From Textual Columns to Query Plans: A Unified Relational-Semantic Execution Framework for Hybrid Query Processing

Nima Shahbazi
Megagon Labs
nima@megagon.ai

Seiji Maekawa
Megagon Labs
seiji@megagon.ai

Nikita Bhutani
Megagon Labs
nikita@megagon.ai

Estevam Hruschka
Megagon Labs
estevam@megagon.ai

Abstract

Real-world table question answering often involves hybrid schemas in which some query-relevant information is explicit in relational columns, while other attributes, predicates, or join conditions are only implicit in free-form text. Existing systems struggle with this setting: Text-to-SQL methods scale to large and multi-table databases but require fully structured schemas, whereas direct LLM-based methods can interpret textual content but are costly and unreliable when applied to large databases. We present OMNITQA, a unified framework for semi-structured table question answering that treats semantic reasoning as a first-class operation within relational query execution. OMNITQA compiles natural-language questions into directed acyclic graphs of relational and LLM-based semantic operators. This enables ambiguity-aware plan diversification, cost-aware optimization, and dual-engine execution over structured and textual data. Across structured and semi-structured benchmarks, OMNITQA consistently improves performance in hybrid settings, outperforming the strongest baselines by 14 accuracy points on average and by 27 points on the most challenging subset, while maintaining competitive accuracy on fully structured datasets.

1 Introduction

Recent advances in large language models (LLMs) have significantly improved table question answering (TQA) systems (Ipeirotis and Zheng, 2025; Liu et al., 2025b; Zhang et al., 2024). However, existing approaches operate under a polarized trade-off: they either scale efficiently over large databases by assuming fully structured schemas, or provide semantic flexibility over unstructured text at the cost of poor scalability. This assumption rarely holds in real-world settings, where datasets frequently feature *hybrid* schemas that blend structured attributes with free-form text such as product descriptions, customer feedback, or clinical notes. Answering even simple queries in such settings may require recovering latent attributes from text, grounding them to implicit schema elements, and performing multi-step reasoning jointly over structured and unstructured data (Figure 1).

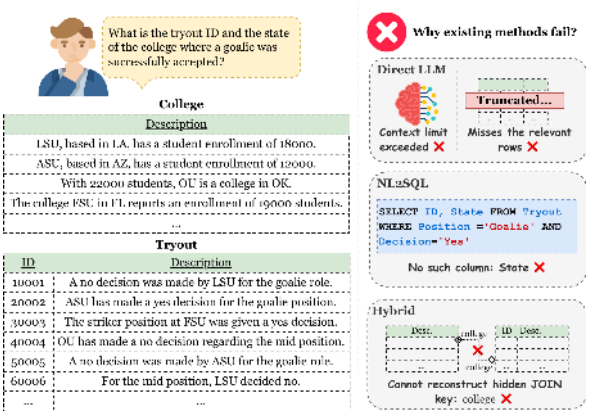


Figure 1: Limitations of existing TQA approaches on querying semi-structured data

Existing paradigms fail to natively support the intersection of structural scaling and semantic depth required for hybrid schemas. Traditional Text-to-SQL approaches scale well over multi-table databases but assume a strict closed world and fail when critical information is embedded within free-form text (Zhou et al., 2025). Conversely, direct LLM-based approaches (Fang et al.; Yu et al., 2025) reason flexibly over serialized tables and documents, but bypass symbolic query execution, leading to high inference cost and poor scalability on large datasets (Zhang et al., 2026). Recent hybrid pipelines (Khoja et al., 2025; Abhyankar et al., 2025; Patel et al., 2025) combine symbolic querying with semantic matching, but typically operate at a coarse granularity and assume that relevant schema attributes are already explicitly represented. They simplify the problem to attribute identification rather than dynamic structure recovery.

Findings of the Association for Computational Linguistics: IJCNLP-AACL 2026, pages 3979–4002
November 6-10, 2026 ©2026 Association for Computational Linguistics

In contrast, we focus on a more fundamental challenge: environments where query-essential attributes are unmaterialized and must be dynamically recovered from unstructured text during query execution. We argue that this requires moving beyond pipeline-based designs toward a unified execution model that tightly interleaves symbolic database operations with semantic reasoning. This unified approach naturally unlocks fine-grained, data-driven optimizations, allowing the resulting system to generalize effectively across long tables, multi-table schemas, complex queries, textual columns, and implicit attributes. Table 1 summarizes the full scope of settings we aim to support.

To this end, we propose OMNITQA[1],, a unified framework that treats semantic reasoning as a first-class operator in a relational execution model. It translates natural language queries into directed acyclic graphs (DAGs) of atomic relational and LLM-based operators to enable fine-grained reasoning over hybrid schemas. To handle natural language and textual ambiguity when recovering latent schema elements, OMNITQA employs a parallel plan diversification strategy. Rather than committing to a single fragile grounding, the planning layer generates and evaluates multiple candidate plans in parallel and then consolidates their outputs to select the best answer.

OMNITQA builds on established abstractions for combining relational processing with LLM-based semantic operations (Patel et al., 2025; Liu et al., 2025a; Glenn et al., 2024). Existing hybrid query optimizers address the efficient execution of specified queries or operator programs, including operator placement and implementation selection (Mang et al., 2026; Russo et al., 2026). Our focus is end-to-end question answering when query-relevant structure is implicit in text: synthesizing alternative relational-semantic plans under ambiguity in the schema, executing them, and consolidating their answers.

To execute these heterogeneous plans efficiently, OMNITQA incorporates a rule-based query optimization layer and a physical dual-engine architecture. It leverages the operator-centric DAG abstraction and applies classic database optimization principles such as filter push-down, operator reordering, and query-aware reduction to minimize expensive LLM invocations. The dual-engine runtime then dynamically routes these optimized operators between a traditional relational database engine and an LLM module. To ensure throughput over large datasets, the engine implements operator-aware batching and partitions intermediate relations to execute semantic operations in parallel.

[1]Code: https://github.com/megagonlabs/OmniTQA

Challenge	NL2SQL	Direct-LLM	Hybrid	OMNITQA
Long tables	✓	✗	✗	✓
Multi-table	✓	✓	✗	✓
Complex queries	✓	✗	✗	✓
Textual columns	✗	✓	✓	✓
Implicit schema	✗	✓	✗	✓

Table 1: Comparison of various TQA approaches. ✗ denotes settings that are not natively supported, although limited or indirect handling may still be feasible.

We evaluate OMNITQA on both structured and semi-structured benchmarks. The results demonstrate that our framework breaks the traditional trade-off. It significantly outperforms existing TQA systems on semi-structured datasets while remaining competitive on fully structured datasets.

Our main contributions are as follows:

- We formalize hybrid query processing over semi-structured data, where latent schema elements must be dynamically recovered before symbolic execution can proceed.
- We design and implement an operator-centric TQA framework that models hybrid queries as relational-semantic DAGs, powered by a physical dual-engine runtime with operator-aware batching for scalable inference.
- We introduce a parallel plan diversification strategy to robustly recover latent structure under textual ambiguity, and showcase extensibility of the framework via an initial query optimizer.
- We conduct extensive experiments showing that OMNITQA outperforms existing TQA systems on semi-structured datasets by 14 accuracy points on average and 27 points on the most challenging subset, while maintaining competitive performance on structured benchmarks.

2 Preliminaries

Data Model. Let $\mathcal{S} = \{T_1, \ldots, T_n\}$ be a database schema over an instance $\mathcal{D}$. The global attribute set $\mathcal{A} = \bigcup_i \mathcal{A}_i$ is partitioned into two disjoint subsets: structured attributes $\mathcal{A}_\text{s}$ and unstructured text attributes $\mathcal{A}_\text{u}$, such that $\mathcal{A} = \mathcal{A}_\text{s} \cup \mathcal{A}_\text{u}$. Structured attributes contain atomic values

queryable via traditional relational operators Ω_{R} (e.g., projection, selection, join, aggregate). Unstructured attributes contain free-form text fields that implicitly encode latent entities, attributes, or predicates, requiring a set of semantic operators Ω_{S} (e.g., extraction, matching, joins).

Table Question Answering. Given a structured schema ($\mathcal{A} = \mathcal{A}_{\mathrm{s}}$) and a natural language query Q, the objective is to derive a symbolic execution plan composed exclusively of relational operators. Executing this plan over $\mathcal{D}$ yields the answer Y.

Semi-Structured Table Question Answering. Given a hybrid schema ($\mathcal{A} = \mathcal{A}_{\mathrm{s}} \cup \mathcal{A}_{\mathrm{u}}$) and a query Q, the objective is to dynamically recover the latent structure embedded within $\mathcal{A}_{\mathrm{u}}$ and ground it alongside explicit elements in $\mathcal{A}_{\mathrm{s}}$. A hybrid plan of relational and semantic operations is required to jointly reason over $\mathcal{A}_{\mathrm{s}}$ and $\mathcal{A}_{\mathrm{u}}$.

2.1 Design Considerations

To scale across long tables, multi-table schemas, complex queries, textual columns, and implicit attributes (Table 1), a hybrid query processing system should adhere to three core principles:

- **Operator-Centric Decomposition.** Decompose queries into fine-grained plan with sub-tasks that can be directed to a symbolic or semantic engine. Such decomposition is essential to handle complex queries and multi-table operations over implicit text attributes.
- **Plan Diversification.** Mitigate natural language and text column ambiguity by generating and exploring alternative logical plans. This ensures robust structural grounding before executing downstream database operations.
- **Cost Optimization.** Apply principles from database query optimization to reduce expensive LLM inferences and allow the framework to scale to long tables efficiently.

3 Proposed System: OMNITQA

To operationalize these principles, we introduce OMNITQA, which cleanly decouples query planning from physical execution. We illustrate the overall architecture of OMNITQA in Figure 2, which consists of a logical planning layer with plan diversification, and a physical dual-engine runtime.

3.1 Pre-processing

Given a natural language query Q, OMNITQA first executes a lightweight, query-aware pre-processing to prune the database context and build a compact data profile for the planner. This stage implements two primitives. First, an LLM isolates query-relevant tables, explicit attributes $\mathcal{A}_{\mathrm{s}}$, and unstructured text fields $\mathcal{A}_{\mathrm{u}}$, while eliminating orthogonal schema elements to minimize the plan search space. Next, it constructs a representative data snapshot for the planner by combining semantically relevant rows (via vector search over text fields) with randomly sampled tuples. This provides the planner grounded context regarding the underlying data distribution. We provide implementation details of pre-processing in Appendix A.1. The output of this stage consists of a pruned database context and a query-aware data preview, which together serve as input for the planning stage.

3.2 Logical Planning & Plan Diversification

The logical planning phase consists of four components: *schema grounding*, *atomic decomposition*, *plan diversification*, and *plan optimization*.

Schema Grounding. Building directly on the preprocessing layer, schema grounding utilizes the dual data preview to map the entities and predicates expressed in Q to concrete elements of the refined database schema $\mathcal{S}$ (e.g., mapping the phrase "customer feedback" to a specific text column named `usr_cmnt`). This process resolves linguistic variations and ensures all query-relevant tables and attributes are bound.

Operator-Centric Decomposition. Once the relevant schema elements are identified, the planner decomposes Q into a directed acyclic graph (DAG) representing a logical query plan, denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each node $v \in \mathcal{V}$ corresponds to an atomic operator drawn from the unified operator space $\Omega = \Omega_{\mathrm{R}} \cup \Omega_{\mathrm{S}}$, while directed edges $e \in \mathcal{E}$ define strict data-flow dependencies. To ensure fine-grained execution and deterministic optimization, we require every atomic node v to satisfy two structural constraints: (1) it contains at most one logical condition, and (2) the condition involves a single column or variable.

By modeling queries as explicit operator DAGs, OMNITQA extends classical query optimization to semantic spaces. Unlike prior hybrid pipelines (Khoja et al., 2025; Abhyankar et al., 2025)

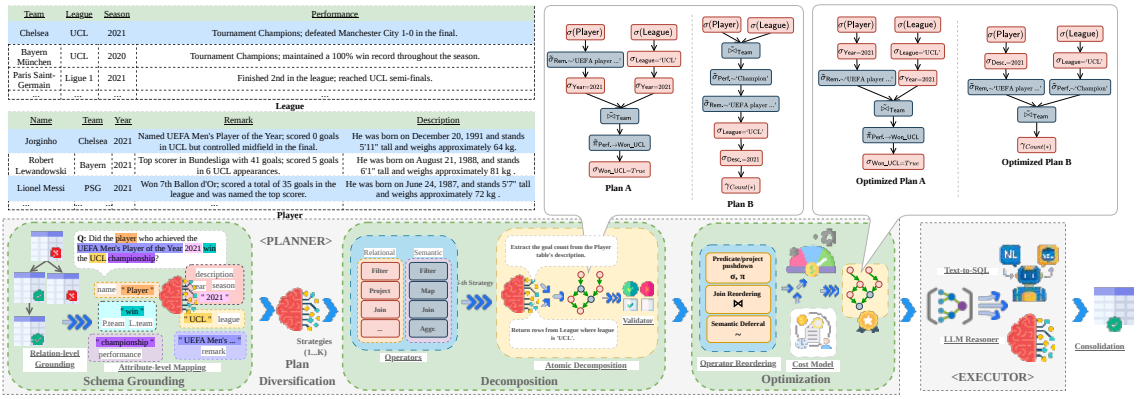


Figure 2: OMNITQA architecture. Given the UEFA Soccer database and the query *"Did the player who achieved the UEFA Men's Player of the Year 2021 win the UCL championship?"*, OMNITQA constructs a data preview (blue) to ground query expressions to schema elements, then generates and optimizes candidate logical plans to resolve ambiguity over where the UEFA Men's Player of the Year information is stored, e.g., in "Remark" or "Description".

where semantic operators only process pre-defined schema attributes, nodes in Ω_{S} are dynamically tasked with recovering latent relational structures from text fields $\mathcal{A}_{\mathrm{u}}$. They can convert unstructured text into structured relations to unblock downstream symbolic operations $v \in \Omega_{\mathrm{R}}$. We provide the details of relational Ω_{R} and semantic Ω_{S} operators in Table 5 in Appendix A.2.

Plan Diversification. To mitigate structural failures arising from natural language ambiguity or unmaterialized schemas, OMNITQA implements a parallel *plan diversification* strategy. Rather than generating a single, error-prone schema grounding, the planner generates a set of K distinct candidate execution plans. These candidate plans systematically explore alternative structural interpretations across four primary dimensions: **1) Schema Mapping Diversity:** We map an ambiguous query term such as "return player" to multiple candidate attributes such as `player_id` and `player_name`. **2) Risk-Profile Variations:** We generate *strict* plans that prioritize deterministic relational operators, as well as *fuzzy* plans that use semantic operators to capture subjective predicates. **3) Operator Substitution:** We explore logically equivalent transformations, such as using a semantic `FILTER` instead of a semantic `MAP` followed by relational filtering. **4) Semantic Intent Modeling:** We interpret vague expressions such as "top player" using alternative semantic criteria, such as `goals_scored` or `awards_won`.

Plan Optimization. The execution plans generated by the planner may not be cost-efficient, particularly when semantic operators are involved. Because query optimization over hybrid operator spaces is NP-hard (Chatterji et al., 2002), we employ a LLM-based heuristic optimizer that accounts for the asymmetry between inexpensive relational operations and expensive semantic reasoning. We enforce four optimization invariants over the operator DAGs: **1) Selection Pushing**: Relational selection operators (σ) are pushed down past semantic operators toward base relations to prune irrelevant tuples early. **2) Projection Pruning**: Unnecessary attributes are dropped immediately after their last dependent node to reduce the width and memory footprint of intermediate relation. **3) Join Reordering**: Highly selective relational joins are prioritized to limit intermediate result size. **4) Adaptive Semantic Deferral**: Semantic operators are systematically deferred until relational pruning is complete unless a downstream relational operator is expected to expand the result set, in which case the semantic step is moved earlier to serve as a pre-filter. We also provide a deterministic implementation of this optimization strategy in Appendix A.3.

3.3 Physical Execution & Plan Consolidation

The physical execution engine takes the K optimized candidate plans and executes them concurrently over the database instance $\mathcal{D}$ to produce K candidate answers, which are subsequently consolidated into a single final output. To balance structural scaling with semantic expressiveness, the runtime introduces a dual-engine architecture. Relational operators (Ω_{R}) are dynamically routed to a native SQL database engine that executes compiled query statements directly on the host database. Conversely, semantic operators (Ω_{S}) are dispatched

to an LLM-based reasoning module. To prevent the context-window explosions or excessive row-wise API latency that a naive data-passing approach would incur, the engine employs a block-based batching strategy.

Operator-driven Batching. The execution engine treats semantic operators as chunked primitives to enable substantial intra-operator parallelism. Intermediate relations are partitioned into manageable blocks of size β and executed concurrently via three asynchronous batching strategies: **1) Semantic Map & Filter**: These operators evaluate tuples independently. Input relations are partitioned into chunks processed asynchronously. To reduce output tokens, the LLM returns only derived values (for Map) or row indices satisfying the predicate (for Filter), which are then applied to the original relation. **2) Semantic Join**: To avoid loading massive Cartesian products into a single context window, the engine executes an LLM-driven block nested-loop join. Input relations are streamed in blocks of size β, and each block pair is evaluated independently in parallel. **3) Semantic Aggregate**: For operators requiring global context, the engine employs a recursive map-reduce strategy. Relations exceeding β are partitioned into parallel chunks, partial local aggregations are computed asynchronously, and intermediate metrics are recursively reduced to a final scalar value. The detailed algorithm for operator-driven batching is described in Appendix A.4.

Plan Consolidation. The final stage of the execution step is to select the best answer among the K candidates produced by executing the diversified plans. OMNITQA supports two consolidation strategies: **1) LLM-as-a-Judge Consensus** (Gu et al., 2024). An independent LLM evaluates candidate outputs by comparing them against the original query and representative samples of each plan's results. Using few-shot examples to calibrate the evaluation, the judge selects the output that best aligns with the user's intent. **2) Semantic Majority Voting** (Chen et al., 2024). An ensemble-based strategy that treats each materialized relation as a vote and selects the result that appears most frequently across independent plans.

4 Experiments

We aim to answer four core research questions:

- **RQ1 (Effectiveness):** Does OMNITQA outperform state-of-the-art baselines on hybrid schemas containing implicit text attributes?
- **RQ2 (Generalizability):** Does the performance generalize across datasets and base LLMs?
- **RQ3 (Efficiency):** How does OMNITQA compare to baselines in terms of token cost?
- **RQ4 (Ablation Study):** What is the impact of individual design components?

Datasets. Our primary benchmark is REPAIRTQA (Zhang et al., 2026), which provides paired fully structured and semi-structured variants of the same underlying tables. In the semi-structured setting, query attributes embedded in free-form text fields. It also includes subsets with simple lookup, compositional queries, and multi-table operations across both short and long table settings. To evaluate generalization, we additionally evaluate on semi-structured datasets including FETAQA (Nan et al., 2022), HYBRIDQA (Chen et al., 2020), and TAT-QA (Zhu et al., 2021) datasets. Detailed dataset statistics are provided in Appendix B.1. We also evaluate standard fully structured benchmarks including WIKISQL (Zhong et al., 2017) and WIKITABLEQUESTIONS (Pasupat and Liang, 2015), with results reported in Appendix C.1.

Baselines. We evaluate OMNITQA against three categories of baselines: direct-LLM, Text-to-SQL, and hybrid approaches. **DIRECT-LLM** performs direct prompting without explicit planning or symbolic execution. **NL2SQL** generates SQL queries for direct database execution. **PLAN-OF-SQLS** (Nguyen et al., 2025) decomposes a query into natural language steps, translates each step to SQL, and chains intermediate results. **H-STAR** (Abhyankar et al., 2025) routes operations between SQL and LLMs based on task types, but assumes explicit schema attributes and does not support multi-table reasoning. **WEAVER** (Khoja et al., 2025) interleaves SQL execution with LLM reasoning, but assumes explicit schemas and does not target multi-table or semi-structured settings.

Evaluation. Following REPAIRTQA (Zhang et al., 2026), we use an LLM-based judge instead of exact string matching to evaluate outputs. This accounts for variations in ordering, formatting, and minor textual differences while preserving semantic equivalence for free-form answers. We evaluate

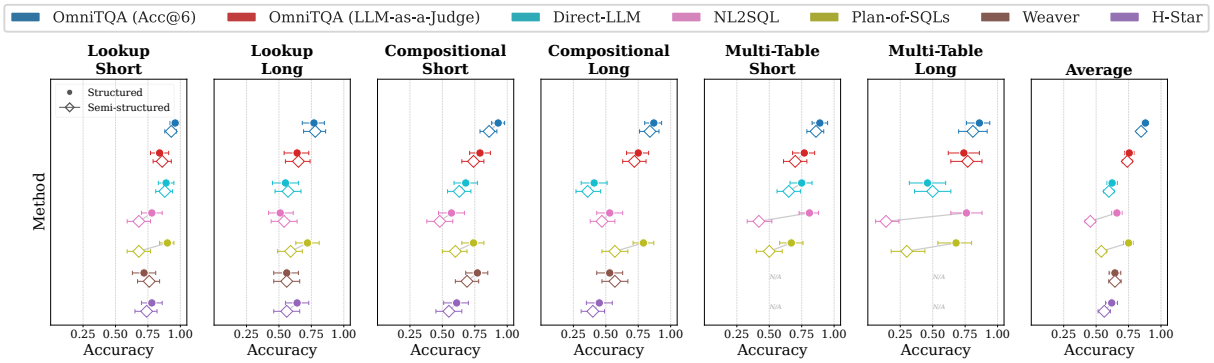


Figure 3: Effectiveness comparison of OMNITQA vs. baselines on structured vs. semi-structured representations of REPAIRTQA evaluated with GEMINI-3-FLASH. **Filled markers** denote the structured setting, and **unfilled markers** denote the semi-structured setting. LLM-as-a-Judge reports selected-answer accuracy; ACC@6 is an oracle upper bound over the six generated candidates. Horizontal bars show approximate 95% confidence intervals from 10,000 parametric bootstrap replicates using the reported accuracies and nominal evaluation sizes.

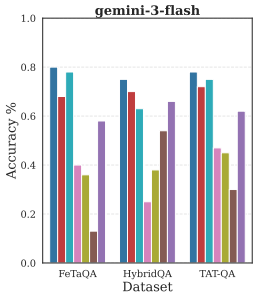


Figure 4: Effectiveness on extended benchmarks.

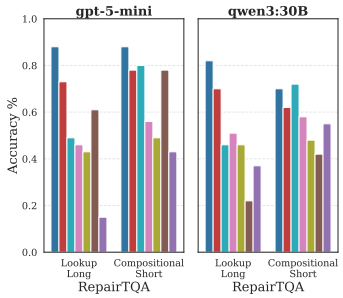


Figure 5: Effectiveness of of OMNITQA vs. baselines using different base LLMs.

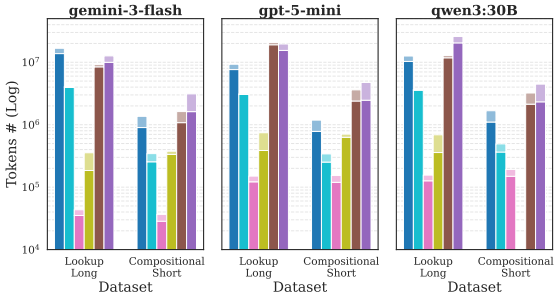


Figure 6: Cost comparison of OMNITQA vs. baselines. Darker and lighter shades denote *input* and *output* tokens, respectively.

systems along two dimensions: *effectiveness* (answer accuracy) and *efficiency* (token usage).

Settings. We evaluate across multiple base LLMs, including Gemini-3-Flash-Preview, GPT-5-Mini, and a locally deployed Qwen3:30B via Ollama. Vector embeddings are generated using the `all-MiniLM-L6-v2` encoder and cosine similarity is used for retrieval during pre-processing. We provide additional model details in Appendix B.2. By default, we set the plan diversification factor to $K = 6$ and the execution block size to $\beta = 100$; sensitivity analyses for $K \in [1, 6]$ and $\beta \in [10, 1000]$ are provided in Appendix C.3. Unless stated otherwise, we use Gemini-3-Flash-Preview for all experiments. For fair comparison, all methods use the same underlying LLMs. We report two complementary accuracy measures. LLM-as-a-Judge denotes the accuracy of the single answer selected from the $K = 6$ executed candidates by an LLM without access to the test question's reference answer. The selected answer is then scored against the reference using the correctness-evaluation protocol described above. ACC@6 counts a question as correct if at least one of the same six candidates is correct under that protocol. It is therefore a diagnostic upper bound for answer selection within the generated candidate set, rather than an inference-time consolidation method. Comparisons with single-answer baselines use LLM-as-a-Judge selected-answer accuracy. The gap to ACC@6 indicates the remaining opportunity to improve answer selection when a correct candidate has already been generated.[2] Additional implementation details and prompts are provided in Appendix A.1 and Appendix D, respectively.

We estimate approximate 95% confidence intervals using 10,000 bootstrap samples based on the reported accuracy and evaluation size ($n = 100$, or $n = 50$ for Multi-Table Long). For the Average panel, we average across the available subsets in each sample. These intervals capture uncertainty from question sampling, not variation across model runs.

[2]We discuss other consolidation variants in Appendix C.2

4.1 RQ1: Effectiveness

We first evaluate the methods on structured and semi-structured subsets in REPAIRTQA to measure their effectiveness in recovering latent structure.

OMNITQA achieves state-of-the-art accuracy across semi-structured settings. As shown in Figure 3, OMNITQA consistently outperforms all baselines. Using LLM-as-a-Judge, it outperforms the strongest baseline evaluated across all six subsets (Direct-LLM) by 14 points on average in the semi-structured setting (74% vs. 60% in the rightmost subplot). The corresponding approximate 95% confidence intervals are 70.2–77.7% for OMNITQA and 55.8–63.8% for Direct-LLM, with bootstrap standard errors of 1.89 and 2.05 percentage points, respectively. The largest gain appears on the multi-table long subset, where OMNITQA achieves 77% accuracy compared to 50% for the next-best baseline (Direct-LLM). This setting is particularly challenging because it requires both accurate schema grounding and complex join reasoning over implicit attributes. These gains suggest that plan diversification helps latent structure under ambiguity. While LLM-as-a-Judge does not reach the ACC@6 upper bound, it effectively selects strong candidates from the diversified plan set and consistently outperforms all baselines.

Direct-LLM struggles on long tables, while Text-to-SQL degrades sharply on semi-structured data. Baseline architectures fail under specific table topologies. Direct-LLM remains competitive on when serialized tables fit comfortably within the model context window. Conversely, Text-to-SQL baselines excel on explicit schemas but struggle when query-relevant attributes are embedded in free-form text. This degradation becomes more severe on multi-table subsets that require both schema grounding and join reasoning.

Hybrid baselines shows moderate robustness but remain limited on complex settings. Hybrid baselines outperform Text-to-SQL systems on semi-structured data, but still struggle on long-table and multi-table subsets. This suggests that coarse-grained integration between symbolic execution and semantic reasoning is insufficient for robust latent structure recovery at scale.

OMNITQA significantly narrows the performance gap between structured and semi-structured settings. All methods experience some degradation when moving from structured to semi-structured tables. However, OMNITQA exhibits the smallest performance drop, indicating stronger robustness to implicit schema representations. This smaller degradation is accompanied by competitive selected-answer accuracy on fully structured tables, including against strong Text-to-SQL baselines such as Plan-of-SQLs. When query-relevant attributes are explicitly represented, relational operations already address much of the task, leaving less need for semantic structure recovery. Together, these results highlight the contribution of a unified framework that remains competitive on explicit schemas while providing substantial gains when answering questions requires recovering structure from text.

Dataset	OMNITQA	w/ Naive $\hat{R}^{\mathcal{Q}}$	w/ QDMR	w/o Opt. $\mathcal{G}$	w/o Div. $\mathcal{G}$	w/o Pru. $\mathcal{S}$
Lookup Long	**0.65**	0.59	0.43	0.65	0.53	0.65
Compositional Short	**0.74**	0.74	0.46	0.74	0.68	0.73
Multi-Table Short	**0.70**	0.64	0.33	0.70	0.59	0.68

Table 2: Ablation study: accuracy (LLM-as-a-Judge).

Dataset	OMNITQA	w/o Opt. $\mathcal{G}$
Lookup Long	**16.7M**	20.4M
Compositional Short	**1.3M**	1.6M
Multi-Table Short	**0.6M**	0.7M

Table 3: Ablation study: cost (LLM-as-a-Judge).

4.2 RQ2: Generalizability

OMNITQA generalizes well across semi-structured benchmarks. Figure 4 shows the accuracy of OMNITQA and baselines on three additional semi-structured datasets. Because these benchmarks contain relatively small tables, Direct-LLM performs reasonably well. However, Text-to-SQL methods fail due to the implicit schema representations. Overall, OMNITQA consistently achieve strong performance across all datasets.

OMNITQA is robust across different base LLMs, particularly on long-table subsets. Figure 5 compares method performances with GPT-5-Mini and Qwen3:30B. OMNITQA shows similar trends as Gemini-3-Flash-Preview, with strong gains on long-table subsets, indicating robustness to the choice of base LLM. In contrast, hybrid base-

lines exhibit larger performance variation across LLMs, suggesting higher sensitivity to model capabilities and prompting strategies. OMNITQA's modular architecture and parallel plan diversification seem to provide a more stable performance across different LLMs.

4.3 RQ3: Efficiency

We evaluate computational cost in terms of token usage. Figure 6 reports input and output tokens (stacked bars) with darker segments indicating input tokens and lighter segments output tokens. We compare configurations using one versus six candidate plans. LLM-as-a-Judge has similar cost to the six-plan setting and is omitted for clarity. All results are averaged over the full test set.

OMNITQA achieves achieves superior accuracy and robustness at competitive cost. A single-plan configuration of OMNITQA has cost comparable to Direct-LLM, while delivering substantially higher accuracy and robustness. In contrast, Text-to-SQL methods, especially NL2SQL, are the most cost-efficient due to minimal LLM involvement. Finally, we observed that across all settings, input tokens dominate the total cost.

4.4 RQ4: Ablation Study

We conduct an ablation study of OMNITQA's key components. While we focus here on accuracy via LLM-as-a-Judge, identical relative trends hold ACC@6 (see Appendix Table 7).

Data preview is essential for long tables. Replacing data preview with a uniformly sampled subset consistently degrades performance, particularly on long tables (Table 2). The effect is smaller on short tables, where relevant information is more likely to appear in random samples.

Operators designed for semi-structured settings are crucial for accurate decomposition. Replacing our operator model with a QDMR-based decomposition (Wolfson et al., 2020) results in significant accuracy degradation across all settings, reaching as high as 49%. This highlights that the design of operators for semi-structured table reasoning is a non-trivial task.

Plan optimization improves efficiency without affecting accuracy. Removing optimization increases execution cost by up to 20% (Table 2), while leaving accuracy unchanged. This suggests that our optimization strategy successfully minimize inference overhead without modifying the plan's underlying semantics.

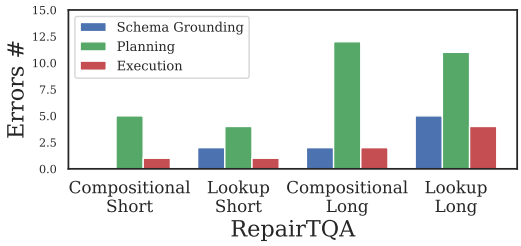


Figure 7: Error categorization in REPAIRTQA.

Plan diversification is necessary for robust performance. Replacing our diversification strategy with naive sampling of K independent plans leads to consistent accuracy degradation. This suggests that stochastic LLM variation is insufficient for handling schema ambiguity or structural recovery.

Schema pruning provides consistent but modest gains. Removing schema pruning leads to a modest decrease in accuracy, especially on complex queries. Table 3 shows that pruning also reduces token cost by up to 18%. This suggests that pruning effectively reduces the search space for the planner, leading to efficient and accurate plan generation.

4.5 Error Analysis

We conduct a qualitative analysis of failure cases to identify sources of error. While some errors in REPAIRTQA stem from data inconsistencies or evaluation issues (detailed in Appendix C.5), we focus on errors attributable to the system itself. We categorize failures into three stages of OMNITQA: *schema grounding*, *planning*, and *execution*, based on manual inspection of errors in single-table subsets of REPAIRTQA.

Figure 7 shows that the majority of errors are due to planning, which includes decomposition and plan generation. Common issues include incomplete projections (e.g., missing requested attributes such as middle name when 'full name' is required). Because the planner relies on a compact data snapshot, it occasionally generates filters using misaligned semantic literals (e.g., substituting status='extinct' with 'inactive' or 'disappeared'). This also leads to incorrect operator choices. For example, it incorrectly maps 'Northern California' to a region column value that is not present in the data, or treats 'most verbose complaint' as word count rather than service duration. We provide additional examples in Appendix C.6.

5 Related Work

Text-to-SQL and Table QA. Text-to-SQL methods execute efficiently over structured databases (Liu et al., 2025c; Hong et al., 2025; Liu et al., 2024; Deng et al., 2025), but they assume explicit schemas and struggle when query-relevant information is embedded in text fields. LLM-based TQA methods instead serialize tables and text for direct reasoning (Zhang et al., 2025; Yu et al., 2025; Wu et al., 2025b,a), offering greater semantic flexibility but sacrificing relational efficiency and scalability on long or multi-table settings.

Semantic Operator Systems. Recent database systems integrate LLMs as fine-grained semantic operators (Patel et al., 2025; Liu et al., 2025a; Glenn et al., 2024; Su et al., 2026), while hybrid pipelines route sub-tasks between relational engines and LLMs (Khoja et al., 2025; Abhyankar et al., 2025). Related query-optimization work focuses on the execution of a specified hybrid computation: PLOP optimizes semantic-operator placement within a supplied plan (Mang et al., 2026), Abacus selects operator implementations under quality, cost, and latency objectives (Russo et al., 2026), and Sema optimizes SQL queries containing semantic expressions (Qi et al., 2026).

OMNITQA targets environments where query-relevant schemas are unmaterialized and must be dynamically reconstructed during query execution (Chen et al., 2020; Zhang et al., 2026). Given a natural-language question, the system must determine which latent attributes, predicates, and relationships are needed, how to recover them from text, and how to compose this recovery with relational operations. OMNITQA represents these decisions within relational-semantic DAGs and explicitly diversifies schema groundings and operator decompositions. The resulting candidates can express different interpretations of the question and textual evidence; executing them and consolidating their answers addresses uncertainty in the intended computation. This differs from searching for efficient implementations of a specified computation, which remains the role of optimization within each candidate. While BlendSQL and SEMA-SQL also generate hybrid queries from natural language (Glenn et al., 2024; Lin et al., 2026), our focus is on resolving latent-schema ambiguity through this diversification and consolidation process, evaluated through end-to-end answer accuracy on structured and semi-structured, single- and multi-table QA.

6 Conclusion

We presented OMNITQA, a unified framework for hybrid query processing over semi-structured data where query-relevant structure must be dynamically recovered during execution. OMNITQA translates queries into directed acyclic graphs (DAGs) that combine relational and semantic operators. This enables cost-aware planning, dual-engine execution, and plan diversification to robustly recover latent schema elements while maintaining scalability. Our experiments show that OMNITQA consistently sets a new accuracy standard on semi-structured benchmarks while maintaining low token costs and high accuracy on structured settings.

7 Limitations

Despite these advances, challenges remain in planning quality, cost, and latency. In particular, generating diverse high-quality plans remains a bottleneck. Since optimization is inherently limited by the quality of the initial plan, incorporating human-in-the-loop to audit and prune plans could significantly improve reliability and cost-efficiency. Furthermore, semantic operators continue to incur significant overhead. A tiered execution approach involving a high-recall, efficient hash-based FILTER or JOIN to prune the search space prior to the more expensive LLM-based execution can be helpful. This can be further augmented by model cascading strategies, which utilize a hierarchy of models to route simpler logic to smaller, cost-effective models while reserving larger LLMs for complex reasoning tasks.

8 Ethical Considerations

We made use of AI tools such as ChatGPT and Copilot to support coding and refining this paper, but all content was carefully reviewed and edited by us to ensure it adheres to our standards and aligns with our research objectives. All external artifacts, datasets, and models utilized in this study were used in strict accordance with their original intended purposes and respective licensing agreements, adhering to established ethical guidelines for research.

References

Nikhil Abhyankar, Vivek Gupta, Dan Roth, and Chandan K Reddy. 2025. H-star: Llm-driven hybrid sql-text adaptive reasoning on tables. In *Proceedings of the 2025 Conference of the Nations of the Americas*

Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 8841–8863.

Sourav Chatterji, Sai Surya Kiran Evani, Sumit Ganguly, and Mahesh Datt Yemmanuru. 2002. On the complexity of approximate query optimization. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 282–292.

Lingjiao Chen, Jared Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. 2024. Are more llm calls all you need? towards the scaling properties of compound ai systems. *Advances in Neural Information Processing Systems*, 37:45767–45790.

Wenhu Chen, Hanwen Zha, Zhiyu Chen, Wenhan Xiong, Hong Wang, and William Wang. 2020. Hybridqa: A dataset of multi-hop question answering over tabular and textual data. In *Findings of EMNLP*.

Yu Chen and Ke Yi. 2017. Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 759–774.

Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. Reforce: a text-to-sql agent with self-refinement, consensus enforcement, and column exploration. *arXiv preprint arXiv:2502.00675*.

Xi Fang, Weijie Xu, Fiona Anting Tan, Ziqing Hu, Jiani Zhang, Yanjun Qi, Srinivasan H Sengamedu, and Christos Faloutsos. Large language models (llms) on tabular data: Prediction, generation, and understanding-a survey. *Transactions on Machine Learning Research*.

Parker Glenn, Parag Dakle, Liang Wang, and Preethi Raghavan. 2024. Blendsql: A scalable dialect for unifying hybrid question answering in relational algebra. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 453–466.

Google. 2025. Gemini 3 Flash: frontier intelligence built for speed. Google Blog.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, and 1 others. 2024. A survey on llm-as-a-judge. *The Innovation*.

Joseph M Hellerstein and Michael Stonebraker. 1993. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 267–276.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-generation database interfaces: A survey of llm-based text-to-sql. *IEEE Transactions on Knowledge and Data Engineering*.

Panos Ipeirotis and Haotian Zheng. 2025. Natural language interfaces for databases: What do users think? *arXiv preprint arXiv:2511.14718*.

Rohit Khoja, Devanshu Gupta, Yanjie Fu, Dan Roth, and Vivek Gupta. 2025. Weaver: Interweaving sql and llm for table reasoning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 28270–28296.

Yin Lin, Tianjing Zeng, Zhongjun Ding, Rong Zhu, Bolin Ding, H. V. Jagadish, and Jingren Zhou. 2026. SEMA-SQL: Beyond traditional relational querying with large language models. arXiv preprint arXiv:2604.23477. *Preprint*, arXiv:2604.23477.

Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and 1 others. 2025a. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*, page 2.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024. A survey of nl2sql with large language models: Where are we, and where are we going. *arXiv preprint arXiv:2408.05109*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025b. A survey of text-to-sql in the era of llms: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025c. A survey of text-to-sql in the era of llms: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering*.

Qiuyang Mang, Yufan Xiang, Hangrui Zhou, Runyuan He, Jiaxiang Yu, Hanchen Li, Aditya Parameswaran, and Alvin Cheung. 2026. PLOP: Cost-based placement of semantic operators in hybrid query plans. arXiv preprint arXiv:2604.09944. *Preprint*, arXiv:2604.09944.

Magnus Müller, Daniel Flachs, and Guido Moerkotte. 2021. Memory-efficient key/foreign-key join size estimation via multiplicity and intersection size. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 984–995. IEEE.

Linyong Nan, Chiachun Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, and 1 others. 2022. Fetaqa: Free-form table question answering. *Transactions of the Association for Computational Linguistics (TACL)*, 10:35–51.

Anh Nguyen and 1 others. 2025. Interpretable llm-based table question answering. *Transactions on Machine Learning Research (TMLR)*.

OpenAI. 2025. Model release blog: Introducing gpt-5. *Technical report*.

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 (ACL)*.

Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic operators and their optimization: Enabling llm-based data processing with accuracy guarantees in lotus. *Proceedings of the VLDB Endowment*, 18(11):4171–4184.

Kangkang Qi, Dongyang Xie, Wenbo Li, Hao Zhang, Yuanyuan Zhu, Jeffrey Xu Yu, and Kangfei Zhao. 2026. Sema: A high-performance system for LLM-based semantic query processing. arXiv preprint arXiv:2603.11622. *Preprint*, arXiv:2603.11622.

Matthew Russo, Chunwei Liu, Sivaprasad Sudhir, Gerardo Vitagliano, Michael Cafarella, Tim Kraska, and Samuel Madden. 2026. Abacus: A cost-based optimizer for semantic operator systems. *Proceedings of the VLDB Endowment*, 19(5):1060–1073.

P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34.

Yunxiang Su, Tianjing Zeng, Zhongjun Ding, Yin Lin, Rong Zhu, Zhewei Wei, Bolin Ding, and Jingren Zhou. 2026. Large language model-enhanced relational operators: Taxonomy, benchmark, and analysis. *arXiv preprint arXiv:2603.02537*.

Tomer Wolfson, Mor Geva, Ankit Gupta, Matt Gardner, Yoav Goldberg, Daniel Deutch, and Jonathan Berant. 2020. Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*.

Niklas Wretblad, Fredrik Riseby, Rahul Biswas, Amin Ahmadi, and Oskar Holmström. 2024. Understanding the effects of noise in text-to-SQL: An examination of the BIRD-bench benchmark. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*.

Jian Wu, Linyi Yang, Dongyuan Li, Yuliang Ji, Manabu Okumura, and Yue Zhang. 2025a. Mmqa: Evaluating llms with multi-table multi-hop complex questions. In *The thirteenth international conference on learning representations*.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, and 1 others. 2025b. Tablebench: A comprehensive and complex benchmark for table question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25497–25506.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Peiying Yu, Guoxin Chen, and Jingjing Wang. 2025. Table-critic: A multi-agent framework for collaborative criticism and refinement in table reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 17432–17451.

Weixu Zhang, Yifei Wang, Yuanfeng Song, Victor Junqiu Wei, Yuxing Tian, Yiyan Qi, Jonathan H Chan, Raymond Chi-Wing Wong, and Haiqin Yang. 2024. Natural language interfaces for tabular data querying and visualization: A survey. *IEEE transactions on knowledge and data engineering*, 36(11):6699–6718.

Xuanliang Zhang, Dingzirui Wang, Longxu Dou, Qingfu Zhu, and Wanxiang Che. 2025. A survey of table reasoning with large language models. *Frontiers of Computer Science*, 19(9):199348.

Yue Zhang, Seiji Maekawa, and Nikita Bhutani. 2026. Same content, different representations: A controlled study for table QA. In *The Fourteenth International Conference on Learning Representations*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

Wei Zhou, Bolei Ma, Annemarie Friedrich, and Mohsen Mesgar. 2025. Table question answering in the era of large language models: A comprehensive survey of tasks, methods, and evaluation. *arXiv preprint arXiv:2510.09671*.

Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. 2021. Tat-qa: A question answering benchmark on a hybrid of tabular and textual content in finance. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics (ACL)*.

A Further Deep Dive into OMNITQA's Components

A.1 PREPROCESSING

OMNITQA preprocessing is divided into two phases. A query-agnostic phase first normalizes the database instance and enriches the schema before any question is observed. A query-aware phase then constructs a question-specific view of the data that supports downstream planning and execution.

Attribute Category	Relational Type Binding	Extracted Profile Summaries
Numeric	{*int, float*}	Min, Max, Average, Variance
Categorical	{*string, enum*}	Cardinality, Top-k frequent values
Textual	{*varchar, text*}	Min/Max length, unique count, sample snippets, semantic summaries
Temporal	{*date, timestamp*}	Range $[t_{start}, t_{end}]$, Granularity (e.g., year, day)
Relational	{*PK, FK*}	Referential constraints, join key mappings, table dependencies

Table 4: Data Profiling for Schema Augmentation

Query-agnostic Preprocessing. The query-agnostic phase is executed once per database instance to improve structural reliability and reduce irrelevant context before planning begins. It consists of three operations. First, **table cleaning and normalization** resolves structural inconsistencies such as duplicate attributes, malformed columns, and noisy fields so that downstream SQL execution remains stable. Second, **heuristic schema pruning** removes low-value attributes that are unlikely to help question answering, including columns with more than 95% NULL values, near-constant columns, and clearly non-informative content such as password hashes, Base64 strings, or code fragments. Third, **schema augmentation** attaches lightweight profiles to the remaining attributes, including inferred types, statistical summaries, and relational constraints such as primary and foreign keys. These profiles provide the planner with stronger relational grounding; Table 4 summarizes the extracted metadata.

Query-aware Preprocessing. The query-agnostic phase improves schema quality, but the planner still needs question-conditioned evidence to align natural language predicates with concrete attributes and values, especially in semi-structured tables where relevant signals may remain implicit in text. To address this, OMNITQA constructs a query-aware view tailored to the input question.

This phase has three components. First, **semantic schema pruning** performs a high-recall LLM pass that keeps attributes likely to be relevant to the query while reducing token overhead. Second, **schema grounding** maps query entities and predicates to candidate tables and attributes using the pruned schema, attribute profiles, and retrieved examples. Third, a **query-aware data preview** combines *semantic search* and *random sampling*: semantic search retrieves the top-k_1 rows most relevant to the predicates in a question, while random sampling contributes k_2 tuples that capture the general format and diversity of the relation. Together, these steps produce a refined database instance and a preview that ground both plan generation and execution.

Preprocessing example. Consider the UEFA Soccer database and question in Figure 2: *"Did the player who achieved the UEFA Men's Player of the Year 2021 win the UCL championship?"* To illustrate preprocessing, suppose the full tables contain additional columns omitted from the figure. `Player` also contains `NameCopy` (an exact duplicate of `Name`), `LegacyNote` (98% NULL), `Salary`, and `Nationality`; `League` also contains `ImportHash` (a non-key export checksum), `Stadium`, and `Attendance`. This illustrative extension has 16 columns across the two tables.

(1) Query-agnostic preparation. Before observing the question, cleaning removes the redundant `NameCopy` column. Heuristic pruning removes `LegacyNote`, whose NULL rate exceeds the 95% threshold, and the non-informative `ImportHash`, leaving 13 columns. Schema augmentation then profiles the remaining fields, identifying year values in `Year/Season`, categorical team names, and textual content in `Remark`, `Description`, and `Performance`. Valid fields such as `Salary` and `Attendance` remain available at this stage because their relevance depends on the question.

(2) Query-aware schema selection. For the UEFA question, semantic pruning excludes `Salary`, `Nationality`, `Stadium`, and `Attendance`: these fields contain meaningful data but do not help determine the award recipient or championship outcome. The nine columns shown in Figure 2 are retained, including the team and year fields needed to connect the tables. Both `Remark` and `Description` remain candidate sources of award evidence, preserving uncertainty for subsequent planning rather than prematurely choosing one field.

(3) Data-preview construction. For illustration, let $k_1 = k_2 = 1$ per table. Semantic retrieval

could select Jorginho's row, whose `Remark` includes "Named UEFA Men's Player of the Year," and Chelsea's 2021 UCL row, whose `Performance` includes "Tournament Champions." Random sampling could additionally include Lionel Messi's player row and Bayern München's 2020 league row. The preview thus combines relevant evidence with examples of other values and text patterns.

(4) Context passed to planning. The planner receives the nine-column schema, its profiles, and the four-row preview. In this example, preprocessing reduces schema width from 16 to 13 to 9 columns while retaining alternative evidence fields and candidate links. Award and championship conditions are evaluated during subsequent execution over the retained database relations; execution is not restricted to the preview rows.

A.2 PLANNING: Hybrid Operator Model

We define a comprehensive set of operators that form the building blocks of OMNITQA's execution engine. These operators are categorized into two classes: Relational Operators, which perform traditional database operations, and Semantic Operators, which leverage LLM capabilities for tasks that require understanding and reasoning over unstructured data. Table 5 provides a detailed overview of each operator, including its function, instruction template, and description.

A.3 PLANNING: Deterministic Optimization of Query Plans

As previously mentioned in Section 3.2, the execution plan generated by the planner may not be cost-efficient, particularly when semantic operators are involved. Aside from the LLM-instructed approach discussed in Section 3.2,OMNITQA also extends classical relational query optimization techniques (Selinger et al., 1979; Hellerstein and Stonebraker, 1993) to account for the asymmetry between inexpensive relational operations and expensive semantic reasoning in a deterministic manner. Specifically, we adopt predicate migration as a *cost-aware semantic placement* strategy. While semantic operators are deferred to later stages to minimize unnecessary LLM calls, the optimizer evaluates the *cardinality impact* of downstream relational operators. If a join or set operation is expected to significantly increase the number of tuples, semantic filters are applied earlier on smaller base relations to avoid costly expansion. The optimizer applies the following transformations:

Algorithm 1 Heuristic-Based Plan Optimization of OMNITQA

Input: Plan $\mathcal{G}$, Thresholds ε, τ
Output: Optimized Plan $\mathcal{G}^*$

function OPTIMIZEPLAN($\mathcal{G}, \varepsilon, \tau$)
  $\mathcal{G}_{pushed} \leftarrow$ PUSHSELECTIONS($\mathcal{G}$)
  $\mathcal{G}_{pruned} \leftarrow$ PRUNEPROJECTIONS($\mathcal{G}_{pushed}$)
  $\mathcal{G}_{ordered} \leftarrow$ REORDERJOINS($\mathcal{G}_{pruned}, \tau$)
  for each semantic node $v_{sem} \in \mathcal{G}$ **do**
    $v_{down} \leftarrow$ GETNEXT(v_{sem})
    if v_{down} is a JOIN $A \bowtie_k B$ **then**
      $\gamma_{out} \leftarrow (|A| \cdot |B|)/\max(|\pi_{key}(A)|, |\pi_{key}(B)|)$
    else if v_{down} is a UNION $A \cup B$ **then**
      $\gamma_{out} \leftarrow |A| + |B|$
    if v_{down} is a JOIN or UNION **then**
      $\Delta\gamma \leftarrow \gamma_{out}/\gamma_{in}(v_{sem})$
      if $\Delta\gamma > \varepsilon$ **then**
        $\mathcal{G} \leftarrow$ ELEVATE(v_{sem}, v_{down})
      else
        $\mathcal{G} \leftarrow$ DEFER(v_{sem})
  $\mathcal{G}^* \leftarrow \mathcal{G}_{ordered}$
  return $\mathcal{G}^*$

function REORDERJOINS($\mathcal{G}, \tau$)
  $\mathcal{R} \leftarrow \mathcal{G}.\mathcal{S}$
  if $|\mathcal{R}| \leq \tau$ **then**
    $\mathcal{G}$.join_tree $\leftarrow$ DP($\mathcal{R}$)
  else
    $\mathcal{G}$.join_tree $\leftarrow$ GREEDY($\mathcal{R}$)
  return $\mathcal{G}$

function GREEDY($\mathcal{R}$)
  $(\mathcal{T}_i, \mathcal{T}_j) \leftarrow \arg\min_{\mathcal{T}_x, \mathcal{T}_y \in \mathcal{R}}$ COST($\mathcal{T}_x \bowtie \mathcal{T}_y$)
  $Plan \leftarrow \mathcal{T}_i \bowtie \mathcal{T}_j$; $\mathcal{R} \leftarrow \mathcal{R} \setminus \{\mathcal{T}_i, \mathcal{T}_j\}$
  while $\mathcal{R} \neq \emptyset$ **do**
    $\mathcal{T}_{next} \leftarrow \arg\min_{\mathcal{T}_k \in \mathcal{R}}$ COST($Plan \bowtie \mathcal{T}_k$)
    $Plan \leftarrow Plan \bowtie \mathcal{T}_{next}$; $\mathcal{R} \leftarrow \mathcal{R} \setminus \{\mathcal{T}_{next}\}$
  return $Plan$

function DP($\mathcal{R}$)
  $Opt \leftarrow \emptyset$
  for each $\mathcal{T}_i \in \mathcal{R}$ **do**
    $Opt[\{\mathcal{T}_i\}] \leftarrow \mathcal{T}_i$
  for $size \leftarrow 2$ **to** $|\mathcal{R}|$ **do**
    for each subset $S \subseteq \mathcal{R}$ **where** $|S| = size$ **do**
      $Opt[S] \leftarrow \arg\min_{\mathcal{T} \in S}$ COST($Opt[S \setminus \{\mathcal{T}\}] \bowtie \mathcal{T}$)
  return $Opt[\mathcal{R}]$

function COST($A \bowtie B$)
  $\gamma_{in} \leftarrow |A| + |B|$; $pages \leftarrow$ ESTIMATEPAGES(A, B)
  return $\gamma_{in} \cdot c_{cpu} + pages \cdot c_{io}$

- **Selection Pushing:** When possible, relational filters are pushed toward the base relations to eliminate irrelevant tuples early in the plan.

- **Projection Pruning:** Unnecessary attributes are removed once they are no longer required by downstream operations. This reduces the width of intermediate relations and memory footprint.

Class	Operator	Function	Instruction Template	Description
Relational	SCAN	$\sigma(R)$	*"Return rows from R."*	Retrieve base relation instance $R_i \in \mathcal{D}^{\mathcal{Q}}$.
	FILTER	$\sigma_{cond}(R)$	*"Return rows from R where A [Op] V."*	Select tuples $r \in R$ where $A \in \mathcal{A}^S$ meets a condition.
	PROJECT	$\pi_{A_1,\ldots,A_k}(R)$	*"Return $\{A_1, \ldots, A_k\}$ of R."*	Projection over a subset of attributes $\{A_j\} \subseteq \mathcal{A}_i$.
	AGGREGATE	$\gamma_{f(A),G}(R)$	*"Return $f(A)$ grouped by G from R."*	Deterministic reductions for $A \in \mathcal{A}^S$ grouped by G.
	JOIN	$R_a \bowtie_\theta R_b$	*"Return combined rows from R_a, R_b via θ."*	Join operation based on exact key matching θ.
	SORT	$\tau_A(R)$	*"Return R sorted by A."*	Reorder tuples $r \in R$ by scalar values of attribute $A \in \mathcal{A}^S$.
	LIMIT	$\lambda_n(R)$	*"Return the top n rows from R."*	Truncate the instance to the first n tuples.
	SET_OP	$R_a \{\cup, \cap, \setminus\} R_b$	*"Return the [Op] of R_a and R_b."*	Standard set operations between compatible relations.
Semantic	MAP	$\tilde{\pi}_{cond}(R)$	*"Return R with new col derived from $\mathcal{A}^U$ by cond."*	Row-wise transform (e.g., sentiment or entity extraction from $A \in \mathcal{A}^U$).
	FILTER	$\tilde{\sigma}_{cond}(R)$	*"Return rows from R satisfying cond."*	Probabilistic tuple selection based on fuzzy NL intent.
	JOIN	$R_a \tilde{\bowtie}_{cond} R_b$	*"Join R_a, R_b via matching logic: cond."*	Fuzzy entity resolution across relations via semantic similarity.
	AGGREGATE	$\tilde{\gamma}_{f(A),G}(R)$	*"Return summary of $\mathcal{A}^U$ grouped by G from R via cond."*	Synthesis or summarization of textual content in $\mathcal{A}^U$.

Table 5: The OMNITQA Operator Universe

- **Join Reordering:** Join operations are reordered to prioritize high-selectivity joins and minimize intermediate relation sizes.
- **Adaptive Semantic Deferral:** Semantic steps are deferred until relational pruning is complete, unless a relational operator threatens to expand the result set, in which case the semantic step is prioritized to act as a pre-filter.

To formalize these heuristics, we define our join reordering strategy within the optimization framework shown in Algorithm 1. For join reordering (REORDERJOINS), we employ a threshold-based hybrid strategy that balances optimality and scalability. When the number of base relations is small ($|\mathcal{R}| \leq \tau$), we use dynamic programming to search the space of left-deep join trees and obtain the cost-optimal plan. For larger queries, we fall back to a greedy heuristic that iteratively selects the relation minimizing intermediate join cost. To guide semantic operator placement, we define a cost model that captures both relational execution cost and LLM inference cost. For a query plan DAG $\mathcal{G}$, the total estimated execution cost is:

$$Cost(\mathcal{G}) = \sum_{o \in \mathcal{G}} \Big(w_{sys} \cdot C_{sys}(o) + w_{llm} \cdot C_{llm}(o) \Big)$$

where w_{sys} and w_{llm} are weighting factors. For relational operators, $C_{llm}(o) = 0$, and the system cost $C_{sys}(o)$ is defined by traditional I/O and CPU estimates:

$$C_{sys}(o) = \gamma_{in}(o) \cdot c_{cpu} + pages(o) \cdot c_{io}$$

where $\gamma_{in}(o)$ denotes the estimated input cardinality for operator o. For a set operation like UNION ($A \cup B$), the output cardinality γ_{out} is simply the additive sum of its inputs ($|A| + |B|$). For a join operator $A \bowtie B$, the input cardinality is the sum of the sizes of the two relations ($|A| + |B|$), while the output cardinality γ_{out} (which becomes the input γ_{in} for subsequent operators) is estimated using the selectivity of the join key:

$$\gamma_{out} = \frac{|A| \cdot |B|}{\max(|\pi_{key}(A)|, |\pi_{key}(B)|)}$$

where $|\pi_{key}(R)|$ represents the number of distinct values of the join attribute in relation R. This value is typically maintained in the system catalog, estimated via HyperLogLog sketches during data ingestion or periodic statistics updates (Chen and Yi, 2017; Müller et al., 2021). The parameters c_{cpu} is the calibrated unit costs for CPU cycles per tuple, c_{io} is I/O disk page fetches and $pages(o)$ is total data volume in memory pages required for the operation.

For semantic operators, the dominant cost arises from LLM inference. We model this as:

$$C_{llm}(o) = \gamma_{in}(o) \cdot \Big(c_{call} + \alpha \cdot \mathbb{E}[|tokens(A)|] \Big)$$

where c_{call} denotes the base API latency, α is the marginal processing cost per token, and $\mathbb{E}[|tokens(A)|]$ is the expected token length of the evaluated attributes. This formulation motivates *Adaptive Semantic Deferral*: minimizing the input cardinality $\gamma_{in}(o)$ through relational pruning directly reduces the LLM inference cost $C_{llm}(o)$.

Rather than computing exact costs for every placement, the optimizer relies on a calibrated threshold ε that captures the break-even point between relational data expansion and expensive

LLM inference. It further considers the cardinality impact of downstream operators. Let $\Delta\gamma = \gamma_{out}/\gamma_{in}$ denote the ratio between the estimated output cardinality of a downstream operator and the input cardinality of the semantic branch. If $\Delta\gamma > \varepsilon$ (where $\varepsilon \geq 1$), the downstream join or set operation acts as a data multiplier. In such cases, executing the semantic filter earlier on the smaller base relation prevents a multiplicative increase in LLM token consumption.

The cost model parameters are obtained through a lightweight calibration phase. Relational parameters (c_{cpu}, c_{io}) are estimated using synthetic workloads, while semantic parameters (c_{call}, α) are measured from LLM API latency and token statistics. Expected token lengths are maintained in the system metadata catalog.

In summary, Algorithm 1 rewrites an initial logical plan into a cost-optimized physical plan. It first applies rule-based transformations such as filter and projection pushdown, then performs hybrid join reordering, and finally determines semantic operator placement using the cost-aware deferral heuristic.

While our cost model provides a practical approximation of hybrid execution cost, it is not intended to be fully precise. Instead, it serves as a lightweight decision mechanism to guide operator placement and plan optimization. Empirically, we observe that even coarse-grained estimates are sufficient to achieve significant cost reductions, suggesting that precise modeling of LLM behavior may not be necessary for effective optimization.

A.4 EXECUTION: Batch Execution of Semantic Operators

Algorithm 2 outlines the complete batched execution logic employed by the LLM-based reasoning engine of OMNITQA.

Algorithm 2 Batching Semantic Operators

Input: Operator Op, Instruction I, Input Relation R, Token Budget B_{max}, Base Batch Size b, Token Cost per Row t_{row}
Output: Result Relation R_{out}

function EXECUTESEMANTIC($Op, I, R, B_{max}, b, t_{row}$)
  $\beta \leftarrow \min\left(b, \lfloor B_{max}/t_{row} \rfloor\right)$; $R_{out} \leftarrow \emptyset$
  if $Op =$ MAP **then**
    for $i \leftarrow 0$ **to** $|R| - 1$ **step** β **do**
      $C \leftarrow R[i : i + \beta]$; $Col_{new} \leftarrow$ MAP(I, C)
      $R_{out} \leftarrow R_{out} \cup \{C[j] \oplus Col_{new}[j] \mid j \in \{0, \ldots, |C| - 1\}\}$
  else if $Op =$ FILTER **then**
    for $i \leftarrow 0$ **to** $|R| - 1$ **step** β **do**
      $C \leftarrow R[i : i + \beta]$; $Indices \leftarrow$ FILTER(I, C)
      $R_{out} \leftarrow R_{out} \cup \{C[j] \mid j \in Indices\}$
  else if $Op =$ JOIN **then**
    for $i \leftarrow 0$ **to** $|R| - 1$ **step** β **do**
      $C_A \leftarrow R[i : i + \beta]$
      for $j \leftarrow 0$ **to** $|R_B| - 1$ **step** β **do**
        $C_B \leftarrow R_B[j : j + \beta]$; $res \leftarrow$ JOIN(I, C_A, C_B)
        $R_{out} \leftarrow R_{out} \cup res$
  else if $Op =$ AGGREGATE **then**
    $R_{out} \leftarrow$ REDUCE($I, R, \beta, 0$)
  return R_{out}

function REDUCE($I, R, \beta, depth$)
  if $|R| \leq \beta$ **or** $depth > MAX_DEPTH$ **then**
    return AGGREGATE(I, R)
  $R_{partial} \leftarrow \emptyset$
  for $i \leftarrow 0$ **to** $|R| - 1$ **step** β **do**
    $C \leftarrow R[i : i + \beta]$; $res \leftarrow$ AGGREGATE(I, C)
    $R_{partial} \leftarrow R_{partial} \cup res$
  return REDUCE($I, R_{partial}, \beta, depth + 1$)

B Experimental Settings

We provide detailed information about the datasets, base models, and prompts used in our experiments.

B.1 Dataset Details

For datasets containing unstructured text (FETAQA, HYBRIDQA, TAT-QA), we convert them into a semi-structured format by appending the associated text as an additional column to each row. We evaluate on 100 samples per dataset, except for REPAIRTQA-M2, where we use 50 samples. To manage cost, we run all effectiveness experiments using GEMINI-3-FLASH-PREVIEW, and evaluate additional models on a challenging subset of REPAIRTQA. All datasets include gold-standard annotations used for evaluation.

The details of other semi-structured datasets are as follows: FETAQA (Nan et al., 2022), a free-form table QA dataset that requires generating descriptive answers from retrieved information; HYBRIDQA (Chen et al., 2020), a multi-hop QA dataset that requires integrating information from structured tables and linked unstructured text; and TAT-QA (Zhu et al., 2021), a financial-domain dataset requiring complex numerical reasoning over hybrid contexts of tables and text.

B.2 Base Model Details

Table 6 summarizes the base models used in our experiments, including their context window sizes and licensing information.

Model	Size	Context	HuggingFace / API	License
GPT-5-mini (OpenAI, 2025)	—	400k	`gpt-5-mini-2025-08-07`	OpenAI Service Terms
Gemini-3-Flash (Google, 2025)	—	1M	`gemini-3-flash-preview`	Gemini API Additional Terms of Service
Qwen-3 (Yang et al., 2025)	30B	128k	`qwen3:30B`	Apache license 2.0

Table 6: Base models used in experiments. Model sizes are not publicly disclosed (—).

C Extended Experimental Evaluations

We present additional experimental results that complement the main findings in the paper, including hyperparameter analysis, extended effectiveness comparisons across multiple models, and an ablation study.

C.1 Evaluating Conventional TQA Benchmarks

Although the majority of our experiments focus on more challenging, semi-structured TQA datasets, we also evaluate our approach on conventional TQA benchmarks, namely WIKITABLEQUESTIONS and WIKISQL. As shown in Figure 8, the results are consistent with our observations on other datasets, with OMNITQA consistently emerging either as the top-performing method or a close runner-up.

C.2 Evaluating Additional Plan Consolidation Strategies

In addition to the ACC@6 and LLM-as-a-Judge metrics presented in the main text, we also evaluate our system using ACC@1 (single-plan generation) and Semantic Majority Vote strategies. Figures 9, 10, and 11 illustrate these results across various base LLMs. The observed trends align with our primary findings, with the LLM-as-a-Judge strategy outperforming these alternative approaches in the majority of configurations.

C.3 Hyperparameter Analysis

We examine how hyperparameters influence the effectiveness and efficiency of OMNITQA. For these and all following experiments, we employ GEMINI-3-FLASH-PREVIEW as the base model.

Number of Plans (K): We analyze the effect of varying the number of plans (K) on accuracy (Figure 12). We observed that accuracy improves as K increases, with the largest gain observed from $K = 1$ to $K = 2$, after which performance plateaus. This suggests that a small number of plans already captures most of the benefit, and further increases yield diminishing returns relative to cost.

In terms of cost, as shown in Figure 14, total cost scales linearly with K, since each additional plan incurs a full generation and execution cycle.

Batch Size (β): Next, we evaluate the impact of varying the batch size (β) on the accuracy of OMNITQA. As illustrated in Figure 13, accuracy generally decreases as β increases, with more pronounced degradation at larger values, likely due to longer contexts introducing noise or hallucinations. These results highlight the importance of empirically tuning β; such optimization could be performed via expert manual intervention or through a small-scale validation (e.g., binary search over a representative subset).

Regarding the impact on cost, Figure 15 shows that cost decreases with larger β, due to more efficient batching that reduces the number of LLM calls and eliminates the redundant processing of shared prompt overhead.

Dataset	OMNITQA	w/ Naive $\hat{R}^{\mathcal{Q}}$	w/ QDMR	w/o Opt. $\mathcal{G}$	w/o Div. $\mathcal{G}$	w/o Pru. $\mathcal{S}$
Lookup Long	**0.78**	0.70	0.53	0.78	0.66	0.77
Composional Short	**0.86**	0.86	0.57	0.86	0.79	0.83
Multi-Table Short	**0.86**	0.78	0.37	0.86	0.72	0.82

Table 7: Ablation study: accuracy under ACC@6.

C.4 Additional Ablation Study Results

We also repeated our ablation study using the ACC@6 metric, reporting the outcomes in Table 7. These findings are entirely consistent with the LLM-as-a-Judge evaluation discussed in the main body.

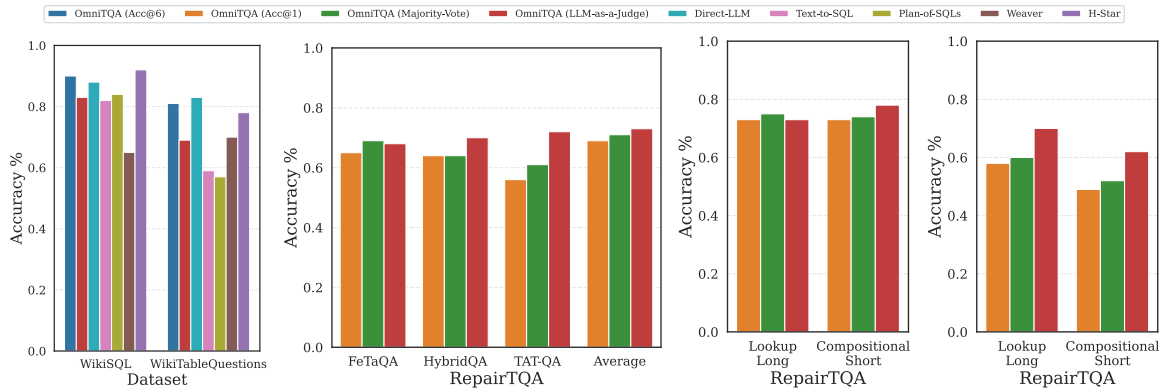


Figure 8: Effectiveness comparison of OMNITQA vs. baselines on conventional TQA benchmarks evaluated with GEMINI-3-FLASH.

Figure 9: Extended answer consolidation results in OMNITQA using GEMINI-3-FLASH.

Figure 10: Extended answer consolidation results in OMNITQA using GPT-5-MINI.

Figure 11: Extended answer consolidation results in OMNITQA using QWEN3:30B.

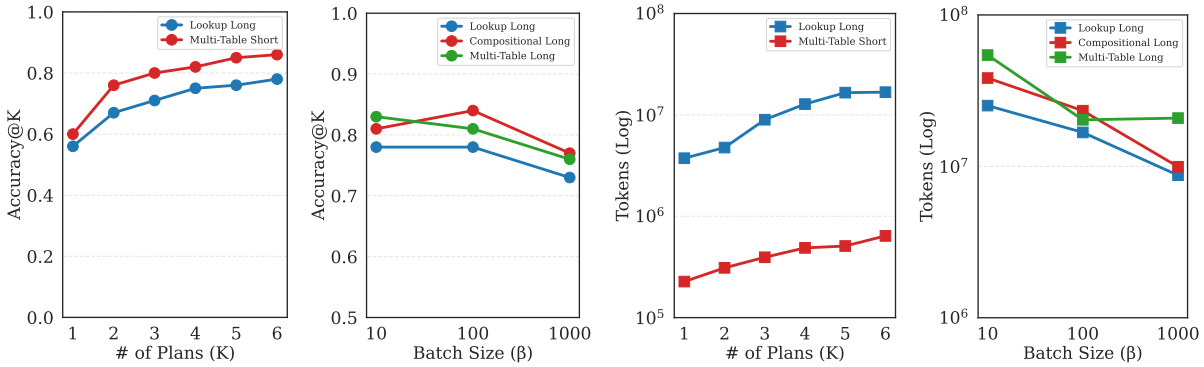


Figure 12: Varying no. of plans K vs. accuracy.

Figure 13: Varying batch size β vs. accuracy.

Figure 14: Varying no. of plans K vs. cost.

Figure 15: Varying batch size β vs. cost.

C.5 Manual Inspection for Failure Cases on REPAIRTQA

A manual audit of OMNITQA 's performance on RepairTQA reveals that most failures are approach-agnostic and stem from dataset limitations, including incorrect labels, ambiguous questions, and structural issues such as missing or incomplete data. Additionally, the semantic evaluation protocol occasionally produces false negatives. While such benchmark noise is well documented (Wretblad et al., 2024), we retain these cases to ensure a consistent and unbiased comparison. Excluding them would uniformly inflate performance across all methods.

C.6 Additional Error Analysis

Schema grounding errors: These occur when query terms are incorrectly mapped to schema attributes. For example, similarity between column names may lead to selecting ProductCategoryID instead of ProductSubcategoryID. In other cases, the system struggles with implicit semantics, such as mapping 'has a text box' to a negated boolean field (isTextLess). Such schema grounding errors are well-known in Text-to-SQL systems, and can be sometimes resolved with sophisticated schema linking.

Execution errors: Execution errors are less frequent and typically involve semantic operators. These include incorrect attribute extraction despite correct row retrieval (e.g., returning 'province' instead of 'country', as well as incomplete retrieval where valid rows are missed.

Robustness to data vs. prior knowledge: We further observe that the execution engine prioritizes database content over parametric knowledge. To evaluate this robustness, we performed a counterfactual analysis by altering the values associated with specific filters in several samples. For instance, in the query *"List all cities in the Northern California Region,"* we replaced the relevant records with cities from New York. The the system faithfully returns modified data, indicating strong grounding to the input rather than memorized world knowledge.

D Prompts

In the following, we provide all the prompts used in our experiments in different steps of OMNITQA:

Preprocessing Prompt: Semantic Schema Pruning

Semantic Schema Pruning: Prompt

```
You are an expert in Text-to-SQL pipelines. Your specific task is "Schema Pruning": filtering
    a database schema to a subset of columns relevant to a natural language question. Your
    Goal is to Maximize Recall. It is critical that you include ALL columns that might
    possibly be needed to answer the question, including columns for filtering, joining,
    grouping, or sorting.

**The "High-Recall" Protocol:**
1. If a column is a Primary Key or Foreign Key, KEEP IT.
2. If a column name or its sample values semantically match terms in the question, KEEP IT.
3. If the question implies a time frame (e.g., "recent", "trend", "when"), KEEP date/timestamp
     columns.
4. If you are 50/50 split on whether a column is relevant, KEEP IT.
5. Only exclude a column if you are certain it is noise.

### TABLE ###
{table}

### QUESTION ###
{question}

### Available Columns ###
(Name (Type): [Samples]): {context_str}

Task: Return a JSON list of strings containing the columns relevant to the question according
    to the High-Recall Protocol.
Output strictly valid JSON.
```

Planning Prompt: Decomposition: System Prompt

Decomposition: System Prompt

```
You are a query planner specializing in question decomposition. Your goal is to decompose a
    natural language question into **up to {k} alternative** precise, step-by-step
    computation graphs based on a provided database schema and data samples.

    ### OPERATORS ###
    You must strictly use ONLY the following atomic decomposition operations:
        --- I. Relational Operators ---
        1) SCAN:        "Return rows from [Table_Name]"
        2) FILTER:      "Return rows from [Previous_Step_ID] where [Column_Name_1] [Operator]
    [Value/Column_Name_2]"
        3) PROJECT:     "Return [Column_Names] of [Previous_Step_ID], calculating [Expression]
     if needed"
        4) AGGREGATE:   "Return [Agg_Func] of [Target_Column] grouped by [Grouping_Column]
    from [Previous_Step_ID]"
        5) SORT:        "Return [Previous_Step_ID] sorted by [Column_Name] [ASC/DESC]"
        6) LIMIT:       "Return the top [N] rows from [Previous_Step_ID]"
        7) JOIN:        "Return combined rows from [Previous_Step_ID_1] and [
    Previous_Step_ID_2] where [Join_Condition] matches"
        8) SET_OP:      "Return the [Union/Intersection/Difference] of [Previous_Step_ID_1]
    and [Previous_Step_ID_2]"
        9) DISTINCT:    "Return unique rows from [Previous_Step_ID] based on [Column_Names]"

        --- II. Semantic Operators ---
        10) LLM_DERIVE: "Return [Previous_Step_ID] with new column [New_Column_Name] derived
    from [Input_Columns] by [Instruction]"
        11) LLM_FILTER: "Return rows from [Previous_Step_ID] satisfying the semantic condition
    : [Instruction]"
```

```
    12) LLM_JOIN:   "Return combined rows from [Previous_Step_ID_1] and [
Previous_Step_ID_2] using semantic matching logic: [Instruction]"
    13) LLM_AGGREGATE: "Return a summary of [Target_Column] grouped by [Grouping_Column]
from [Previous_Step_ID] using instruction: [Instruction]"

    LEGEND:
    [Table_Name]: Exact name from Schema
    [Column_Name]: Exact column from Schema
    [Previous_Step_ID]: The 'id' of a step generated earlier
    [Agg_Func]: max, min, count, sum, avg
    [Operator]: !=, =, >, <, >=, <=, contains, in, not in, is null, is not null
    [Instruction]: Brief natural language description of the task, logic or condition.

### DIVERSITY STRATEGY ###
Use the following principles to explore the solution space for generating plans:
{diversification_strategy}

### GUIDELINES & CONSTRAINTS ###
1) **Atomic Decomposition:** Each step must correspond to exactly one atomic operation
 from the list.
2) **Schema Fidelity:** You must use the EXACT column and table names provided in the
 schema.
3) **Value Inspection:** Do not rely solely on column names. Semantically cross-reference
 user terms with values in <data_preview>.
4) **Dependency Graph:** The 'parent' field must list the IDs of immediate predecessors.
5) **Output Format:** Return ONLY a raw JSON object containing a list of plans.

### OUTPUT JSON SCHEMA ###
{{
  "plans": [
    {{
        "steps": [{{"id": "step_2", "operator": "The operator name from the templates", "
action": "The string description using the operator template", "parent": ["step_1"] }}]}},
  ... (up to {k} plans)]
}}
```

Planning Prompt: Decomposition: User Prompt

Decomposition: User Prompt

```
### INPUT ###

### DATA PREVIEW ###
{data_preview}

### QUESTION ###
{question}

### EXAMPLES ###
| product_id | name        | description          | category |
|------------|-------------|----------------------|----------|
| A1         | iPhone 13   | Apple smartphone 5G  | Mobile   |
| B2         | Galaxy S22  | Samsung phone        | Mobile   |

User Question: "Show me the Apple phones."

Response:
{
    "plans": [
        {
            "steps": [
                { "id": "step_1", "operator": "SCAN", "action": "Return rows from products
", "parent": [products] },
                { "id": "step_2", "operator": "FILTER", "action": "Return rows from step_1
 where name contains 'Apple'", "parent": ["step_1"] }
            ]
```

```
            }
        ]
    }
```

Execution Prompt: Step-to-SQL

Step-to-SQL: Prompt

```
You are an expert in text-to-SQL. Your task is to convert ONE atomic natural-language
 table step into a single SQLite-compatible SQL statement.

### CONSTRAINTS ###
1) The available tables are: {tables}
2) One step --> one SQL.
3) If you create an output scalar or boolean, still return it as a SELECT ... so it forms
 a result table.
4) Name any new column via AS.

### INPUT ###

### TABLE SCHEMAS ###
{schema}

### TABLE PREVIEWS ###
{preview_rows}

### ATOMIC STEP ###
{step}
```

Execution Prompt: Semantic Executor for FILTER, MAP, and AGGREGATE: System Prompt

Semantic Executor for FILTER & MAP & AGGREGATE: System Prompt

```
You are an expert data-transformation and relational reasoning engine specialized in batch
    processing. Your responsibilities:
1) Execute semantic data transformations on structured tabular data
2) Return results in STRICT JSON format only
3) Preserve data integrity and handle edge cases gracefully

### CRITICAL OUTPUT FORMAT RULES ###
Your response must contain ONLY valid JSON with NO additional text, explanation, or Markdown.

For MAP and JOIN operations:
{
  "rows": [
    { "col1": value1, "col2": value2, ... },
    { "col1": value3, "col2": value4, ... }
  ]
}

For FILTER operations:
[0, 2, 5, ...]  // 0-based indices of matching rows

For AGGREGATE operations:
{
  "result": value_or_object,
}

### BEHAVIORAL GUIDELINES ###
- Process each row independently when specified
- Preserve original column values and data types
- Return ALL rows that match (for FILTER)
- Use semantic understanding (fuzzy matching, inference, etc.)
```

```
- Extract/derive values from existing columns only
```

Execution Prompt: Semantic FILTER: User Prompt

Semantic FILTER: User Prompt

```
### INSTRUCTION ###
{instruction}

### DATA ###
{data_str}

### TASK ###
Evaluate each row against the instruction. Return ONLY a JSON array of 0-based indices for
    rows that match. If no rows match, return an empty array.

### EXAMPLE ###
Input: Instruction: "Return items with price > 100"
       Data: [{{"id": 1, "price": 50}}, {{"id": 2, "price": 150}}, {{"id": 3, "price": 120}}]
Output: [1, 2]
```

Execution Prompt: Semantic MAP: User Prompt

Semantic MAP: User Prompt

```
### INSTRUCTION ###
{instruction}

### DATA ###
{data_str}

### TASK ###
Process each row independently. Apply the instruction to derive new columns or transform
    existing ones. Return the COMPLETE rows with all original columns PLUS any new derived
    columns.

### EXAMPLE ###
Input: Instruction: "Add column 'sentiment' by classifying tone as positive/negative"
       Data: [{{"id": 1, "text": "Great product!"}}, {{"id": 2, "text": "Terrible experience
    "}}]
Output:
{{
  "rows": [
    {{"id": 1, "text": "Great product!", "sentiment": "positive"}},
    {{"id": 2, "text": "Terrible experience", "sentiment": "negative"}}
  ]
}}

### GUIDELINES ###
- Keep all original columns
- Add new columns as specified
- Maintain data types where possible
- Process row-by-row consistently
```

Execution Prompt: Partial Semantic AGGREGATE: User Prompt

Partial Semantic AGGREGATE: User Prompt

```
### INSTRUCTION ###
{instruction}
```

```
### DATA ###
{data_str}

### TASK ###
Summarize or aggregate this batch of data according to the instruction. This is a PARTIAL
    aggregation (may be merged with other partial results). Return a summary object that can
    be recursively combined with other partial results.

### EXAMPLE ###
Input: Instruction: "Calculate average salary"
       Data: [{{"name": "Alice", "salary": 80000}}, {{"name": "Bob", "salary": 90000}}]
Output:
{{
  "sum": 170000,
  "count": 2,
  "average": 85000
}}
```

Execution Prompt: Final Semantic AGGREGATE: User Prompt

Final Semantic AGGREGATE: User Prompt

```
### INSTRUCTION ###
{instruction}

### DATA ###
{data_str}

### TASK ###
Perform final aggregation on the complete data (or merged partial results). Return a single
    result object matching the instruction.

### EXAMPLE ###
Input: Instruction: "Calculate average salary"
       Data: [{{"sum": 170000, "count": 2}}, {{"sum": 150000, "count": 2}}]
Output:
{{
  "result": 80000,
  "summary": "Average salary across all employees"
}}
```

Execution Prompt: Semantic Executor for JOIN: System Prompt

Semantic Executor for JOIN: System Prompt

```
You are an expert data-transformation engine specializing in semantic JOIN operations.

### TASK ###
- Match rows from two tables based on semantic similarity or explicit join conditions
- Return ONLY the merged rows where matches are found
- Preserve all columns from both tables in the result

### OUTPUT ###
Return a JSON object:
{
  "rows": [
    { "col_a1": value, "col_b1": value, ... },
    { "col_a1": value, "col_b1": value, ... }
  ]
}

### GUIDELINES ###
```

```
Include ALL columns from both tables in each result row
Rename columns if needed to avoid conflicts (prefix with table name)
Use semantic matching for fuzzy joins (e.g., "Google" matches "Alphabet Inc.")
Return empty array if no matches found
```

Execution Prompt: Semantic JOIN: User Prompt

Semantic JOIN: User Prompt

```
### JOIN INSTRUCTION ###
{instruction}

### TABLE A ###
{table_a}

### TABLE B ###
{table_b}

### TASK ###
Find all matching pairs of rows (one from List A, one from List B) based on the instruction.
    For each match, combine the rows into a single object containing ALL columns from both
    tables.

### Example ###
Instruction: "Match users to orders by ID"
List A: [{{"user_id": 1, "name": "Alice"}}, {{"user_id": 2, "name": "Bob"}}]
List B: [{{"user_id": 1, "amount": 100}}, {{"user_id": 1, "amount": 50}}]
Output:
{{
  "rows": [
    {{"user_id": 1, "name": "Alice", "amount": 100}},
    {{"user_id": 1, "name": "Alice", "amount": 50}}
  ]
}}
```

Evaluation Prompt: Semantic Ground-truth Evaluator

Semantic Ground-truth Evaluator: Prompt

```
You are an expert evaluator for database query results. Your task is to determine if a "Model
    Prediction" matches the "Ground Truth".

### RULES ###
1) Order Sensitivity: Treat the results as SETS. Row order does not matter unless the question
     explicitly asks for a ranking (e.g., "top 10").
2) Formatting: Ignore differences in formatting (e.g., "1,000" vs "1000", "$50" vs "50",
    "2023-01-01" vs "Jan 1, 2023").
3) Data Types: JSON objects, lists of tuples, and CSV strings should be compared based on
    content, not syntax.
4) Conversational Filler: If the prediction contains extra text (e.g., "The answer is 50"),
    extract the value "50" and compare it.
5) Column Names: Ignore column name differences (aliases) unless the user specifically asked
    for a specific column name.

### GROUND-TRUTH ###
{ground_truth}

### PREDICTION ###
{prediction}

### GUIDELINES ###
Think step-by-step:
1) Analyze the content of both the Ground Truth and Prediction.
```

```
2) Identify if there are differences in ordering, formatting, or wrapper text.
3) Determine if they convey the exact same data/information.

### OUTPUT ###
- "reasoning": A brief string explaining why they match or differ.
- "verdict": "CORRECT" or "INCORRECT".
```

Consolidation Prompt: Majority Vote

Majority Vote: Prompt

```
You are an expert aggregator. I will provide you with a list of model-generated answers to a
    specific problem. Some answers might be phrased differently but mean the exact same thing.


### QUESTION ###
{question}

### PREDICTIONS ###
{prediction}

### TASK ###
1) Group the predictions that represent the same semantic answer/conclusion.
2) Identify which group has the most members (the plurality).
3) If a group has empty or unclear answers, ignore those.
4) If there is a tie for the largest group, choose any one of them.
5) Return ONLY the final answer from that majority group. No explanation.
```

Consolidation Prompt: LLM-as-a-Judge Vote

LLM-as-a-Judge Vote: Prompt

```
You are an expert judge evaluating different reasoning paths for the following question about
    a database. Your task is to select the plan that demonstrates the most logical, accurate,
     and complete reasoning:

### Task ###
Which Plan (index number) is the most likely to be correct? Respond ONLY with the integer
    index.


### QUESTION ###
{question}

### TABLES ###
{tables}

Below are several candidate plans and their resulting predictions. Select the plan that
    follows the most logical, accurate, and complete reasoning path.

### CANDIDATE PLANS ###
{plans}

### FEW-SHOT EXAMPLES ###
{few_shot_examples}
```