via SQL) from logical reasoning (delegated to LLMs), treating SQL outputs as intermediate steps rather than final answers. In addition, our **Parallel Decomposition** enhances robustness by splitting complex questions into parallelizable sub-questions, avoiding the inter-step dependencies of traditional sequential reasoning. Together, CoQ offers a novel design that directly addresses these challenges, establishing an effective new paradigm for table understanding.

To comprehensively evaluate the effectiveness of CoQ, we conduct extensive experiments using both the closed-source models (GPT-3.5, GPT-4.1) and the open-source models (LLaMA 2, DeepSeek-V3), across five representative table understanding benchmarks: WikiTQ, TabFact, FeTaQA, IM-TQA, and Open-WikiTable. Experimental results show that CoQ consistently outperforms state-of-the-art (SOTA) baselines, increasing accuracy from 61.11% to 74.77%, and reducing the invalid SQL generation rate from 9.48% to 3.34%.

In summary, we propose CHAIN-OF-QUERY (CoQ), a novel multi-agent collaboration framework for SQL-aided table understanding with the following contributions:

- We design natural-language-style table schemas to abstract away structural noise and irregularities in raw tables, enabling LLMs to focus on high-level semantics.
- We propose a novel Clause-by-Clause SQL Generation Strategy, which constructs queries incrementally to reduce error propagation and improve reliability.
- We introduce a Hybrid Reasoning Division Strategy that separates mechanical reasoning

(handled by SQL) and logical reasoning (handled by LLMs).

- Extensive experiments show CHAIN-OF-QUERY consistently outperforms previous baselines, achieving substantial improvements across diverse settings.

2 Related Work

We review previous work from three aspects: Traditional LLM-Based Table Understanding, Multi-Agent Table Understanding, and Text-to-SQL.

Traditional LLM-Based Table Understanding. Since the structure of tables (e.g., non-sequential cell order) differs significantly from plain text typically used in pre-training (Raffel et al., 2023), some work (Zhang et al., 2023a,b; Li et al., 2024b; Zhuang et al., 2024; He et al., 2025) fine-tuned LLMs to adapt to tabular data. While effective on specific datasets, these methods depend on crafted instructions and expensive training, limiting scalability. More recent prompt-based approaches (Sui et al., 2023a,b; Chen et al., 2024) exploit LLMs' general reasoning ability, offering better generalization across tasks. However, these methods typically adopt single-turn prompting strategies, failing to support multi-hop reasoning.

Multi-Agent Table Understanding. Recent work has explored LLM-powered agents (Wu et al., 2023; Wang et al., 2024a) that perform multi-step reasoning and invoke table operation tools. Dater (Ye et al., 2023) reformulates questions into cloze-style prompts and retrieves values via LLM-generated SQL. Chain-of-Table (Wang et al., 2024c) and Tree-of-Table (Ji et al., 2024) construct dynamic reasoning paths using pre-defined Python functions. Although effective, these approaches rely on coding-based steps. This makes them fragile, as code errors can cause the entire reasoning chain to fail. For example, Zhou et al. (2025) reports that about half of the failures come from invalid code generation. Moreover, most of these methods are limited by fixed function sets, restricting their reasoning flexibility.

Text-to-SQL. Generating SQL queries from natural language is a long-standing challenge. Early methods primarily rely on fine-tuning LLMs on annotated SQL datasets (Pourreza and Rafiei, 2024; Li et al., 2024a). To reduce training cost and improve compositional reasoning, Tai et al. (2023) and Pourreza and Rafiei (2023) explored Chain-of-Thought prompting in Text-to-SQL. OpenTab (Kong et al., 2024a) integrates LLM-based SQL generation into a tabular data RAG system. Recently, MAC-SQL (Wang et al., 2025b) and MAG-SQL (Xie et al., 2024b) build multi-agent frameworks that refine generated queries to improve SQL quality. Although these methods perform well on benchmarks such as Spider (Yu et al., 2019) and BIRD (Li et al., 2023), they all tend to generate fully grounded SQL queries that directly return the final answer, making query construction difficult. Moreover, LLMs tend to produce overly complex queries in an attempt to handle questions holistically (Shen et al., 2025), reducing reliability.

3 CHAIN-OF-QUERY Approach

3.1 Problem Definition of Table Understanding

Given a 2-tuple $X = (Q, T)$, where Q is a natural language question and $T = \{S, D\}$ is a table composed of schema S and data content D, the goal of table understanding is to identify a relevant subset of data $D_r \subseteq D$ that is necessary to answer Q, and then derive the final answer A by reasoning over both Q and D_r. Here, the schema S describes the structure of the table, while D contains the individual cell-level values.

3.2 Overview of CHAIN-OF-QUERY

We propose CHAIN-OF-QUERY as a multi-agent framework that decomposes table understanding into modular sub-tasks, with each agent invoking dedicated strategies for its assigned role. As shown in Figure 1, it comprises four specialized agents: **Semantic Splitter**, **SQL Query Generator**, **Dynamic Planner**, and **Answer Generator**, each responsible for a distinct stage of the pipeline. The Semantic Splitter constructs natural-language-style table schemas and decomposes questions into parallel sub-questions. The SQL Query Generator applies the Clause-by-Clause SQL Generation Strategy. The Dynamic Planner incorporates the Hybrid Reasoning Division Strategy. The Answer Generator synthesizes final answers based on SQL outputs and inferences from the LLM. We describe each component and its corresponding strategies in the following sections.

3.3 Semantic Splitter: Schema Abstraction and Query Decomposition

The Semantic Splitter serves as the entry point of the entire agentic workflow. This module is de-

signed to optimize the input for SQL generation by addressing two key challenges: (1) LLMs' limited ability to interpret complex table structures, which often leads to inaccurate SQL generation; and (2) interference between sub-questions within a single complex query, where independent sub-questions could be handled separately to avoid entanglement. To address these challenges, we introduce a natural-language-style representation of table schemas for query generation, and incorporate a parallel decomposition mechanism to identify and isolate separable sub-queries for clean execution. The following subsections detail this process.

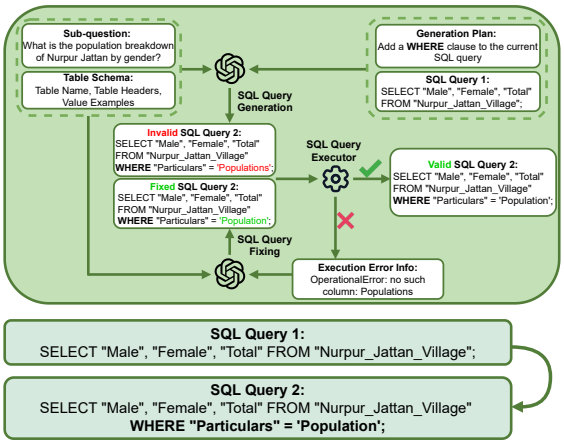

Figure 2: Illustration of the SQL Query Generator and the Clause-by-Clause Generation Strategy.

3.3.1 Natural-Language-Style Table Schema

LLMs' limited understanding of tabular data hinders their ability to generate accurate SQL based on raw tables. To address this, we ask: Is a natural-language-style schema sufficient for SQL generation? Our key insight is that SQL generation primarily depends on high-level schema information—not the detailed table contents—and that expressing the schema in natural language aligns better with LLMs' strengths as language models.

With the goal of addressing these limitations in handling raw and irregular table structures, we construct a concise natural-language-style schema as a surrogate for full-table input. It consists of three components: *table name*, *headers*, and *value examples*. The table name employs concise keywords to summarize the main content and purpose of the table at a high level, providing essential context for the LLM to understand the table. The table headers comprise multiple (column name, column value type) pairs. These pairs not only provide a natural language abstraction of the table structure, but also serve as a reference for value formatting in downstream SQL generation. The value examples are a few sampled rows of the table paired with their corresponding column names. We adopt "PIPE" format (i.e., a pipe-delimited string with "|" separators) to encode the value examples, as studies found it to improve LLM performance on table tasks (Sui et al., 2023a; Wang et al., 2024c).

By replacing the full table with our designed schema, LLMs can generate accurate SQL without being distracted by complex layouts or unnecessary content. This approach also improves scalability, as it avoids importing long tables in full, reducing input length and simplifying the reasoning process.

3.3.2 Parallel Decomposition

The second component of the Semantic Splitter is a Parallel Decomposer, which focuses on separating input questions into independent sub-questions. Unlike the traditional sequential decomposition strategies where each sub-question depends on the previous answer and introduces fragility due to error propagation, our approach targets questions without interdependencies. We defer the handling of sequential dependencies to our Clause-by-Clause Generator in Section 3.4.2.

Many complex tabular questions involve comparing or aggregating information from semantically distinct and disjoint table regions, which do not require sequential reasoning. To address this, the Parallel Decomposer splits such questions into independent sub-questions, each focused on a localized region of the table. This removes inter-step dependencies, enables concurrent processing, and simplifies SQL generation, as each sub-query operates over a narrow, self-contained scope without interference from others.

3.4 Clause-wise SQL Generation for Precision and Robustness

After outlining our approach to transforming and preprocessing inputs to improve LLMs' understanding, we now focus on the next key component: generating accurate SQL with LLMs to support effective table understanding. The SQL Query Generator constructs executable SQL queries to retrieve intermediate information relevant to the task and support mechanical reasoning. As shown in Figure 2, an initial query is generated by an LLM, then validated before being passed to the Dynamic Planner, which determines whether further clauses

should be appended to improve precision. We now detail the generation process.

3.4.1 SQL Query Generation with Validation

Given the schema and sub-question, an LLM generates an SQL query to retrieve relevant information. The query is validated by an SQL executor through direct execution; if it fails, the error message is passed to a secondary LLM for correction. The revised query is re-executed and, if successful, forwarded to the Dynamic Planner. To enhance query quality, we introduce a clause-by-clause generation strategy described below.

3.4.2 Clause-by-Clause SQL Generation

Table understanding tasks often require complex, multi-step reasoning, such as aggregation and mathematical operations that in turn demand SQL queries with multiple clauses or even nested structures. This makes single-shot SQL generation particularly challenging, as even minor errors can propagate and cascade through the query, ultimately leading to execution failures.

To improve the reliability and controllability of SQL generation, we propose a Clause-by-Clause Strategy that incrementally builds queries by isolating the construction of each clause. The generator starts by producing a basic `SELECT-FROM` query that selects columns relevant to the sub-question. However, without condition filtering, this query may return a large number of rows, posing scalability challenges. To mitigate this, we sample a few representative rows (similar to the natural-language-style table schema construction) as the Planner only needs to observe data patterns rather than precise values.

Subsequent clauses (e.g., `WHERE`, `GROUP BY`) are appended one at a time, under the guidance of the Dynamic Planner. By decomposing SQL construction into discrete clauses, each step focuses solely on its own logic and syntax, independent of the full query context. Each intermediate query is validated before proceeding. If execution fails, the error can be traced to the most recently added clause, allowing for more targeted correction. If correction fails, the agent reverts to the last validated query, ensuring robustness. As shown in Figure 2, if a newly generated clause introduces an error to the query (SQL2) and cannot be resolved, the agent uses the last valid query (SQL1) to ensure executable SQL. This incremental process forms a chain of increasingly precise queries, where each step builds on a validated state, preventing error propagation and ensuring stability.

3.5 Balancing SQL and LLM Reasoning through Dynamic Planning

We build the Dynamic Planner that determines whether to continue SQL generation, guided by a Sufficiency-based Early Stopping Mechanism. Together with the Answer Generator, it implements our Hybrid Reasoning Division Strategy: SQL handles mechanical reasoning, while LLMs perform logical reasoning over intermediate results to produce the final answer.

3.5.1 Hybrid Reasoning Division Strategy

Existing SQL-aided approaches treat the SQL output as the final answer, requiring highly precise queries. This often leads to long, fragile SQL programs, especially when reasoning is complex or question intent is ambiguous. However, table understanding is inherently open-ended: the goal is to derive a natural language answer, not merely to retrieve an exact value. Based on these insights, we introduce a Hybrid Reasoning Division Strategy that decomposes table understanding into two stages: mechanical reasoning (e.g., filtering, arithmetic) is offloaded to SQL, while logical reasoning (e.g., comparison, inference) is handled by LLMs. Crucially, SQL execution results are treated as intermediate data rather than final answers, enabling LLMs to apply their stronger inference capabilities over SQL outputs to arrive at the final answer. This division leverages the respective strengths of SQL and LLMs. Our case study (Appendix A) further explains this design.

3.5.2 Sufficiency-based Early Stopping

To support this hybrid reasoning strategy, we introduce the Sufficiency-based Early Stopping Mechanism. It enables the Planner to halt clause generation once the retrieved data are sufficient. This prevents unnecessary SQL complexity and ensures that reasoning responsibilities are dynamically and appropriately balanced between SQL and LLMs.

3.6 Answer Generator

The Answer Generator produces the final answer by aggregating sub-answers from each sub-question. It first generates a sub-answer based on the corresponding SQL result, then combines all sub-answers into a complete natural language response to the original question. As the final stage of the

Table 1: Results on WikiTQ and TabFact using GPT-3.5 and LLaMA 2. **Acc.** = accuracy (%), **Inv.** = invalid SQL rate (%, lower is better). Underline = second-best, **bold** = best. Improvements are over second-best. CoQ achieves the highest accuracy with a large margin, driven by schema abstraction, clause-by-clause SQL generation, and hybrid reasoning.

Method	WikiTQ				TabFact			
	GPT-3.5		LLaMA 2		GPT-3.5		LLaMA 2	
	Acc.	Inv.	Acc.	Inv.	Acc.	Inv.	Acc.	Inv.
Generic Table Understanding								
End-to-End QA	43.39	N/A	35.48	N/A	67.45	N/A	53.46	N/A
Table-to-Text	16.07	N/A	14.21	N/A	49.72	N/A	48.06	N/A
Few-Shot QA	52.56	N/A	35.52	N/A	71.54	N/A	62.01	N/A
Chain-of-Thought	53.48	N/A	36.05	N/A	65.37	N/A	60.52	N/A
Binder	56.74	N/A	30.92	N/A	79.17	N/A	62.76	N/A
Dater	52.81	N/A	41.44	N/A	78.01	N/A	65.12	N/A
Chain-of-Table	59.94	N/A	42.61	N/A	80.20	N/A	67.24	N/A
Tree-of-Table	61.11	N/A	44.01	N/A	81.92	N/A	69.33	N/A
SQL-aided Table Understanding								
Basic Text-to-SQL	47.40	14.07	32.18	18.81	64.93	19.66	63.27	33.02
OpenTab	55.39	10.24	37.41	15.16	78.57	12.73	56.84	27.83
MAC-SQL	52.92	10.34	36.88	17.65	76.06	18.13	56.68	31.54
MAG-SQL	55.87	9.48	38.25	16.13	78.84	13.28	59.91	29.49
CHAIN-OF-QUERY (Ours)	**74.77** (+13.66)	**3.34** (-6.14)	**58.91** (+14.90)	**13.18** (-1.98)	**92.31** (+10.39)	**2.74** (-9.99)	**78.80** (+9.47)	**23.88** (-3.95)

pipeline, this component ensures that localized reasoning results are coherently integrated into a unified natural language response.

4 Experiments

In this section, we empirically evaluate the effectiveness of CoQ and aim to address the following questions: **RQ1**: How does CoQ compare with generic table understanding methods? **RQ2**: How does CoQ perform relative to traditional SQL-aided methods? **RQ3**: How well does CoQ generalize to real-world, structurally complex tabular workloads?

4.1 Experimental Setup

Datasets and Evaluation Metrics. We evaluate our CoQ framework on five widely used table understanding benchmarks: WikiTQ (Pasupat and Liang, 2015), FeTaQA (Nan et al., 2022), TabFact (Chen et al., 2020), IM-TQA (Zheng et al., 2023), and Open-WikiTable (Kweon et al., 2023). WikiTQ and FeTaQA are table QA datasets requiring short-span and free-form answers, respectively. TabFact is a fact verification task based on tables. IM-TQA features complex real-world table styles, while Open-WikiTable involves multi-table scenarios. Collectively, these datasets cover diverse reasoning types, table structures, and domains, forming a robust testbed for evaluating generalization. Dataset details are provided in Appendix F.

We use official accuracy metrics for WikiTQ, IM-TQA, and Open-WikiTable, and standard binary accuracy for TabFact. For FeTaQA, we report BLEU (Papineni et al., 2002) and ROUGE (Lin, 2004) scores to align with prior work. We also track the rate of invalid SQL generation.

Baselines. We compare our method against two categories of baseline approaches: (a) *Generic table understanding*, including End-to-End QA, Few-Shot QA, Table-to-Text (Min et al., 2024), Chain-of-Thought (Wei et al., 2022), Binder (Cheng et al., 2023), Dater (Ye et al., 2023), Chain-of-Table (Wang et al., 2024c), and Tree-of-Table (Ji et al., 2024); (b) *SQL-aided table understanding*, including Basic Text-to-SQL (Rajkumar et al., 2022), OpenTab (Kong et al., 2024a), MAC-SQL (Wang et al., 2025b), and MAG-SQL (Xie et al., 2024b).

Implementation Details. To align with baselines, we adopt GPT-3.5 and LLaMA 2 as the backbone LLMs, and additionally evaluate with the recent stronger models LLaMA 3.1, DeepSeek-V3, and GPT-4.1. Model configurations are detailed in Appendix H. Prompts include few-shot examples sampled from the training set, with illustrative cases shown in Appendix I.

4.2 Main Results

4.2.1 Overall Performance

Here, we compare CoQ with both generic and SQL-aided table understanding methods on three datasets involving multi-hop reasoning. As shown in Tables 1 and 2, CoQ consistently achieves both higher answer accuracy and lower SQL error rates. On WikiTQ, it reaches 74.77% with GPT-3.5, a +13.66% gain over the second-best. On TabFact, it attains 92.31%, exceeding the best baseline by +10.39%. On FeTaQA, CoQ reports the highest BLEU (22.19) and ROUGE scores (R-1: 0.65, R-2: 0.42, R-L: 0.54), and the lowest error rate (7.74%). We also assess how CoQ scales with stronger models in Table 3. These results highlight the effectiveness of CoQ, which is analyzed below.

Table 2: FeTaQA results with GPT-3.5. **BLEU** and **ROUGE** (R-1/2/L) evaluate answer quality. CoQ achieves the best results across BLEU, ROUGE, and error rate, enabled by its schema abstraction, clause-by-clause SQL generation, and hybrid reasoning.

Method	BLEU	R-1	R-2	R-L	Inv.
E2E QA	16.94	0.60	0.38	0.50	N/A
Tab2Text	9.43	0.40	0.22	0.33	N/A
CoTab	20.45	0.62	0.40	0.52	N/A
BT2SQL	16.92	0.60	0.37	0.49	14.23
OpenTab	18.19	0.60	0.37	0.49	10.71
MAC-SQL	17.56	0.58	0.35	0.47	13.98
MAG-SQL	18.81	0.60	0.37	0.48	11.13
CoQ (Ours)	**22.19** (+1.74)	**0.65** (+0.03)	**0.42** (+0.02)	**0.54** (+0.02)	**7.74** (-2.97)

4.2.2 Comparison Against Generic Methods

We now provide a detailed comparison with generic table understanding methods, which rely solely on LLMs' natural language capabilities without explicit SQL assistance. Their improvements are modest (e.g., Tree-of-Table improves accuracy by just 1.17% over Chain-of-Table on WikiTQ). This indicates a performance plateau for current methods. On the other hand, Table-to-Text methods linearize entire tables into textual descriptions to eliminate structured content. However, the generated text often exhaustively covers all table content, making it difficult for LLMs to locate key information. This leads to low accuracy and poor scalability on large tables. Empirically, the table-to-text baseline performs the worst among all methods (e.g., 16.07% on WikiTQ).

In contrast, CoQ consistently achieves substantial improvements across all datasets and LLM backbones. For example, it outperforms Tree-of-Table on WikiTQ by +13.66% with GPT-3.5 and +14.90% with LLaMA 2. Similar trends hold for TabFact and FeTaQA. These results confirm the effectiveness of CoQ's SQL-aided design in enhancing the accuracy and robustness of structured reasoning. Additionally, its natural-language-style table schemas abstract away structural noise to avoid forcing LLMs to interpret tabular structures or irrelevant content.

4.2.3 Comparison Against SQL Methods

We also compare CoQ with SQL-aided baselines and find that it consistently achieves higher accuracy and lower SQL error rates across all datasets and model backbones. A key limitation of existing SQL-aided methods lies in their rigid, one-shot formulation: they attempt to generate a single SQL query that directly yields the final answer. This approach works reasonably well for traditional Text-to-SQL tasks, where answers typically correspond to certain cell values. However, in table understanding, answers often require multi-step reasoning, abstraction, or comparisons across multiple rows or columns. Forcing all logic into one complex SQL query in such settings leads to bloated and error-prone programs. For example, while MAG-SQL performs well on standard Text-to-SQL benchmarks (Xie et al., 2024b), it underperforms on reasoning-intensive table understanding tasks. MAG-SQL only achieves 55.87% accuracy and a 9.48% invalid SQL rate on WikiTQ with GPT-3.5, compared to 74.77% accuracy and just 3.34% invalid SQL with CoQ. Similar trends are observed on TabFact and FeTaQA.

In contrast, CoQ benefits from our Clause-by-Clause SQL Generation, which ensures that generated queries remain concise and valid. Our Hybrid Reasoning Division further ensures that queries are simplified but aligned with the reasoning scope by offloading higher-level inference to the LLM. Moreover, our natural-language-style table schemas abstract away structural noise, reducing the LLM's burden in interpreting complex layouts for SQL generation. As a result, CoQ achieves improved reliability without compromising reasoning depth.

Table 3: Results on WikiTQ with LLaMA 3.1, DeepSeek-V3, and GPT-4.1. CoQ delivers greater gains than upgrading to stronger LLMs, enabled by hybrid reasoning and robust SQL generation.

Method	**LLaMA 3.1**		**DeepSeek-V3**		**GPT-4.1**	
	Acc.	Inv.	Acc.	Inv.	Acc.	Inv.
Table-to-Text	16.41	N/A	48.39	N/A	50.05	N/A
Few-Shot QA	39.94	N/A	66.07	N/A	70.97	N/A
Chain-of-Table	54.17	N/A	71.64	N/A	75.78	N/A
Basic Text-to-SQL	44.13	16.71	64.75	12.10	70.79	7.50
MAG-SQL	45.49	11.44	65.79	9.14	71.71	5.82
CHAIN-OF-QUERY (Ours)	**62.18** (+8.01)	**5.62** (-5.82)	**81.85** (+10.21)	**2.30** (-6.84)	**84.92** (+9.14)	**1.54** (-4.28)

Table 4: Results on IM-TQA and Open-WikiTable. Top-K = accuracy at top-K candidates. Note that Top-K accuracy is reported as a decimal rather than a percentage. CoQ maintains SOTA performance on structurally complex and multi-table datasets, enabled by its natural-language-style schema abstraction.

Method	**IM-TQA**		**Open-WikiTable**			
	Acc.	Inv.	Top-1	Top-2	Top-5	Top-10
Few-Shot QA	52.47	N/A	0.336	0.357	0.383	0.388
Chain-of-Table	48.80	N/A	0.404	0.430	0.463	0.467
Basic Text-to-SQL	63.16	7.86	0.429	0.458	0.490	0.496
OpenTab	67.28	6.94	0.491	0.523	0.556	0.565
MAG-SQL	68.90	6.06	0.457	0.476	0.516	0.524
CHAIN-OF-QUERY (Ours)	**74.96** (+6.06)	**2.80** (-3.26)	**0.527** (+0.036)	**0.552** (+0.029)	**0.592** (+0.076)	**0.608** (+0.083)

4.3 Scalability to Stronger LLMs

Our initial experiments used GPT-3.5 and LLaMA 2 to ensure fair comparisons with baselines. To assess how CoQ scales with stronger models, we further evaluate it on LLaMA 3.1, DeepSeek-V3, and GPT-4.1 using the WikiTQ dataset. As shown in Table 3, CoQ consistently outperforms other methods regardless of model strength. Notably, even with GPT-3.5, CoQ achieves 74.77% accuracy (Table 1), surpassing the performance of Few-Shot QA with GPT-4.1 (70.97%). This demonstrates that our framework contributes more to overall performance than simply upgrading to a more powerful language model. When paired with GPT-4.1, CoQ further boosts accuracy to 84.92% and reduces the invalid SQL rate to just 1.54%, confirming that its combination of hybrid reasoning and robust SQL generation remains effective with stronger LLMs.

4.4 Generalization to Real-World Tables

The previously explored datasets primarily feature clean, well-structured tables. However, real-world scenarios often involve messy, irregular tables. To evaluate CoQ's performance in such settings, we conduct additional experiments on the IM-TQA and Open-WikiTable datasets. IM-TQA contains structurally diverse (transposed, nested, and irregular formats) tables, while Open-WikiTable reflects multi-table databases in practical applications.

As shown in Table 4, CoQ maintains strong performance across both structurally complex and multi-table settings, consistent with our earlier results. A key contributor to this robustness is our natural-language-style schema, which abstracts away structural noise and allows LLMs to focus on semantic content rather than layout-specific details.

4.5 Ablation Study: Deep Dive into CoQ's Key Components and Mechanisms

We conduct an ablation study on WikiTQ using GPT-3.5, disabling one CoQ component at a time.

As shown in Table 5, removing natural-language-style schemas leads to a clear performance drop (–10.12%) and more invalid SQL (+1.13%). This highlights the benefit of abstracting away structural noise. Removing the Parallel Task Decom-

Table 5: Ablation results on WikiTQ (GPT-3.5), evaluating each CoQ component. CoQ achieves optimal performance through its modular design, with each component contributing significantly to accuracy and SQL validity.

Method	Acc.	Inv.
CHAIN-OF-QUERY	**74.77**	**3.34**
w/o Natural-Language-Schema	64.65 (-10.12)	4.47 (+1.13)
w/o Parallel Task Decomposition	72.95 (-1.82)	3.86 (+0.52)
w/o Clause-by-Clause SQL Generation	57.73 (-17.04)	8.82 (+5.48)
w/o Hybrid Reasoning Division	55.96 (-18.81)	5.27 (+1.93)

Table 6: Comparison of the number of LLM calls per question across methods on WikiTQ dataset. CoQ maintains high performance with low LLM usage, enabled by hybrid reasoning division that avoids over-generation.

Method	# Calls
Binder	50
Dater	100
Chain-of-Table	≤ 25
Tree-of-Table	≤ 29
MAC-SQL	$\leq$ **19**
MAG-SQL	≤ 25
CHAIN-OF-QUERY	$\leq \underline{22}$

position causes a minor accuracy drop (–1.82%), suggesting its limited impact on simple cases but usefulness in complex ones. Disabling the Clause-by-Clause SQL Generation causes a substantial drop in accuracy (–17.04%) and a sharp rise in invalid SQL (+5.48%), as clause-level generation supports incremental validation and accurate clause selection. This reduces the risk of incorrect queries and ensures appropriate SQL operations are applied. The largest drop (–18.81%) occurs without the Hybrid Reasoning Division, which prevents over-generation and delegates logical reasoning to the LLM. Without this control, the system tends to over-generate, increasing query complexity without improving answer quality.

Overall, the ablation results confirm that each CoQ component contributes meaningfully to performance, highlighting the effectiveness of its modular design in balancing precision and adaptability.

4.6 Analysis of LLM Usage

In addition to accuracy, we examine CoQ's LLM usage by analyzing the upper bound of calls required per question (Table 6). Among all methods, CoQ offers a favorable trade-off between reasoning depth and LLM usage, requiring at most 22 calls. In contrast, other strong baselines require up to 25–100 calls due to extensive branching and fixed steps. While MAC-SQL is slightly more efficient (within 19 calls), it underperforms on complex tasks. Notably, we report upper bounds here; in practice, **CoQ averages just 7.63 LLM calls per instance** (Appendix E). This low average stems from our Hybrid Reasoning Division: by delegating logical reasoning to the LLM, CoQ avoids generating overly complex SQL and reduces error correction. It issues only the essential clauses needed to retrieve sufficient information, minimizing execution overhead.

5 Conclusion

We introduce CHAIN-OF-QUERY, a multi-agent framework for SQL-aided table understanding, built upon three key insights: natural-language-style schema, clause-by-clause generation and hybrid reasoning division. They enable robust, interpretable, and precise reasoning over tables. CoQ achieves SOTA performance across five benchmarks, demonstrating strong generalization with low error rates and few LLM calls. It offers insights into effective integration of reasoning with structured query assistance in complex tabular tasks.

6 Limitations

While our experiments span five benchmarks across diverse domains in both English and Chinese, and CHAIN-OF-QUERY demonstrates strong generalization across datasets within these two languages, its effectiveness in other languages remains unexplored. Future work could extend the framework to multilingual settings and evaluate its adaptability to a wider range of linguistic phenomena.

7 Ethics Statement

This work uses five publicly available datasets: WikiTQ, TabFact, FeTaQA, IM-TQA, and Open-WikiTable, all of which are widely used in prior research and contain no personally identifiable information. No additional data collection or human annotation was performed. Our experiments use GPT-3.5, GPT-4.1, LLaMA 2, and DeepSeek-V3

in standard inference-only settings. While we are not aware of specific ethical concerns related to our datasets or methods, we acknowledge potential risks: the reliance on pretrained LLMs may propagate social or cultural biases, and large-scale model deployment could raise environmental concerns.

References

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, and 12 others. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

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

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)*, Addis Ababa, Ethiopia.

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. *ICLR*, abs/2210.02875.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 181 others. 2025. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Xinyi He, Yihao Liu, Mengyu Zhou, Yeye He, Haoyu Dong, Shi Han, Zejian Yuan, and Dongmei Zhang. 2025. Tablelora: Low-rank adaptation on table structure understanding for large language models. *arXiv preprint arXiv:2503.04396*.

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.

Deyi Ji, Lanyun Zhu, Siqi Gao, Peng Xu, Hongtao Lu, Jieping Ye, and Feng Zhao. 2024. Tree-of-table: Unleashing the power of llms for enhanced large-scale table understanding. *arXiv preprint arXiv:2411.08516*.

Kezhi Kong, Jiani Zhang, Zhengyuan Shen, Balasubramaniam Srinivasan, Chuan Lei, Christos Faloutsos, Huzefa Rangwala, and George Karypis. 2024a. Opentab: Advancing large language models as open-domain table reasoners. *arXiv preprint arXiv:2402.14361*.

Kezhi Kong, Jiani Zhang, Zhengyuan Shen, Balasubramaniam Srinivasan, Chuan Lei, Christos Faloutsos, Huzefa Rangwala, and George Karypis. 2024b. Opentab: Github repository. GitHub Repository, accessed 2024-05-16.

Sunjun Kweon, Yeonsu Kwon, Seonhee Cho, Yohan Jo, and Edward Choi. 2023. Open-wikitable: Dataset for open domain question answering with complex reasoning over table. *Preprint*, arXiv:2305.07288.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024a. Codes: Towards building open-source language models for text-to-sql. *Preprint*, arXiv:2402.16347.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Preprint*, arXiv:2305.03111.

Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2024b. Table-gpt: Table fine-tuned gpt for diverse table tasks. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

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

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

Dehai Min, Nan Hu, Rihui Jin, Nuo Lin, Jiaoyan Chen, Yongrui Chen, Yu Li, Guilin Qi, Yun Li, Nijun Li, and Qianren Wang. 2024. Exploring the impact of table-to-text methods on augmenting llm-based question answering with domain hybrid data. *Preprint*, arXiv:2402.12869.

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*, 10:35–49.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305*.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Preprint*, arXiv:2304.11015.

Mohammadreza Pourreza and Davood Rafiei. 2024. Dts-sql: Decomposed text-to-sql with small large language models. *Preprint*, arXiv:2402.01117.

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. *Preprint*, arXiv:1910.10683.

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

Jiawei Shen, Chengcheng Wan, Ruoyi Qiao, Jiazhen Zou, Hang Xu, Yuchen Shao, Yueling Zhang, Weikai Miao, and Geguang Pu. 2025. A study of in-context-learning-based text-to-sql errors. *Preprint*, arXiv:2501.09310.

Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. 2023a. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. *Web Search and Data Mining*.

Yuan Sui, Jiaru Zou, Mengyu Zhou, Xinyi He, Lun Du, Shi Han, and Dongmei Zhang. 2023b. Tap4llm: Table provider on sampling, augmenting, and packing semi-structured data for large language model reasoning. *Conference on Empirical Methods in Natural Language Processing*.

Chang-You Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. Exploring chain-of-thought style prompting for text-to-sql. *Preprint*, arXiv:2305.14215.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025a. Mac-sql: Github repository. GitHub Repository, accessed 2024-05-16.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and 1 others. 2025b. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024a. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6).

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. 2024b. Chain-of-table: Code repository. GitHub Repository, accessed 2024-05-16.

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. 2024c. Chain-of-table: Evolving tables in the reasoning chain for table understanding. *ICLR*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*.

Wenxuan Xie, Gaochen Wu, and Bowen Zhou. 2024a. Mag-sql: Github repository. GitHub Repository, accessed 2024-05-16.

Wenxuan Xie, Gaochen Wu, and Bowen Zhou. 2024b. Mag-sql: Multi-agent generative approach with soft schema linking and iterative sub-sql refinement for text-to-sql. *arXiv preprint arXiv:2408.07930*.

Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, and Xia Hu. 2023. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *Preprint*, arXiv:2304.13712.

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 preprint arXiv:2301.13808*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Tianping Zhang, Shaowen Wang, Shuicheng Yan, Jian Li, and Qian Liu. 2023a. Generative table pretraining empowers models for tabular prediction. *arXiv preprint arXiv:2305.09696*.

Tianshu Zhang, Xiang Yue, Yifei Li, and Huan Sun. 2023b. Tablellama: Towards open large generalist models for tables. *arXiv preprint arXiv:2311.09206*.

Mingyu Zheng, Yang Hao, Wenbin Jiang, Zheng Lin, Yajuan Lyu, QiaoQiao She, and Weiping Wang. 2023. IM-TQA: A Chinese table question answering dataset with implicit and multi-type table structures. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5074–5094, Toronto, Canada. Association for Computational Linguistics.

Wei Zhou, Mohsen Mesgar, Annemarie Friedrich, and Heike Adel. 2025. Efficient multi-agent collaboration with tool use for online planning in complex table question answering. *Preprint*, arXiv:2412.20145.

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

A Case Study

Table 7: Case study: Fabrice Santoro's Grand Slam results and win–loss record.

Name	2001	2002	n_win_loss
Australian Open	2R	3R	22–18
French Open	2R	2R	17–20
Wimbledon	2R	1R	11–14

Question: Did Fabrice Santoro win more at the Australian Open or Wimbledon?

SQL by MAG-SQL:

```
WITH Wins_and_losses AS (
    SELECT name,
        CAST(SUBSTR(n_win_loss, 1, INSTR(n_win_loss, '-') - 1) AS INT) AS wins
    FROM Fabrice_Santoro
    WHERE name IN ('Australian Open','Wimbledon')
)
SELECT
    name
FROM Wins_and_losses
ORDER BY wins DESC
LIMIT 1;
```

• SQL result:

Name
Australian Open

SQL by CoQ:

```
SELECT name, n_win_loss
FROM Fabrice_Santoro
WHERE name IN ('Australian Open','Wimbledon');
```

• SQL result:

Name	n_win_loss
Australian Open	22-18
Wimbledon	11-14

• Then the LLM compares 22-18 vs. 11-14 and answers: Australian Open.

Figure 3: Comparison of MAG-SQL vs. CHAIN-OF-QUERY.

This example highlights the fundamental difference between MAG-SQL and CHAIN-OF-QUERY in handling structured reasoning.

As shown in Figure 3, MAG-SQL attempts to answer the question entirely within SQL. It constructs a nested query that parses the `n_win_loss` field (e.g., 22-18) to extract the number of wins using string manipulation functions like `SUBSTR` and `INSTR`. It then casts the result to integer and selects the entry with the highest number of wins. While functional, this query is long, brittle, and tightly coupled to a specific string format. Any minor format mismatching would cause execution failure or incorrect results. This reflects a common issue with traditional SQL-aided approaches: in pursuit of returning the final answer in one shot, they overcomplicate query logic and increase error risk.

CoQ, in contrast, adopts a different strategy. It first generates a simple SQL query that merely retrieves the relevant rows and raw `n_win_loss` records. Then, the LLM compares the values ("22–18" vs. "11–14") and reasons that the Australian Open had more wins. This hybrid division—SQL for extraction, LLM for comparison—reduces SQL complexity and improves robustness, as the LLM is better suited to handle slight variations or contextual interpretations in text-formatted values.

In short, MAG-SQL complicates SQL to achieve end-to-end reasoning, whereas CoQ simplifies SQL and leverages the LLM where it excels, demonstrating the strength of our Hybrid Reasoning Division Strategy.

B Examples of CHAIN-OF-QUERY

B.1 SQL Generation Example Using Parallel Decomposition

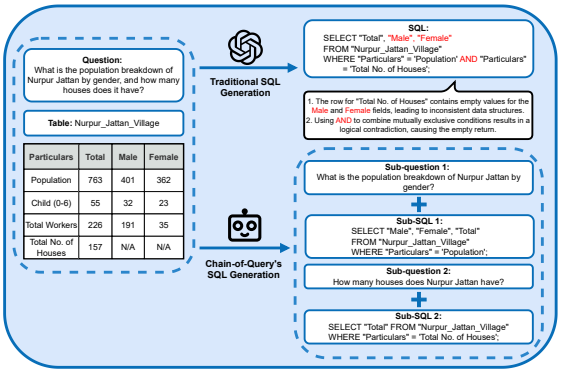

Figure 4: Comparison of traditional vs. CHAIN-OF-QUERY SQL generation via parallel sub-question decomposition.

Figure 4 illustrates the difference between traditional SQL generation and our CHAIN-OF-QUERY framework using parallel sub-question decomposition. In the traditional approach, the LLM is required to answer the entire question using a single SQL query, which leads to structurally complex queries. Due to limited understanding of both the table structure and SQL semantics, the model may incorrectly use conjunctions like AND to combine incompatible conditions. This often results in logical inconsistencies or empty outputs, especially when the involved rows contain mismatched structures or missing values.

In contrast, CHAIN-OF-QUERY decomposes the original question into two independent subquestions, each focusing on a distinct aspect of

the table. These sub-questions are processed in parallel and translated into simpler, more focused SQL queries. This strategy improves robustness, avoids conflicting constraints, and simplifies table access by localizing reasoning to narrower regions.

B.2 SQL Generation Example with Early Stopping

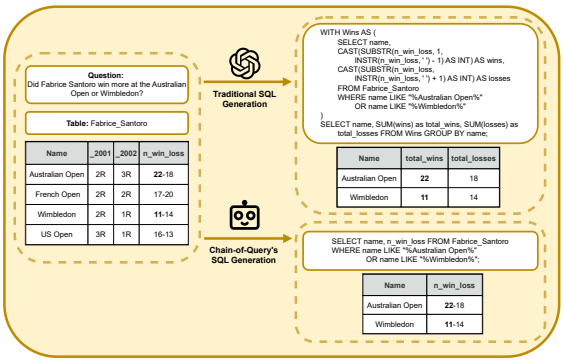

Figure 5: Example of Sufficiency-based Early Stopping in SQL generation.

Figure 5 illustrates the difference between traditional SQL generation and our CHAIN-OF-QUERY framework with the Sufficiency-based Early Stopping Mechanism. While the traditional approach constructs a complex query involving nested operations (e.g., `WITH`, `CAST`, `SUBSTR`) to compute explicit win/loss counts, our method halts early once sufficient information has been retrieved. Specifically, our method issues a simpler SQL query that directly extracts raw win–loss strings (e.g., "22–18", "11–14") from the relevant rows. The LLM then completes the comparison based on these outputs. This early stopping mechanism reduces unnecessary query complexity and enhances robustness by delegating the final reasoning step to the language model. Notably, despite its increased complexity, the traditional query yields essentially the same informational content as the simpler query used in our approach. Moreover, such complexity introduces more opportunities for execution failures and semantic mismatches.

C Clause Option List of Clause-by-Clause SQL Generation Strategy

Based on common patterns of information retrieval and reasoning operations in table understanding tasks, we define five types of SQL clauses used in our clause-by-clause generation strategy:

- `SELECT-FROM` clause: Serves as the foundational component of an SQL query, used to select task-relevant columns from the table.
- `WHERE` clause: Filters table rows based on specific conditions relevant to the question.
- `WITH AS` clause: Defines a Common Table Expression (CTE), enabling the creation of temporary virtual tables for intermediate transformations. This includes generating new columns or modifying existing ones without altering the original table. Such operations help organize and extract new information to support downstream reasoning.
- Aggregate function clause: Applies aggregation functions such as `COUNT`, `SUM`, `AVG`, `MAX`, and `MIN` to summarize or compute over table content. These operations are often essential for high-level reasoning and abstraction.
- `ORDER BY` clause: Sorts the table based on specified columns, facilitating reasoning that depends on ranking or positional relationships in the data.

D Algorithm of Sufficiency-based Early Stopping Mechanism

Algorithm 1: SUFFICIENCY-BASED EARLY STOPPING

Data: (Q, T), where Q is a natural language question; $T = (S, D)$ is a table consisting of schema S and data content D.
Result: $\hat{D}$ is the extracted subset of table's data content used to answer the question.

Function *Sufficiency-Early-Stop* (Q, T)*:*
  chain $\leftarrow$ `[Generator(Q, T)]`
  repeat
    sql $\leftarrow$ chain$[-1]$
    $D' \leftarrow$ `ExecuteSQL(T, sql)`
    plan $\leftarrow$ `Planner(Q, S, sql, D')`
    if plan $\neq$ *STOP* **then**
      next_sql $\leftarrow$ `Generator(Q, T, chain, plan)`
      chain.append(next_sql)
  until plan $=$ *STOP*
  $\hat{D} \leftarrow D'$
  return $\hat{D}$

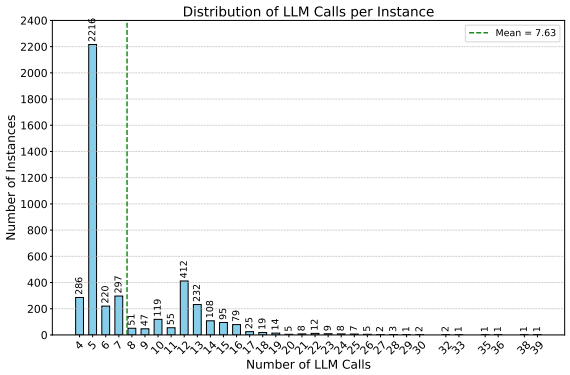

Figure 6: Empirical LLM Calls per Instance (WikiTQ). CoQ completes most instances within very few LLM calls, enabled by hybrid reasoning, early stopping, and clause-by-clause generation.

E Empirical Statistics on LLM Calls per Instance

For WikiTQ, as shown in Figure 6, CoQ requires **on average only 7.63 LLM calls per instance**, with a median of just 5. Notably, 57% of the instances complete within 5 calls, and only 0.16% exceed 30 calls, with the maximum observed at 39. These statistics highlight the low cost of our approach, especially when compared to other multi-agent baselines that incur significantly higher LLM usage.

This low usage arises from two key factors. First, our Hybrid Reasoning Division and Early Stopping prevent the construction of overly long SQL queries by stopping clause generation once sufficient information has been retrieved, avoiding unnecessary or overly complex SQL clauses. Second, our Clause-by-Clause Generation allows the LLM to focus on one clause at a time rather than constructing an entire SQL query at once. This not only simplifies generation at each step but also substantially reduces the chance of syntax or logic errors, minimizing the need for repeated LLM corrections. Together, these strategies ensure that our method maintains high SQL generation quality and minimizes redundant LLM usage.

F Dataset Details

We use five publicly available datasets in our experiments: **WikiTQ**, **TabFact**, **FeTaQA**, **IM-TQA**, and **Open-WikiTable**. Below we summarize their features, sources, formats, licensing, and usage details.

F.1 WikiTQ

WikiTQ (Pasupat and Liang, 2015) is a question answering dataset derived from Wikipedia, a resource with broad topical coverage. It contains 2,108 tables and 22,033 natural language questions, each paired with an answer derivable from a single table. Answers typically correspond to one or more table cells.
License: CC BY-SA.
Language: English.
Split Used: Standard test set (2,273 examples) following prior work.
Content: No personally identifiable or offensive content observed.

F.2 TabFact

TabFact (Chen et al., 2020) is a table-based fact verification dataset over Wikipedia tables covering over 16 domains. Each example includes a table and a natural language statement, labeled as either entailed or refuted. It comprises 117,854 examples with binary labels.
License: CC BY-SA.
Language: English.
Split Used: Standard test set (12,779 examples) following prior work.
Content: No personally identifiable or offensive content observed.

F.3 FeTaQA

FeTaQA (Nan et al., 2022) is a complex table QA dataset focused on multi-hop reasoning. Each entry includes a Wikipedia table, a natural language question, and a free-form textual answer. It focuses on free-form, multi-step reasoning beyond cell-level lookup.
License: CC BY-SA.
Language: English.
Split Used: Standard test set (2,003 examples) following prior work.
Content: No personally identifiable or offensive content observed.

F.4 IM-TQA

IM-TQA (Zheng et al., 2023) is a table QA dataset that is built by collecting tables from open websites of more than 10 domains. It features complex table styles, including hierarchical, nested, and messy real-world structures. Each entry includes a structurally complex table, several natural language questions, and answers relevant to certain cells.

License: CC BY-SA.
Language: Chinese.
Split Used: Standard test set (464 examples) following prior work.
Content: No personally identifiable or offensive content observed.

F.5 Open-WikiTable

Open-WikiTable (Kweon et al., 2023) is a dataset for open domain question answering with complex reasoning over multi-table settings. Each entry includes a natural language question, a table related to the question, and the answer.
License: CC BY-SA.
Language: English.
Split Used: Standard test set (6,602 examples) following prior work.
Content: No personally identifiable or offensive content observed.

F.6 Usage and Compliance

We use all datasets strictly for academic research in inference-only mode, without additional annotation or redistribution. Usage is fully aligned with the datasets' intended purposes and license terms.

G Reproducibility Statement

G.1 CHAIN-OF-QUERY

We conduct all experiments using the following models as the backbone LLMs: GPT-3.5-turbo (Brown et al., 2020), GPT-4.1 (OpenAI et al., 2024), LLaMA-2-13B (Touvron et al., 2023), and DeepSeek-V3 (DeepSeek-AI et al., 2025). Model configurations are provided in Appendix H. Prompt examples for each agent in the CHAIN-OF-QUERY framework are provided in Appendix I.

G.2 Baselines

Our implementation is based on the same environment as Chain-of-Table (Wang et al., 2024b). To ensure fair comparison, we directly reuse the GPT-3.5 and LLaMA 2 results for the following baselines as reported in their paper: Few-Shot QA, Chain-of-Thought, Binder, and Dater.

We re-run OpenTab, MAC-SQL, and MAG-SQL using their official open-source implementations (Kong et al., 2024b; Wang et al., 2025a; Xie et al., 2024a), following the same inference settings described in their respective repositories.

We used the Table-to-Text prompt provided by (Min et al., 2024) to obtain text format tables as one baseline.

Prompt examples for Basic Text-to-SQL (BT2SQL) and End-to-End QA (E2E QA) are included in Appendices J and K, respectively.

G.3 Tooling and Package Settings

All experiments are conducted in a Python 3.10 environment using standard open-source libraries. We make no modifications to any third-party packages beyond parameter configuration.

For preprocessing and database operations, we use `pandas` for basic table parsing and manipulation with no manual adjustments, and access SQLite databases via the `sqlite3` module along with the lightweight `records` wrapper.

We further evaluated CoQ on a temporal multi-table setting based on the Open-WikiTable dataset, following the setup of the baseline (Kong et al., 2024b). This setup involves table retrieval from a large corpus using BM25 to select the top-k tables.

To evaluate natural language outputs, we report BLEU and ROUGE scores. BLEU is computed using the `nltk` library, while ROUGE is calculated using the `rouge-score` package, both with default settings. Accuracy is computed using the official evaluation scripts provided by the dataset authors.

Additionally, we use a custom script to compute the invalid SQL rate, defined as the proportion of model-generated SQL queries that fail to execute due to syntax or runtime errors.

H LLMs' Inference Configurations

To ensure the reproducibility and stability of LLM outputs across all agents in our framework, we carefully configure the decoding parameters for each model.

For `GPT-3.5-turbo` model, `GPT-4.1` model, and `DeepSeek-V3` model, we adopt a deterministic configuration: temperature is set to 0.0 and `top_p` to 1.0. This disables sampling and enforces greedy decoding, ensuring consistent outputs across repeated runs. All responses are generated using official APIs from OpenAI and DeepSeek. Results are reported from a single run, without sampling variability.

For `LLaMA 2` and `LLaMA 3.1`, we deploy the 13B-chat and the 8B-instruct models respectively using Hugging Face Inference Endpoints, hosted on 8 A100 GPUs. Due to platform constraints, exact settings of `temperature` = 0.0 and `top_p`

= 1.0 are not available. As a practical approximation, we set `temperature` = 0.01 and `top_p` = 0.9 for all agents, which yields nearly deterministic outputs while maintaining compatibility with the deployment platform. All LLaMA results are likewise reported from a single inference run.

I Prompts Examples of Chain-of-Query

Our prompts are uniform across datasets and not domain-specific.

I.1 Prompt of Parallel Decomposition (WikiTQ)

[Instruction]
Your task is to decompose a question into subquestions. The decomposition should be based on the presence of interrogative words (e.g., what, which, who, where, how, when, why) in the original question. You should write the subquestions in the format of a python list. Solve the task step by step if needed.

[Constraints]
Subquestions should be independent and self-contained. Avoid using pronouns like "he," "she," "it," or "they" if they refer to an entity introduced in another subquestion. Restate the entity or essential context in each subquestion so that it makes complete sense on its own. Ensure each subquestion is a complete, grammatically correct sentence.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** Subquestions: “‘python subquestions = ["first subquestion", "second subquestion", ...] “‘

I.2 Prompt of Sub-answer Generation (WikiTQ)

[Instruction]
Your task is to answer a question related to a given table based on the execution result attained by running SQLite. Solve the task step by step if you need to. The table schema, a few example rows, a SQLite query and the execution result will be provided. Assume that you can always find the answer, so you must give an answer that makes sense to the question based on the given table. Your answer should be as short as possible. Do not use sentences if one or two words will do.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** Answer: [Your answer]

I.3 Prompt of Final Answer Generation (WikiTQ)

[Instruction]
Your task is to generate a final, coherent answer by combining the provided subanswers. The final answer should fully and naturally address the original question, using information from all the subanswers. Integrate the information fluently into a single, cohesive sentence, without explicitly listing or numbering the subanswers. Write the final answer in clear, natural, and grammatically correct English. Solve the task step by step if needed. The original question, subquestions and subanswers will be provided.

[Constraints]
Use all the subanswers when forming the final answer. Do not simply list the subanswers separately; integrate them fluently into a single, cohesive sentence. If an entity is introduced clearly in one subanswer, you may refer back to it later using appropriate pronouns such as "he," "she," "it," or "they." Avoid repeating the full name or full description unnecessarily when a pronoun would maintain clarity and improve fluency. Ensure that the final answer sounds fluent and directly addresses the original question. If any subanswers overlap or feel redundant, merge and rephrase them appropriately. If relevant, you may keep specific numbers or data in your final answer to make it more informative, even if the numbers are not explicitly asked for in the question. Avoid run-on or grammatically incorrect structures even though the answer should be a single sentence.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** Answer: [Your answer]

I.4 Prompt of SELECT-FROM Clause Generation

[Instruction]
Your task is to fill in the missing column names in an incomplete SQLite query so that it extracts the columns required to interpret the question correctly. Solve the task step by step if you need to. The table schema, a few example rows, question, and an incomplete SQLite query will be provided.

[Constraints]
The SQLite does not need to directly answer the question. You must complete the SELECT clause so it contains: 1. All value columns relevant to the question. 2. All context columns needed to understand what each row represents (e.g., identifiers, categories, descriptions). Only insert the missing column names in the parentheses of the SELECT statement without modifying any other part of the given SQL query. Don't add other clauses like WHERE and GROUP BY. Always include context columns like description, name, category, or type when they are essential to interpreting the selected values.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** SQL: "'sql [the completed SQL] "'

I.5 Prompt of WHERE Clause Generation

[Instruction]
Your task is to fill in the WHERE clause in an incomplete SQLite query so that it extracts a useful subset of rows that are most relevant to answering the question, even if the query is not final. Solve the task step by step if you need to. The table schema, a few example rows, question, and an incomplete SQLite query will be provided.

[Constraints]
The SQLite does not need to directly answer the question. Insert the necessary condition(s) or subquery in the WHERE clause, do not modify any other part of the provided SQL query. Use relaxed fuzzy matching (e.g., LIKE, IN) as much as possible when unsure of exact values. When filtering based on a value from the question (e.g., a team, location, or result), ensure the corresponding column contains similar values that match the target entity. Avoid filtering on columns when it is unclear if they support the value mentioned in the question. Do not assume a column's semantics solely based on its name. Do not invent or hallucinate values unless they are clearly implied by the question and supported by the schema. Do not add additional SQL clauses such as GROUP BY or ORDER BY.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** SQL: "'sql [the completed SQL] "'

I.6 Prompt of WITH AS Clause Generation

[Instruction]
Your task is to write a WITH ... AS SELECT statement that restructures the original table to enable accurate computation or aggregation (e.g., summing, grouping). These computations are needed to answer a provided question. You should extract or combine relevant parts of one or more columns selected in the provided SQLite query to create new columns in the new table. Solve the task step by step if you need to. The original table schema, a few example rows, the question, and a basic SQLite query (choosing one or more columns) will be provided.

[Constraints]
The question requires an operation (such as summing numeric values) that cannot be performed correctly without splitting or concatenating current column(s). Use SQLite syntax and functions (such as SUBSTR, INSTR, or string concatenation) as needed. Be mindful of data types in the SELECT subquery to ensure correct comparisons, computations, and transformations. Preserve all columns selected by the provided SQLite query in the new table, even if they do not require transformation.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** SQL: “‘sql [the completed SQL] “‘

I.7 Prompt of Aggregate Function Clause Generation

[Instruction]
Your task is to choose exactly one aggregate function from the list (COUNT, AVG, MAX, MIN, SUM) and rewrite the provided SQLite query. This query retrieves information from a table to answer a given question. Solve the task step by step if you need to. The table schema, a few example rows, question, and a basic SQLite query will be provided.

[Constraints]
The SQLite does not need to directly answer the question. Only consider following aggregate functions: COUNT, AVG, MAX, MIN, SUM. Aggregate functions must not be nested. The aggregate function should be chosen based on the semantics of the question (e.g., "total" suggests SUM, "most" suggests MAX). Modify the basic query by applying the chosen function to the relevant column. Keep the structure of the original query intact, only apply aggregation where needed. Use aggregate functions properly over the whole table.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** SQL: “‘sql [the completed SQL] “‘

I.8 Prompt of ORDER BY Clause Generation

[Instruction]
Your task is to add an ORDER BY clause to the provided SQLite query. This query retrieves information from a table to answer a given question. Solve the task step by step if you need to. The table schema, a few example rows, question, and a basic SQLite query will be provided.

[Constraints]
The SQLite does not need to directly answer the question. Modify the query by applying an ORDER BY clause to the relevant column. The sorting order (ASC or DESC) should be determined based on the semantics of the question. Apply the ORDER BY clause to the relevant column. Ensure that the original query structure remains unchanged except for the addition of the ORDER BY clause.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** SQL: “‘sql [the completed SQL] “‘

I.9 Prompt of Dynamic Planner (Sufficiency)

[Instruction]
Your task is to decide whether the current SQL result is sufficient to answer the question. The SQLite query retrieves information from a table to answer a given question. Solve the task step by step if needed. The table schema, a few example rows, the question, a SQLite query and its result will be provided.

[Constraints]
Answer "Yes" if and only if the current SQL result is sufficient to answer the question or with reasonable interpretation. Otherwise, answer "No". Base your decision solely on the given question and the provided basic SQLite query. The SQLite query does not need to be a complete or final answer. If it includes most key information that allows someone to infer the answer correctly, "Yes" is still acceptable, even if the question suggests that additional filtering, aggregation, or sorting might be needed.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** Decision: [Yes or No]

I.10 Prompt of Dynamic Planner (WHERE Clause)

[Instruction]
Your task is to decide whether answering the question require using a SQL WHERE clause to filter rows in the table? Solve the task step by step if you need to. The table schema, the total number of rows, a few example rows, and the question will be provided.

[Constraints]
Filtering must use a WHERE clause. Don't consider other filtering methods like GROUP BY. Check the number of rows first. If the question refers to a subset like "top 10," but the entire table already contains only that subset (e.g., the table has exactly 10 rows), then no filtering is needed and the answer is No. Operations like ORDER BY, LIMIT, or aggregation across all rows do not count as filtering. You must make the decision. Output Yes or No as the decision.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** Decision: [Yes or No]

I.11 Prompt of WHERE Clause Correction

[Instruction]
When executing the SQLite query below, an error occurred in the WHERE clause. Please correct the WHERE clause based on the provided question, table schema and error information. Solve the task step by step if needed. Identify which conditions are irrelevant, overly strict, or mismatched. The corrected WHERE clause should retrieve rows that can help interpret the question accurately, even if loosely filtered. The table schema, a few example rows, question, the incorrect SQLite query and error information will be provided. Only fix the WHERE clause, do not change any other part of the query. After fixing the query, verify it carefully. If possible, include verifiable evidence in your analysis.

[Constraints]
Modify only the WHERE clause; do not alter SELECT, GROUP BY, or other clauses. Ensure that the condition(s) in WHERE match the logic required by the question. Use correct syntax and valid condition(s) based on the provided schema. Handle data types appropriately. e.g., avoid comparing numeric strings without conversion. Use relaxed fuzzy matching (e.g., LIKE, IN) as much as possible when unsure of exact values. If the SQL query might return no results because of overly strict or irrelevant conditions, modify it to be more flexible: - Remove conditions when value mismatches are likely. - Relax LIKE or inequality comparisons that don't match the table's value patterns. - Do not remove constraints that clearly align with the question. Remember: a runnable and interpretable query is better than an overly strict query that returns nothing.

[Response format]
Your response should be in this format: Analysis: **[Your analysis]** SQL: "'sql [the correct SQL] "'

J Prompt Example of Basic Text-to-SQL (WikiTQ)

[Instruction]
Given the table schema and three example rows out of the table, write a SQLite program to extract the sub-table that contains the information needed to answer the question. The SQLite does not need to directly answer the question. Assume you always have enough information when executing the SQLite. Output only the SQL, with no explanation.

[Response format]
Your response should be in this format: SQL: “‘sql [the completed SQL] “‘

K Prompt Example of End-to-End QA (WikiTQ)

[Instruction]
Your task is to answer a question related to a given table. Assume you can always find the answer, so you must give an answer that makes sense to the question based on the given table. Your answer should be as short as possible. Do not use sentences if one or two words will do. Output only the answer, with no explanation.