SPARQ: A Cost-Efficient Framework for Offline Table Question Answering via Adaptive Routing

Yang Liu[1], Mengyi Yan[2*], Jiao Xue[4], Weilong Ren[3], Yutong Ye[1], Haoyi Zhou[1], Jianxin Li[1*], Zhumin Chen[2]
[1]Beihang University [2]Shandong University [3]Shenzhen Institute of Computing Sciences
[4]Inspur Cloud Information Technology Co., Ltd.
{ly_act, yutongye, haoyi, lijx}@buaa.edu.cn, {yanmy, chenzhumin}@sdu.edu.cn, renweilong@sics.ac.cn, xuejiao02@inspur.com

***Abstract*—Table Question Answering (TQA), which aims to answer natural language questions over tabular data, has recently attracted growing interest in the database community. While state-of-the-art (SOTA) methods based on online Large Language Models (LLMs) achieve remarkable accuracy, they suffer from several drawbacks, including data privacy risks, high latency, high cost, and overthinking due to excessively long reasoning chains. To address these challenges, we propose SPARQ (Sufficient Precise Adaptive Routing for TableQA), a cost-efficient TQA framework designed for robust offline deployment. SPARQ extends the operator pool for table reasoning and introduces an adaptive query routing mechanism that dynamically selects optimal operators, assisted by a verifier with rollback/fallback strategies. Through extensive evaluations, we demonstrate that SPARQ achieves remarkable performance in an offline setting: it improves accuracy by over 5% on WikiTQ and Tab Fact datasets, while reducing the average end-to-end latency by up to $10.19\times$ on consumer-grade hardware (*e.g.,* RTX 4090). To the best of our knowledge, this is the first systematic framework that deploys offline LLMs to achieve SOTA performance for TQA under realistic hardware constraints, balancing both effectiveness and efficiency. Code, full version and artifacts are provided.[1]**

***Index Terms*—Table Question Answering, Offline Large Language Models, Adaptive Query Routing.**

I. Introduction

Table Question Answering (TQA), a core task in table reasoning, bridges natural language understanding and structured data analysis [1], [2]. It aims to derive concise factual answers, detailed explanations, or fact-checking results from a user's textual query over schema-free tables [3], *e.g., which country was the first to reach new heights with a skyscraper?* TQA is inherently challenging, as it requires translating ambiguous natural language into executable reasoning steps that combine logical, numerical, and textual inference over unstructured or semi-structured tables [4], [5]. Recently, the database community has increasingly studied TQA as a natural interface for querying heterogeneous data without predefined schemas [6]–[12]. Despite these difficulties, TQA holds great promise: it empowers non-experts to query and reason over large-scale data using plain language, producing interpretable and verifiable analytical results [2].

Research in TQA has evolved rapidly across three major stages. (a) *Supervised methods* rely on annotated datasets to train sequence-to-sequence models that map table–query pairs to answers [13]–[18]. (b) *LLM-based approaches* leverage in-context learning, often enhanced by retrieval mechanisms such as RAG [19] and table decomposition [20], [21]. Specifically, diverse strategies have emerged, including Chain-of-Thought prompting [22], multi-agent collaboration [9], and program generation via SQL/Python [23]. (c) *Hybrid frameworks*, *e.g.,* H-STAR [24] and TableRAG [19], integrate symbolic and semantic reasoning via hierarchical operators that repeatedly invoke LLMs and SQL engines for information aggregation.

Despite their success, most existing systems are designed around *online LLM services* (*i.e.,* cloud-hosted models such as GPT [25], [26] and PaLM [27]), with relatively little attention to *offline LLMs* (*i.e.,* locally deployed models such as LLaMA [28] and Qwen [29], [30]). This design bias becomes increasingly problematic as the community shifts toward reducing monetary cost and latency while preserving practical deployability [31], [32]. As a result, existing methods face three fundamental challenges: (a) *Monetary cost and efficiency.* Hybrid reasoning pipelines often invoke LLMs repeatedly, incurring high monetary cost and latency, and frequently "overthink" even for simple queries [7], [10], [21]. (b) *Reasoning and privacy limitations of* LLMs. Even advanced LLMs struggle with structured or long-form inputs and exhibit weak numerical reasoning, leading to hallucinations and error propagation [6], [33], [34]. Querying online LLMs further exposes sensitive user or table information to external servers, risking privacy leakage [35]–[37]. (c) *Data and resource constraints.* Training-heavy methods require extensive annotations and centralized data, which are costly and often infeasible for individual users or organizations; moreover, while offline deployment naturally avoids external data exposure, it must operate under strict memory and compute budgets.

As online LLMs become increasingly powerful yet expensive, there is rising interest in low-cost, low-latency alternatives that can run on consumer-level hardware [38]. A naive solution is to directly replace online models with smaller offline ones, which are faster and incur zero API cost. However, as shown below, this naive substitution performs poorly.

Example 1: Fig. 1 illustrates the pitfalls of existing TQA approaches for the query *"How many games were played after October 1st?"* on a football match table from the WikiTQ dataset [5], where the correct answer is "11". (a) Feeding the entire table to an offline LLM (*i.e.,* Answer 1) overwhelms its context window and numerical reasoning ability, producing a hallucinated answer of "10". This demonstrates the *long-context collapse* problem [6], [34], where LLMs fail to reason

*Corresponding authors.
[1]https://github.com/authurlord/SPARQ

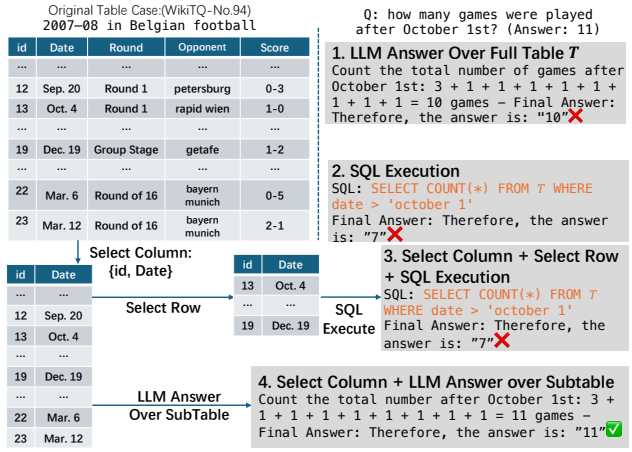


Fig. 1: Comparison of TQA strategies on WikiTQ: (a) offline LLMs (Answer 1), (b) SQL-based reasoning (Answer 2), (c) hybrid method (Answer 3), and (d) a concise yet sufficient Select Column + LLM Answer strategy (Answer 4).

effectively over long and structured inputs. (b) Symbolic methods relying on auto-generated SQL (*i.e.,* Answer 2) also fail because the Date column is semi-structured (*e.g.,* "Oct. 4") and spans multiple years, which string-based comparisons misinterpret, leading to an incorrect result of "7". (c) Even hybrid pipelines that attempt to filter rows and columns before execution (*i.e.,* Answer 3) simply propagate these underlying errors and returns a wrong answer "7", since they omit next-year games in rows 20–23. This indicates that increasing CoT reasoning depth does not ensure correctness; it often adds computational overhead while failing to address fundamental issues such as handling semi-structured data.

These observations suggest that *naively swapping an offline LLM into an online-oriented TableQA pipeline does not work*. This naturally raises three key questions: (1) Can we accurately answer TQA queries using offline LLMs with no extra monetary cost? (2) Can such offline models achieve accuracy comparable to or even surpassing state-of-the-art online systems despite hardware and model-size constraints? (3) Can we accelerate table reasoning by adaptively selecting the optimal strategy for each query, balancing computational efficiency with practical deployability?

Contributions & Organization. To address the challenges discussed above, this paper presents SPARQ (Sufficient and Precise Adaptive Routing for TQA), a cost-efficient framework that enables accurate and efficient offline table reasoning on consumer-level hardware.

The main contributions are summarized as follows:

(1) Preliminary analysis (Section III). We systematically analyze the failure modes of existing TQA methods and identify common pitfalls on hard samples. Guided by these insights, we distill key empirical findings and show that offline models benefit from *minimal yet sufficient* information for effective TableQA.

(2) Framework (Section IV). We propose SPARQ, a cost-efficient and adaptive framework for offline TQA that integrates a dynamic query router with a mutual-information verifier. The router and verifier collaborate to approximate a *minimally sufficient* context for each query, enabling accurate reasoning with low computational cost. SPARQ also features an extended pool of semantic and symbolic operators for flexible reasoning and employs coordinated CPU–GPU scheduling to ensure efficient and scalable execution.

(3) Query routing mechanism (Section V-A). We develop a lightweight and semantic-aware router E that dynamically selects the most suitable operator set for each query based on its predicted query cost. This adaptive mechanism minimizes redundant computation, mitigates LLM overthinking on simple queries, and prevents information loss from overly aggressive pruning on complex ones, balancing accuracy and efficiency.

(4) Verification model (Section V-B). To ensure reliable routing, we introduce a verification model Q that learns a unified representation space for natural-language queries, symbolic code, and tabular data through contrastive learning [39], [40]. By maximizing cross-modal mutual information, Q evaluates in real time whether the pruned subtable and selected operations contain sufficient evidence to answer the query, enabling automatic rollback or fallback when necessary.

(5) Scheduling strategy (Section V-C). We accelerate SPARQ 's execution with a cost-aware dynamic batching scheduler optimized for offline TQA. Leveraging cost estimates from E, the scheduler groups queries of similar difficulty and overlaps CPU-based retrieval with GPU-based inference. This joint optimization minimizes idle time and achieves on average $12.19\times$ speedup on 4B–30B models, making SPARQ efficient and practical for on-premise deployment.

(6) Experimental evaluation (Section VI). Extensive experiments show that SPARQ, fully deployed offline on two RTX 4090 GPUs, beats existing methods by over 5% in accuracy and achieves average speedup of $10.19\times$ in end-to-end latency.

Finally, we review related work in Section II and conclude the paper in Section VII.

II. Preliminary and Related Work

The TQA task maps a query q and table T to an answer a, which may be short-form or natural language. Although LLMs have become the backbone of TQA due to their cross-modal reasoning capabilities, "out-of-the-box" LLMs still face persistent challenges in multi-step inference, symbolic operations, and external knowledge grounding [41], [42]. Related literature can be classified into the following themes.

Large Language Models for TQA. LLMs have revolutionized natural language processing and human–computer interaction [7], [43]. They can be broadly categorized into cloud-hosted proprietary models and on-premise open-weight models. (In this paper, we refer to the former as "online models" and the latter as "offline models"). Online models (*e.g.,* GPT [25] and PaLM [27]) offer robust general capabilities via remote APIs, yet their reliance on cloud infrastructure often entails high latency, privacy risks, and restricted deploy-

ability in isolated environments. In contrast, offline models (*e.g.,* Qwen [44], LLaMA [28]) enable local deployment and full data control, but their constrained parameter scales often result in less stable performance on complex reasoning tasks.

Despite their broad linguistic ability, transformer-based LLMs perform poorly on TQA due to weak numerical reasoning [33], hallucinations [45], and limited understanding of relational structures [8]. Although some research has focused on pre-training table-tailored LLMs to mitigate these issues [8], [46], but they remain costly and underperform in practice. Thus, neither generic nor specialized training-based LLMs provide an ideal solution for TQA.

Natural Language to SQL. Translating natural language questions into executable SQL (NL2SQL) has long been studied. Early works relied on handcrafted grammars [47], [48] but struggled with linguistic ambiguity. This led to a paradigm shift towards model-based methods, which learn complex mapping between natural language and code from large datasets [49]–[51]. With the rise of LLMs, few-shot prompting enabled direct SQL/code generation [25], [26], enhanced by constrained decoding and syntax verification [52], [53]. Benchmarks like Spider [43], CoSQL [54], WikiSQL [55], and BIRD [56] further accelerated such progress.

Recent TQA systems extend this paradigm through iterative SQL reasoning. ReAcTable [7] integrates step-by-step code execution; LEVER [53] adds verification of execution results; and TabSQLify [23] performs multi-step SQL-based table extraction, feeding subtables back to LLMs for final answers. However, such approaches often suffer from information loss and low SQL parsing success rate with offline LLMs, making purely programmatic pipelines unreliable for robust TQA.

Chain-of-Thought Reasoning. Chain-of-Thought (CoT) enhances LLMs' multi-step reasoning by decomposing complex problems into sequential steps [42]. Multi-agent frameworks extend this idea by coordinating specialized agents that collaboratively solve complex tasks [9], [10], [57]. However, as a test-time scaling solution [58], CoT-based methods often suffer from *LLM overthinking* [59], [60]: the tendency to apply overly complex reasoning to simple problems. This incurs significant computational overhead without improving final accuracy [11], an issue widely observed in domains like mathematical QA [61], [62].

Methods like DATER [20], CoT [22], H-STAR [24] and RoT [21] adopt CoT-style step-by-step reasoning, while AutoTQA [9], [10] employs multi-agent decomposition for multi-table queries. However, these methods rely heavily on online LLMs; when transferred to smaller offline models, their performance drops sharply and hallucinations increase due to limited context capacity. Unlike these paradigms, SPARQ employs a lightweight router to assign queries to pre-defined operators and a verifier to assess sufficiency, balancing effectiveness, efficiency, and stability in offline settings.

III. Motivation and Preliminary Analysis

Motivation. In practice, directly substituting offline LLMs into pipelines designed for online models leads to critical failure modes: (1) Even though offline LLMs may support large context windows (*i.e.,* the input can fit), their effective reasoning range over long or complex tables is often much smaller [45], [63], [64]. As context length increases, their numerical reasoning and code generation abilities degrade sharply, exhibiting a "reasoning cliff" well before the architectural context limit; and (2) while existing systems rely on iterative multi-round symbolic execution with online LLMs, the same complex pipelines often induce severe "overthinking" when used with offline models. For smaller local models, long reasoning chains amplify error propagation, whereas simpler, direct operators frequently yield more reliable results.

We validate these observations through controlled experiments in which all methods are evaluated under a unified offline setting, and we further explore strategies to mitigate these issues, particularly for offline LLMs. The details are described below.

Datasets. We conducted our tests on the WikiTQ dataset [5], a widely used benchmark for TQA. Following H-STAR [24], We selected 476 challenging samples from its validation set whose table token count exceeds 4,000. On average, these tables contain 78.5 (resp. 7.73) rows (resp. columns), up to 753 (resp. 21).

Methods. We tested eleven methods: seven representative single-operator approaches, two state-of-the-art (SOTA) hybrid methods for TQA, and two hand-drafted hybrid methods.

1) Base: a baseline that provides LLM with full table T and three few-shot demonstrations. Following [42], it uses chain-of-thought, prompting LLM to divide queries into several sub-tasks. The demonstrations and CoT prompt are shared across all methods below.
2) Select_Column: prompts the LLM to select relevant columns, forming a subtable $T' \subset T$ for the final QA.
3) Select_Row: prompts the LLM to generate two SQL queries that retrieve relevant rows; their intersecton serves as additional QA context.
4) Execute_SQL: iteratively generates SQL queries to answer the question, for up to three rounds; In each round, three SQL candidates are executed on the full table, and their combined results are provided as additional context for the next round, until LLM generates direct answer.
5) Execute_Py: iteratively generates Python codes over pandas DataFrames, up to 3 times, until successful execution.
6) RAG: a retrieval-augmented approach [19] that treats table rows and columns as independent documents and retains up to 50 rows and 10 columns by their semantic similarity with the question.
7) RAG +Rewrite: A hybrid retrieval approach that first leverages LLM for multi-level query rewriting, and then combines dense retrieval (semantic similarity) with sparse retrieval (BM25), merged by reciprocal rank fusion. A

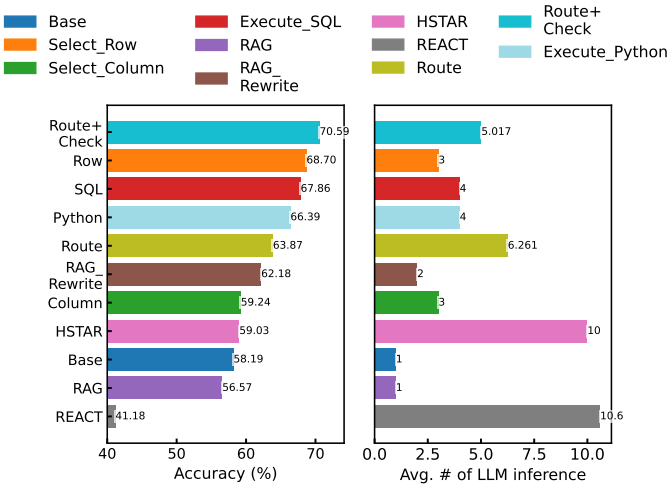


Fig. 2: Preliminary test result.

subsequent re-ranker model weights and reorders these results to select the final rows and columns, retaining up to 50 rows and 10 columns.

8) H-STAR [24]: a SOTA CoT-based approach that performs three sequential steps: column selection, row selection, and SQL execution.
9) ReAcTable [7]: a SOTA NL2SQL-based approach that iteratively performs 5 rounds of subtable selection and SQL execution, followed by majority voting(*s-vote*).
10) Route/Route +Check: following [65], it first prompts LLM to generate a combination of table operators from the operators mentioned above, and then executes them in order. For Route +Check, we manually verify the LLM-selected operators and remove the unnecessary operators.

Configuration. All experiments were conducted using Qwen3-4B-Instruct-2507 [44] as the core offline LLM for TQA, generating SQL/Code and all intermediate results. For retrieval, we adopted a hybrid configuration that combines BGE-m3 for dense embedding, BGE-v2-reranker-m3 for re-ranking, and BM25 for sparse matching [66], [67]. All tests run on a workstation with two NVIDIA RTX 4090 GPUs.

Metric. We used two metrics: (a) accuracy, measured as the ratio of correctly generated answers; and (b) computation cost, quantified by the average number of LLM calls per query, which reflects both efficiency and resource consumption.

Test findings. Fig. 2 shows the performance of different methods. Our key observations are as follows.

(1) Offline LLMs struggle with long-form tables. Base performs poorly, with 0.582 accuracy on the hard dataset. This can be mitigated by (a) providing smaller and more informative inputs, *e.g.,* Select_Row outperforms Base by 18.06% by querying the LLM with pruned tables; or (b) leveraging SQL-based computation, *e.g.,* Execute_SQL beats Base by 16.62% by executing LLM-generated SQL queries on the full tables.

(2) Symbolic execution is powerful but brittle. While Select_Row, Select_Column, Execute_SQL and Execute_Py outperform Base by up to 18.06%, their reliance on strict syntax introduces significant fragility. Internal analysis shows high failure rates that reach 30.30%, primarily due to syntax or execution errors and an intolerance for "dirty" table data. Although symbolic methods excel at numerical computation, their rigid execution requirements hinder further gains in real-world TQA with offline LLM settings.

(3) Overthinking wastes resources without accuracy gains. Complex hybrid methods often incur heavy computational overhead without proportional gains, *e.g.,* H-STAR and ReAcTable require $3.3\times$ and $3.5\times$ more LLM calls than Select_Row, yet suffer accuracy drops of 14.08% and 40.06%, respectively. Notably, ReAcTable underperforms the Base model by 29.23% despite nine additional inferences per query. This degradation stems from aggressive multi-round pruning in H-STAR that discards critical data, and inconsistent symbolic results in ReAcTable's voting scheme. Thus, increased reasoning depth does not guarantee accuracy, *i.e.,* simpler, targeted queries often yield superior outcomes.

(4) Conventional RAG paradigms are unsuitable for TQA. Semantic RAG underperforms Base by 2.78%. By treating rows and columns as isolated text chunks, standard RAG disrupts the table's relational integrity and obscures cross-attribute dependencies essential for numerical and categorical reasoning. However, context-aware retrieval, incorporating query rewriting, hybrid dense–sparse search, and re-ranking, is a powerful asset. For instance, RAG+Rewrite outperforms the Base and simple RAG by 6.86% and 9.92% respectively, proving that retrieval can resolve complex queries if it preserves structural context (see Section VI for more).

(5) TQA benefits from minimal yet sufficient information. We evaluated two prototype methods, Route and Route + Check, designed to test adaptive operator selection. Route employs an LLM to select pruning operators and intersects their results to form a subtable. It surpasses Base by 9.76% while reducing context size by 73.3% with an average of 2.08 operators per query. However, Route + Check achieves the highest performance by manually pruning redundant operators from the LLM's selection. This method outperforms Base and Route by 21.31% and 10.52% respectively, while reducing inference time by 19.86%, and simplifying the operator set by 1.26 on average. The resulting average table reduction stabilizes at a more moderate 56.42%. These results indicate that (a) LLM-based router tends to over-prune by producing overly complex operator sets, which may lead to insufficient context for reasoning; and (b) identifying a smaller yet sufficient operator set is key to achieving better TQA performance.

Our findings.

These preliminary results highlight a fundamental dilemma in offline TQA. First, the reasoning-size conflict necessitates a delicate balance between pruning tables to circumvent the "reasoning cliff" and preserving sufficient data to prevent information loss. Second, the complexity-efficiency conflict emerges when symbolic tools, though required for complex queries, trigger "overthinking" and prohibitive latency during

TABLE I: Notation Table

Notation	Description
$o^i \in \mathcal{O}$	A pool of table operators, each is denoted as o^i.
$\mathcal{S} \in \boldsymbol{\mathcal{S}}$	A set of operators from $\mathcal{O}$.
(q, T)	Query(question)-Table pair for TQA task.
E, Q	Router model E and verify model Q.
LLM	A pre-trained offline Large Language Model(LLM).
Context($\mathcal{S}$)	LLM-generated result after conducting a operator set $\mathcal{S}$

multi-round orchestration on offline LLMs.

The success of Route + Check suggests that the solution lies in an adaptive, minimally sufficient context. However, manual checking is impractical. Therefore, an effective offline TQA system must (1) predict the optimal path using a cost-aware router to select the simplest operator set that stays within the stable reasoning regime (Section V-A). (2) execute verification to ensure information sufficiency without re-triggering expensive LLM generation (Section V-B).

IV. Framework Design

This section introduces SPARQ, an adaptive framework for offline TQA that integrates a dynamic query router with a mutual-information verifier. We first present an extended operator pool used in SPARQ (Section IV-A), followed by an overview of the overall framework (Section IV-B). The detailed design of its components will be described in Section V. Table I summarizes the notations in the paper.

A. An Extended Pool of Semantic and Symbolic Operators

Table operators. SPARQ utilizes a modular operator pool $\mathcal{O}$ to facilitate compositional table reasoning, integrating semantic and symbolic processing to avoid the brittleness of one-shot generation [23], which often fails on offline LLMs due to syntax or execution errors. Specifically, the preprocessing operator o^{prep} normalizes raw data and generates statistical summaries to create a compact, noise-reduced input for offline models. To pinpoint relevant information, the extraction operators (o^{col} and o^{row}) employ a dual-path strategy that combines LLM-based enumeration with SQL-based selection to identify key substructures while minimizing hallucinations. The retrieval operator o^{Ret} serves as a hybrid semantic engine for RAG-style evidence access; it first expands the query q into multi-granular variants (general, balanced, and specific) [68] to capture diverse linguistic nuances. It then computes similarity scores by fusing dense SBERT embeddings [69] with sparse BM25 features [70] through Weighted Reciprocal Rank Fusion [71], ultimately extracting the top-M rows and top-N columns as a contextually rich subtable. Finally, the SQL execution operator o^{SQL} handles symbolic computation and ensures robustness by filtering infeasible execution paths. This modular design allows SPARQ to bridge the gap between semantic retrieval and symbolic logic, granting offline models the efficient contextual access typically reserved for online systems. For more details, please refer to the Appendix A

Remark. Unlike conventional TQA pipelines that rely on fixed symbolic or neural reasoning paths, SPARQ adopts a modular operator design that enables flexible composition and self-correction. In particular, the newly introduced retrieval operator o^{Ret} bridges semantic retrieval with symbolic computation (Section III), allowing offline LLMs to access contextual evidence efficiently, an ability limited to online RAG systems.

B. Overview

We first define the notion of *minimal sufficiency* and then present the overall workflow of SPARQ.

Minimally sufficient. Given a query–table pair (q, T) and a pretrained LLM, a subtable $T^* \subset T$ is *minimally sufficient* if it (a) contains the information necessary to answer q correctly (sufficiency), and (b) minimizes the token length required by the LLM (minimality). However, identifying such a subtable is extremely challenging, as it requires enumerating all possible row–column combinations of T and evaluating each candidate with the LLM, leading to an exponential complexity of $O(2^{m+n} \cdot \text{COST}(\text{LLM}))$, where m and n denote the number of rows and columns in T, respectively, and COST(LLM) represents the cost of LLM inference on a single subtable. Ambiguous queries or overly long or incomplete tables usually incur a high COST(LLM) and often lead to error propagation.

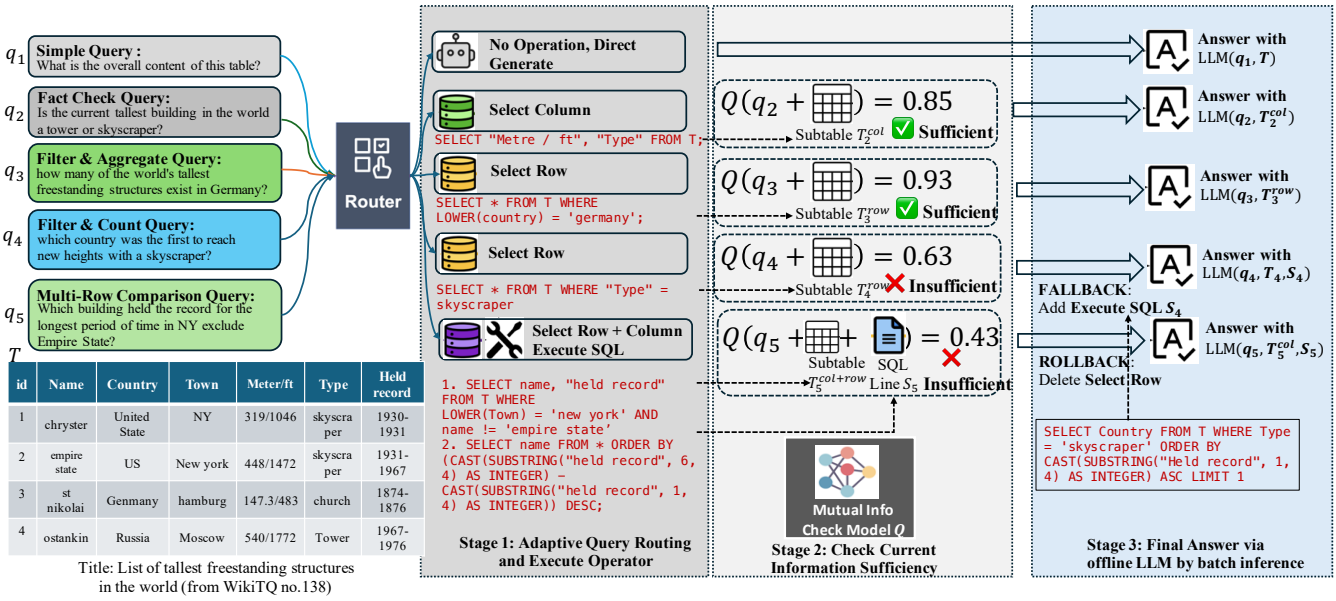
Framework of SPARQ. Given a query–table pair (q, T) and a pretrained LLM, SPARQ aims to retrieve a subtable from T that approximates the *minimally sufficient* one; that is, it contains just enough information for the LLM to accurately answer q while minimizing inference cost. In the sequel, we first introduce the core components of SPARQ and then describe its overall workflow.

Component. SPARQ comprises four cooperative components: (1) an operator pool $\mathcal{O}$ that provides a unified interface for both semantic and symbolic table manipulation (Section IV-A); (2) a query router E that dynamically selects an operator set $\mathcal{S} \subseteq \mathcal{O}$ most suitable for a given query–table pair (q, T), where each operator $o_i \in \mathcal{S}$ is assigned a fitness probability $o_i.p$ to guide the TQA process (Section V-A); (3) a mutual-information verifier Q that evaluates whether the resulting subtable retains sufficient information to answer q, triggering rollback when necessary (Section V-B); and (4) an execution optimizer that orchestrates CPU-based database operations and GPU-based inference in parallel to reduce latency and resource consumption (Section V-C). Together, these components enable SPARQ to approximate a minimally sufficient subtable for accurate TQA without reliance on online LLMs.

Workflow. As shown in Fig. 4, SPARQ takes as input a query–table pair (q, T) and a predefined operator pool $\mathcal{O}$. It operates in two phases: an *offline phase* and an *online phase*.

In the offline phase, SPARQ selects a small set of hard samples with ground-truth labels (*e.g.,* a subset of the WikiTQ validation set) and collects LLM predictions by executing various operator sets $\mathcal{S} \subset \mathcal{O}$ in a set $\boldsymbol{\mathcal{S}}$, thereby generating positive and negative supervision signals following a self-distillation paradigm (line 1). It then trains the lightweight router E and verifier Q on this data (line 2).

In the online phase, SPARQ first preprocesses T with o^{prep}



id	Name	Country	Town	Meter/ft	Type	Held record
1	chryster	United State	NY	319/1046	skyscraper	1930-1931
2	empire state	US	New york	448/1472	skyscraper	1931-1967
3	st nikolai	Genmany	hamburg	147.3/483	church	1874-1876
4	ostankin	Russia	Moscow	540/1772	Tower	1967-1976

Title: List of tallest freestanding structures in the world (from WikiTQ no.138)

Fig. 3: Overview of SPARQ. Given a query–table pair (q, T), the router E selects a cost-efficient operator set $\mathcal{O}$. Operators are executed with parallel batching on CPU/GPU. After each step the verifier Q checks information sufficiency; failures trigger ROLLBACK, and persistent insufficiency triggers a FALLBACK before final QA.

Input: A query-table pair (q, T), a pretrained offline LLM, and an operator pool $\mathcal{O}$.
Output: The answer a of query q on T.

/* Offline */
1. T^{train} := On a small set of hard samples, collect LLM's predictions from a set $\boldsymbol{\mathcal{S}}$ of operator sets $\mathcal{S} \subset \mathcal{O}$;
2. E, Q := Train small router and verifier models on T^{train};

/* Online */
3. T := Preprocessing T with table operator o^{prep};
4. $\mathcal{S} := E(q, T)$; // Router E selects the suitable $\mathcal{S}$ in $\boldsymbol{\mathcal{S}}$.
5. $(T^{\mathcal{S}}, \text{Context}(\mathcal{S})) := Q(\mathcal{S})$; // Verifier Q produces a subtable $T^{\mathcal{S}}$ with optional execution result $\text{Context}(\mathcal{S})$.
6. a := querying LLM with $(q, T^{\mathcal{S}}, \text{Context}(\mathcal{S}))$.

Fig. 4: Workflow of SPARQ .

for data cleaning and normalization (line 3). The router E selects an operator set $\mathcal{S} \in \boldsymbol{\mathcal{S}}$ for (q, T), and the LLM executes it to produce intermediate results $\text{Context}(\mathcal{S})$ (*e.g.,* SQL commands from o^{SQL} or rewritten queries from o^{Ret}) (line 4). The verifier Q then assesses $\mathcal{S}$ with the generated context, finalizes execution, and outputs a refined subtable $T^{\mathcal{S}}$ (line 5). Finally, SPARQ queries the LLM with $(q, T^{\mathcal{S}}, \text{Context}(\mathcal{S}))$ to produce the final answer a (line 6).

Remark. (1) Unlike conventional TQA frameworks with fixed pipelines, SPARQ is specifically engineered to empower offline LLMs, bridging the performance gap through a modular and adaptive architecture. By coupling the router E with the verifier Q, SPARQ dynamically approximates the minimally sufficient subtable, ensuring that smaller local models can reason effectively over structured data. (2) While virtual private cloud (VPC) solutions [72], [73] offer logical data isolation, they remain vulnerable under an *honest-but-curious* threat model [74], [75], and impose operational costs prohibitive for small-to-medium enterprises (SMEs) and individual users. SPARQ bypasses these barriers via hardware-level optimization, ensuring competitive TQA performance with full data control and zero API overhead, making it a practical solution for decentralized TQA.

V. Model and Algorithm

This section details the core components of SPARQ, including (a) an adaptive query router that dynamically selects a cost-efficient operator set approximating the minimally sufficient subtable (Section V-A); (b) a mutual-information verifier that ensures information sufficiency through rollback control (Section V-B); and (c) system-level optimizations that enable efficient execution on commodity hardware (Section V-C).

A. Adaptive Query Router

Challenge. For a given query–table pair (q, T) and an offline LLM, multiple operator sets may produce the same correct answer. For instance, both a single operator o^{row} and a composite set $\{o^{\text{row}}, o^{\text{SQL}}\}$ can succeed. However, different operators incur varying computational costs: o^{SQL} requires three LLM generations, o^{row} and o^{col} involve two calls, and o^{Ret} performs about $3(m+n)$ similarity computations for a table with m rows and n columns plus one LLM generation for query rewrite (*e.g.,* as general, balanced or specific forms). As the operator set expands, computational cost typically increases, creating an inherent trade-off between reasoning depth and efficiency.

Mechanism. We formally define the query time cost, denoted as $\text{COST}(q, T, \mathcal{S})$, as the total runtime required to generate the final answer using an operator set $\mathcal{S}$. This cost is jointly determined by the logical intricacy of the query q (e.g., multi-step reasoning overhead) and the structural scale of the table T (e.g., token consumption and context processing latency). While multiple operator sets may yield the correct

answer, potentially involving redundant operations, there exists a theoretical optimal cost corresponding to the minimal set of operators required for correctness. However, this optimal cost is unknown in advance as it depends on the dynamic interaction between the query logic and table content.

Example 2: Consider the query in Fig. 1: *"How many games were played after October 1st?"*. The logical intricacy is low (simple count), but the structural scale is high (large table). **(1) Naive Path** (o^{Base}) incurs minimal cost but fails to ingest the full context, leading to hallucination. **(2) High-Cost Path** ($o^{\mathsf{SQL}}, o^{\mathsf{row}}$) leads to a wrong answer due to format error and dirty data. **(3) Optimal Path** (o^{col}) reduces structural scale by filtering irrelevant columns while retaining sufficient information with medium cost. SPARQ approximates this optimum by predicting $\{o^{\mathsf{col}}\}$ and verifying its sufficiency.

Consequently, SPARQ adopts an approximate-and-verify co-design paradigm. The Router E predicts an operator set $\mathcal{S}$ to approximate this theoretical minimal cost, while the Verifier Q (see below) ensures the sufficiency of the selected path to prevent under-approximation, balancing reasoning accuracy with computational efficiency. For each operator $o_i \in \mathcal{S}$, we derive a fitness probability $o_i.p$ from E to quantify its contribution utility. Specifically, we calculate $o_i.p$ as the marginal probability of o_i based on the router's predicted distribution over all candidate sets: $o_i.p = \sum_{\mathcal{S}_k : o_i \in \mathcal{S}_k} P(\mathcal{S}_k \mid q, T)$. This metric indicates how likely o_i is to effectively assist the LLM in answering q over T, aggregating confidence from all potential reasoning paths. Once $\mathcal{S}$ is determined, the verifier Q evaluates its information sufficiency and triggers rollback when necessary . Naturally, we have the optimization goal as:

$$\min_{\mathcal{S} \subseteq \mathcal{O}} \ \text{COST}(\mathcal{S}) \quad \text{s.t.} \quad Q(q, T^{\mathcal{S}}, \mathsf{Context}(\mathcal{S})) \geq \tau \qquad (1)$$

where $Q(\cdot)$ is the mutual-information-based sufficiency score (Eq. 3), and τ is predefined threshold (detailed in Section V-B).

Example 3: As illustrated in Fig. 3, given a table T, the router E dispatches queries of varying cost (q_1–q_5) to different operator sets $\mathcal{S}$ based on a cost–benefit analysis:

- $q_1 \rightarrow \{o^{\mathsf{Base}}\}$: q_1 only needs a general table description, which can be answered directly without transformation.
- $q_2 \rightarrow \{o^{\mathsf{col}}\}$: q_2 focuses on specific attributes (*e.g.,* "Meter/ft", "Type"), requiring column extraction before inference.
- $q_3 \rightarrow \{o^{\mathsf{row}}\}$: q_3 involves filtering rows by "country", followed by aggregation by the LLM.
- $q_4 \rightarrow \{o^{\mathsf{row}}\}$: q_4 requires filtering/ranking rows to identify the first country, with temporal or ranking reasoning via LLM.
- $q_5 \rightarrow \{o^{\mathsf{col}}, o^{\mathsf{row}}, o^{\mathsf{SQL}}\}$: q_5 demands multi-step reasoning: filtering by location, computing date differences, and comparing durations; accordingly, column extraction, row extraction, and SQL-based arithmetic are applied simultaneously.

Router design. We introduce two designs for the router E.

Training-free router. The training-free router utilizes a pre-trained LLM's intrinsic reasoning to select optimal operators from $\mathcal{O}$. Given (q, T), SPARQ employs in-context prompting to predict the most suitable operator set. This approach ensures high flexibility and zero training overhead, making it ideal for resource-constrained environments.

Training-based router. We introduce a lightweight training-based router as a complementary option when the training-free approach lacks sufficient accuracy. Given the operator pool $\mathcal{O}$, we define a set of operator sets $\boldsymbol{\mathcal{S}} = \{\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_k\}$, where each $\mathcal{S} \in \boldsymbol{\mathcal{S}}$ represents a unique combination of operators from $\mathcal{O}$, sorted by increasing cost, *i.e.,* $\mathcal{S}_0$ denotes the base case (no operator from $\mathcal{O}$ is actually applied), and $\mathcal{S}_k$ corresponds to the most costly configuration.

To construct the training data, We first sample a small set of hard validation tables T^{train} with long contexts ($>$ 4K tokens). Given the operator pool $\mathcal{O}$, we enumerate a candidate operator-set space $\boldsymbol{\mathcal{S}}$, ordered by query time cost. Given a set of queries q, SPARQ queries the offline LLM with triplets $(q, \mathsf{Context}(\mathcal{S}), T^{\mathcal{S}})$ to obtain predictions p. Each triplet is assigned a reward score $\mathcal{R}(q, \mathcal{S}, T^{\mathcal{S}})$, defined as the product of three components: a prediction evaluation score, an operator-cost penalty, and a context-length penalty:

$$\mathcal{R}(q, \mathcal{S}, T^{\mathcal{S}}) = \mathsf{acc}(p, l) \times \rho_{\text{cost}}(\mathcal{S}) \times \rho_{\text{len}}(\mathcal{S}), \qquad (2)$$

where $\mathsf{acc}(p, l)$ measures the answer accuracy between prediction p and ground-truth label l, $\rho_{\text{cost}}(\mathcal{S}) = \exp(-\lambda_c \cdot \text{COST}(\mathcal{S}))$ penalizes computationally expensive operator sets, and $\rho_{\text{len}}(\mathcal{S})$ penalizes overly long LLM contexts.

Using this training data, the router E is trained as a multi-class classifier to predict the optimal operator-set distribution $P(\mathcal{S} \mid q, T)$. We formulate this as a knowledge distillation task, where the pre-computed reward $\mathcal{R}$ serves as soft labels. In addition, we use the InfoNCE [76] loss to contrast high-reward and low-reward operator sets, guiding E toward selecting cost-efficient operator paths for queries of varying cost.

Remark. An operator set with too few operators may produce an overly large context for the LLM, while an excessively long set risks information loss. Both situations can degrade answer accuracy. Verifier Q is proposed to address this (see below).

B. Mutual-Information Verifier Q

Motivation. As discussed in Section V-A, (a) an over-aggressive operator set $\mathcal{S}$ may cause information loss by filtering out critical evidence; and (b) each operator $o_i \in \mathcal{S}$ is associated with a fitness probability $o_i.p$ that o_i can effectively assist the LLM in answering a query q on T. To avoid an operator set causes information loss due to excessive execution, SPARQ introduces a mutual-information verifier Q as a real-time information sufficiency checker.

Inspired by the retrieve–rerank paradigm in RAG systems, Q assesses whether the current subtable $T^{\mathcal{S}}$, produced by executing the operator set $\mathcal{S}$ on T, contains sufficient information to answer the query q, while additionally considering the intermediate execution result $\mathsf{Context}(\mathcal{S})$ generated by the LLM, which serves as an auxiliary reranker.

Mechanism. Given a query–table pair (q, T) and a subtable

$T^{\mathcal{S}}$ with $\mathsf{Context}(\mathcal{S})$ generated by applying operator set $\mathcal{S}$ on T, Q estimates a sufficiency score $Q(q, T^{\mathcal{S}}, \mathsf{Context}(\mathcal{S})) \in [0, 1]$ that quantifies the mutual information (MI) between the query and the selected evidence (see below). Here a higher Q indicates that the subtable preserves the essential information required for reasoning, while a lower score suggests potential information loss.

Rollback strategy. If the score falls below a predefined threshold (τ), SPARQ initiates a rollback by reverting to a larger sub-table $T^{\mathcal{S}'}$ where $\mathcal{S}' = \mathcal{S} \setminus \{o_{\min}\}$, excluding the extraction operator with the lowest fitness probability $o_i.p$. The verifier Q then re-evaluates $T^{\mathcal{S}'}$ iteratively until a sufficient context is found or the base case ($\mathcal{S}_0$) is reached, *i.e.,* no operator from $\mathcal{O}$ is applied. Such design aims to avoid potential error accumulation via multi-step generation, without invoking LLM re-generation. Only table extraction operator *e.g.,* $\{o^{\mathsf{row}}, o^{\mathsf{col}}, o^{\mathsf{ret}}\}$ will trigger rollback.

Fallback strategy. If the base case still fails to provide sufficient information to answer the query q, SPARQ invokes a fallback mechanism that prompts the LLM to generate an SQL statement over the full table T to retrieve additional evidence. The retrieved results are then incorporated into the context for a final LLM inference to produce the answer. We treat this *Fallback* as a safeguard, since SQL parsing failures in fallback cases are typically more complex than in other scenarios.

Example 4: Continue with Example 3. We consider two failure cases, q_4 and q_5, where the router dispatches them wrongly.

Rollback. For q_5, o^{row} erroneously filters the target row (e.g., misinterpreting NY), resulting in $T_5^{\mathcal{S}} = \emptyset$. The rollback mechanism restores missing evidence by reverting the operator set $\mathcal{S}$ to $\{o^{\mathsf{col}}, o^{\mathsf{SQL}}\}$, specifically removing o^{row} due to its lowest fitness probability p.

Fallback. For q_4, neither o^{row} nor o^{Base} provides sufficient information. Therefore, SPARQ triggers the fallback strategy, prompting the LLM to generate an SQL query (shown in the bottom-right corner of Fig. 3) to retrieve supplementary evidence. The resulting subtable $S_4 = o^{\mathsf{SQL}}(q, T)$, combined with q and T, is then used to produce the final answer.

Model training. We first outline the challenges of training the verifier Q and then describe its training procedure.

Challenges. Training the verifier Q is challenging due to the heterogeneity of TableQA's three modalities: (a) the unstructured query q, (b) the semi-structured subtable $T^{\mathcal{S}}$ containing mixed data types, and (c) the structured execution result $\mathsf{Context}(\mathcal{S})$ encoding logical evidence. Discrepancies in representation and granularity hinder consistent assessment of whether $T^{\mathcal{S}}$ sufficiently covers q. Consequently, a unified representation space is necessary to align these semantics and quantify information sufficiency.

To tackle above challenge, we define Q as a learnable function that holistically evaluates the alignment between q, $T^{\mathcal{S}}$, and $\mathsf{Context}(\mathcal{S})$. It is calculated as a weighted aggregation of the semantic alignment scores across all three modality pairs, thereby assessing both the relevance of the evidence to the query and the internal consistency of the evidence itself:

$$\begin{aligned} Q(q, T^{\mathcal{S}}, \mathsf{Context}(\mathcal{S})) = \sigma(\alpha \cdot f_\theta(q, T^{\mathcal{S}}) + \beta \cdot f_\theta(q, \mathsf{Context}(\mathcal{S})) \\ + \gamma \cdot f_\theta(T^{\mathcal{S}}, \mathsf{Context}(\mathcal{S}))), \end{aligned} \tag{3}$$

where the function f_θ produces an alignment score refactored to $[-1, 1]$, and the non-negative coefficients α, β, γ are balancing weights defined in the training objective (see Equation 4).

Training via cross-modal contrastive alignment. Verifier Q shares the router's training set T^{train}. Positive samples are high-reward operator sets $\mathcal{S}$ yielding sufficient subtables $T^{\mathcal{S}}$. Negative samples are synthesized via hard sampling and augmentation: (i) mismatched query–table pairs, (ii) cross-query context substitution, and (iii) random subtable corruption (*e.g.,*, dropping rows or columns). Q is trained using a cross-modal contrastive learning framework that maximizes the mutual information between semantically aligned modality pairs: $(q, T^{\mathcal{S}})$, (q, S), and $(T^{\mathcal{S}}, S)$. Each pair is encoded by a Transformer-based CrossEncoder f_θ, which takes two linearized inputs, $L_T(\cdot)$ for tables and $L_S(\cdot)$ for SQL, and outputs a scalar similarity score $s = f_\theta(\text{text}_1, \text{text}_2)$. We optimize f_θ using a listwise ranking loss that encourages positive pairs to score higher than negatives, effectively maximizing their mutual information [76]. For a generic modality pair (X, Y), the InfoNCE is defined as:

$$\mathcal{L}_{XY} = -\log \frac{\exp(s^{XY+})}{\exp(s^{XY+}) + \sum_{j=1}^{k} \exp(s_j^{XY-})}, \tag{4}$$

where s^{XY+} denotes the score of a positive pair and s_j^{XY-} represents the scores of k negative pairs. The total objective aggregates all modality pairs:

$$\mathcal{L}_{\text{total}}(\theta) = \alpha \cdot \mathcal{L}_{qT} + \beta \cdot \mathcal{L}_{qS} + \gamma \cdot \mathcal{L}_{TS}, \tag{5}$$

where α, β, and γ are balancing coefficients controlling the contributions of the query–table, query–SQL, and table–SQL alignment losses, respectively. By default, we set $\alpha = \beta = 1$ and $\gamma = 0.2$, due to the inherent weak ability of f_θ in understanding structural and symbolic context. The training set for Q is T^{train}, augmented with hard negative samples within each batch, *e.g.,* alternative queries, SQL statements, or subtables extracted from the same table T, to enhance discrimination without relying on router E.

Remark. (1) Unlike conventional discriminators that rely on heuristic thresholds or binary feedback [77], Q measures sufficiency in a continuous and differentiable manner, enabling adaptive self-correction during reasoning. (2) We train the verifier Q by aligning three heterogeneous modalities, rather than focusing on pairwise alignments such as query-to-table as in [78], query-to-SQL as in [12], [79] or table-to-SQL/Code in [20].

C. Optimization Strategies

To mitigate GPU underutilization caused by "CPU-GPU bubbles" in sequential execution, we propose a decoupled and parallelized inference pipeline. As illustrated in Fig. 8 (now located in Appendix D), our design synchronizes CPU-intensive tabular preprocessing with GPU-accelerated LLM

inference through a producer-consumer architecture. Key optimizations include multi-threaded asynchronous SQL execution and a resource-aware dynamic batching strategy that maximizes KV cache reuse and memory efficiency. By interleaving these heterogeneous tasks, the system substantially enhances throughput while maintaining robustness against malformed queries. We refer the reader to Appendix D for a comprehensive technical description of the scheduling algorithms and implementation details. Notably, our operator set is mutually independent, which can execute in parallel, with results combined via intersection. However, the current scheduler does not leverage this property, which we leave for future work.

VI. Experiment

In this section, we aim to answer the following questions:

- **Q1:** *Effectiveness v.s. Online Methods*. Can SPARQ achieve competitive or even superior accuracy *w.r.t.* online TQA systems while being fully deployed offline to ensure data privacy?
- **Q2:** *Effectiveness v.s. Offline Methods*. Can SPARQ consistently outperform other methods when all models are deployed in the same offline environment, particularly across LLMs of different parameter scales?
- **Q3:** *Efficiency*. How does the scheduling system of SPARQ utilize hardware resources to minimize end-to-end latency and computational overhead during offline inference?
- **Q4:** *Adaptive*. Can SPARQ dynamically adjust its operator selection and retrieval strategy to handle queries and tables of varying cost while maintaining information sufficiency?
- **Q5:** *Benefits*. How do SPARQ's components (*i.e.,* adaptive router, verification model, and symbolic–semantic operators) contribute to its overall performance and robustness?

A. Experimental Settings

Datasets. We evaluated SPARQ on five diverse benchmarks, ranging from standard academic datasets to challenging real-world enterprise environments. First, we used three widely adopted Wikipedia-based benchmarks: (a) *WikiTQ* [5] contains complex questions over Wikipedia tables that require compositional reasoning to produce short and factual answers. (b) *FeTaQA* [3] is a dataset with questions that demand reasoning over tables to generate long-form and descriptive answers. (c) *TabFact* [15] is a fact-checking benchmark where a textual statement must be verified as either *True* or *False* based on a given table. To assess generalization beyond well-formatted data, we incorporated two emerging enterprise-centric benchmarks: (d) *TableBench* [80], featuring 886 complex query–table pairs from proprietary financial reports with highly heterogeneous schemas, and (e) *NeedleInATable* (NIAT) [81], designed to emulate "dirty" enterprise spreadsheets with nested headers, merged cells, and extremely long contexts ($>$ 100k tokens) to rigorously test robustness against structural noise. We fine-tune E and Q with 487 (resp. 476, 376, 92, 199) labeled hard samples from validation set on WikiTQ (resp. TabFact, FeTaQA, TableBench, NIAT).

Baselines. We compared the performance of our SPARQ approach with various strong baselines, including both training-based and training-free paradigms. Note that, considering the nature of offline TQA, we only consider methods with publicly released code and evaluation artifacts.

Training-Based Methods. For WikiTQ and TabFact, we consider LEVER [53], TaPas [13] and TAPEX [14] as Pre-Trained model (PLM)-based baselines. For the LLM-based training method, we adopt Table-GPT (7B) [8] and TableGPT2 (70B) [46]. Following [23], we further evaluate PLM-based T5 [82] on the open-form FeTaQA benchmark.

LLM-Based Training-Free Method. We considered (a) program-aided reasoning methods: DATER [20], TabSQLify [23] and ReAcTable [7]; and (b) hybrid symbolic and semantic approaches: H-STAR [24] and Chain-of-Table [22]. By default, we report DATER and ReAcTable under OpenAI Codex [26], TabSQLify and HSTAR under gpt-3.5-turbo [25], and Chain-of-Tables under PaLM2 [27]. For H-STAR, we additionally report their performance under GPT-4o-mini and llama-3.1-70B [28], denoted as $\mathsf{HSTAR}_{\mathsf{gpt4}}$ and $\mathsf{HSTAR}_{\mathsf{70B}}$, respectively. Since the above methods are built upon different PLMs and LLMs, and some of them are not publicly available (*e.g.,* PaLM-2 [27]), we reported and compared with the best performance in their papers in Table II and XV.

Environment. All experiments are conducted on 2 RTX 4090 GPUs powered by 64GB RAM and 32 processors with Intel(R) Xeon(R) Platinum 8432C @2.00GHz. For large-scale baselines (*i.e.,* $\mathsf{TABLEGPT2}_{\mathsf{70B}}$ and $\mathsf{HSTAR}_{\mathsf{70B}}$), we additionally use two A800 GPUs with 80G VRAM. To deploy the QA-LLM, we use vLLM [38] as the backend.

LLM Models and Configurations. For offline LLM backbones of our SPARQ approach, we employ the dense model Qwen3-4B [29]($\mathsf{SPARQ}_{\mathsf{4B}}$) and the sparse MoE model Qwen3-30B [30] ($\mathsf{SPARQ}_{\mathsf{30B}}$) with FP8 quantization, respectively. $\mathsf{SPARQ}_{\mathsf{30B}}$ uses the MoE model that activates only 3B parameters during inference, which holds computational cost and resource usage comparable to $\mathsf{SPARQ}_{\mathsf{4B}}$. Moreover, the above two models are also used as training-free routers, denoted as E_{4B} and E_{30B}, respectively. Besides, we use online LLM Qwen3-Max [83] to evaluate our generalizability for cloud-hosted LLMs. For the default training-based router E, we adopt BGE-m3 [67] as the base model, and for the verification module Q, we use BGE-Reranker-v2-m3 [84]. For hyper-parameter setting, we set the sampling number of $o^{\mathsf{col}}, o^{\mathsf{row}}$ as 2, and o^{SQL} as 3. The acceptance threshold for τ of Q is set to 0.8. The maximum length of selected $\mathcal{S}$ is set to 3 to avoid aggressive pruning. For constructing T^{train}, we set operator pool size as $|\mathcal{O}| = 4$ excluding o^{Base}, and all possible solutions number $k = |\mathcal{S}| = 15$.

Evaluation Metrics. Following prior work [5], [7], [21], [24], [80], [81], we use *denotation accuracy* [5] for WikiTQ, *Exact Match*(EM) for NIAT, *binary classification accuracy* for TabFact, and *ROUGE-L* [85] for FeTaQA and TableBench to

TABLE II: Results on WikiTQ and TabFact. The best result is marked in **bold**, second best in underline. Entries marked with / were not reported in the paper and are irreproducible with unavailable artifacts.

Method	WikiTQ	TabFact
Approaches requiring training		
TaPas (ACL′20)	48.8	83.9
TAPEX (ICLR′22)	57.5	86.7
LEVER (ICML′23)	62.9	/
Table-GPT (SIGMOD′24)	52.8	/
TABLEGPT2$_{7B}$	61.4	77.8
TABLEGPT2$_{70B}$	71.4	85.4
Approaches without training		
DATER(SIGIR′23)	65.9	85.6
TabSQLify (NAACL′24)	64.7	79.5
ReACTable (VLDB′24)	68.0	86.1
Chain-Of-Tables (ICLR′24)	67.3	86.6
HSTAR$_{gpt3.5}$ (NAACL′25)	69.56	86.5
HSTAR$_{gpt4}$	74.93	89.42
HSTAR$_{70B}$	75.76	89.23
SPARQ$_{4B}$	77.03	91.30
SPARQ$_{30B}$	**79.79**	**92.19**

evaluate long-form answer quality. Each experiment was run 3 times, and the average is reported.

B. Effectiveness Analysis (Q1&Q2)

SPARQ **Outperforms Online Methods (Q1)**. To answer Q1, we compare our SPARQ approach with various training-based and training-free online-LLM based solutions in Tables II and XV, except TABLEGPT2$_{7B}$, TABLEGPT2$_{70B}$ and HSTAR$_{70B}$ that are offline deployed. For fair comparison, results for baselines utilizing online LLMs are directly cited from their respective original papers. From these tables, we can find that SPARQ consistently outperforms all baselines with online models. SPARQ consistently outperforms these online models; for instance, on WikiTQ (Table II), SPARQ$_{4B}$ surpasses coding-specialized baselines like DATER and ReAcTable by 16.89% and 13.28%, respectively. Furthermore, SPARQ$_{4B}$ leads H-STAR (powered by GPT-3.5 and GPT-4o-mini) by 10.73% and 2.8%. Notably, SPARQ achieves these gains using fewer than 5% of the parameters required by online LLMs, validating its superior performance and privacy-preserving, low-cost offline deployment.

We additionally evaluate FetaQA dataset in Table XV, and TableBench with SPARQ, ReAcTable, and H-STAR with Qwen3-Max in Table XVI in Appendix. Although gains are smaller (as these models are already strong), SPARQ still improves efficiency and robustness. For example, on TableBench, SPARQ achieves 0.55 Rouge-L, outperforming ReAcTable (0.40), whereas with offline LLM Qwen3-4B, SPARQ (0.491) substantially surpasses ReAcTable (0.051). However, token consumption for all online LLM methods remains substantial ($\geq 3.8M$ tokens), and their latency is considerably higher than offline counterparts (at least $2.02\times$), detailed in Appendix A.

SPARQ **Outperforms Offline Methods (Q2)**. To answer Q2, we offline deploy some baseline methods, including large offline LLMs (*i.e.,* TABLEGPT2$_{7B}$, TABLEGPT2$_{70B}$, and HSTAR$_{70B}$ in Table XV) and small models (*i.e.,* all baselines compared in Table III).

Offline Large Models. Offline TQA often lacks the massive data and compute required for full-model training. SPARQ bypasses this by fine-tuning only its lightweight router E and verifier Q on select hard negatives. Despite its smaller scale, SPARQ$_{4B}$ beats TABLEGPT2$_{70B}$ by 7.88% and even exceeds the 70B H-STAR by 1.67% while requiring just 5.7% of the parameters. This confirms that SPARQ's "minimal-but-sufficient" context approach is a more efficient path to high-performance TQA than mere parameter scaling.

Offline Small Models. Table III presents a unified comparison where all baseline methods and SPARQ are benchmarked on the same offline models, served via a vLLM-based, OpenAI-compatible server. We observe that baseline methods suffer a significant performance degradation when migrated to offline models, particularly those with fewer parameters (*e.g.,* 4B). As noted in recent literature, smaller models are highly susceptible to hallucinations, formatting errors, and repetitive outputs when confronted with complex tasks like SQL generation or insufficient context [86]. On the Qwen3-4B model, for example, SPARQ outperforms all baselines by over 14% on both TabFact and WikiTQ. Unlike baselines that suffer from error accumulation across overthought reasoning chains (Section III), SPARQ utilizes a targeted route-check mechanism. By detecting and rolling back erroneous operations, it maintains a sufficient and stable context for the LLM, effectively bypassing the reliability issues of long-form reasoning in small models. As a direct comparison, H-STAR's multi-step filtering process results in a SQL execution success rate of only 20.22%, with the accuracy drastically dropping to 8.98% due to information loss. While SPARQ$_{4B}$ maintains a 77.79% SQL execution success rate and exhibits no significant performance degradation compared to SPARQ$_{30B}$.

SPARQ **Demonstrates Stability Across Various LLMs (Q2)**. SPARQ employs a model-ability-aware scheduling strategy to maintain stable performance and latency across model scales (Table III). For smaller models, SPARQ applies test-time scaling [87] by increasing sampling and tightening verification thresholds, effectively trading inference compute for accuracy. Conversely, for capable large models like Qwen3-30B, it reduces sampling and relaxes verification to bypass intermediate steps and minimize latency. While baselines fluctuate 10%–40% across scales, SPARQ's performance remains consistently high, validating its dynamic balance between effectiveness and efficiency.

C. Efficiency Analysis (Q3)

To answer Q3, in Tables III and IV, we compare the efficiency of our SPARQ with various baseline methods, where all methods are evaluated within an identical offline environment and utilizing the same underlying models. The runtime of most training-free, LLM-based methods can be divided into two primary components: *prediction time*, which encompasses all queries to the LLM (e.g., for CoT generation, SQL generation, and final answer synthesis), and *execution time*, which includes

TABLE III: Comparison of accuracy and end-to-end latency on TabFact, WikiTQ, NIAT and TableBench datasets using the same offline backend LLM with different training-free baseline methods.All offline model is deployed with 2 RTX 4090 GPUs. Latency means the average latency per query in seconds measured using 16 threads for all methods if supported.

Method	Qwen3-4B								Qwen3-30B							
	TabFact		WikiTQ		NIAT		TableBench		TabFact		WikiTQ		NIAT		TableBench	
	Acc	Latency	Acc	Latency	EM	Latency	Rouge-L	Latency	Acc	Latency	Acc	Latency	EM	Latency	Rouge-L	Latency
ReAcTable	25.84	0.40	0.58	0.59	0.95	0.93	0.05	1.57	66.90	0.56	33.47	0.63	26.08	**0.56**	0.39	1.90
TabSQLify	6.23	3.14	63.58	5.08	55.67	4.58	0.24	4.94	45.06	2.96	72.26	4.46	67.98	3.68	0.31	3.85
H-STAR	60.33	1.66	8.98	2.06	13.99	1.49	0.02	2.63	88.24	2.26	70.33	2.48	56.37	2.16	0.26	3.74
Chain-of-Tables	17.74	1.20	49.15	2.02	63.02	1.29	0.30	1.60	83.75	2.12	58.31	1.96	69.96	1.22	0.37	1.97
SPARQ (Ours)	**91.30**	**0.31**	**77.03**	**0.49**	**66.58**	**0.72**	**0.49**	**1.07**	**92.19**	**0.33**	**79.79**	**0.55**	**73.45**	0.95	**0.52**	**1.32**

TABLE IV: Runtime and throughput analysis with single thread on WikiTQ dataset using the same offline model Qwen3-4B on 2 RTX 4090 GPUs. Time is in second(s), throughput is in tokens/s.

Method	Avg. prediction time	Avg. Execution time	Avg. I/O token throughput	Avg. End-to-End Latency
TabSQLify	1.96	1.49	2870 / 125	3.45
ReAcTable	3.52	0.07	**11601** / 114	3.59
H-STAR	6.53	8.66	1709 / 179	15.19
SPARQ_{4B}	**0.7612**	**0.0860**	5260 / **1184**	**0.8472**

TABLE V: **Left:** Accuracy on WikiTQ across table sizes. **Right:** *Reasoning cliff* on NIAT: performance drops as token length increases.

Method	Small	Med.	Large
DATER	62.5	42.3	34.6
CoT	68.1	52.3	44.9
TabSQL	68.2	57.9	52.3
H-STAR	71.6	65.2	64.8
SPARQ_{4B}	79.0	74.8	70.3
SPARQ_{30B}	**82.3**	**77.9**	**74.7**

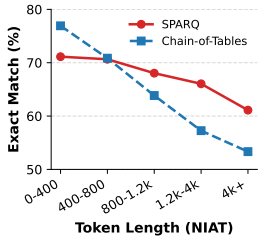


TABLE VI: Ablation study of the accuracy performance for SPARQ_{4B} on TabFact and WikiTQ datasets.

Method	TabFact	WikiTQ
SPARQ_{4B}	**91.30**	**77.04**
w. Training-Free Router E_{4B}	86.86	75.53
w/o Verification Model Q	87.20	74.06
w/o Table Extraction	87.10	74.38
w/o SQL Operator	87.70	74.24
w/o Retrieval Operator	87.55	76.26

the execution of external tools such as SQL, Python code, or retrieval operators. Table IV provides a detailed breakdown of these metrics, where "avg. I/O token throughput" serves as a measure of the LLM's responsiveness. To minimize end-to-end latency, for Table III we set multi-thread number to 16 for all methods except TabSQLify which only supports single-thread, while for Table IV, we set thread to 1 for all methods, towards a clearer breakdown of prediction/execution steps. Our findings are reported below.

SPARQ **Achieves Superior Latency via Coordinated CPU-GPU Scheduling**. For the experiments in Table III, we compare SPARQ with the baselines using our batched pipeline. SPARQ achieves an average speedup of $1.49\times$ (resp. $1.37\times$) over the second-fastest baseline ReAcTable on TabFact (resp. WikiTQ), while consistently outperforming ReAcTable in accuracy. In the experiments reported in Table IV, by decoupling and interleaving these heterogeneous CPU-bound and GPU-bound tasks as detailed in Fig. 8, SPARQ achieves an average speedup of over $10.19\times$ in end-to-end latency compared to baseline methods, with improvements reaching up to $17.93\times$ in high-contention scenarios. For LLM inference, SPARQ attains a $5.26\times$ average speedup in prediction time and $8.82\times$ higher output tokens throughput than the baselines, demonstrating the benefits of batch inference.

D. Retrieval Quality Analysis (Q4)

To answer Q4, in Table V and Figure 9, we provide a quantitative and qualitative analysis of SPARQ's retrieval quality under different complexity configurations, where the evaluation metrics follow existing works [7], [23], [24].

SPARQ **Achieves Robust Performance Across All Sample Complexities via its Adaptive Strategy**. Table V (Left) details SPARQ's performance on WikiTQ stratified by table size (*Small*: <2k tokens, *Medium*: 2k–4k, *Large*: >4k). SPARQ's adaptive routing effectively allocates operators based on structural scale: preserving context for *Small* tables while deploying extraction operators (*e.g.,*Execute_SQL) for larger ones. Consequently, SPARQ achieves performance gains of over 10% across all groups. Crucially, as visualized in Table V (Right), we observe the "reasoning cliff" phenomenon [45] on the NIAT benchmark: while baseline methods (*e.g.,* Chain-of-Tables) suffer a steep accuracy drop ($> 20\%$) as input length exceeds 4k tokens, SPARQ_{4B} maintains robustness in the 4000+ regime. This empirical evidence refutes the notion that 4k tokens is a trivial bound for offline models, confirming the necessity of structural simplification.

E. Ablation Study (Q5)

To answer Q5, Fig. 5 and Table VI show the ablation study of all SPARQ's components and an analysis of the operator set $\mathcal{S}$, respectively, demonstrating the effectiveness of each component within the SPARQ framework. Due to space constraints, we provide a detailed analysis of table extraction and SQL/Python execution operators in Appendix I–J.

A Training-Free Router Is Not a Plug-and-Play Solution. As discussed in existing work [65], we find that a training-free router cannot accurately assess the quality and success rate of operators for offline LLMs. Its routing decisions are often overly aggressive (Fig. 5), favoring excessively long operator sequences. This leads to a significant performance degradation on both the WikiTQ and TabFact datasets (Table VI). Simi-

TABLE VII: Cross-domain generalization. Diagonal entries (bold) indicate in-domain performance.

Test Dataset	Training Source		
	TableBench	WikiTQ	TabFact
TableBench(Rouge-L)	**0.491**	0.465	0.471
WikiTQ(Acc,%)	74.30	**77.03**	75.10
TabFact(Acc,%)	87.50	89.47	**91.30**

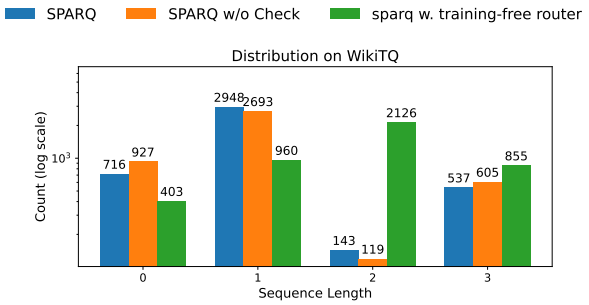


Fig. 5: Distribution of $|\mathcal{S}|$ on WikiTQ by SPARQ_{4B}.

larly, removing the check model Q results in a comparable performance drop, which underscores the critical importance of an information-aware verification step.

Cross-Domain Generalization. To verify that SPARQ learns intrinsic reasoning patterns rather than overfitting to specific dataset artifacts, we evaluate the transferability of our router across domains (Table VII). We train the router on one dataset and test it on others. As shown, the performance drop when transferring to unseen domains is minimal. For instance, on the challenging TABLEBENCH, models trained on WIKITQ and TABFACT achieve 0.465 and 0.471 Rouge-L, respectively, comparable to the in-domain baseline (0.491). This implies a relative degradation of less than 5%, confirming that the learned routing and verification policies generalize well across different schemas and query distributions.

F. Hyper-parameter Setting

Threshold τ for verify model. As shown in Fig. 6 (left), the value of τ presents a trade-off between TQA accuracy and operational complexity. A permissive threshold (e.g., $\tau = 0.2$) allows Q to approve most operator sets $\mathcal{S}$, which increases the average number of operators by 6.5% but degrades accuracy by 2.66% due to a higher risk of error propagation. A tight threshold (e.g., $\tau = 0.99$) causes Q to reject most operator sets, forcing frequent rollbacks to the Base model or fallbacks to SQL execution, leads to a substantial performance drop of over 25%, where we observe that more than 20% of answers are incorrect, primarily due to errors from processing improperly long contexts or faulty SQL results.

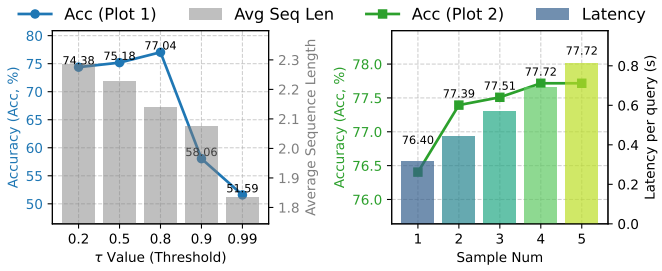


Fig. 6: Hyperparameter Analysis for SPARQ_{4B}.

Sampling Number. As shown in Fig. 6 (right), increasing the sample number for $o^{\text{row}}, o^{\text{col}}, o^{\text{SQL}}$ can correct formatting errors, yielding a modest performance gain of 1.32%. However, this benefit comes at a significant latency cost. For complex task of SQL generation, increasing the sample number from 1 to 5 inflates the end-to-end latency by 2.56×. To balance accuracy and efficiency, we select sample number of 2 for o^{row} and o^{col}, and 3 for the more computationally intensive o^{SQL}.

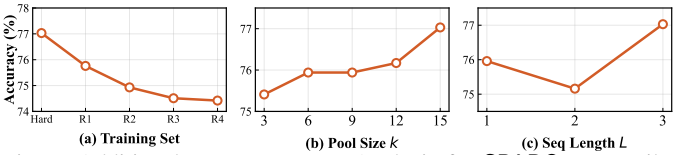


Fig. 7: Additional Hyperparameter Analysis for SPARQ_{4B} on WikiTQ dataset.

Robustness to Training Distribution. To address concerns regarding overfitting to "hard" patterns, we compare our default strategy (476 hard samples contain $>$ 4k tokens) against random sampling the same number of training examples in the validation set(R1-R4 in Fig. 10(a)). While hard samples yield peak accuracy (77.03%), the model maintains robust performance (avg. ∼74.9%) even when trained on random subsets. This moderate drop confirms that SPARQ learns intrinsic "minimal sufficient" reasoning capabilities rather than memorizing specific heuristics, and that the hard-sample strategy is an effective optimization for complex queries.

Sensitivity to Search Configuration (k and L). Fig. 10(b)-(c) illustrates the impact of search breadth and depth. Reducing the operator pool size k from 15 to 3 degrades accuracy, underscoring the need to cover long-tail operator combinations. For candidate count L, we observe a dip at $L = 2$ (75.16%) compared to $L = 1$ (75.96%), suggesting that blindly increasing L increases the risk that routers select overly aggressively pruned paths. However, extending to $L = 3$ provides sufficient diversity for the verifier to identify the optimal set, recovering accuracy to 77.03%.

VII. CONCLUSION

In this paper, we presented SPARQ, an offline framework for TableQA that addresses the key challenges of data privacy, latency, and cost in online systems. By integrating an adaptive query router with a mutual-information verifier, SPARQ dynamically selects the most efficient reasoning path for each query while ensuring information sufficiency. We verify the effectiveness and efficiency of SPARQ via extensive tests.

VIII. ACKNOWLEDGMENT

This work was supported in part by China NSFC 62225202, NSFC 62502326, NSFC 62572119, Shenzhen Natural Science Foundation JCYJ20250604184338052, Technology Innovation Guidance Program of Shandong Province YDZX2024088 and General Program of the China Postdoctoral Science Foundation 2025M784286.

AI-Generated Content Acknowledgement

The authors confirm that no part of the paper's main content, including text, figures, tables, or code, was generated by artificial intelligence (AI) tools. Minor assistance was limited to grammar checking and reference formatting using standard productivity software. No generative AI system was used in the conception, writing, or analysis of the scientific content. This statement is included in accordance with the ICDE 2026 policy on the disclosure of AI-generated content.

References

[1] N. Jin, J. Siebert, D. Li, and Q. Chen, "A survey on table question answering: recent advances," in *China Conference on Knowledge Graph and Semantic Computing*. Springer, 2022, pp. 174–186.

[2] X. Zhang, D. Wang, L. Dou, Q. Zhu, and W. Che, "A survey of table reasoning with large language models," *Frontiers of Computer Science*, vol. 19, no. 9, p. 199348, 2025.

[3] L. Nan, C. Hsieh, Z. Mao, X. V. Lin, N. Verma, R. Zhang, W. Kryściński, N. Schoelkopf, R. Kong, X. Tang, M. Mutuma, B. Rosand, I. Trindade, R. Bandaru, J. Cunningham, C. Xiong, and D. Radev, "FeTaQA: Free-form table question answering," *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 35–49, 2022.

[4] X. Deng, H. Sun, A. Lees, Y. Wu, and C. Yu, "Turl: Table understanding through representation learning," *ACM SIGMOD Record*, vol. 51, no. 1, pp. 33–40, 2022.

[5] P. Pasupat and P. Liang, "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)*. Association for Computational Linguistics, 2015, pp. 1470–1480. [Online]. Available: https://aclanthology.org/P15-1142/

[6] Y. Chung, G. T. Kakkar, Y. Gan, B. Milne, and F. Özcan, "Is long context all you need? leveraging llm's extended context for nl2sql," *VLDB*, vol. 18, no. 8, p. 2735–2747, Sep. 2025.

[7] Y. Zhang, J. Henkel, A. Floratou, J. Cahoon, S. Deep, and J. M. Patel, "ReAcTable: Enhancing react for table question answering," *Proc. VLDB Endow.*, vol. 17, no. 8, pp. 1981–1994, 2024. [Online]. Available: https://db.cs.cmu.edu/publications/

[8] P. Li, Y. He, D. Yashar, W. Cui, S. Ge, H. Zhang, D. Rifinski Fainman, D. Zhang, and S. Chaudhuri, "Table-gpt: Table fine-tuned gpt for diverse table tasks," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–28, 2024.

[9] J.-P. Zhu, P. Cai, K. Xu, L. Li, Y. Sun, S. Zhou, H. Su, L. Tang, and Q. Liu, "Autotqa: Towards autonomous tabular question answering through multi-agent large language models," *VLDB*, vol. 17, no. 12, pp. 3920–3933, 2024.

[10] J. Zhu, P. Cai, K. Xu, L. Li, Y. Sun, S. Zhou, H. Su, L. Tang, and Q. Liu, "Unitqa: A unified automated tabular question answering system with multi-agent large language models," in *SIGMOD*, Jun. 2025, pp. 279–282.

[11] G. Xiao, D. He, J. Wang, and M. Balazinska, "Cents: A flexible and cost-effective framework for llm-based table understanding," *Proceedings of the VLDB Endowment*, vol. 18, no. 11, pp. 4574–4587, 2025.

[12] A. Biswal, L. Patel, S. Jha, A. Kamsetty, S. Liu, J. E. Gonzalez, C. Guestrin, and M. Zaharia, "Text2sql is not enough: Unifying ai and databases with tag," *CIDR*, 2025.

[13] J. Herzig, P. K. Nowak, T. Müller, F. Piccinno, and J. Eisenschlos, "TaPas: Weakly supervised table parsing via pre-training," in *ACL*. Association for Computational Linguistics, 2020, pp. 4320–4333.

[14] Q. Liu, B. Chen, J. Guo, M. Ziyadi, Z. Lin, W. Chen, and J.-G. Lou, "Tapex: Table pre-training via learning a neural sql executor," in *ICLR*, 2022.

[15] W. Chen, H. Wang, J. Chen, Y. Zhang, H. Wang, S. Li, X. Zhou, and W. Y. Wang, "Tabfact: A large-scale dataset for table-based fact verification," in *ICLR*, 2020.

[16] H. Zhang, Y. Wang, S. Wang, X. Cao, F. Zhang, and Z. Wang, "Table fact verification with structure-aware transformer," in *EMNLP*, 2020, pp. 1624–1629.

[17] T. Yu, C.-S. Wu, X. V. Lin, bailin wang, Y. C. Tan, X. Yang, D. Radev, richard socher, and C. Xiong, "Grappa: Grammar-augmented pre-training for table semantic parsing," in *ICLR*, 2021.

[18] Z. Gu, J. Fan, N. Tang, P. Nakov, X. Zhao, and X. Du, "Pasta: Table-operations aware fact verification via sentence-table cloze pre-training," in *EMNLP*, 2022, pp. 4971–4983.

[19] S.-A. Chen, L. Miculicich, J. M. Eisenschlos, Z. Wang, Z. Wang, Y. Chen, Y. Fujii, H.-T. Lin, C.-Y. Lee, and T. Pfister, "TableRAG: Million-token table understanding with language models," in *Thirty-eighth Conference on Neural Information Processing Systems*, 2024. [Online]. Available: https://openreview.net/forum?id=41lovPOCo5

[20] Y. Ye, B. Hui, M. Yang, B. Li, F. Huang, and Y. Li, "Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning," in *SIGIR*, 2023, pp. 174–184.

[21] X. Zhang, D. Wang, K. Xu, Q. Zhu, and W. Che, "RoT: Enhancing table reasoning with iterative row-wise traversals," in *EMNLP*, 2025.

[22] Z. Wang, H. Zhang, C.-L. Li, J. M. Eisenschlos, V. Perot, Z. Wang, L. Miculicich, Y. Fujii, J. Shang, C.-Y. Lee, and T. Pfister, "Chain-of-table: Evolving tables in the reasoning chain for table understanding," in *ICLR*, 2024. [Online]. Available: https://openreview.net/forum?id=4L0xnS4GQM

[23] M. M. H. Nahid and D. Rafiei, "TabSQLify: Enhancing reasoning capabilities of LLMs through table decomposition," in *NAACL*, 2024.

[24] N. Abhyankar, V. Gupta, D. Roth, and C. K. Reddy, "H-STAR: LLM-driven hybrid SQL-Text adaptive reasoning on tables," in *Proceedings of the 2025 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2025. [Online]. Available: https://aclanthology.org/2025.naacl-long.445/

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *NIPS*, vol. 33, pp. 1877–1901, 2020.

[26] M. Chen, J. Tworek *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[27] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen *et al.*, "Palm 2 technical report," *arXiv preprint arXiv:2305.10403*, 2023.

[28] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[29] "Qwen3-4B model card," 2025, https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507.

[30] "Qwen3-30B model card," 2025, https://huggingface.co/Qwen/Qwen3-30B-A3B-Instruct-2507-FP8.

[31] Z. Wan, X. Wang, C. Liu, S. Alam, Y. Zheng, J. Liu, Z. Qu, S. Yan, Y. Zhu, Q. Zhang *et al.*, "Efficient large language models: A survey," *TMLR*, 2024.

[32] L. Chen, M. Zaharia, and J. Zou, "Frugalgpt: How to use large language models while reducing cost and improving performance," *Transactions on Machine Learning Research*, 2024.

[33] J. Ahn, R. Verma, R. Lou, D. Liu, R. Zhang, and W. Yin, "Large language models for mathematical reasoning: Progresses and challenges," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, 2024, pp. 225–237.

[34] Q. Zhang, C. Hu, S. Upasani, B. Ma, F. Hong, V. Kamanuru, J. Rainton, C. Wu, M. Ji, H. Li *et al.*, "Agentic context engineering: Evolving contexts for self-improving language models," *arXiv preprint arXiv:2510.04618*, 2025.

[35] Y. Zhang, A. Floratou, J. Cahoon, S. Krishnan, A. C. Müller, D. Banda, F. Psallidas, and J. M. Patel, "Schema matching using pre-trained language models," in *ICDE*. IEEE, 2023, pp. 1558–1571.

[36] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (llm) security and privacy: The good, the bad, and the ugly," *High-Confidence Computing*, vol. 4, no. 2, p. 100211, 2024.

[37] B. Yan, K. Li, M. Xu, Y. Dong, Y. Zhang, Z. Ren, and X. Cheng, "On protecting the data privacy of large language models (llms) and llm agents: A literature review," *High-Confidence Computing*, vol. 5, no. 2, p. 100300, 2025.

[38] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *SOSP*, 2023, pp. 611–626.

[39] Y. Mao, X. Yan, Q. Guo, and Y. Ye, "Deep mutual information maximin for cross-modal clustering," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, 2021, pp. 8893–8901.

[40] Z. Wang, Y. Zhao, H. Huang, J. Liu, A. Yin, L. Tang, L. Li, Y. Wang, Z. Zhang, and Z. Zhao, "Connecting multi-modal contrastive representations," *Advances in Neural Information Processing Systems*, vol. 36, pp. 22 099–22 114, 2023.

[41] Z. Cheng, T. Xie, P. Shi, C. Li, R. Nadkarni, Y. Hu, C. Xiong, D. Radev, M. Ostendorf, L. Zettlemoyer *et al.*, "Binding language models in symbolic languages," in *ICLR*, 2023.

[42] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, vol. 35. Curran Associates, Inc., 2022, pp. 24 824–24 837. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[43] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 3911–3921. [Online]. Available: https://aclanthology.org/D18-1425/

[44] The Qwen Team, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[45] P. Shojaee, I. Mirzadeh, K. Alizadeh, M. Horton, S. Bengio, and M. Farajtabar, "The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity," *arXiv preprint arXiv:2506.06941*, 2025.

[46] A. Su, A. Wang, C. Ye, C. Zhou, G. Zhang, G. Chen, G. Zhu, H. Wang, H. Xu, H. Chen *et al.*, "Tablegpt2: A large multimodal model with tabular data integration," *arXiv preprint arXiv:2411.02059*, 2024.

[47] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in *Proceedings of the 2015 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, 2015, pp. 416–432.

[48] C. Quirk, R. Mooney, and M. Galley, "Language to code: Learning semantic parsers for if-this-then-that recipes," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2015, pp. 878–888.

[49] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems 27*, 2014.

[50] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 933–944.

[51] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[52] T. Scholak, N. Schucher, and D. Bahdanau, "Picard: Parsing incrementally for constrained auto-regressive decoding from language models," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 9895–9901.

[53] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. Wang, and X. V. Lin, "Lever: Learning to verify language-to-code generation with execution," in *ICML*. PMLR, 2023, pp. 26 106–26 128.

[54] T. Yu *et al.*, "Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases," in *EMNLP*, 2019, pp. 1962–1979.

[55] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *arXiv preprint arXiv:1709.00103*, 2017.

[56] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo *et al.*, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," *Advances in Neural Information Processing Systems*, vol. 36, pp. 42 330–42 357, 2023.

[57] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversation," in *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024.

[58] Y. Xu, X. Guo, Z. Zeng, and C. Miao, "Softcot++: Test-time scaling with soft chain-of-thought reasoning," *arXiv preprint arXiv:2505.11484*, 2025.

[59] Y. Ge, S. Liu, Y. Wang, L. Mei, L. Chen, B. Bi, and X. Cheng, "Innate reasoning is not enough: In-context learning enhances reasoning large language models with less overthinking," *arXiv preprint arXiv:2503.19602*, 2025.

[60] Z. Wei, L. Pang, J. Liu, J. Deng, S. Xu, Z. Duan, J. Wang, F. Sun, X. Cai, H. Shen *et al.*, "The evolution of thought: Tracking llm overthinking via reasoning dynamics analysis," *arXiv preprint arXiv:2508.17627*, 2026.

[61] C.-H. Chiang and H.-Y. Lee, "Over-reasoning and redundant calculation of large language models," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2024, pp. 161–169.

[62] C. Fan, M. Li, L. Sun, and T. Zhou, "Missing premise exacerbates overthinking: Are reasoning models losing critical thinking skill?" *arXiv preprint arXiv:2504.06514*, 2025.

[63] Y. Du, M. Tian, S. Ronanki, S. Rongali, S. B. Bodapati, A. Galstyan, A. Wells, R. Schwartz, E. A. Huerta, and H. Peng, "Context length alone hurts LLM performance despite perfect retrieval," in *Findings of the Association for Computational Linguistics: EMNLP 2025*. Association for Computational Linguistics, 2025, pp. 23 281–23 298. [Online]. Available: https://aclanthology.org/2025.findings-emnlp.1264/

[64] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[65] W. Yeo, K. Kim, S. Jeong, J. Baek, and S. J. Hwang, "Universalrag: Retrieval-augmented generation over corpora of diverse modalities and granularities," *arXiv preprint arXiv:2504.20734*, 2025.

[66] F. Crestani, M. Lalmas, C. J. Van Rijsbergen, and I. Campbell, ""is this document relevant?... probably" a survey of probabilistic models in information retrieval," *ACM Computing Surveys (CSUR)*, vol. 30, no. 4, pp. 528–552, 1998.

[67] J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu, "M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation," in *Findings of the Association for Computational Linguistics ACL 2024*, 2024, pp. 2318–2335.

[68] X. Ma, Y. Gong, P. He, N. Duan *et al.*, "Query rewriting in retrieval-augmented large language models," in *EMNLP*, 2023.

[69] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in *EMNLP*, 2019.

[70] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: BM25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[71] G. V. Cormack, C. L. Clarke, and S. Buettcher, "Reciprocal rank fusion outperforms condorcet and individual rank learning methods," in *SIGIR*, 2009, pp. 758–759.

[72] Microsoft, *Azure OpenAI Service Documentation*, 2024, https://learn.microsoft.com/en-us/azure/ai-services/openai/.

[73] A. W. Services, *Amazon Bedrock User Guide*, 2024, https://docs.aws.amazon.com/bedrock/.

[74] K. Bonawitz *et al.*, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 1175–1191.

[75] O. Goldreich, *Foundations of Cryptography, Volume 2: Basic Applications*. Cambridge University Press, 2009.

[76] A. van den Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," in *arXiv preprint arXiv:1807.03748*, 2018.

[77] Z. Shao, S. Cai, R. Lin, and Z. Ming, "Enhancing text-to-SQL with question classification and multi-agent collaboration," in *Findings of the Association for Computational Linguistics: NAACL 2025*. Association for Computational Linguistics, 2025.

[78] H. Li, J. Zhang, C. Li, and H. Chen, "Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 11, 2023, pp. 13 067–13 075.

[79] X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Zhang, J. Fan, G. Li, N. Tang, and Y. Luo, "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*, 2025.

[80] X. Wu, J. Yang, L. Chai, G. Zhang, J. Liu, X. Du, D. Liang, D. Shu, X. Cheng, T. Sun *et al.*, "Tablebench: A comprehensive and complex benchmark for table question answering," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 24, 2025, pp. 25 497–25 506.

[81] L. Wang *et al.*, "Needleinatable: Exploring long-context capability of large language models towards long-structured tables," in *NeurIPS*, 2025.

[82] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[83] Q. Team, "Qwen3-max: Just scale it," September 2025. [Online]. Available: https://qwen.ai/blog?id=qwen3-max

[84] C. Li, Z. Liu, S. Xiao, Y. Shao, and D. Lian, "Llama2vec: Unsupervised adaptation of large language models for dense retrieval," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 3490–3500.

[85] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, 2004, pp. 74–81.

[86] A. Srivatsa, Z. Li, M. Glass, A. Gliozzo, and M. Sadoghi, "LM^2: A simple yet effective method for decomposed reasoning," *arXiv preprint arXiv:2404.02255*, 2024.

[87] D. Ding, A. Mallick, S. Zhang, C. Wang, D. Madrigal, M. D. C. H. Garcia, M. Xia, L. V. S. Lakshmanan, Q. Wu, and V. Rühle, "BEST-route: Adaptive LLM routing with test-time optimal compute," in *ICML*, 2025. [Online]. Available: https://openreview.net/forum?id=tFBIbCVXkG

[88] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, and L. Ceze, "Flashinfer: Efficient and customizable attention engine for LLM inference serving," in *MLSys*, 2025. [Online]. Available: https://openreview.net/forum?id=RXPofAsL8F

[89] A. Kuzmin, M. Van Baalen, Y. Ren, M. Nagel, J. Peters, and T. Blankevoort, "Fp8 quantization: The power of the exponent," *Advances in Neural Information Processing Systems*, vol. 35, pp. 14 651–14 662, 2022.

APPENDIX

A. Detailed Operators and corresponding prompts in SPARQ

In this section, we list a set of five operators, which are commonly used in SQL and LLM-based tableQA solutions. In the following, we present the specific operator and corresponding prompts. For simplicity, the demonstration examples are omitted from the presented prompts.

Base QA Operator o^{Base}: A baseline that provides LLM with the full table T and three few-shot demonstrations. Following CoT [42], it uses chain-of-thought, prompting LLM to divide TQA into several sub-tasks and lead to the final result. The prompt list is listed in Figure 18.

Column Extraction Operator o^{col}: SPARQ employs a column extraction operator, denoted as o^{col}, to identify query-relevant columns via two parallel paths: (1) direct enumeration via LLM reasoning $t^{\text{col}}(\cdot)$, and (2) SQL-based projection $S^{\text{col}}(\cdot)$. The union $o^{\text{col}}(q,T) = t^{\text{col}}(\cdot) \cup S^{\text{col}}(\cdot)$ forms the sub-table T^{col}, mitigating hallucination and syntax errors arising from single-path. Prompt is listed in Figure 13.

Row extraction Operator o^{row} SPARQ applies a row-based operator, O^{row}, to filter query-relevant rows via two complementary paths: (1) direct enumeration via LLM reasoning $t^{\text{row}}(\cdot)$, and (2) SQL-based selection $S^{\text{row}}(\cdot)$. The merged result $T^{\text{row}} = t^{\text{row}}(\cdot) \cup S^{\text{row}}(\cdot)$ forms the refined sub-table for subsequent reasoning. The prompt is listed in Figure 14.

Retrieval operator o^{Ret}**.** The retrieval operator O^{Ret} is a key innovation of SPARQ, introducing hybrid semantic retrieval that combines dense and sparse search. Given (q,T), the LLM expands q into multi-granular variants (general, balanced, specific), while rows and columns are linearized and scored using SBERT embeddings [69] as dense retrieval and BM25 [70] as sparse retrieval to measure pairwise similarity, and applies weighted reciprocal rank fusion [71] to merge them. The top-M rows and top-N columns are retained as T^{Ret}, with additional text or hyperlinks extracted for web tables. By bridging symbolic reasoning with RAG-style retrieval, SPARQ enables offline LLMs to efficiently access contextual evidence. The rewrite prompt is listed in Figure 15.

SQL execute operator. SPARQ employs an SQL execution operator, O^{SQL}, to perform symbolic computation by generating and executing SQL from (q,T). It supports filtering, comparison, and aggregation functions (e.g., COUNT, AVG), and enhances robustness by returning [UNKNOWN] when execution is infeasible, so that such operations are automatically skipped. Prompt is listed in Figure 17.

Training-Free Router $E_{4\text{B}}$ **and** $E_{30\text{B}}$**.** Following [65], we additionally test training-free router, which performance is evaluated in Section VI-E. Prompt is listed in Figure 16.

B. Model Parameters of different models in SPARQ

The model parameters utilized within SPARQ are detailed in Table VIII. For non-GPU components, our implementation is based on `Python 3.10`, incorporating several key libraries for specific functionalities. We employ `sqlite3` and `sqlalchemy` for database construction and efficient SQL execution; `BM25` is used to handle sparse retrieval tasks; We utilize `pandas` for core data manipulation, enhanced by `pandarallel` to facilitate multithreaded data preprocessing.

C. Detailed Statistics of Safeguard Mechanisms

To assess the computational overhead of our safeguard mechanisms, we monitored the trigger frequency of both *Rollback* (internal re-routing) and *Fallback* (external SQL execution) across multiple benchmarks. As shown in Table IX, the Fallback mechanism is invoked sparingly, serving only as a necessary safety net for the most challenging queries.

D. Detailed optimization strategies

In TQA systems, the end-to-end inference pipeline typically comprises two distinct stages: (1) CPU-intensive tabular preprocessing and database operations; and (2) GPU-accelerated LLMs inference. A naïve implementation executes these two stages sequentially in a single thread, processing one query at a time adapted by most online model systems. While simple, this design leads to severe GPU underutilization due to frequent idle periods, commonly referred to as CPU-GPU bubbles, during which the GPU stalls while waiting for the CPU to complete preprocessing for the next query.

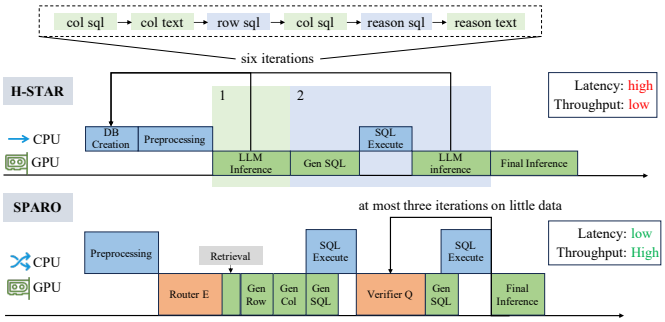


Fig. 8: Scheduling Pipeline of baseline H-STAR and SPARQ.

To address this inefficiency, we introduce a batched pipeline execution strategy that decouples and parallelizes the CPU-bound and GPU-bound stages of the workflow. Our design consists of two core components as depicted in Fig. 8.

Multi-threaded preprocessing. We parallelize tabular preprocessing using multithread and optimize the database operations through an asynchronous scheduling pipeline that enables streaming management of the SQL execution. To enhance robustness, we introduce an optimized parsing algorithm that automatically detects and filters out malformed SQL queries, thereby preventing erroneous queries from blocking the execution pipeline. This approach maximizes CPU throughput and reduces per-query preprocessing latency, especially in database-augmented TQA pipelines where SQL execution and robustness to malformed queries are critical.

Offline batching for GPU inference. Rather than dispatching

TABLE VIII: SPARQ component names, attributes, corresponding example design parameters, and model names.

SPARQ **Components**	**Attributes**	**Attributes**	**Model Name**
Embedding Model	Model size of the embedding model for o^{Ret}.	568M	BGE-M3
Backbone of Router	Model size of backboned router model E	568M	BGE-M3
Vector Dimensionality	The number of dimensions for each embedded vector	1,024	
Backbone of Check Model	Model size of backboned Check model V	560M	BGE-Reranker-Large
QA Model for $\mathsf{SPARQ}_{\mathsf{4B}}$	Model size of the LLM used for QA and operator generation.	4B	Qwen3-4B [29]
QA Model for $\mathsf{SPARQ}_{\mathsf{30B}}$	Model size of the LLM used for QA and operator generation.	30B	Qwen3-30B [30]

TABLE IX: Statistics of Rollback and Fallback mechanisms across different benchmarks. The *Rate* columns indicate the percentage of the total test set where the respective mechanism was triggered. Low Fallback rates confirm that the computationally expensive safeguard is invoked only sparingly.

Dataset	**Total Size**	**Rollback**		**Fallback**	
		Count	**Rate**	**Count**	**Rate**
WikiTQ	4,344	1,957	45.05%	528	12.15%
TableBench	886	173	19.53%	71	8.01%
TabFact	2,024	390	19.27%	295	14.57%
FetaQA	2,003	289	14.43%	60	3.00%
NIAT	3,989	920	23.06%	191	4.79%

each preprocessed query immediately, we batch structurally and semantically similar queries to maximize KV cache reuse and throughput. A resource-aware dynamic loader manages multiple GPU models, temporarily loading lightweight components (e.g., routing and verification) and releasing them upon completion to free memory for LLM inference. This offline batching exploits LLM parallelism, substantially improving GPU utilization and efficiency.

Batching combines multiple preprocessed queries into a single GPU operation, enabling concurrent processing that reduces CPU–GPU bubbles. For the offline TQA, this approach is far more effective than multithread (validated in Section VI-C). Notably, our operator set is mutually independent which can execute in parallel, with results combined via intersection. However, the current scheduler does not leverage this property, which we leave for future work.

TABLE X: hyper-parameters for different operators in both $\mathsf{SPARQ}_{\mathsf{4B}}$ and $\mathsf{SPARQ}_{\mathsf{30B}}$. temp. represents temperature hyper-parameter for LLM inference.

Operators	**temp.**	**top_p**	**output_tokens**	**samples**	**examples**
o^{col}	0.7	0.8	2048	2	3
o^{row}	0.7	0.8	2048	2	3
o^{ret}	0.7	0.8	1024	1	3
o^{sql}	0.7	0.8	2048	3	4
o^{Base}	0.0	1.0	512	1	4
Final QA	0.0	1.0	512	1	4
$E^{\mathsf{4B}}, E^{\mathsf{30B}}$	0.0	1.0	128	1	3

E. Inference Parameter for SPARQ

The hyperparameters used to query LLM for each operator are listed in Table X. All datasets and methods within SPARQ adhere to this identical setting. It is worth noting that we additionally set `top_k=20`, `min_p=0` and `presence_penalty=1` to mitigate the issue of hallucination. For clarification, the term `samples` indicates the number of generations performed for the same query, while `examples` denotes the number of demonstrations for each query. Furthermore, E^{4B} and E^{30B} represent the training-free router instances discussed in detail in Section VI-E.

TABLE XI: Number of generated samples for different methods on dataset WikiTQ.

Method	**# samples / step**	**Total # samples**
DATER	Decompose Table: 40 Generate Cloze: 20 Generate SQL: 20 Query: 20	100
Chain-of-Table	Dynamic Plan ≤ 5 Generate Args ≤ 19 Query: 1	≤ 25
TabSQLify	Table Decompose: 1 Query: 1	2
H-STAR	Column Extraction: 4 Row Extraction: 4 Query: 2	10
ReACTable(s-vote)	Generate SQL: ≤ 10 Generate Python: ≤ 5 Query: 1	10.20
SPARQ (Ours)	Column Selection: 2 Row Selection: 2 SQL Execute: 3 Rewrite+Retrieval: 1 Query: 1	5.01

F. Analysis: Generated Samples

In Table XI, we analyze the sample generation efficiency of various LLM-based solutions by comparing the average number of samples generated per query. This analysis serves as an extended discussion of the motivation presented in Section III and Figure 2. DATER utilizes self-consistency refinement at each step, resulting in 100 samples per query. In contrast, Chain-of-Table uses a more resource-efficient methods, denoted as `Dynamic Plan`,`Generate Args Plan` and `Query`, which significantly save the LLM generated sample number per query. H-STAR applies a fixed pipeline with intertwined steps of symbolic and textual extraction of column, rows and SQL generation. ReAcTable employs the ReACT method to generate multiple candidates with SQL and Python code in a sandbox environment, then conducts majority voting to integrate multiple results. TabSQLify generates the fewest samples, with one generation each for the table decomposition and query steps. While SPARQ use the dynamic route and check mechanism, to dispatch different level of queries into different subset of operators. Detailed discussion is listed in Section III.

G. Implementation Detail for baseline methods

We run H-STAR, ReAcTable, Chain-of-Table and TabSQLify using their official implementation and prompts in

GitHub. In Table III, for baseline ReAcTable, Chain-of-Table and H-STAR, we conduct their native multithread implementation to 16 for best efficiency, while TabSQLify is in single thread, as it cannot natively support multithreading.

In Table III, we made minor modifications to ensure a fair comparison of the baselines. In detail, for H-STAR, Chain-of-Tables and TabSQlify, we use its default hyper-parameter designed for GPT-3.5-Turbo(*e.g.,* prompts, number of samples and demonstration examples), except that we set `top_p=0.7` and `top_k=0.8`, following the suggestion of qwen-3 series [29]. For ReAcTable, we use *s-vote* among its variant methods, which shows stability in their original paper.

In Table IV, the average prediction time and the I/O token throughput for all methods were calculated by client-side measurement of the server's response and I/O tokens across all requests. This was performed on an identical offline `vllm`-based server (OpenAI-capatible) using the same model `qwen3-4B`. To maximize generation speed, the `vllm` server was enabled with `FlashInfer` [88] and `fp8`-quantization [89] for both model weights and KV-Cache.

H. Detailed Time Breakdown Analysis

To provide a transparent and granular breakdown of latency sources, we recorded the execution time of each pipeline step. **Experimental Setup:** All time statistics reported in Table XII were collected on a **single NVIDIA RTX 4090 GPU**. To ensure a clear dissection of computational costs without the confounding factors of parallelism, the pipeline was executed in a **single-threaded sequential mode**. This strict setup allows us to precisely attribute the total runtime to specific modules and identify system bottlenecks.

The pipeline execution is categorized into five logical stages: (1) **Preprocessing** (Steps 1 & 3): Includes data loading and the construction of ephemeral SQLite databases for each table. (2) **Router** (Steps 2, 4 & 5): Encompasses the construction of routing prompts, the inference of the Router model, and the parsing of generated plans. (3) **Operator Execution** (Steps 6-9): Represents the core execution phase, divided into retrieval (*RAG*), context pruning (*Select Row/Col*), and symbolic reasoning (*SQL Generation & Execution*). (4) **Verification** (Steps 10 & 11): The safeguard mechanism that iteratively checks for missing information or execution errors. (5) **Final QA** (Steps 12 & 13): The final generation phase where the model synthesizes the answer based on the processed context.

Analyzing the breakdown reveals two critical insights regarding efficiency and task characteristics. First, the overhead of our proposed architectural components is minimal. The **Router** typically consumes less than 1.5% of the total time (e.g., 42.3s out of 3291.6s on NIAT), confirming that separating planning from reasoning incurs negligible latency cost. More importantly, the **Verification** overhead is exceptionally low, accounting for merely **0.6%** on WikiTQ and **0.2%** on TableBench. This empirically validates that our safeguard mechanism effectively amortizes its cost by triggering expensive fallbacks only when absolutely necessary. Second, the dominant latency sources align with dataset characteristics: WikiTQ, known for complex symbolic logic, spends the majority of time on *SQL Generation* (52.8%), whereas TabFact, which requires verifying long-context claims, heavily relies on *Select Operators* (48.9%) to prune tables. This demonstrates that SPARQ dynamically allocates computational resources to the operators most suited for the specific data modality.

I. Analysis of Iterative Code Generation Baselines

To benchmark SPARQ against pure code-generation approaches, we implemented two iterative baselines: **SPARQ-Py** and **SPARQ-SQL**. These methods prompt the LLM to generate Python or SQL code to solve the query. If execution fails (e.g., syntax errors or runtime exceptions), the model is reprompted with the error message up to 3 times. If all attempts fail, the system falls back to direct LLM generation.

Table XIII details their performance in terms of Accuracy, Latency (per query), and Execution Success Rate (ESR). **Key Observations:** (1) **Execution Instability:** The code generation methods suffer from inconsistent execution success rates. For instance, on NIAT (Qwen-4B), SPARQ-SQL only achieves a 28.85% execution success rate, heavily relying on the fallback mechanism. (2) **Performance Gap:** SPARQ consistently outperforms these baselines. On NIAT (Qwen-30B), SPARQ achieves **73.45%** accuracy with **0.95s** latency, surpassing SPARQ-Py (52.05%, 0.84s) in accuracy by a large margin and outperforming SPARQ-SQL (72.27%, 1.57s) in latency significantly. (3) **Efficiency Trade-off:** While SPARQ-Py is sometimes faster due to simple logic generation, it lacks the semantic depth required for complex reasoning, leading to lower accuracy. SPARQ strikes a superior balance by combining structural extraction with symbolic reasoning.

J. Complementary Roles of Operators

To further understand the contribution of different operator types, we conducted an ablation study removing specific operator groups, detailed in Table VI. We observed that the removal of either table extraction operators or the Execute_SQL operator leads to a notable decline in performance, though these ablations impact different subsets of samples.

Specifically, removing **table extraction** severely impairs the LLM's ability to parse and reason over long tables, as the model loses the ability to simplify the context window. In contrast, removing the **Execute_SQL** operator deprives the LLM of a mechanism to verify its own calculations, resulting in frequent numerical reasoning errors. This confirms that SPARQ's hybrid design is essential for handling the diverse challenges of real-world TQA.

K. Error Analysis: Case Study

We provide additional case studies among different operator sequences in Figure 11 and Figure 12. Both cases are selected from the test split of the WikiTQ dataset.

L. Reasoning cliff test on TabelBench

We evaluated SPARQ on TableBench to show that long-form table deeply effects the TAQ performance. As shown

TABLE XII: Time breakdown (in seconds) of different pipeline stages across benchmarks. **Preprocessing** includes data loading and database construction. **Router** includes query construction and inference. **Verification** sums the check model iteration and fallback SQL addition. **Final QA** includes prompt assembly and the final reasoning generation.

Dataset	Preproc.	Router	Operator Execution			Verify	Final QA	Total
	(Step 1,3)	(Step 2,4,5)	RAG	Select	SQL	(Step 10,11)	(Step 12,13)	Time
TableBench	11.6	12.3	0.0	148.0	279.4	2.4	598.9	1052.5
NIAT	5.0	42.3	9.9	679.6	1410.5	5.8	1138.4	3291.6
TabFact	27.8	21.8	9.5	1293.9	584.2	8.0	699.1	2644.1
WikiTQ	58.2	51.8	9.7	1007.8	2944.2	34.4	1469.0	5575.0

TABLE XIII: Performance of iterative code generation baselines. **Acc.** denotes Rouge-L for TableBench and Exact Match (EM) for NIAT. **Lat.** is the average per-query latency in seconds. **ESR** (Execution Success Rate) indicates the percentage of queries successfully solved by code execution without reverting to fallback.

Dataset	Method	Accuracy/EM	Latency	ESR
	Backbone: Qwen3-4B			
TableBench (N=886)	SPARQ	49.08	1.07s	-
	SPARQ-Py	41.65	0.88s	45.71%
	SPARQ-SQL	35.19	1.35s	60.95%
	Backbone: Qwen3-30B			
	SPARQ	52.00	1.32s	-
	Execute-Py	49.46	0.76s	74.83%
	Execute-SQL	45.75	1.13s	81.72%
	Backbone: Qwen3-4B			
NIAT (N=3,989)	SPARQ	66.58	0.72s	-
	Execute-Py	44.95	0.52s	53.75%
	Execute-SQL	64.08	0.56s	28.85%
	Backbone: Qwen3-30B			
	SPARQ	73.45	0.94s	-
	Execute-Py	52.05	0.84s	62.55%
	Execute-SQL	72.27	1.58s	48.12%

TABLE XIV: Table size trends on TableBench with Qwen3-30B

Tokens	Rouge-L
$0-300$	0.4750
$400-550$	0.5437
$550-800$	0.5267
$800-1200$	0.5011
$1200+$	0.4753

in Table XIV, the Rouge-L score exhibits a non-linear trend: performance peaks at moderate table lengths (400–550 tokens, Rouge-L = 0.5437) and gradually declines as tables become longer, returning to baseline levels ($\geq$1200 tokens). This "reasoning cliff" suggests that excessively long tables may overwhelm the model's contextual reasoning capacity.

M. Performance on FeTaQA

We evaluated SPARQ with training-based and training-free baselines on FeTaQA. The results detailed in Table XV show SPARQ outperforms the baseline in the vast majority of cases.

N. The Verification Module Minimizes Information Loss from Aggressive Routing

Fig. 9 illustrates the average number of table cells retrieved per query by SPARQ compared to baseline methods. We observe that a router model, due to its limited input and representational capacity, is naturally biased towards selecting more complex and aggressive extraction paths, which introduces a

TABLE XV: Experimental results on FeTaQA.

Model	Rouge-1	Rouge-2	Rouge-L
Approaches requiring training			
T5-small	0.55	0.33	0.47
T5-base	0.61	0.39	0.51
T5-large	0.63	0.41	0.53
Approaches without training			
DATER	0.66	0.45	0.56
ReAcTable	**0.71**	0.46	0.61
TabSQLify	0.58	0.35	0.48
$\text{HSTAR}_{\text{gpt3.5}}$	0.62	0.39	0.52
Chain-of-Tables	0.66	0.44	0.56
Ours			
SPARQ_{4B}	0.60	0.36	0.49
$\text{SPARQ}_{\text{30B}}$	0.69	**0.51**	**0.65**

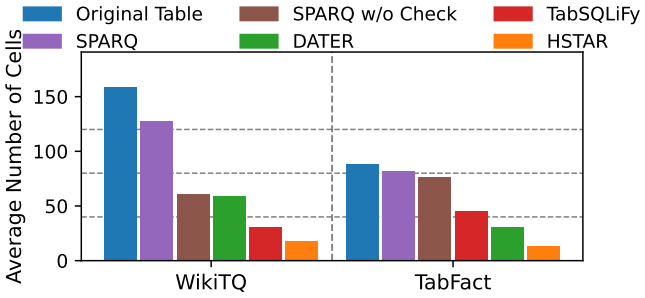


Fig. 9: Comparison of average table cells retrieved for final QA for SPARQ_{4B} and baselines with online LLM baselines.

significant risk of information loss. The verification model, Q, counteracts this tendency by triggering a rollback mechanism for erroneous selections that result in an insufficient context. This corrective action was activated in 51.9% of table size on the WikiTQ dataset and 7.4% on TabFact. Although this makes SPARQ's retrieval strategy appear more conservative than the baselines (i.e., it retains more cells on average), this is a direct consequence of prioritizing information sufficiency.

O. Performance with online LLMs

To verify the scalability of SPARQ beyond offline settings, we evaluated its performance using the state-of-the-art online model, **Qwen3-Max** [83], on the challenging TableBench dataset. The results, summarized in Table XVI, highlight three critical advantages:

- **Superior Accuracy:** SPARQ achieves a Rouge-L score of **0.55**, significantly outperforming the agent-based ReAcTable (0.40) and the decomposition-based H-STAR

TABLE XVI: Experimental results on TableBench with Qwen3-Max.

Method	Qwen3-4B		Qwen3-30B		Qwen3-Max		
	Rouge-L	Latency	Rouge-L	Latency	Rouge-L	Latency	Token Consumption
H-STAR	0.02	2.63	0.26	3.74	0.35	17.23	5.1M
ReAcTable	0.05	**1.57**	0.39	1.90	0.40	15.73	48.0M
SPARQ	**0.49**	1.71	**0.52**	**1.32**	**0.55**	**2.67**	**3.8M**

TABLE XVII: Experimental results on the robustness of SPARQ to table width variations. The metrics of Qwen3-4B and Qwen3-30B are Rouge-L.

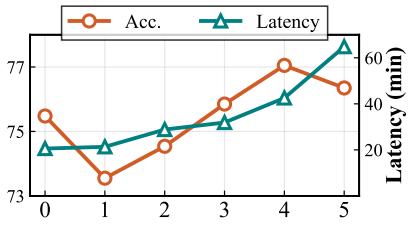
Dataset	Avg. Columns	Max Columns	Qwen3-4B	Qwen3-30B
TableBench_num	6.63	20	0.6488	0.7269
TableBench_cols	42.31	49	0.6638	0.7579

(0.35). This confirms that our routing mechanism effectively enhances even the most capable frontier models.

- **Drastic Cost Reduction:** A key bottleneck of agentic frameworks is token consumption. ReAcTable consumes **48.0M** tokens due to its verbose thought loops. In contrast, SPARQ consumes only **3.8M** tokens ($\approx 12\times$ reduction) while achieving higher accuracy, making it a far more economically viable solution for enterprise deployment.
- **Low Latency:** SPARQ maintains a latency of **2.67s**, which is comparable to direct inference and dramatically faster than ReAcTable (15.73s) and H-STAR (17.23s). This efficiency stems from our non-iterative routing design, avoiding the high latency penalty of multi-turn agent interactions.

P. Analysis of the robustness of SPARQ *to table width*

To evaluate the robustness of SPARQ to tables width, we additionally evaluate SPARQ on two synthesized variants: TableBench_num (avg. 6.63 columns, up to 20) and TableBench_cols (avg. 42.31 columns, up to 49). There variants are constructed via feature derivation, with results detailed in Table XVII. SPARQ achieves consistent performance across both datasets on Qwen3-4B and Qwen3-30B (Rouge-L 0.6488 and 0.7269 vs. 0.6638 and 0.7579). This robustness is enabled by the Column-Filter operator, which prunes irrelevant attributes and prevents the LLM from being overwhelmed by excessive table width.



(d) Op Set $|\mathcal{O}|$

Fig. 10: Additional Hyperparameter Analysis of operator set expansion for SPARQ_{4B} on WikiTQ dataset.

Q. Hyper-parameter: impact of Operator Set Expansion ($|\mathcal{O}|$).

Fig. 10(d) evaluates the cost-benefit trade-off of operator expansion. The system achieves peak accuracy (77.05%) with the inclusion of the o^{SQL} operator (Index 4), which offloads symbolic logic effectively. However, further expanding to o^{Python} (Index 5) causes a performance drop and a sharp latency spike. This indicates that unconstrained code execution is not always beneficial for offline models, given their limited code-generation success rate (69%) and lower tolerance to noisy data types compared to structured SQL.

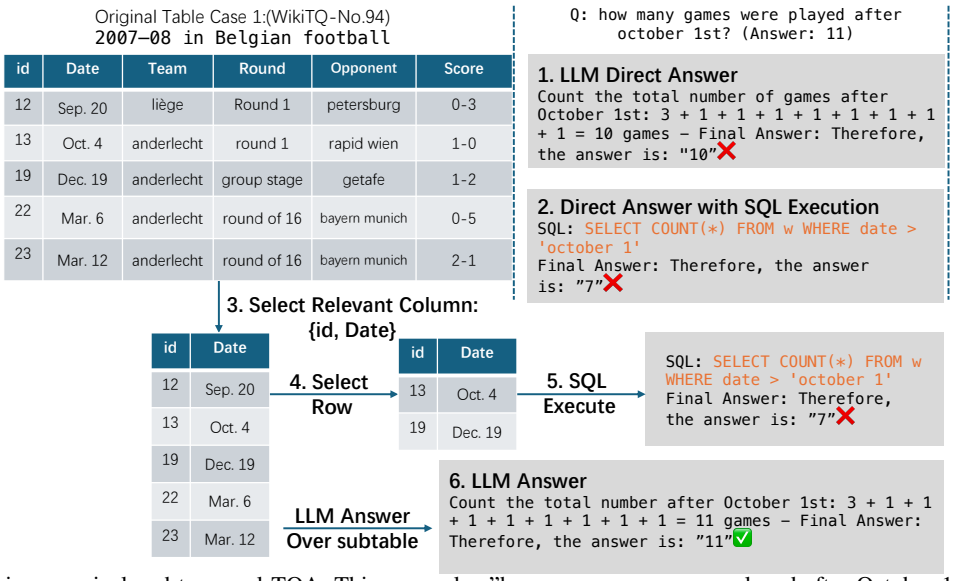


Fig. 11: Failure case in numerical and temporal TQA. This case asks, "how many games were played after October 1st?" (Correct Answer: 11). The LLM's direct answer yields "10" due to a counting error, exposing its weak native numerical ability. The direct SQL-based answer yields "7" because it fails to select games in next year(*e.g.,* row 23), highlighting the fragility of symbolic methods with non-normalized data.

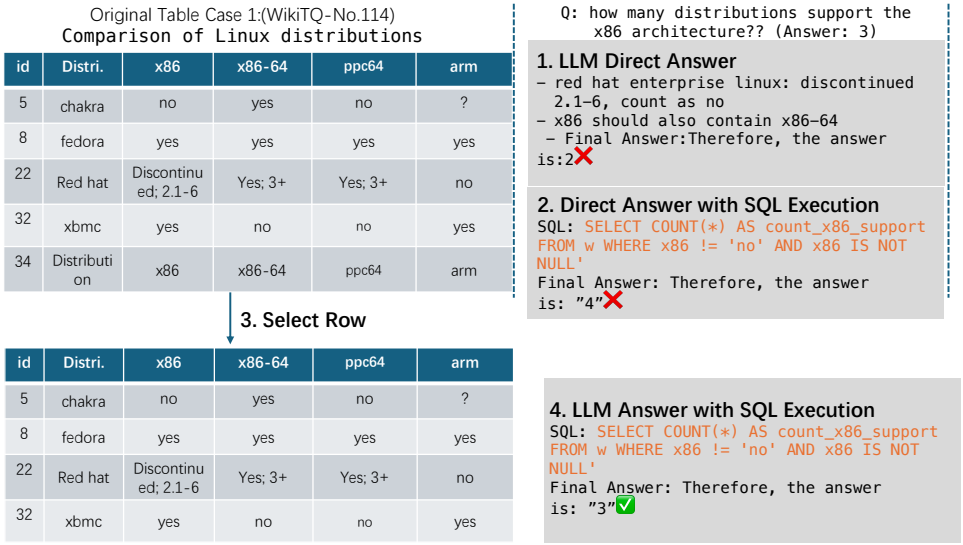


Fig. 12: Failure modes in semantic and categorical TQA. This case asks, "how many distributions support the x86 architecture?" (Correct Answer: 3). The LLM's direct answer incorrectly yields "2" by misinterpreting the context of "discontinued" and confusing "x86" with "x86-64", showcasing its shortage in semantic understanding with irrelevant context. Direct SQL-based answer incorrectly counts "4" due to its inability to filter out last row as header, again demonstrating the limited fault tolerance of symbolic methods for dirty tables.

Instruction for o^{col}

(system message) You are given the table schema, and the table along with the corresponding statement. Write a simple SQLite program for selecting the required columns only, to help answer the question correctly. The SQLite program need not directly answer the question. Assume you always have enough information when executing the SQLite. Fuzzy match data if unsure.
(task description)
1. Plan:
- Identify critical values and ranges from the table related to the statement.
- Make use of your domain knowledge to find the correct approach to solve the question.
- Always select the column with special aggregate values like 'total'.
2. Retrieval:
- Generate a simple SQL program extracting the relevant columns
- SQL: SELECT COLUMNS FROM w;
- Evidence: f_{col}(column names)
(output format)
Response Format: Begin your response with 'Output' and include:
- Plan: Write the plan for column extraction along with a reasoning chain
- Retrieval: Write a simple SQL query
Evidence: f_{col}(column names)
(instruction) Before you return the answer, review your outputs and make sure you have followed all the given instructions.
(input)

```
CREATE TABLE figure skating at the asian winter games(
    row_id int,
    rank int,
    nation text,
    gold int,
    silver int,
    bronze int,
    total int)
/*
SELECT * FROM w;
row_id rank nation gold silver bronze total
0 1 china 13 9 13 35
1 2 japan 7 10 7 24
2  3 uzbekistan 1 2 3 6
3 6 total   24 23 26 73
*/
columns: ['row_id', 'nation', 'gold', 'silver', 'bronze', 'total']
statement: the number of gold and silver medals are equal
```

Fig. 13: Column Selection Prompt o^{col}

Instruction for o^{row}

(system message) You are given the table schema, and the table along with the corresponding statement. Your task is to write an SQLite program to create a subtable to help answer the question correctly. The SQLite program need not answer the question. Try to use fuzzy-match for values if you are not sure about the values.
(task description)
Process:
1. Decompose: Find all the subtasks from the main question.
2. Retrieval: Generate an SQLite program for each subtask and then combine using an 'OR' statement.
(output format)
Response Format: Your response must strictly have the following components:
- Decompose: Divide the question into smaller manageable sub-tasks; one task for each condition
- Retrieval: A basic SQLite program for each condition followed by a combined SQLite
NOTE
- Strictly use the 'OR' statement to when you combine the multiple conditions in the final SQLite program.
Add rows if not present in the response.
Rows: f_row([rows])
(demonstration)

```
<input>
CREATE TABLE 2005 pba draft(
  row\_id int,
  pick int,
  player text,
  country\_of\_origin text)
/*
3 example rows:
SELECT * FROM w LIMIT 3;
row\_id pick player country\_of\_origin
1 2 alex cabagnot united states
2 3 dennis miranda philippines
*/
statement: the player after alex cabagnot is from philippines
<output>
Take a deep breath and think step by step
1. Decompose:
  - Part 1: Identify the row for alex cabagnot
    Response 1: SELECT * FROM w WHERE player = 'alex cabagnot';
  - Part 2: Identify the player after alex cabagnot
    Response 2: SELECT * FROM w WHERE pick = (SELECT pick FROM w WHERE player LIKE
    '%alex cabagnot%') + 1;
  - Part 3: Check if the player after alex cabagnot is from the Philippines
    Response 3: SELECT * FROM w WHERE country\_of\_origin = 'philippines';
2. Retrieval:
   # Using 'OR' to combine the conditions from #1, #2, #3
   SQL: SELECT * FROM w WHERE player = 'alex cabagnot' OR pick = (SELECT pick FROM
   w WHERE player LIKE 'alex cabagnot') + 1 OR country\_of\_origin = 'philippines';
```

(input)

```
CREATE TABLE jeev milkha singh(
 row_id int,
 tournament text,
 events int,
 cuts made int)
/*
SELECT * FROM w LIMIT 3;
row_id tournament  events cuts made
0 masters tournament 3 2
1 us open 4 3
2 the open championship 2 1
*/
columns: ['row_id', 'tournament', 'events', 'cuts made']
statement: the number of cut made exceeds the number of event by 2
```

Fig. 14: Row Selection Prompt o^{row}

Instruction for o^{Ret}

(ROLE) You are an expert in data retrieval strategies for hybrid search systems. Your goal is to rewrite a user's question into a set of three distinct queries, each optimized for a different aspect of a hybrid (dense + sparse) retrieval pipeline.
(CONTEXT)
The user provides an '[Original Query]' and a '[Table Sample]'. The table sample is a small representative extract from a much larger table and will always include a 'row_id' or similar index column. Your task is to use the content and schema of the sample to generate the queries.
(TASK: GENERATE THREE QUERY VARIANTS)
Based on the user's query and the table sample, generate three rewritten queries with increasing specificity:
1. [**GENERAL**] Query (Optimized for Dense/Vector Search):** A descriptive, semantic sentence capturing the user's core intent.
2. [**BALANCED**] Query (Optimized for Hybrid Search):** A concise query blending semantic intent with the most critical column names.
3. [**SPECIFIC**] Query (Optimized for Sparse/Keyword Search like BM25):** A "keyword-stuffed" query that aggressively lists relevant column names and specific cell values from the sample. This should be a space-separated list of terms, not a grammatical sentence.
(Output Format)
You MUST provide the output in the following exact format:
[GENERAL]: Your general, semantic query here
[BALANCED]: Your balanced, semantic + keyword query here
[SPECIFIC]: Your specific, keyword-stuffed query here
(demonstration)

```
<input>
/*
col : rank | cyclist | team
row 0 : alejandro valverde (esp) | caisse d'epargne
row 1 : alexandr kolobnev (rus) | team csc saxo bank
row 2 : davide rebellin (ita) | gerolsteiner
row 3 : paolo bettini (ita) | quick step
row 4 : franco pellizotti (ita) | liquigas
row 5 : denis menchov (rus) | rabobank
row 6 : samuel sánchez (esp) | euskaltel-euskadi
row 7 : stéphane goubert (fra) | ag2r-la mondiale
row 8 : haimar zubeldia (esp) | euskaltel-euskadi
row 9 : david moncoutié (fra) | cofidis
*/
columns: ['rank', 'cyclist', 'team']
Q: which country had the most cyclists finish within the top 10?
</input>
<output>
[GENERAL]: Determine which country is represented by the highest number of cyclists in the top
10 ranking by counting the nationalities mentioned in the cyclist column.
[BALANCED]: Count cyclists by country from the 'cyclist' column to find the most frequent
nationality.
[SPECIFIC]: country count cyclist rank esp rus ita fra alejandro valverde (esp) alexandr
kolobnev (rus) davide rebellin (ita) samuel sánchez (esp) haimar zubeldia (esp)
```

(input)

```
table caption: 2007 New Orleans Saints season
/*
col : game site | result/score
row 0 : rca dome | l 41 \ 10
row 1 : raymond james stadium | l 31 \ 14
row 2 : louisiana superdome | l 31 \ 14
row 4 : louisiana superdome | l 16 \ 13
row 9 : louisiana superdome | l 37 \0 29
*/
columns: ['game site', 'result/score']
Q: what number of games were lost at home?
```

Fig. 15: Rewrite Prompt for o^{Ret}

Instruction for training-free router E_{4B}

(system message) Given a query and a table, determine the necessary operations to answer the query. The order of the operators in the output list represents their importance, with the first operator being the most crucial.
(task description)
The possible operations are:
'Base': No specific operation is needed; the answer can be found by directly observing the table.
'Execute_SQL': Required for complex calculations or aggregations (e.g., COUNT, SUM, AVG, GROUP BY).
'Select_Column': Needed when the query specifically asks for information from a subset of columns.
'Select_Row': Needed when the query requires filtering the table to find specific rows based on certain conditions.
'RAG': Employ a Retrieval-Augmented Generation (RAG) system. This involves rewriting the query and performing a semantic search to find relevant content. This is useful when the query's intent isn't directly translatable to a simple SQL condition and may require external knowledge.
(Output Format) Your output must be a single JSON list. Here is an example template for the output:
["Select_Column", "Select_Row", "Execute_SQL"]
(demonstration)

```
<input>
Q: how many german racers finished the race?
Table Content:
CREATE TABLE 00__German_motorcycle_Grand_Prix (
pos text,
no int,
rider text,
manufacturer text,
laps int,
time_retired text,
grid int,
points int
)
/*
All rows of the table:
SELECT * FROM 00__German_motorcycle_Grand_Prix;
pos  no           rider manufacturer  laps time_retired  grid  points
1   75   mattia pasini       aprilia  27.0   39:44.091   3.0    25.0
2   19 álvaro bautista       aprilia  27.0       +0.01   2.0    20.0
3   52     lukáš pešek         derbi  27.0      +0.111   1.0    16.0
... (33 rows omitted) ...
*/
columns: ['pos', 'no.', 'rider', 'manufacturer', 'laps', 'time/retired', 'grid', 'points']
<output>
["Base", "Select_Row"]
```

(input)

```
Q: which country is represented by the most drivers?
Table Content:
CREATE TABLE _00__Gran_Premio_Telmex (
pos int,
no int,
driver text,
team text,
laps int,
time_retired text,
grid int,
points int
)
/*
All rows of the table:
SELECT * FROM _00__Gran_Premio_Telmex;
pos no             driver                  team  laps time_retired  grid  points
1  1 sébastien bourdais    newman/haas racing    66  1:51:31.146     2      34
2  9     justin wilson               rusport    66     +pt3.528s     1      29
3  5          will power        team australia    66    +pt46.536s     4      26
... (8 rows omitted) ...
*/
columns: ['pos', 'no', 'driver', 'team', 'laps', 'time/retired', 'grid', 'points']
```


Fig. 16: Training-Free Router Prompt E_{4B}

Instruction for o^{SQL}

(system message) Your task is to determine whether you need an SQLite program to solve the question. You must understand the question and identify if it involves any calculations. The final SQLite program must be simple.
(task description)
Final Output: If the question has any mathematical component, then you must write an SQLite program to extract a subtable to answer the question. Otherwise, simply return 'None'
(Output Format) Begin your response with 'Output: ' and always include the following:
- Final Output: A step-by-step reasoning followed by 'None' if SQL is not required otherwise the SQLite program as the solution.
Exclude aggregate values like 'total', 'average', etc from your SQLite programs.
Follow all instructions before returning the answer. Be careful. Think step by step.
(demonstration)

```
<input>
CREATE TABLE asian games (
   row_id int,
   rank int,
   lane int,
   player text)
/*
All rows of the table:
SELECT * FROM w;
row_id   rank  lane  player
0     5  olga tereshkova (kaz)
1     6  manjeet kaur (ind)
2     3  asami tanno (jpn)
*/
columns: ['row_id','rank','lane','player']
Q: which country has the highest number of athletes?
Final Output:
We need to count the occurrences of athletes from each country. Since this involves
mathematical operations, we will use SQLite to solve this.
 - Step 1: Extract the country from the player column, assuming it is enclosed within parentheses.
   SELECT SUBSTR(`player`, INSTR(`player`, '(') + 1, INSTR(`player`, ')') - INSTR(`player`,
   '(') - 1) AS country FROM w;
 - Step 2: Count the occurrences of athletes from each country and find the country with the
 highest count.
   SELECT SUBSTR(`player`, INSTR(`player`, '(') + 1, INSTR(`player`, ')') - INSTR(`player`, '(') - 1)
SQL: SELECT SUBSTR(`player`, INSTR(`player`, '(') + 1, INSTR(`player`, ')') - INSTR(`player`,
'(') - 1) AS country, COUNT(*) AS num_athletes FROM w GROUP BY country ORDER BY num_athletes
DESC LIMIT 1;
```

(input)

```
<input>
CREATE TABLE Fabrice Santoro(
row_id int,
name text,
career\nwin-loss text)
/*
All example rows:
SELECT * FROM w;
row_id   name  career\nwin-loss
0 australian open   22{18
1 indian wells 17{20
2 total 39-38
*/
columns: ['row_id', 'name', 'career\nwin-loss']
statement: how many games did he win in total?
```

Fig. 17: SQL Execute Operator Prompt o^{SQL}

Instruction for o^{Base}

(system message) You are an expert on table data. You must use the table data and the additional evidence to answer the given question.
(task description)
Procedure:
- Divide the main statement into sub-tasks and answer each sub-task
- Based on the answers, check whether the statement is supported by the table
NOTE
Please be extremely careful, pause, make sure all instructions have been followed and only then output the answer.
(Output Format)
Begin your response with 'Output: ' and always include the following:
- Decompose: Divide the main question into sub-tasks and answer each sub-task
- Final Answer: Strictly output as a short phrase starting by 'therefore, the answer is: "AnswerName1", "AnswerName2"...' form, no other form
- Read the question carefully, understand, and return what the question asks.
- Be careful, make sure you have followed all instructions and only then return the output.
(demonstration)

```
<input>
table caption: 2007 New Orleans Saints season
/*
col : game site | result/score
row 0 : rca dome | l 41 / 10
row 1 : raymond james stadium | l 31 / 14
row 2 : louisiana superdome | l 31 / 14
row 4 : louisiana superdome | l 16 / 13
row 9 : louisiana superdome | l 37 / 29
row 10 : reliant stadium | l 23 / 10
row 12 : louisiana superdome | l 27 / 23
row 15 : louisiana superdome | l 38 / 23
row 16 : soldier field | l 33 / 25
*/
columns: ['game site', 'result/score']
Q: what number of games were lost at home?
<output>

Here is an additional evidence to help the answering proces
Additional Evidence:
/*
col : games_lost_at_home
row 0 : 5
*/
Using the table and the additional evidence to answer the question
1. Decompose:
    - #1: From the additional evidence, number of games lost at home = 5
          From the table, counting the occurrences of "louisiana superdome" in the
          "game site" and 'result/score' for loss column = 5
2. Final Answer: Therefore, the answer is: "5"
```

(input)

```
<input>
table caption: Matthew Morrison
/*
col : year | title
row 0 : 2007 | music and lyrics
row 1 : 2007 | dan in real life
row 2 : 2007 | i think i love my wife
*/
columns: ['year', 'title']
Q: what movies other than 'music and lyrics' was morrison involved with in 2007?
```

Fig. 18: Base QA Operator Prompt o^{Base}