Weaver : Interweaving SQL and LLM for Table Reasoning

Anonymous ACL submission

Abstract

Querying tables with unstructured data is challenging due to the presence of text (or image), either embedded in the table or in external paragraphs, which traditional SQL struggles to process, especially for tasks requiring semantic reasoning. While Large Language Models (LLMs) excel at understanding context, they face limitations with long input sequences. Existing approaches that combine SQL and LLMs typically rely on rigid, predefined workflows, limiting their adaptability to complex queries. To address these issues, we introduce Weaver , a modular pipeline that dynamically integrates SQL and LLMs for table-based question answering (TableQA). Weaver generates a flexible, step-by-step plan that combines SQL for structured data retrieval with LLMs for semantic processing. By decomposing complex queries into manageable subtasks, Weaver improves accuracy and generalization. Our experiments show that Weaver consistently outperforms state-of-the-art methods across four TableQA datasets, reducing both API calls and error rates. Code and data publicly available [1].

1 Introduction

Tables play a critical role across various domains such as finance (e.g., transaction records), healthcare (e.g., medical reports), and scientific research. However, many real-world tables are **semi-structured** (Gupta et al., 2020a), containing a combination of structured fields and unstructured content (such as free-form text or images), which makes reasoning and information retrieval challenging. Extracting insights from such data demands both logical and semantic reasoning. While SQL and Python-based methods excel in handling structured data, they fall short in dealing with unstructured text, missing entries, or implicit inter-column relationships.

[1] https://anonymous.4open.science/r/weaver-84DB/

Question: Which Country had the most competitors?

1990 British Grand Prix

Rank	Driver	Constructor	Laps	TimeRetired
1	Alain Prost	Ferrari	64	1:18:31
2	Thierry Boutsen	Williams-Renault	64	39.092
3	Ayrton Senna	McLaren-Honda	64	43.088
4	Éric Bernard	Lola-Lamborghini	64	401:03:00
5	Nelson Piquet	Benetton-Ford	64	1:20:02
6	Aguri Suzuki	Lola-Lamborghini	63	1 Lap
7	Alex Caffi	Arrows-Ford	63	1 Lap
8	Jean Alesi	Tyrrell-Ford	63	1 Lap

Gold Answer: Italy

Figure 1: Table and Question Answer example from WikiTQ Dataset

Recent advances in Large Language Models (LLMs) have demonstrated strong capabilities in natural language understanding and contextual reasoning, opening new avenues for complex tasks. However, LLMs still face key limitations, particularly with long contexts and numerical or temporal reasoning. For instance, in the WikiTableQuestions (Pasupat and Liang, 2015) dataset, the query *"Which country had the most competitors?"* Figure 1 requires inferring the competitors' countries from a *"driver"* column—information not explicitly present. While traditional tools like SQL or Python cannot resolve such gaps, LLMs can leverage their pre-trained knowledge to do so. Yet, the subsequent grouping and counting by country is something LLMs struggle with but SQL handles well. Solving such queries effectively requires a hybrid approach that combines the strengths of LLMs and programmatic methods. This raises a key question: *Can SQL and LLMs be seamlessly interwoven?*

Some methods, such as Binder (Cheng et al., 2022) and BlendSQL (Glenn et al., 2024), integrate LLMs into SQL workflows by treating them as function calls. For example, Binder combines LLM reasoning with SQLite to support hybrid queries. While effective for simpler tasks, these approaches struggle with complex queries, as LLMs often fail to generate accurate multi-step logic. This highlights the need to decompose complex queries into

smaller, manageable steps. Other approaches, like H-Star (Abhyankar et al., 2024) and ReAcTable (Zhang et al., 2024), use programmatic techniques to prune tables but rely heavily on costly API calls. Meanwhile, methods like ProTrix (Wu and Feng, 2024) limit reasoning to just two steps, making them insufficient for multi-hop queries. These rigid pipelines often constrain LLMs to final answer extraction and cannot handle questions requiring external or implicit knowledge.

To address these challenges, we introduce a modular, planning-based framework that dynamically alternates between SQL for logical operations and LLMs for semantic reasoning. By decoupling these components, our approach overcomes the limitations of monolithic systems and significantly improves performance on complex Table QA tasks. The process begins with selecting relevant columns and generating natural-language descriptions to resolve schema inconsistencies. An LLM then generates a step-by-step reasoning plan, combining SQL queries for structured operations with LLM prompts for semantic inference or column augmentation. Each step produces an intermediate table, enabling transparent reasoning and easy backtracking. A final answer extraction step retrieves the result from the processed table. This flexible design integrates with standard database engines (e.g., MySQL, SQLite) and supports various deployment settings. Our approach offers the following key contributions:

- We propose *Weaver* , a modular and interpretable framework for hybrid query execution that dynamically decomposes complex queries into modality-specific steps (e.g., SQL, LLM, VLM) without manual effort.
- We conduct extensive experiments on multiple hybrid QA benchmarks, including multimodal datasets, showing that *Weaver* outperforms existing methods by large margins, particularly on complex, multi-hop reasoning queries.
- We introduce a query plan optimization strategy that improves execution efficiency with minimal accuracy loss. *Weaver* also stores all intermediate outputs, enabling transparency, effective human-in-the-loop debugging.

Our results show superior accuracy over existing baselines, especially for queries requiring implicit reasoning beyond explicit table values. By bridging structured and unstructured reasoning, our approach sets a new benchmark for complex Table QA, offering a scalable solution for real-world applications.

2 Our Approach

This study addresses question answering tasks over tables containing both structured and unstructured data. Each instance consists of a table (T), a user query (Q), an optional paragraph (P), and a predicted answer (A). Queries span multiple categories: short-form queries (WikiTQ) involving direct lookups or aggregations; fact-checking queries (TabFact (Chen et al., 2019)) that require claim verification; numerical reasoning queries (FinQA (Chen et al., 2021b)) necessitating multi-step calculations; and multi-modal queries (FinQA and OTT-QA (Chen et al., 2021a)) that demand reasoning over extensive textual contexts inside and outside tables. Complex queries, such as "Which country had the most competitors?", frequently require semantic inference when explicit data is unavailable. Previous approaches typically rely on single-step executions, limiting flexibility and interpretability. In contrast, our method dynamically integrates SQL for structured data operations and LLMs for semantic inference, providing adaptable and accurate query resolution.

2.1 SQL-LLM Weaver

We introduce Weaver , a novel methodology integrating SQL and LLM specifically designed for TableQA tasks involving complex semantic reasoning and free-form responses. Weaver operates through distinct, structured phases:

1. Pre-processing: We begin by preprocessing the tables to mitigate SQL-related errors due to naming conflicts and data inconsistencies. This involves renaming columns conflicting with SQL reserved words (e.g., `Rank`), removing special characters, and standardizing column names. Subsequently, an LLM identifies and extracts relevant columns for the query, generating descriptive metadata for these columns. This metadata clarifies schema interpretations, resolves formatting issues, and defines accurate data types, as illustrated in Figure 2. For external unstructured text, relevant information is retained using an LLM to ensure context alignment.

2. Planning: In this phase, an LLM generates a dynamic, step-by-step plan using few-shot prompt-

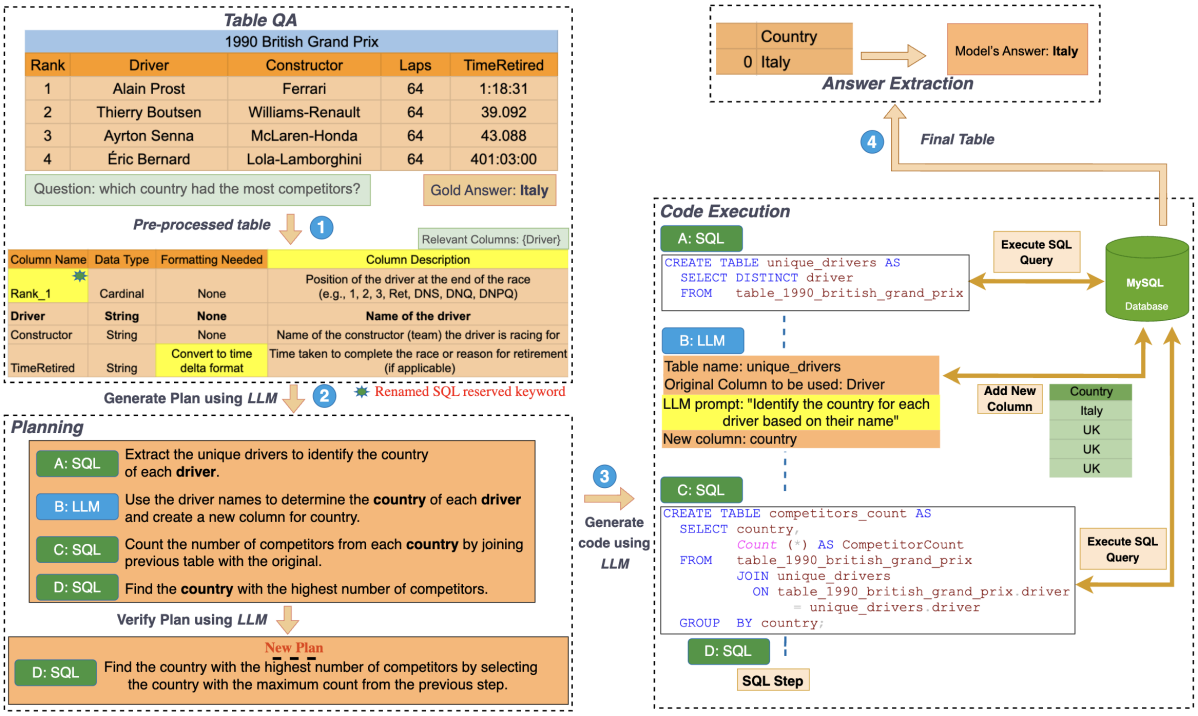

Figure 2: Step-by-step TableQA execution using Weaver approach.

ing based on the previously derived metadata. The plan consists of sequential subtasks, each categorized explicitly as either SQL or LLM operations.

(a.) SQL Step: SQL steps manage structured data tasks, including filtering rows, formatting column data types, mathematical operations and data aggregations. For example, Figure 2 demonstrates an SQL step that generates an intermediate table, "unique_drivers."

(b.) LLM Step: LLM steps handle tasks beyond SQL's capabilities, such as deriving new columns through semantic inference, sentiment analysis, or interpreting complex textual data. LLMs leverage either contextual paragraphs or their pretrained knowledge to perform these tasks. Each LLM step carefully integrates outputs back into the structured data tables to ensure coherence and consistency for subsequent SQL steps. Figure 2 illustrates how an LLM infers a "country" column from the "driver" column in the intermediate table "unique_drivers". The LLM is guided through structured prompts that leverage its pretrained knowledge and reasoning capabilities. Relevant information (extracted from external unstructured text in pre-processing step) is also passed to LLM if present.

Plan Verification. Prompting techniques like self-refinement (Madaan et al., 2023) and verification (Weng et al., 2023) are known to enhance LLM reasoning by reducing errors and improving consistency. To leverage this, we use a secondary LLM to verify the initial plan, ensuring its logical consistency, robustness, and completeness. Gaps such as insufficient reasoning or formatting issues are addressed by refining the plan, as shown in step (D) in Figure 2. This verification improves the pipeline's reliability and mitigates cascading errors.

3. Code Execution: Following verification, the pipeline executes the plan sequentially, combining SQL queries and LLM-generated prompts.

(a.) SQL Step - Query Generation: SQL operations involve formatting, filtering, joining, aggregating, and grouping data, with intermediate tables stored at each stage. SQL efficiently handles structured data operations, reducing reliance on LLM steps.

(b.) LLM Step - Prompt Generation: LLMs dynamically create prompts for operations exceeding SQL capabilities, such as generating new columns or interpreting textual insights. These LLM-generated prompts interact with structured intermediate tables from SQL steps, ensuring coherent integration.

Robust error handling and fallback mechanisms ensure pipeline robustness, utilizing the recent successful intermediate table if execution errors occur.

4. Answer Extraction: In the final pipeline stage, the intermediate table and user query are inputted to an LLM, which generates a natural language answer. Leveraging few-shot learning en-

sures output consistency and contextual accuracy, effectively resolving complex queries. Figure 2 illustrates this process with a sample TableQA example.

2.2 Optimization

We experimented to further enhance the efficiency of our pipeline, optimizing the planning strategy by minimizing unnecessary LLM calls and prioritizing SQL-based operations.

One key optimization involves pushing SQL operations such as filtering, aggregation, and formatting early in the pipeline as shown in Figure 3 in Appendix A.5. This reduces the volume of data that needs to be processed by the LLM, significantly reducing latency and computational overhead. Furthermore, parallelizing LLM inference across these optimized chunks further enhances efficiency, allowing the system to handle large-scale tabular reasoning tasks effectively, as shown Figure 3. To further streamline execution, sequential SQL steps are merged, reducing SQL calls and improving performance, see Figure 4 in Appendix A.5.

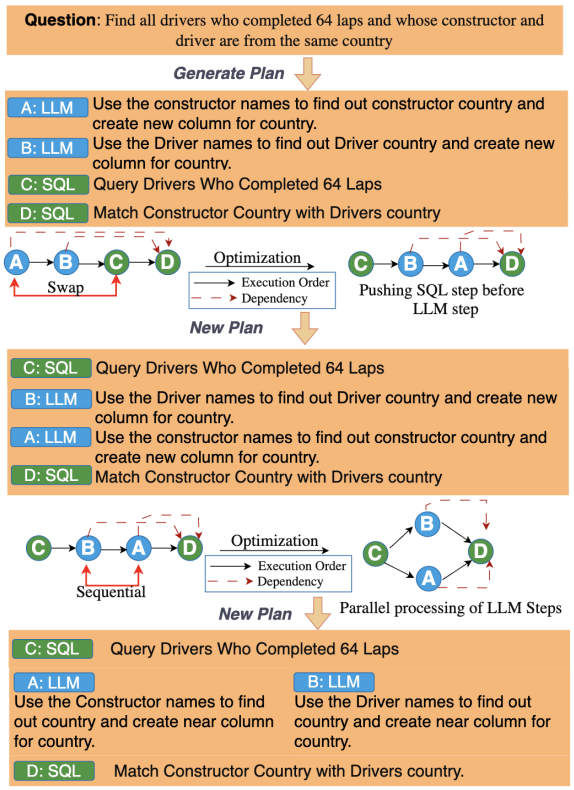

Figure 3: Optimization using SQL Reordering and LLM call parallelization.

Given the computational demands of LLMs for large tables we split data into smaller context-aware chunks before sending them to the LLM for inference. This prevents input truncation, maintains logical coherence between batches, and ensures optimal utilization of the context window of the model, as demonstrated in Figure 5, in the Appendix A.5. This optimization strategy further enhances *Weaver* planning and execution efficiency.

3 Experiments

Benchmarks. As hybrid multi-hop TableQA remains an emerging research area, there is no dedicated benchmark to evaluate such tasks. To fill this gap, we curate a hybrid subset by filtering relevant examples from several established table-based datasets for the evaluation of *Weaver* .

Source Datasets. For a comprehensive evaluation of *Weaver* 's ability to handle complex hybrid queries, we assess its performance across four diverse datasets: WikiTQ, TabFact (a fact verification dataset), FinQA (a numerical reasoning dataset), and OTT-QA (a short-form answering dataset). We also evaluated our approach on 3000 queries each of multimodal (MM) datasets, FinQA_{MM} and $\text{OTT-QA}_{\text{MM}}$, which require reasoning in both tabular data, and the accompanying textual context (usually a paragraph outside tables). We also evaluated Weaver on the $\text{MMTabQA}_{\text{MM}}$ dataset (Mathur et al., 2024), which involves reasoning over tables that include both text and images. This dataset contains 1,600 queries and 206 tables, with each query requiring the integration of textual and visual reasoning, including SQL, LLM, and VLM calls. Unlike traditional table QA tasks, these benchmarks challenge the model to integrate information from structured (tables) and unstructured (text) modalities. Details on the datasets in Appendix B.

Filtering Methodology. We define "hybrid" queries as those that require both SQL operations and LLM-based reasoning. These queries are more complex, as they necessitate not only structured data retrieval but also advanced reasoning capabilities, such as entity inference or free-text interpretation, which SQL alone cannot provide. To identify such queries, we use Binder-generated queries that incorporate *user-defined LLM functions (UDFs)*. Queries involving UDFs indicate the need for semantic reasoning beyond SQL's capabilities. It is important to note that we did not validate the correctness of Binder's outputs; its role was purely to flag queries with hybrid characteristics. These candidate queries were then manually validated to confirm they genuinely required both SQL execution and LLM reasoning steps to arrive at the correct answer. For FinQA, we utilized the "qa" ob-

ject to identify queries requiring multiple reasoning steps, excluding simple table lookups to ensure the selected queries involved more than just direct data retrieval. These flagged queries were then manually reviewed to confirm their need for both SQL execution and LLM reasoning.

Dataset Statistics. After filtering, the hybrid versions of the datasets consist of: (a) WIKITQ: 510 examples (original: 4,344), (b) TABFACT: 303 examples (original: 2,000), and (c) FINQA: 1,006 examples (original: 8,281). These represent the final queries filtered to create the hybrid versions: $\text{WikiTQ}_{\text{hybrid}}$, $\text{TabFact}_{\text{hybrid}}$, and $\text{FinQA}_{\text{hybrid}}$.

Evaluation Metrics Traditional Exact Match requires an exact match between the model's generated answer and the gold answer, which can unfairly penalize correct responses that differ only in format, for example, 2nd April 2024 and 04/02/2024 are semantically the same. To address this, we introduce Relaxed Exact Match (REM) metric which implements a three-step evaluation framework. First, we standardize the model's output to align with the gold answer, handling variations such as unit representation and common abbreviations. For example, if the model returns the final table with column- "Year" with value- "17", and the expected answer is "17 years", we transform the output to match the reference format. Once the answers are format-aligned, we apply the standard EM metric to determine whether the processed output matches the gold answer. However, automated transformations can sometimes introduce unintended errors, such as incorrect unit conversions or context misalignment. To prevent such issues, we also perform human evaluation to ensure accurate answer matching.

LLM Models. In our research, we use state-of-the-art large language models (LLM) such as Gemini-2.0-Flash (DeepMind, 2024), GPT-4o-mini-2024-07-18, GPT-4o-2024-08-06 (OpenAI et al., 2024) and the open-source DeepSeek R1-distill Llama-70B [2] (DeepSeek-AI et al., 2025), (Shi et al., 2024) for table reasoning tasks. Our model inputs include in-context examples, the table, and the question for each step of the pipeline.

3.1 Baseline Methods

We evaluated our approach against several baselines that are broadly categorized into 4 categories:

[2]https://github.com/meta-llama/llama- models/blob/main/-models/llama3_3/LICENSE

1. Query Engines (Binder, BlendSQL): These methods generate hybrid SQL queries with LLMs as *User Defined Functions*. They leverage SQL to interpret tabular data and execute queries to retrieve relevant information., **2. End-to-End LLM QA:** This approach leverages LLMs for question answering without intermediate query structuring. The model receives a query and table as input, generating answers based on learned patterns and reasoning. We employ GPT-4o, GPT-4o-mini, Gemini-2.0-Flash, and DeepSeek R1-distill LlaMA 70B for all tasks, **3. Pruning-Based Methods (ReacTable, H-Star):** These methods first apply SQL or Python-based pruning techniques, such as filtering columns or rows, before passing the refined table to an LLM for final answer extraction, and **4. Planning Based Approach (ProTrix):** ProTrix employs a two-step "Plan-then-Reason" framework. It first plans the reasoning, and assigns SQL to filter the table. Finally, it uses LLM to extract the final answer. By comparing our approach with these methods, we highlight the unique strengths of SQL-LLM Weaver , which combines SQL-based filtering with LLM-driven reasoning for more effective query resolution.

3.2 Results and Analysis

Our Weaver performs well on three challenging benchmarks; we present key findings next.

First, are Hybrid Queries harder? The results in Table 1 compare GPT-3.5-turbo on the original dataset with GPT-4o-mini on the hybrid set. Despite leveraging a more capable model (GPT-4o-mini) for the hybrid queries, we observe substantial performance drops—H-Star and Binder see accuracy declines of 9.5% and 32.7%, respectively, on $\text{WikiTQ}_{\text{hybrid}}$.

	Original (GPT-3.5)	Hybrid (GPT-4o-mini)
Binder	56.7%	24%
ReAcTable	52.4%	27%
H-Star	69.5%	59%
ProTrix	65.2%	61.4%

Table 1: Baselines result comparison on WIKITQ after filtering on hybrid part.

Notably, GPT-4o-mini outperforms GPT-3.5-turbo on benchmarks like MMLU and MATH (Source: OpenAI), yet still struggles on hybrid queries. This underscores their inherent difficulty and highlights the limitations of current methods in handling multi-step, semantically complex reasoning in Table QA.

Does *Weaver* Help? Table 2 demonstrates that Weaver consistently outperforms state-of-the-art baselines across all datasets. On WikiTQ, Weaver surpasses the best-performing baseline ProTrix by 5.5% across all four models. On TabFact, it achieves a breakthrough 91.2% using DeepSeek model, surpassing the 90% benchmark. On FinQA, it achieves accuracy of 65%, outperforming baselines by 4.6% on DeepSeek R1-distill Llama 70B.

	WikiTQ	**TabFact**	**FinQA**
	GPT-4o-mini		
End-to-End QA	60.4	84.4	44.7
*Binder**	24.0	62.0	13.0
BlendSQL	26.0	68.5	37.0
*ReAcTable**	29.9	37.4	-
H-Star	59.0	83.0	40.1
ProTrix	61.4	81.5	46.4
Weaver	**65.0**	**89.4**	**49.3**
	GPT-4o		
End-to-End QA	66.4	80.8	58.3
*Binder**	27.3	60.3	17.0
BlendSQL	42.0	68.3	34.3
*ReAcTable**	45.4	45.4	-
H-Star	61.0	**87.0**	46.0
ProTrix	61.7	80.5	54.3
Weaver	**70.7**	83.4	**60.8**
	Gemini-2.0-Flash		
End-to-End QA	67.5	81.8	29.4
*Binder**	12.9	60.4	21.3
BlendSQL	31.1	60.1	19.7
*ReAcTable**	20.4	37.6	-
H-Star	63.5	**86.1**	38.7
ProTrix	62.2	80.8	42.9
Weaver	**69.6**	85.4	**44.5**
	DeepSeek R1-distill Llama 70B		
End-to-End QA	**76.4**	82.5	52.4
*Binder**	26.4	62.7	24.4
BlendSQL	32.2	50.8	36.7
*ReAcTable**	52.2	45.6	-
H-Star	68.7	55.6	50.3
ProTrix	41.4	81.1	60.4
Weaver	73.0	**91.2**	**65.0**

Table 2: Experimental results for various models on short-form QA, fact verification, and numerical reasoning tasks. *: with self-consistency. Best result in **bold**, second-best in underlined. A hyphen (-) indicates missing results due to incompatibility or untested scenarios.

Weaver vs Query Engines. Binder and BlendSQL struggle with hybrid queries due to their rigid single-step execution framework. We observe that only 61% and 66% of the hybrid queries execute successfully in Binder and BlendSQL on WikiTQ. The reported accuracies for these methods are calculated based on successfully executed queries. Although BlendSQL slightly outperforms Binder with a modest 2% it frequently encounters type errors when integrating LLM-generated outputs into SQL operations. Weaver outperforms BlendSQL by 39%, 20.9% and 12.3% accuracy on WikiTQ, TabFact and FinQA using GPT-4o-mini.

Weaver vs Pruning Methods. H-Star and ReAcTable while effective for structured queries, perform poorly on semantic tasks. H-Star attains 59.0% and 63.5% accuracy on $\text{WikiTQ}_{\text{hybrid}}$ using GPT-4o-mini and Gemini-2.0-Flash, respectively, but struggles with row extraction. In some cases, its row-filtering heuristics discard critical contextual data essential for reasoning. H-Star's higher accuracy on TabFact with GPT-4o stems from the dataset's suitability for pruning techniques. [3] Furthermore, *Weaver* with GPT-4o-mini and DeepseekR1-Distill-LLAMA, despite their smaller size, performs competitively highlighting its effectiveness in resource-constrained environments where lightweight models are preferred.

Weaver vs Planning Method. ProTrix follows a two-step pipeline (planning and execution), achieves 61.4% and 62.2% on $\text{WikiTQ}_{\text{hybrid}}$ with GPT-4o-mini and Gemini-2.0-Flash. However, it fails in scenarios requiring intermediate semantic processing. For example, queries demanding dynamic column generation (e.g., inferring Country from Constructor) reveal its inability to seamlessly integrate SQL and LLM reasoning, leading to a 9% accuracy gap (GPT-4o) compared to Weaver .

Weaver on Multimodal Method. We evaluated Weaver on two multimodal datasets—FinQA_{MM} and $\text{OTT-QA}_{\text{MM}}$—requiring multi-hop reasoning over both structured (tables) and unstructured (text) data. As shown in Table 3, Weaver outperformed baselines, with notable improvements in FinQA_{MM} and moderate gains in $\text{OTT-QA}_{\text{MM}}$.

	FinQA_{MM}	$\text{OTT-QA}_{\text{MM}}$	FinQA_{MM}	$\text{OTT-QA}_{\text{MM}}$
	GPT-4o-mini		GPT-4o	
End2End QA	45.9	61.2	57.6	**68.7**
Weaver	**63.2**	**63.7**	**68.0**	65.2
	Gemini-2.0-Flash		DeepSeek R1	
End2End QA	37.9	64.1	54.8	59.9
Weaver	**60.8**	**66.7**	**66.2**	**62.8**

Table 3: Experimental results on short-form question answering on dataset with table and paragraphs.

In FinQA_{MM}, Weaver excelled in numerical and multi-hop reasoning, where end-to-end models struggled with irrelevant information and sequential logic, such as calculating net values. For $\text{OTT-QA}_{\text{MM}}$, which involved less structured computation and more world knowledge, Weaver still showed

[3] We didn't test ReAcTable on FinQA since its prompts are tailored to other sets; modifying them changes baseline.

consistent gains. Unlike baselines, Weaver effectively retrieved and integrated key table and paragraph segments, ensuring relevant information was used. These results highlight the strength of our modular, reasoning-focused approach, which integrates information step-by-step instead of relying on holistic attention.

On $\text{MMTabQA}_{\text{MM}}$ dataset, Weaver achieved an accuracy of 53.02% using gpt-4o-mini model, significantly outperforming the end-to-end QA baseline, which scored 46.33%. These results highlight the strength of our modular, reasoning-driven approach, combining structured data (tables), unstructured data (text), and visual inputs (images) for superior performance. This underscores Weaver 's versatility and scalability in addressing complex multimodal (table, text, images) QA tasks.

Efficacy Analysis. We conducted an analysis to assess the effectiveness and scalability of Weaver in $\text{WikiTQ}_{\text{hybrid}}$, focusing on 98 large tables with over 30 rows and average token length of 17,731. Our results show that Weaver achieved an accuracy of 65.6%, outperforming ProTrix (37.5%) and H-Star (35.9%) by 28.1% and 29.7%, respectively, on these large tables Weaver maintained the same accuracy on the entire original dataset, demonstrating its ability to handle complex queries while remaining both scalable and robust. These results highlight Weaver 's ability to tackle a wide range of table-based tasks with consistent performance.

In addition to delivering reliable performance, *Weaver* offers the critical advantage of transparent, interpretable reasoning. By following a structured execution plan, it ensures that final answers are tightly aligned with preceding reasoning steps, enhancing traceability and reducing spurious outputs. This directly addresses a core limitation of large language models—hallucination and memorization. Unlike end-to-end LLMs, which may produce correct answers without valid reasoning, *Weaver* only yields correct outputs when the underlying plan is sound, ensuring both reliability and interpretability.

Efficiency Analysis. Table 4 demonstrate the efficiency of our proposed Weaver framework using number of API calls. We make six API calls which are much lower compared to approaches that use self-consistency (Binder) with 50 calls and H-Star which uses ~ 8 calls to reach the answer.

ProTrix uses only two fixed API calls and relies on the LLM solely for generating the plan. However, it does not involve the LLM during execution. This limits its ability to handle multi-step queries requiring reasoning at each step. For instance, it may fail to infer information from individual rows or perform operations such as applying a SQL GROUP BY on an LLM-inferred 'country' column. Such steps are often essential to arrive at the correct final answer. These metrics demonstrate how our approach minimizes computational overhead while maintaining accuracy.

API calls/ Query	GPT-4o	GPT-4o-mini	Gemini-2.0
ProTrix	2	2	2
Binder	50	60	53
H-STAR	8	8	8
Weaver	5.31	5.87	5.85

Table 4: Number of API calls comparison per TableQA.

#LLM Optimization Effect		
#LLMs Drops	15	
#SQL Drops	19	
#SQL Merge	113	
#SQL Reorder	4	
	Before Opt.	**After Opt.**
#LLMs	74	59
#SQL	532	513
#Total Steps	616	469
Accuracy (%)	65	64

Table 5: Effect of optimization on GPT-4o-mini plans on WIKITQ. Opt. stands for plan optimization.

Optimization: We experimented with optimizing our planning and execution strategy on 200 TableQA queries. Table 5 demonstrates how our optimization strategy focuses on reducing unnecessary computational steps without compromising accuracy. To achieve this, we implemented several techniques.

1. LLM Step Reduction: By identifying and eliminating redundant LLM steps, we reduced the 15 LLM calls. This optimization ensures that LLMs are only used when necessary, lowering computational costs.

2. SQL Step Optimization: We achieved a reduction of 19 SQL steps by eliminating unused operations. Additionally, we merged 113 sequential SQL steps into fewer, more efficient queries and reordered 4 steps to optimize execution flow, making it more efficient.

Optimizing GPT-4o-mini reduces LLM steps by 20% and SQL steps by 24.8%, significantly improving efficiency. The accuracy slightly drops from 65% to 64%, but this trade-off is minimal, especially under practical constraints. Our goal is to create a scalable TableQA pipeline that balances

accuracy with computational cost, particularly for large tables. The optimization achieves this by maintaining modular reasoning while reducing latency, API calls, and input tokens, demonstrating that efficiency gains don't sacrifice performance.

Error Analysis Table 6 illustrates the effectiveness of our approach in minimizing errors in SQL execution and plan generation. Our approach reduces SQL errors by 30% and plan generation by 86% compared to ProTrix in both GPT-4o, GPT-4o-mini and Gemini-2.0-Flash. However, GPT-4o-mini exhibits a higher SQL error rate due to its smaller model size, which limits its ability to generate accurate SQL queries.

	GPT-4o	GPT-4o-mini	Gemini-2.0
		ProTrix	
SQL Error	51.2%	25.9%	27.0%
Plan Generation	11.0%	17.0%	9.0%
		Weaver	
SQL Error	15.0%	42.5%	16.0%
Plan Generation	1.0%	3.0%	1.0%

Table 6: Error in SQL and Plan Generation on WIKITQ.

In Weaver , SQL errors arise due to incorrect formatting, unsupported MySQL functions, or hallucinated columns and tables. Planning errors arise when SQL steps replace LLM reasoning or generate unused tables. The Plan Verification Step, Figure 2, mitigates these issues by refining planning for improved reliability in complex table-based reasoning.

Analysis Across Pipeline Stages: We perform a stage-wise analysis to assess the contribution of each component in our pipeline—*filtering*, *planning*, and *execution*. Compared to SQL-only generation, which struggles with multi-step reasoning, our pipeline yields consistent accuracy gains by structuring the task into sub-components. The *filtering* stage removes irrelevant columns, reducing noise and guiding the model's attention. The *planning* stage—central to our method—decomposes complex queries into symbolic and semantic steps. This step is essential and not ablatable. However, skipping plan verification (i.e., executing without validating) leads to a 1% drop in accuracy, indicating that verification adds robustness. Finally, *execution* stage translates structured plans into SQL.

4 Comparision with Related Work

Table-based Question Answering (TableQA) combines table understanding, question interpretation, and NLP. Foundational work such as Text-to-SQL (Rajkumar et al., 2022), Program-of-Thought (Chen et al., 2023), and TabSQLify (Nahid and Rafiei, 2024b) laid the groundwork. Binder and TAG (Biswal et al., 2024) expose the limitations of traditional Text-to-SQL methods in handling complex analytical tasks involving both structured and unstructured data. To address these challenges, several alternative approaches have been explored:

Fine-Tuning Methods: These methods fine-tune LLMs to specialize in reasoning over hybrid tabular and textual data. Models such as (Zhu et al., 2024), (Mittal et al., 2024), and (Patel et al., 2024) are trained to extract, reason, and execute over such inputs. However, fine-tuning requires large task-specific datasets and tends to lack generalization across domains.

Query Engines: This direction integrates LLMs with SQL engines via user-defined functions (UDFs), allowing LLM calls within queries. UQE (Dai et al., 2024), BlendSQL, SUQL (Liu et al., 2024a), and Binder follow this paradigm. While flexible, LLM-generated queries can be error-prone, and these systems often support only limited query structures, reducing adaptability.

Table Pruning and Planning: Approaches like H-Star, ReAcT, ProTrix, and others (Liu et al., 2024b; Nahid and Rafiei, 2024a) enhance efficiency by programmatically pruning rows or columns using SQL or Python. While this reduces processing overhead, these methods often function as black boxes, lacking transparency and vulnerable to cascading errors if early pruning steps are incorrect.

5 Conclusion

We introduce Weaver , a novel approach for table-based question answering on semi-structured tables. Weaver outperforms all baselines by strategically decomposing complex queries into a sequence of LLM- and SQL-based planning steps. By alternating between these modalities, it enables precise, interpretable, and adaptive query resolution. Weaver overcomes prior limitations by effectively handling both complex queries and large tables. Its modular design also supports future extensions, including image-based tables, multi-table reasoning, and integration with free-form text. As future work, we plan to explore fine-tuning and supervision strategies to further improve execution accuracy and plan reliability.

Limitations

While our approach demonstrates strong performance across multiple datasets, it is currently limited to English-language tables, restricting its applicability to multilingual settings. Additionally, our method does not explicitly handle hierarchical tables, where multi-level dependencies introduce additional complexity in reasoning. Another limitation is the inability to process multi-table queries, which require reasoning across multiple relational structures. Furthermore, the lack of well-established benchmarks for hybrid datasets poses a challenge in evaluating and further improving performance in more complex, real-world scenarios.

Ethics Statement

We, the authors, confirm that our research adheres to the highest ethical standards in both research and publication. We have thoughtfully addressed various ethical considerations to ensure the responsible and equitable use of computational linguistics methodologies. In the interest of reproducibility, we provide detailed resources, including publicly available code, datasets (compliant with their respective ethical standards), and other relevant materials. Our claims are supported by the experimental results, although some degree of stochasticity is inherent in black-box large language models, which we mitigate by using a fixed temperature. We also offer thorough details on annotations, dataset splits, models used, and prompting techniques to ensure that our work can be reliably reproduced. We used AI assistants to help refine the writing and improve clarity during the drafting and revision process. No content was generated without human oversight or verification.

References

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

Asim Biswal, Liana Patel, Siddarth Jha, Amog Kamsetty, Shu Liu, Joseph E. Gonzalez, Carlos Guestrin, and Matei Zaharia. 2024. Text2sql is not enough: Unifying ai and databases with tag.

Wenhu Chen, Ming-Wei Chang, Eva Schlinger, William Wang, and William W. Cohen. 2021a. Open question answering over tables and text.

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

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. *arXiv preprint arXiv:1909.02164*.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan Routledge, and William Yang Wang. 2021b. FinQA: A dataset of numerical reasoning over financial data. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3697–3711, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022. Binding language models in symbolic languages. In *The Eleventh International Conference on Learning Representations*.

Hanjun Dai, Bethany Yixin Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya Mangpo Phothilimthana, Charles Sutton, and Dale Schuurmans. 2024. Uqe: A query engine for unstructured databases.

Google DeepMind. 2024. Gemini 2.0.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, and et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

Parker Glenn, Parag Pravin Dakle, Liang Wang, and Preethi Raghavan. 2024. Blendsql: A scalable dialect for unifying hybrid question answering in relational algebra.

Vivek Gupta, Maitrey Mehta, Pegah Nokhiz, and Vivek Srikumar. 2020a. INFOTABS: Inference on tables as semi-structured data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2309–2324, Online. Association for Computational Linguistics.

Vivek Gupta, Maitrey Mehta, Pegah Nokhiz, and Vivek Srikumar. 2020b. INFOTABS: Inference on tables as semi-structured data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2309–2324, Online. Association for Computational Linguistics.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts.

Shicheng Liu, Jialiang Xu, Wesley Tjangnaka, Sina Semnani, Chen Yu, and Monica Lam. 2024a. SUQL: Conversational search over structured and unstructured data with large language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 4535–4555, Mexico City, Mexico. Association for Computational Linguistics.

Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E. Gonzalez, Ion Stoica, and Matei Zaharia. 2024b. Optimizing llm queries in relational workloads.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback.

Suyash Vardhan Mathur, Jainit Sushil Bafna, Kunal Kartik, Harshita Khandelwal, Manish Shrivastava, Vivek Gupta, Mohit Bansal, and Dan Roth. 2024. Knowledge-aware reasoning over multimodal semi-structured tables. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 14054–14073, Miami, Florida, USA. Association for Computational Linguistics.

Akash Mittal, Anshul Bheemreddy, and Huili Tao. 2024. Semantic sql – combining and optimizing semantic predicates in sql.

Md Mahadi Hasan Nahid and Davood Rafiei. 2024a. Normtab: Improving symbolic reasoning in llms through tabular data normalization.

Md Mahadi Hasan Nahid and Davood Rafiei. 2024b. Tabsqlify: Enhancing reasoning capabilities of llms through table decomposition.

OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, and et al. 2024. Gpt-4o system card.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1470–1480, Beijing, China. Association for Computational Linguistics.

Liana Patel, Siddharth Jha, Parth Asawa, Melissa Pan, Carlos Guestrin, and Matei Zaharia. 2024. Semantic operators: A declarative model for rich, ai-based analytics over text data.

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

Yucheng Shi, Peng Shu, Zhengliang Liu, Zihao Wu, Quanzheng Li, Tianming Liu, Ninghao Liu, and Xiang Li. 2024. Mgh radiology llama: A llama 3 70b model for radiology.

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. 2023. Large language models are better reasoners with self-verification.

Zirui Wu and Yansong Feng. 2024. ProTrix: Building models for planning and reasoning over tables with sentence context. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4378–4406, Miami, Florida, USA. Association for Computational Linguistics.

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

Fengbin Zhu, Ziyang Liu, Fuli Feng, Chao Wang, Moxin Li, and Tat Seng Chua. 2024. Tat-llm: A specialized language model for discrete reasoning over financial tabular and textual data. In *Proceedings of the 5th ACM International Conference on AI in Finance*, ICAIF '24, page 310–318, New York, NY, USA. Association for Computing Machinery.

A Appendix: LLM Prompts and Examples

A.1 Prompt Examples

Listing 1: Extract Relevant Column

```
Given column descriptions, Table and Question return a list of columns that can be
relevant to the solving the question (even if slightly relevant) given the table
name and table:
table name: { self.name }

table: { self.table }
Question: { self.question }

Example output: [ 'Score', 'Driver']
Instructions:
1. Do not provide any explanations, just give the cols as a list
2. The list will be used to filter the table dataframe directly so take care of that

Output:
```

Listing 2: Column Description Prompt

```
Give me the column name, data type, formatting that needs to be done, column
descriptions in short for the given table and question. The descriptions should be
useful in planning steps that solve the question asked on that table. Also, give a
small description of the table using table name and table data given.
Table:
table name: { self.name }
{ self.table }
Question: { self.question }
```

Listing 3: Planning Prompt

```
I need a step-by-step plan in plain text for solving a question, given column
descriptions and table rows. Follow these guidelines:
Begin analyzing the question to categorize tasks that require only SQL capabilities
(like straightforward data formatting, mathematical operations, basic aggregations)
and those that need LLM assistance (like summarization, text interpretation, or
answering open-ended queries).
MySQL Query Generation: For parts of the question that involve formatting of column
data type, filtering and mathematical or analytical tasks, generate SQL query code
to answer them directly, without using an LLM call.
LLM-Dependent Task Identification: For tasks that SQL cannot inherently
perform, specify the columns or portions of rows where LLM calls are needed. Add an
extra column in the result set to store the LLM output for each row in the filtered
data subset.
Example -
<Table Name>
<Table>
Question: <Question>
<Column Descriptions>
<Plan>

Solve for this:
Table:
table name: { self.name }
{ self.table }
Question: { self.question }
{ self.description }
Only give the step-by-step plan and remove any other explanations or code.
Output format:
Step 1: SQL
Step 2: Either SQL or LLM
Step 3: ...
Step 4: ...
```

Listing 4: Verify Plan Prompt

```
Suppose you are an expert planner verification agent.
Verify if the given plan will be able to answer the Question asked on this table.
Table name: { self.name }
Table: { self.table }
Column descriptions: { self.description }
Question to Answer: { self.question }
Old Plan: { self.plan }
Is the given plan correct to answer the Question asked on this table (check format
issues and reasoning steps) should be able to guide the LLM to write correct code
and get correct result.
If the plan is not correct, provide better plan detailed on what needs to be done
handling all kinds of values in the column.
- Check if the MySQL step logic adheres to the column format. (Performs calculations
    and formatting and filtering in the table)
- The LLM step's logic will help in getting the correct answer.
If the original plan is correct then return that plan.

Do not provide with code or other explanations, only the new plan.
Output format:
Step 1: Either SQL or LLM - ...
Step 2: SQL or LLM - ...
Step 3: SQL ...

As given in original plan.
```

Listing 5: Code Execution Prompt

```
  MySQL Code Generation:   For parts of the question that involve data
formatting, data manipulations such as filtering, grouping, aggregations, and
creating new tables. Generate optimized MySQL code to
answer those parts directly without using an LLM.

LLM-Dependent Tasks Identification: For tasks that SQL cannot inherently perform
that require sentiment analysis, logical inferences, or questions that involve
interpreting text data, specify only that 1 column where LLM calls are needed. Add
an extra column in the table that stores the LLM output for each row in the filtered
data subset.

 Instructions:
1. Store the output at each step by creating a new table. Use this new table for the
    next steps.
2. The code for MySQL should handle all values in the column (formatting and
   filtering). New columns
from previous LLM steps can be assumed present in table.
3. Don't give any other explanations, only MySQL and LLM steps as the given plan.

Then, Only give step (SQL or LLM) that is needed -
The Output format example -
 Step 1  - SQL: MySQL code, table name to be used in the next query
 Step 2  - LLM:
- Reason: Why we need to use LLM
- Table name:
- original column to be used:
- LLM prompt: The prompt that user can use to solve the problem
- New column name:
 Step 3  - SQL: MySQL code, table name to be used in the next query
 Step 4  - ...
 Step 5  - ...

LLM step format should be same.
Solve for this question, given table and step by step plan as a reference:
Table name: { self.name }
Schema: { self.table.columns }
Column Descriptions: { self.description }
Table: { self.table }
Question: { self.question }
Plan: { self.plan }

First check if taking above Plan will give the desired output.
```

```
Give me code for solving the question, and no other explanations. Keep in mind the
    column data formats (string to appropriate data type, removing extra character,
    Null values) while writing Mysql code.
```

Listing 6: LLM Step Prompt

```
Given a column and step you need to perform on it -
Column: { df.column }
Step to solve the question: { step.prompt }
Question: { self.question }

 Instructions:
- Do not provide any explanation and Return only a list (separate values by '#')
    that can be added
to a dataframe as a new column in a dataframe.
- Do not create a column name already present in the table. (duplicate column)
- Any value should not be more than 3 words (or each value should be as short as
    possible).
- Size of output list Should be same as input list.
```

Listing 7: Answer Extraction Prompt

```
Table: { self.name }
{ self.table }
Question: { self.question }

Answer the question given the table in as short as possible.
If the table has just one column or value consider that as the answer given the
    column name.
Just provide the answer, do not provide any other information.
```

Listing 8: Plan Optimization Prompt

```
You are an expert in SQL and plan optimization. Your task is to optimize the given
SQL plan while ensuring it correctly answers the given question. Use the following
optimization strategies, but only if they maintain correctness:

SQL Merge: Merge sequential SQL steps where possible (e.g., combining filtering,
    aggregation, and sorting in one query).
SQL Reordering: Reorder SQL steps to filter early before applying computationally
    expensive operations like LLM processing.
LLM Merge: Merge sequential LLM steps where the operation is on the same column.
Given Information: ...
Plan: { self.plan }
```

A.2 Table and Question Example

Below is an example of a table and question used for LLM planning.

Listing 9: Table and Question Example for LLM Planning

```
Table: New_York_Americans_soccer
            Year  Division League        Reg_Season               Playoffs
                National_Cup
0           1931       1.0    ASL        6th (Fall)             No playoff
    None
1    Spring 1932       1.0    ASL              5th?             No playoff    1st
    Round
2      Fall 1932       1.0    ASL               3rd             No playoff
    None
3    Spring 1933       1.0    ASL                 ?                      ?
    Final
4       1933/34        NaN    ASL               2nd             No playoff
              ?
5       1934/35        NaN    ASL               2nd             No playoff
              ?
6       1935/36        NaN    ASL               1st  Champion (no playoff)
              ?
```

```
7       1936/37       NaN    ASL     5th, National        Did not qualify
    Champion
8       1937/38       NaN    ASL  3rd(t), National             1st Round
              ?
9       1938/39       NaN    ASL     4th, National        Did not qualify
              ?
10      1939/40       NaN    ASL               4th             No playoff
              ?
11      1940/41       NaN    ASL               6th             No playoff
              ?
12      1941/42       NaN    ASL               3rd             No playoff
              ?
13      1942/43       NaN    ASL               6th             No playoff
              ?
14      1943/44       NaN    ASL               9th             No playoff
              ?
15      1944/45       NaN    ASL               9th             No playoff
              ?
16      1945/46       NaN    ASL               5th             No playoff
              ?
17      1946/47       NaN    ASL               6th             No playoff
              ?
18      1947/48       NaN    ASL               6th             No playoff
              ?
19      1948/49       NaN    ASL            1st(t)                 Finals
              ?
20      1949/50       NaN    ASL               3rd             No playoff
              ?
21      1950/51       NaN    ASL               5th             No playoff
              ?
22      1951/52       NaN    ASL               6th             No playoff
              ?
23      1952/53       NaN    ASL               6th             No playoff
    Semifinals
24      1953/54       NaN    ASL               1st  Champion (no playoff)
    Champion
25      1954/55       NaN    ASL               8th             No playoff
              ?
26      1955/56       NaN    ASL               6th             No playoff
              ?

 Question:  How long did it take for the New York Americans to win the National Cup
    after 1936?
```

A.3 Model Responses

Listing 10: Response to Sample Prompt 1 (Column Description)

```
### Table Description
The table ‘New_York_Americans_soccer‘ contains historical performance data for the
    New York Americans soccer team, detailing their standings in various leagues,
    playoff outcomes, and national cup results from 1931 to 1956. The data includes
    information on the year, division, league, regular season standings, playoff
    participation, and national cup results.

### Column Details

| Column Name        | Data Type   | Formatting Needed                    | Column
    Description                                                                  |
|--------------------|-------------|--------------------------|--------------|
| Year               | String      | Standardize to a consistent format    |
    Represents the year or season of the soccer performance. Some entries may need
    parsing to extract the year. |
| Division           | Float       | Convert to Integer (if applicable)    |
    Indicates the division in which the team played. Some entries are NaN and need
    handling. |
| League             | String      | Standardize to a consistent format    |
    Represents the league in which the team participated, primarily ASL (American
    Soccer League). |
```

```
| Reg_Season          | String      | Clean and standardize standings       |
    Indicates the team's regular season performance, which may include qualifiers
    like "t" or "?" that need clarification. |
| Playoffs            | String      | Standardize and clean                 |
    Indicates playoff participation and outcomes, with entries like "No playoff" and
     "Finals" that need to be standardized. |
| National_Cup        | String      | Standardize and clean                 |
    Indicates the outcome of the national cup, with entries like "Champion" or "1st
    Round" that need standardization. |
```

Listing 11: Response to Sample Prompt 2 (Planning Prompt)

```
Plan: Step 1: SQL - Standardize the Year column to a consistent format and extract
    the year from
entries like "Spring 1932" and "Fall 1932".
Step 2: SQL - Clean and standardize the National_Cup column to identify the years
    when the team won
the national cup.
Step 3: SQL - Filter the data to find the first year after 1936 when the
    National_Cup column
indicates a win.
```

Listing 12: Response to Sample Prompt 3 (Verify Plan Prompt)

```
New Plan: ### Revised Plan:
Step 1: LLM - Standardize the Year column to a consistent format by extracting the
    year from entries
like "Spring 1932" and "Fall 1932". Convert all entries to a four-digit year format
    (e.g., "1932"
instead of "Spring 1932").
Step 2: SQL - Clean and standardize the National_Cup column to identify winning
    entries. Define a
clear criterion for a "win," such as entries that contain "Champion" or "1st Round"
    (if applicable).
Step 3: SQL - Filter the data to find the first year after 1936 where the
    National_Cup column
indicates a win. Ensure to handle any NaN or ambiguous entries appropriately.
```

A.4 Code Execution Results

Listing 13: Response to Sample Prompt 4 (Code Execution Prompt) Step 1

LLM Step -

```
- Reason: Standardize the Year column to correct format.
- Table name: New_York_Americans_soccer
- original column to be used: Year
- LLM prompt: Extract the year from phrases like "Spring 1932" or "Fall 1932" and
    standardize all
entries to a YYYY format. Ensure the output is consistent across all entries.
- New column name: Year_Formatted.

Dataframe create after LLM Step

Year_Formatted Division League    Reg_Season              Playoffs
    National_Cup
0   1931       1.0      ASL        6th (Fall)            No playoff           None
1   1932       1.0      ASL              5th?            No playoff      1st Round
2   1932       1.0      ASL               3rd            No playoff           None
3   1933       1.0      ASL                 ?                    ?          Final
4   1933       NaN      ASL               2nd            No playoff              ?
5   1934       NaN      ASL               2nd            No playoff              ?
6   1935       NaN      ASL               1st  Champion (no playoff)              ?
7   1936       NaN      ASL     5th, National        Did not qualify       Champion
8   1937       NaN      ASL  3rd(t), National             1st Round              ?
9   1938       NaN      ASL     4th, National        Did not qualify              ?
10  1939       NaN      ASL               4th            No playoff              ?
11  1940       NaN      ASL               6th            No playoff              ?
12  1941       NaN      ASL               3rd            No playoff              ?
```

```
13   1942       NaN      ASL              6th             No playoff              ?
14   1943       NaN      ASL              9th             No playoff              ?
15   1944       NaN      ASL              9th             No playoff              ?
16   1945       NaN      ASL              5th             No playoff              ?
17   1946       NaN      ASL              6th             No playoff              ?
18   1947       NaN      ASL              6th             No playoff              ?
19   1948       NaN      ASL           1st(t)                 Finals              ?
20   1949       NaN      ASL              3rd             No playoff              ?
21   1950       NaN      ASL              5th             No playoff              ?
22   1951       NaN      ASL              6th             No playoff              ?
23   1952       NaN      ASL              6th             No playoff      Semifinals
24   1953       NaN      ASL              1st  Champion (no playoff)       Champion
25   1954       NaN      ASL              8th             No playoff              ?
26   1955       NaN      ASL              6th             No playoff              ?
```

Listing 14: Response to Sample Prompt 4 (Code Execution Prompt) Step 2

```
SQL Step -
CREATE TABLE standardized_national_cup AS
SELECT
    Year_Formatted,
    Division,
    League,
    Reg_Season,
    Playoffs,
    CASE
        WHEN National_Cup LIKE '%Champion%' THEN 'Win'
        WHEN National_Cup LIKE '%1st Round%' THEN 'Win'
        ELSE 'No Win'
    END AS National_Cup
FROM New_York_Americans_soccer;
Table created: standardized_national_cup

dataframe create after loading from standardized_national_cup ...
    Year_Formatted  Division League        Reg_Season               Playoffs
        National_Cup
0    1931       1.0    ASL        6th (Fall)             No playoff       No Win
1    1932       1.0    ASL              5th?             No playoff          Win
2    1932       1.0    ASL               3rd             No playoff       No Win
3    1933       1.0    ASL                 ?                      ?       No Win
4    1933       NaN    ASL               2nd             No playoff       No Win
5    1934       NaN    ASL               2nd             No playoff       No Win
6    1935       NaN    ASL               1st  Champion (no playoff)       No Win
7    1936       NaN    ASL     5th, National        Did not qualify          Win
8    1937       NaN    ASL  3rd(t), National              1st Round       No Win
9    1938       NaN    ASL     4th, National        Did not qualify       No Win
10   1939       NaN    ASL               4th             No playoff       No Win
11   1940       NaN    ASL               6th             No playoff       No Win
12   1941       NaN    ASL               3rd             No playoff       No Win
13   1942       NaN    ASL               6th             No playoff       No Win
14   1943       NaN    ASL               9th             No playoff       No Win
15   1944       NaN    ASL               9th             No playoff       No Win
16   1945       NaN    ASL               5th             No playoff       No Win
17   1946       NaN    ASL               6th             No playoff       No Win
18   1947       NaN    ASL               6th             No playoff       No Win
19   1948       NaN    ASL            1st(t)                 Finals       No Win
20   1949       NaN    ASL               3rd             No playoff       No Win
21   1950       NaN    ASL               5th             No playoff       No Win
22   1951       NaN    ASL               6th             No playoff       No Win
23   1952       NaN    ASL               6th             No playoff       No Win
24   1953       NaN    ASL               1st  Champion (no playoff)          Win
25   1954       NaN    ASL               8th             No playoff       No Win
26   1955       NaN    ASL               6th             No playoff       No Win
```

Listing 15: Response to Sample Prompt 4 (Code Execution Prompt) Step 3

```
SQL Step -
CREATE TABLE first_win_after_1936 AS
SELECT
```

```
    Year_Formatted,
    Division,
    League,
    Reg_Season,
    Playoffs,
    National_Cup
FROM standardized_national_cup
WHERE Year_Formatted > '1936' AND National_Cup = 'Win'
ORDER BY Year
LIMIT 1;
Table created: first_win_after_1936
dataframe create after loading from first_win_after_1936 ...
   Year_Formatted Division League Reg_Season               Playoffs National_Cup
0  1953     None    ASL        1st  Champion (no playoff)          Win
```

Listing 16: Response to Sample Prompt 5 (Answer Extraction Prompt)

```
Generated Answer: 17 years
Comparison Result: Yes
Actual answer: 17 years, model answer: 17 years
Answer matched: True
```

A.5 More Detail on Optimization

We have explored several optimization techniques to enhance the efficiency of our pipeline by reducing the number of steps generated during query execution. While some of these strategies are detailed in the section 2, we outline additional key techniques below:

SQL Merging. Figure 4 explains merging sequential SQL steps to optimize the pipeline's performance. Since SQL operations follow a logical structure, combining multiple steps into a single query does not compromise the correctness or integrity of the process. This consolidation reduces the overhead of executing individual queries and improves the overall efficiency of the pipeline by minimizing redundant operations and streamlining execution.

Listing 17: Example of Step Merging

```
Original Plan (Multiple SQL Steps):
SELECT * FROM table WHERE column = 'X';
SELECT * FROM table ORDER BY date DESC;

Optimized (Merged into a Single Step):
SELECT * FROM table WHERE column = 'X' ORDER BY date DESC;
```

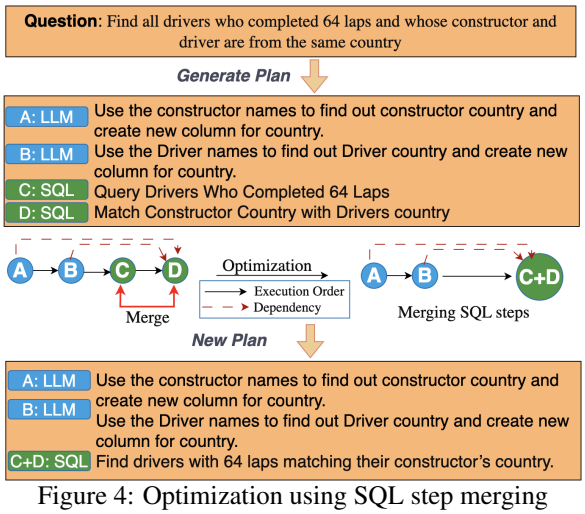

Figure 4: Optimization using SQL step merging

Parallel LLM Execution. Initially, we prompted the LLM to generate a new column by supplying the entire existing column and asking it to return a list of the same length. However, this approach often led to inconsistent results—such as incorrect list lengths, duplicated values, or hallucinated entries—due to the model's sensitivity to long input sequences. Errors were especially prevalent in the middle of the list, consistent with the "Lost in the Middle" effect (Liu et al., 2023).

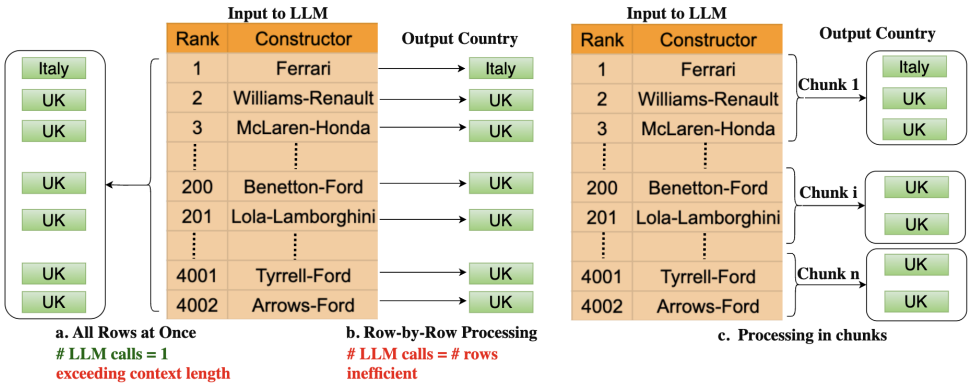

Figure 5: Optimizing LLM Calls: Chunk-based processing on Rows

To improve both reliability and efficiency, we adopted a chunk-wise parallel execution strategy that avoids the overhead of row-by-row processing while enhancing consistency. As illustrated in Figure 5, we segment the input into appropriately sized batches and execute multiple LLM calls in parallel. This design enables simultaneous inference over different parts of the data, substantially reducing latency by eliminating sequential processing bottlenecks. The result is faster response times and improved scalability—making this approach well-suited for large-scale reasoning tasks over semi-structured data.

Column-Wise Batching. Conventional LLM pipelines often chunk inputs row-wise, generating one column value per row across a batch. In contrast, we propose a column-wise batching strategy, depicted in Figure 6, which processes multiple columns for a small chunk of rows in a single call.

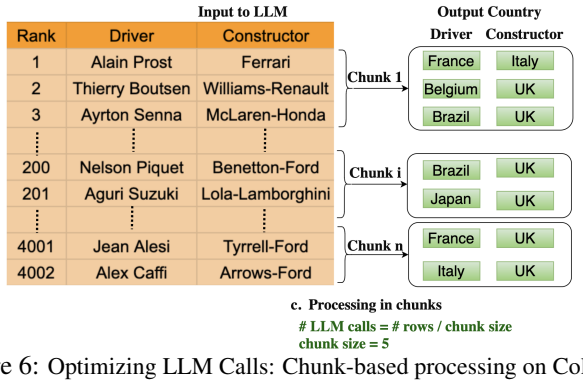

Figure 6: Optimizing LLM Calls: Chunk-based processing on Columns

This approach preserves intra-row context across multiple attributes of the same entity, reducing inconsistencies that arise when attributes are generated independently. It is particularly advantageous in Retrieval-Augmented Generation (RAG) systems and memory-augmented pipelines, where repeated LLM calls over fragmented inputs can be inefficient. By extracting all relevant information in one unified query, column-wise batching lowers computational costs while maintaining high accuracy in entity-level reasoning.

Listing 18: Examples of Different Plan Optimization

```
Question - The Kremlin Cup is held in Russia, and the St. Petersburg Open is also
    held in Russia.

Plan:
Step 1: SQL - Filter the table to select tournaments with the names "Kremlin Cup"
    and "St. Petersburg Open".
```

```
Step 2: SQL - Extract the country information from the Tournament column for the
    selected tournaments.

Step 3: LLM - Summarize the results to confirm that both tournaments are held in
    Russia.

Optimized Plan:
Step 1: SQL - Filter the table to select tournaments with the names "Kremlin Cup"
    and "St. Petersburg Open", and extract the country information from the
    Tournament column in a single query.

Step 2: LLM - Summarize the results to confirm that both tournaments are held in
    Russia.
```

A.6 More Detail on handling Multi-Modal data

The proposed pipeline Figure 8 is modular and designed for extensibility. Each component can be upgraded such as substituting the SQL Query Executor with an expert SQL agent to enhance execution efficiency and accuracy. Likewise, the LLM Semantic Reasoner and VLM (Vision Language Model) components can be replaced with specialized reasoning agents, allowing Weaver to adapt to evolving multi-modal requirements.

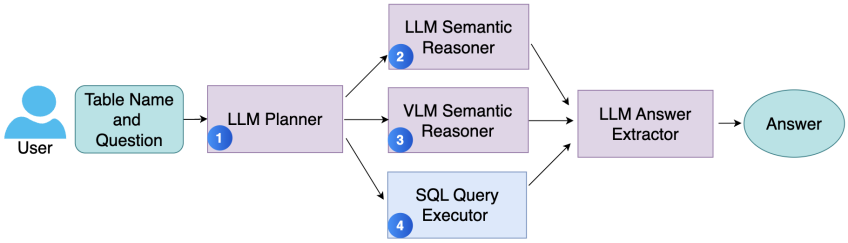

Figure 7: Modular Pipeline for query execution

Handling Paragraph data using LLM step To address unstructured textual content outside the table (i.e., paragraph data), we filter these texts for relevance to both the question and the tabular data. The filtered content is used as an auxiliary knowledge source during LLM steps. This enables the system to either incorporate the external text into the tabular context or leverage it directly in the final answer generation step, depending on the Planner Agent's discretion.

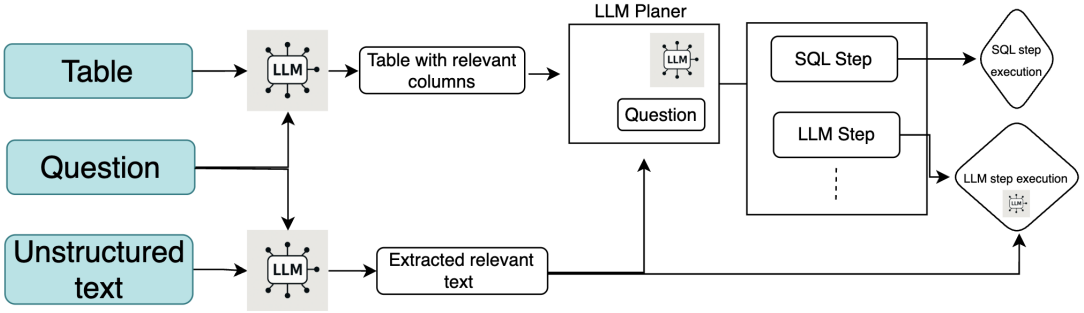

Figure 8: Modular Pipeline for query execution

This design supports robust integration of text, table, and visual data, ensuring that the system remains scalable, accurate, and adaptable across diverse input modalities.

A.7 Additional Analysis

Table 7 shows the number of LLM and SQL steps used by Weaver , across different models on WikiTQ.

Steps	GPT-4o	GPT-4o-mini	Gemini-2.0-Flash
LLM Steps	46	178	122
SQL Steps	1530	1407	1510
Total	1576	1585	1632

Table 7: Number of LLM steps and SQL steps used in *Weaver* on WikiTQ.

Token Usage Analysis: We compared the average token usage (input/output) per TableQA pair across ProTrix, H-STAR, and Weaver using three different LLM backends. As shown in Table 8, Weaver maintains competitive token efficiency, especially in output size, while enabling structured multi-step reasoning. Notably, ProTrix appears lightweight due to its limited planning, whereas H-STAR and Weaver consume similar token budgets despite Weaver offering higher accuracy. Binder is excluded from this comparison due to its excessive API usage (see Table 4).

Token Usage/Query	GPT-4o (I/O)	GPT-4o-mini (I/O)	Gemini-2.0 (I/O)
ProTrix	445.57 / 317.14	564.3 / 388.85	559.94 / 220.44
H-STAR	8,836 / 842	8,994 / 854	10,284 / 702
Weaver	8,568 / 725	8,723 / 796	6,041 / 545

Table 8: Token usage comparison per TableQA (Input/Output tokens).

Experimental Setup and Hyperparameters: All experiments were conducted using publicly available large language models (LLMs) accessed via their official APIs. We used the models in their standard configuration without any fine-tuning. Specifically, we used GPT-4o, GPT-4o-mini (OpenAI), Gemini-2.0-Flash, and DeepSeek R1 with a fixed temperature setting of 0.01 to ensure deterministic and stable outputs. No additional modifications were made to the default API settings beyond controlling temperature for consistency across runs.

B Dataset Details

In this section, we describe the used dataset in details.

- *Short-Form Answering (WikiTQ):* WikiTQ is a dataset designed for short-form answering where the steps to reach the answer can be relatively complex and the expected answer is a short piece of information. For example, the query "Which country had the most competitors?" in Figure 1. This task is ideal for testing how well information retrieval from a semi-structured table can be handled.
- *Fact-Checking (TabFact):* TabFact focuses on fact-checking tasks where the query involves verifying whether a particular statement is true or false. For example, the query, "Is the GDP of Japan in 2022 greater than that of Germany?" evaluates the framework's ability to correctly interpret data points in complex tables and make valid judgments.
- *Numerical Reasoning (FinQA):* FinQA (Financial QA) is a dataset that requires the model to perform arithmetic operations or infer relationships between numerical values across different columns. For example, "What is the total revenue of Company X in 2021 after deducting expenses?" tests the model's ability to handle numerical data and apply operations such as summation, subtraction, or other mathematical reasoning tasks.
- *Multimodal Dataset (FinQA$_{MM}$, OTT-QA$_{MM}$, MMTabQA$_{MM}$):* FinQA$_{\text{MM}}$ and OTT-QA$_{\text{MM}}$ are datasets that require reasoning across both tabular data and accompanying textual context (typically a paragraph) to answer questions correctly. Unlike traditional table QA tasks, these benchmarks challenge the model to integrate information from both structured (tables) and unstructured (text) modalities. For instance, in FinQA, financial metrics are found in the table, while their definitions, dependencies, or contextual cues are only available in the surrounding text. Similarly, OTT-QA includes open-domain trivia questions, where the relevant answers often span both the table and the associated paragraphs.

MMTabQA$_{\text{MM}}$ is a multimodal dataset that requires reasoning across both tabular data and visual information. Unlike traditional table QA tasks, it incorporates both text and images within its tables, challenging the model to integrate structured (tables), unstructured (text), and visual (images) modalities. The data set consists of query demands that combine textual descriptions, numerical data, and visual cues

from images. For example, a question might ask about the relationship between financial data in the table and trends depicted in an image. MMTabQA$_{MM}$ tests the model's ability to perform complex multimodal reasoning, requiring SQL, LLM, and VLM calls to accurately derive answers. This makes it a valuable benchmark for evaluating systems that integrate and reason over multimodal inputs.

C Few-Shot Prompting for Plan Generation

We use a few-shot prompting approach to generate plans during the planning stage. Through our analysis of various complex data problems, we identified three broad categories of transformation challenges that commonly arise. For each category, we manually crafted a representative example—rather than using examples from any particular dataset—to serve as in-context prompts. This was done to avoid memorization bias and to encourage cross-dataset generalization. These examples were specifically designed to capture the reasoning skills required for hybrid queries. We will include them in the appendix for clarity and reproducibility in the final version. Below are the three categories and the corresponding examples with sample tables:

- **Semantic reasoning from textual content**: These are cases where a column contains long or descriptive text, and we want the LLM to reason some semantic information which can be either direct extraction from text (e.g., extract topic from abstract text) or inference from text (e.g., infer sentiment from a text).

Listing 19: Few-shot examples for semantic reasoning from textual content

```
Table: grocery_shop

item_description         sell_price            buy_price
"Indulge your senses with this botanical blend of rosemary and lavender. Gently
    cleanses while nourishing your hair, leaving it soft, shiny, and revitalized."
      7.99   4.99

Question: Which item category has the highest average profit?

Plan:
Step 1: LLM - Item_category column needs to be created using item_description column
    .
Step 2: SQL - Calculate average profit for each category and find the maximum.
```

- **Commonsense or background knowledge inference**: In these cases, the required information is not explicitly present in the table but can be inferred using commonsense knowledge or facts the LLM is likely to have been trained on. We prompt the LLM to infer and populate a new column based on the existing data.

Listing 20: Few-shot examples for commonsense or background knowledge inference

```
Table: order_delivery_history

Order ID   Product   Event              Timestamp (Local)        Location
101        Laptop    Dispatched         2025-01-14 08:00 AM      Los Angeles, USA
101        Laptop    Arrived at Hub     2025-01-15 03:00 AM      Chicago, USA

Question: Which location had the maximum time taken between dispatch at one location
     and arrival or delivery at a subsequent location?

Plan:
Step 1: LLM - Convert local timestamps to UTC time for all events.
Step 2: SQL - Sort events within each Order_ID and Product by Timestamp_UTC.
Step 3: SQL - Pair Dispatched events with the corresponding Arrived at Hub or
    Delivered events for each order/product and calculate the time difference.
Step 4: SQL - Display the final output with paired events, durations, and relevant
    information sorted by Order_ID and Product.
```

- **Preprocessing or normalization of non-SQL-friendly columnse**: These are cases where a column contains values that are too diverse or unstructured to be directly used in a SQL query. Since the full table is not visible to the LLM and generating an exhaustive list of conditions is infeasible, we prompt the LLM to generate a cleaned or normalized version of the column that can be used in downstream SQL queries.

Listing 21: Few-shot examples for preprocessing or normalization of non-SQL-friendly columns

```
Table: Israel at the 1972 Summer Olympics

Name                Placing
Shaul Ladani        19
Esther Shahamorov   Semifinal (5th)
Listeravov Shamil   12th

Question: Who has the highest placing rank?

Plan:
Step 1: LLM - Format the column Placing by extracting only numerical values (e.g. 5
    from Semifinal (5th)) and converting the text into numbers (e.g. Semifinal to 5,
     12th to 12, 19 to 19).
Step 2: SQL - Retrieve the highest placing (rank) from the placing column by
    selecting the minimum number in the list as lower number corresponds to higher
    rank.
```

D Additonal Discussion

LLM Utility in Column Transformation and Semantic Inference: Our core contribution lies in how we effectively leverage LLMs beyond what traditional SQL systems can offer, specifically in tasks involving column transformation and semantic inference. Weaver integrates LLM reasoning for two primary purposes: (i) handling tasks that SQL cannot perform—such as entity extraction or parsing ambiguous formats A.2 and (ii) inferring implicit knowledge based on pretraining, where the LLM is tasked with verifying a plan that involves semantic understanding Figure 2.

A detailed example from the WikiTQ dataset in Appendix A.2(Listing 12) demonstrates the LLM's capability to normalize complex date strings like *"Spring 1932"* or *"1935/36"* into standardized `YYYY` formats. This transformation, impossible through SQL alone, was completed through a single LLM step with high consistency.

Listing 22: Example explaination

```
Processes complex date formats (e.g., "Spring 1932", "1935/36", "1937")
Standardizes them into YYYY format (1932, 1936, 1937 respectively)
```

Additionally, our hybrid planning enables seamless coordination between SQL and LLM steps, as seen in examples involving multi-column reasoning (e.g., from a movies table with columns movie name, review content, movie information, and release date).

Listing 23: Example explaination

```
Question: "What movies suitable for kids with positive reviews should be recommended
     based on their reviews after 2018?"

Plan:
SQL: Filter movies released after 2018 using the release_date column:

SELECT * FROM movies WHERE release_date > 2018;

LLM: Utilize a Large Language Model (LLM) to evaluate whether a movie is suitable
    for children by processing:
movie_info (structured metadata)
review_content (unstructured user reviews)
new column: suitable_movies
```

```
SQL: Apply SQL filtering to retain only movies flagged as suitable by the LLM:
SELECT * FROM movies WHERE suitable_movies = 'Yes';

LLM: For the filtered movies, check which movies have positive reviews based on:
movie_info
review_content
new column: recommended

SQL: Apply SQL filtering to retain only movies which are recommended:
SELECT * FROM movies WHERE recommended = 'Yes';
```

Beyond Semi-structured Text. We define semi-structured tables following prior work (Gupta et al., 2020b), focusing on unstructured or free-form content embedded within structured table formats—such as textual cells requiring semantic interpretation—rather than on hierarchical structures like JSON or HTML trees. While our current benchmarks focus on flat tables, the primary challenges we address stem from this embedded unstructured content. These challenges often require semantic inference, an area where traditional SQL struggles and large language models (LLMs) are critical in bridging the gap. Examples include inconsistencies in formatting, reliance on domain-specific knowledge, and implicit contextual cues necessary for accurate query interpretation.

Additionally, our approach is easily extendable to hierarchical data formats (e.g., HTML, nested JSON, HiTab) through structure decoding. These formats can be transformed into a flattened, normalized table representation and then processed using Weaver . However, as we discuss later, such datasets typically lack the complex hybrid queries involving multi-step reasoning and semantic inference, which are the primary focus of our work.

Filtering Hybrid Queries using Binder. To filter hybrid queries from their original dataset counterparts, we leverage Binder's query structure. Specifically, we identify queries that invoke any UDF, for example 'QA' function as seen in the example below, which prompts the LLM with a yes/no question related to a single column. This serves as a reliable indicator that the query requires semantic reasoning beyond SQL capabilities. The example below demonstrates such a pattern, where the QA function is applied to assess contextual understanding. This filtering process ensures that the LLM is used for its intended purpose, semantic inference or interpretation, aligned with the methodology outlined in the "Our Approach" section.

Listing 24: Example NeuralSQL query using QA function

```
Question: what number of games were lost at home?

NeuralSQL:
SELECT COUNT(*) FROM w WHERE QA("map@is it a loss?"; 'result/score') = 'yes'
AND QA("map@is it the home court of New Orleans Saints?"; 'game site') = 'yes'
```

E Comparison with Existing Datasets

Several benchmark datasets—such as Spider, HiTab, and BIRD—have contributed significantly to multi-hop and multi-table question answering. However, these benchmarks primarily emphasize symbolic SQL reasoning and do not align with the hybrid reasoning focus of Weaver , which requires coordinated symbolic (SQL-based) and semantic (LLM-based) reasoning.

Spider was designed for cross-domain Text-to-SQL parsing over a variety of databases. While it features compositional queries, the questions can typically be answered using SQL alone. Despite recent augmentations that add paraphrasing and perturbation, the core tasks remain fully executable without any semantic interpretation, making Spider insufficient for evaluating hybrid symbolic-semantic pipelines.

HiTab focuses on hierarchical tables and structural decoding challenges. Although it includes tables that require some normalization or flattening, once transformed, the resulting queries involve only 2–3 simple SQL operations (e.g., filtering, aggregation). Importantly, these tasks do not demand semantic reasoning or LLM-based operations, limiting HiTab's suitability for hybrid evaluation.

BIRD is a large-scale, multi-table QA benchmark that emphasizes compositional and multi-hop reasoning over relational tables. It serves as a rigorous testbed for symbolic systems. However, as we detail below, it lacks tasks that truly require hybrid semantic-symbolic coordination.

E.1 Evaluation on BIRD Dataset

We evaluate Weaver on the BIRD benchmark using GPT-4o as the backend. Weaver achieves an accuracy of **91%**, solving nearly all tasks using symbolic SQL execution alone, without invoking any LLM-based semantic reasoning. These results demonstrate that while Weaver is highly effective on BIRD, the dataset's symbolic nature limits its value for testing hybrid reasoning capabilities. Most BIRD tasks can be completed in 3–4 SQL steps involving standard operations like joins, filtering, and sorting. For instance, consider the query:

Listing 25: Example execution plan

```
Question: Please list the lowest ten eligible free rates for students aged 5  17  in
     continuation schools. Eligible free rates Free Meal Count (Ages 5  17  ) /
    Enrollment (Ages 5  17  )

Plan:

SQL: Join the frpm table with the schools table using School Type and SOCType,
    filtering for "Continuation High Schools".
SQL: Compute eligible free rates per school using the formula: Free Meal Count (Ages
     5-17) / Enrollment (Ages 5-17).
SQL: Sort schools by the calculated rate in ascending order.
SQL: Select the ten schools with the lowest eligible free rates.

SQL Queries Generated -

Step 1: SELECT f.* FROM frpm f JOIN schools s ON f.School Type = s.SOCType WHERE s.
    SOCType = 'Continuation High Schools';

Step 2: CREATE TABLE eligible_free_rates AS SELECT School Name, School Type, Free
    Meal Count (Ages 5-17), Enrollment (Ages 5-17), (Free Meal Count (Ages 5-17) /
    Enrollment (Ages 5-17)) AS eligible_free_rate FROM continuation_schools;

Step3: CREATE TABLE sorted_eligible_free_rates AS SELECT * FROM eligible_free_rates
    ORDER BY eligible_free_rate ASC;

Step 4: CREATE TABLE lowest_ten_eligible_free_rates AS SELECT * FROM
    sorted_eligible_free_rates LIMIT 10;
```

Despite the multi-hop joins, these tasks do not require semantic disambiguation, commonsense reasoning, or LLM-assisted table understanding. As such, BIRD was excluded from our main hybrid benchmark comparison table, though its results highlight Weaver 's strong symbolic reasoning capabilities.

Why Existing Datasets Fall Short? While Spider, HiTab, and BIRD advance multi-hop QA in important ways, none capture the core challenges of hybrid queries where symbolic (SQL) and semantic (LLM) steps must be interleaved. Weaver is explicitly designed for such hybrid workflows, requiring intelligent planning, decomposition, and alternating execution paths—capabilities not required in these prior datasets.

E.2 Execution Strategy in the Weaver Pipeline

Why SQL Execution Alone Was Insufficient? In *Weaver* , SQL execution alone does not yield the final answer because the retrieved table may contain extraneous data or errors, particularly if any SQL-LLM steps fail or produce incomplete results. Although SQL retrieves the data, it cannot handle formatting issues, missing values, or the semantic reasoning required to extract the correct answer. Therefore, an LLM is used post-execution to refine the table, correct formatting, and extract only the relevant values, ensuring accurate final answers that SQL alone cannot guarantee.

Why SQL and LLM Outputs Are Not Combined? Unlike methods like H-STAR, which execute SQL and LLMs in parallel, *Weaver* selectively chooses whether SQL or LLM should handle each query

part. This planning-based approach improves efficiency by avoiding the computational cost of parallel execution. As shown in Table 5: Number of API Calls Comparison per TableQA, H-STAR requires 8 LLM API calls per query, whereas *Weaver* averages only 5.4, significantly reducing overhead and improving overall performance.

E.3 Rationale for Using SQL over Python

While Python offers greater expressiveness, we selected SQL for three key reasons that align with the objectives of our framework:

- **Portability:** SQL is the standard query language supported by most database engines, ensuring our approach can be easily integrated into real-world systems.
- **Interpretability:** SQL's declarative nature makes queries more transparent, facilitating step-by-step explainability in the planning pipeline.
- **Efficiency:** SQL operations, such as filtering, aggregation, and joins, are highly optimized for symbolic inference over tabular data, enabling faster execution on large datasets without requiring data export.

SQL is well-suited for our use case because *Weaver* is designed for seamless integration with database systems and big data environments. In contrast, Python-based solutions like Pandas can encounter scalability issues, while SQL is inherently optimized to efficiently handle large-scale tabular data.