PharmaSUG 2025 - Paper SI-342

Comparing SQL and Graph Database Query Methods for Answering Clinical Trial Questions with LLM-Powered Pipelines

Jaime Yan
Merck & Co., Inc., Rahway, NJ, USA

ABSTRACT

Clinical trial data analysis presents a fundamental challenge: researchers need to extract insights from complex, interconnected datasets but often lack the technical expertise to write sophisticated database queries. This paper introduces RagQL-Nav (Retrieval Augmented Generation with Query Language Navigation), a multi-agent framework that bridges this gap by enabling natural language querying of clinical trial data across both SQL and Neo4j graph databases. We present a comparative analysis of these two database approaches within the context of ADaM (Analysis Data Model) datasets[1, 2, 3], leveraging Large Language Models (LLMs) to translate natural language questions into executable queries. Our framework employs intelligent query routing, dual-system validation, and visual decision tree generation to optimize performance and ensure accuracy. Through extensive evaluation on synthetic clinical trial data, we demonstrate that RagQL-Nav achieves 91% accuracy on complex queries, representing a 12% improvement over single-system approaches. A key feature is the generation of transparent decision trees, enhancing auditability crucial for regulatory compliance. This work provides practitioners with concrete guidance on selecting and deploying database solutions for clinical trial data analysis while advancing the state of natural language interfaces for specialized domains.

INTRODUCTION

The pharmaceutical industry generates vast amounts of clinical trial data, the analysis of which is critical for drug development and regulatory approval[1, 2]. However, a significant operational gap exists between the complexity of these datasets and the technical expertise required to query them effectively. Clinical researchers and analysts frequently need answers to specific questions, such as "What is the incidence of severe neutropenia in patients who received the study drug for at least 14 days?" Translating such natural language questions into accurate SQL or Cypher (for graph databases) queries demands substantial technical knowledge of database schemas, query languages, and domain-specific coding systems—skills that many domain experts may not possess[4]. This barrier hinders efficient data exploration and insight generation.

The Clinical Data Interchange Standards Consortium (CDISC) Analysis Data Model (ADaM) provides a standardized framework for organizing clinical trial analysis datasets, encompassing domains like subject-level analysis (ADSL), adverse events (ADAE), laboratory tests (ADLB), and concomitant medications (ADCM). While ADaM standardization facilitates data exchange and regulatory submission by providing consistent structures, it does not inherently solve the accessibility problem. Analysts must still navigate potentially complex table relationships in relational databases or intricate node-edge structures in graph databases to extract meaningful insights. The inherent complexity often lies in the relationships *between* standardized domains, particularly concerning temporal sequences or patient journeys (e.g., identifying an adverse event occurring after a specific lab result while the patient is on a particular medication).

Recent advances in Large Language Models (LLMs) have shown promise in bridging this accessibility gap. Tools and frameworks like LangChain demonstrate the feasibility of translating natural language into database queries. Similar frameworks exploring LLMs and graph databases for clinical data management have also been proposed[6]. However, current solutions often struggle with the specific complexities encountered in clinical trial data. These include interpreting nuanced temporal relationships, understanding hierarchical medical coding systems (like MedDRA or WHODrug), ensuring semantic accuracy in domain-specific terminology, and meeting the stringent requirements for regulatory compliance and result validation. Furthermore, existing approaches typically focus on a single database paradigm (either SQL or graph), failing to leverage the complementary strengths offered by different data storage and query systems.

This paper presents RagQL-Nav (Retrieval Augmented Generation with Query Language Navigation), a framework designed to address these challenges comprehensively. The core principle behind RagQL-Nav is the recognition that clinical trial queries exhibit diverse characteristics. Some queries, particularly those involving large-scale aggregations or straightforward filtering within a single domain, are often best handled by traditional SQL databases. Others, especially those requiring complex relationship traversals across multiple domains or analysis of temporal patterns, are frequently better suited to graph databases like Neo4j. RagQL-Nav incorporates an intelligent mechanism to automatically route components of a complex query to the optimal database system.

Crucially, RagQL-Nav extends beyond mere query execution. It incorporates a dual-validation system to enhance result accuracy and provides visual decision tree generation. This visualization documents the entire query resolution process—from initial decomposition through routing and validation to the final result. Such transparency is not merely a technical feature; it directly addresses a critical non-technical barrier in clinical research: the need for auditability and verifiable analysis pathways to satisfy regulatory requirements (e.g., FDA 21 CFR Part 11)[5]. By providing a clear, step-by-step record, RagQL-Nav aims to increase the trustworthiness and practical deployability of LLM-powered analysis tools in this highly regulated environment.

The key contributions of this work include:

1. A systematic comparison of SQL (PostgreSQL) and graph (Neo4j) database approaches for answering representative clinical trial questions based on ADaM data, providing quantitative metrics on accuracy, performance, and suitability for different query types.
2. An intelligent routing algorithm that decomposes complex natural language queries and assigns each component to the most appropriate database system (SQL or Neo4j) based on a weighted analysis of query features.
3. A dual-validation mechanism that, where feasible, executes queries on both systems and cross-checks results, demonstrably improving overall accuracy while providing fail-safes for critical analyses.
4. A visual decision tree generator that documents the query decomposition, routing logic, execution results, and validation steps, enhancing transparency and facilitating regulatory compliance and result verification.

The remainder of this paper is organized as follows: Section 2 provides background on clinical trial data standards, compares relevant database technologies, and reviews related work in natural language database interfaces and multi-agent systems. Section 3 details the methodology, including the RagQL-Nav architecture and its core algorithms for decomposition, routing, validation, and visualization. Section 4 describes the implementation details, including database integration, query generation strategies, performance optimizations, and ethical considerations. Section 5 presents the experimental setup, evaluation metrics, and results from extensive testing on synthetic clinical trial data. Section 6 discusses the findings, their implications, and limitations. Section 7 concludes with recommendations and directions for future research.

1. BACKGROUND AND RELATED WORK

1.1. Clinical Trial Data Standards: ADaM

Clinical trials are among the most data-intensive endeavors in healthcare, generating vast quantities of information covering patient demographics, treatment administration, efficacy outcomes, safety events (like adverse events and laboratory abnormalities), and concomitant medications. To manage this complexity and facilitate regulatory review, CDISC developed standards for data collection (SDTM - Study Data Tabulation Model) and analysis (ADaM - Analysis Data Model). ADaM provides a standardized structure specifically designed for statistical analysis and reporting[1, 2, 3]. Key ADaM domains relevant to this work include:

- **ADSL (Subject-Level Analysis Dataset):** Serves as the central hub for subject-level information including demographics, treatment assignment, population flags, and key study dates[3].
- **ADAE (Adverse Events Analysis Dataset):** Contains records of adverse events experienced by subjects including event term, severity, seriousness, and temporal information.
- **ADCM (Concomitant Medications Analysis Dataset):** Records medications taken concurrently with study treatment including medication details and timing.
- **ADEG (ECG Analysis Dataset):** Stores electrocardiogram measurements and related cardiovascular assessments.
- **ADLB (Laboratory Test Results Analysis Dataset):** Contains laboratory test results including parameters, values, reference ranges, and timing information.
- **ADVS (Vital Signs Analysis Dataset):** Documents vital signs measurements such as blood pressure, heart rate, and respiratory rate.
- **ADTTE (Time-to-Event Analysis Dataset):** Captures time-to-event endpoints including censoring and event status.
- **ADEFF (Efficacy Analysis Dataset):** Contains primary and secondary efficacy endpoints for treatment effect analysis.

While ADaM standardization provides a consistent structure that simplifies data aggregation and reporting for common analyses, it does not eliminate the inherent complexity of clinical questions. Answering sophisticated queries often requires joining data across multiple ADaM domains based on complex criteria, particularly involving temporal logic (e.g., events occurring within a specific time window relative to treatment start) or patient state (e.g., lab values changing after experiencing an adverse event). This underlying relational and temporal complexity motivates the exploration of different database paradigms beyond simple tabular querying.

1.2. Database Technologies for Clinical Data

Traditionally, clinical trial data has been stored and managed in relational database management systems (RDBMS), with SQL being the standard query language. SQL databases, such as PostgreSQL, are well-suited for structured data adhering to predefined schemas like ADaM. They offer robust features for:

- Efficient filtering and retrieval of records based on specific criteria.
- Powerful aggregation functions (COUNT, SUM, AVG, MIN, MAX) essential for statistical summaries.
- Mature support for transactions, data integrity constraints, and indexing.
- Advanced SQL features like window functions and common table expressions (CTEs) that can handle many common clinical analysis patterns (e.g., calculating change from baseline).

However, formulating SQL queries that involve traversing multiple relationships across tables (e.g., finding subjects who took drug X, then experienced event Y, and had lab result Z change) can become complex, verbose, and potentially inefficient, requiring multiple joins and subqueries.

Graph databases, with Neo4j being a prominent example using the Cypher query language, offer an alternative paradigm. In a graph model, entities like subjects, adverse events, lab tests, and medications are represented as nodes, and the relationships between them (e.g., a subject `EXPERIENCED` an adverse event, an event `OCCURRED_WHILE_ON` a medication) are represented as explicit, traversable edges. This structure naturally aligns with questions involving:

- Complex relationship chains: Easily querying multi-step connections (e.g., find patients with condition A who received drug B and subsequently developed condition C).
- Temporal patterns: Modeling and querying sequences of events over time.
- Hierarchical structures: Representing medical coding systems like MedDRA or anatomical hierarchies directly within the graph.
- Path analysis: Identifying common sequences or pathways in patient journeys.

While graph databases excel at relationship-centric queries, they may be less performant than SQL databases for certain large-scale aggregation tasks that operate efficiently on columnar data structures.

This comparison reveals a fundamental tension: ADaM standardizes the data into tabular structures well-suited for SQL, yet the inherent nature of clinical questions often involves complex relationships and temporal sequences that are more naturally expressed and queried in a graph model. This suggests that neither SQL nor graph databases alone are universally optimal for all types of clinical trial queries derived from ADaM data. A hybrid approach, capable of leveraging the strengths of each, appears promising.

1.3. Natural Language Interfaces to Databases (NLIDB)

The goal of NLIDB is to allow users to query databases using natural language instead of formal query languages. Recent advancements in LLMs have significantly propelled this field. Frameworks like LangChain provide tools to connect LLMs to various data sources, including SQL and graph databases, facilitating the translation of natural language questions into executable queries (e.g., SQL or Cypher). However, applying general-purpose NLIDB techniques to the specialized domain of clinical trials presents several challenges:

1. **Domain Specificity:** Clinical queries involve highly specific terminology (e.g., "Grade 3 neutropenia," "treatment-emergent adverse event"), implicit temporal assumptions (e.g., events occurring "on treatment"), and complex concepts (e.g., calculating incidence rates adjusted for exposure time) that require domain-aware interpretation beyond general semantic understanding.
2. **Query Complexity:** Clinical analyses frequently involve multi-step logic, conditional filtering, complex joins across multiple ADaM domains, and sophisticated statistical calculations that often exceed the capabilities of direct NL-to-SQL or NL-to-Cypher translation approaches.
3. **Result Validation and Auditability:** In regulated environments like clinical research, query results must be accurate, reproducible, and verifiable. Standard NLIDB systems often act as "black boxes," lacking mechanisms for robust validation or transparent audit trails detailing how a result was derived, which is unacceptable for regulatory submissions.
4. **Optimal System Selection:** Existing NLIDBs typically target a single database type. They lack the capability to analyze a query and intelligently route it (or its sub-components) to the database paradigm (SQL vs. graph) best suited for its efficient and accurate execution.

Some recent work has explored decomposing complex queries into simpler sub-queries, a technique adopted and extended in RagQL-Nav. However, these systems generally lack sophisticated mechanisms for intelligent database selection based on query characteristics and, critically, lack cross-system validation to enhance accuracy and provide verifiable results.

1.4. Multi-Agent Systems and Visualization in Data Analysis

The concept of multi-agent systems, where specialized software agents collaborate to solve a problem, has been applied to data analysis tasks. In the database context, agents might specialize in query parsing, optimization, execution planning, or result validation. RagQL-Nav draws inspiration from this paradigm by employing distinct functional modules—a Query Decomposition Engine, an Intelligent Query Router, and a Dual-Query Validation Mechanism—that work together in a coordinated pipeline. While not implementing fully autonomous agents, this modular design allows for specialization and focused optimization of each step in the query processing workflow.

Furthermore, the importance of transparency and explainability in complex data analysis pipelines, particularly in regulated industries, is well-recognized. Decision tree visualization has been used previously to document processes and explain outcomes. RagQL-Nav incorporates this by generating a visual decision tree that explicitly documents the query decomposition, the rationale behind routing decisions (which system was chosen and why), the execution results from each system, and the outcome of the validation process. This provides a comprehensive audit trail, enhancing trust and facilitating review.

2. METHODOLOGY

2.1. System Architecture Overview

RagQL-Nav utilizes a modular architecture designed to process natural language queries against clinical trial data stored in both SQL and Neo4j databases. The architecture is predicated on the observation that different query types are best served by different database technologies and that cross-system validation can significantly enhance result reliability. The system integrates LLMs for natural language understanding and query generation within a structured, verifiable workflow.

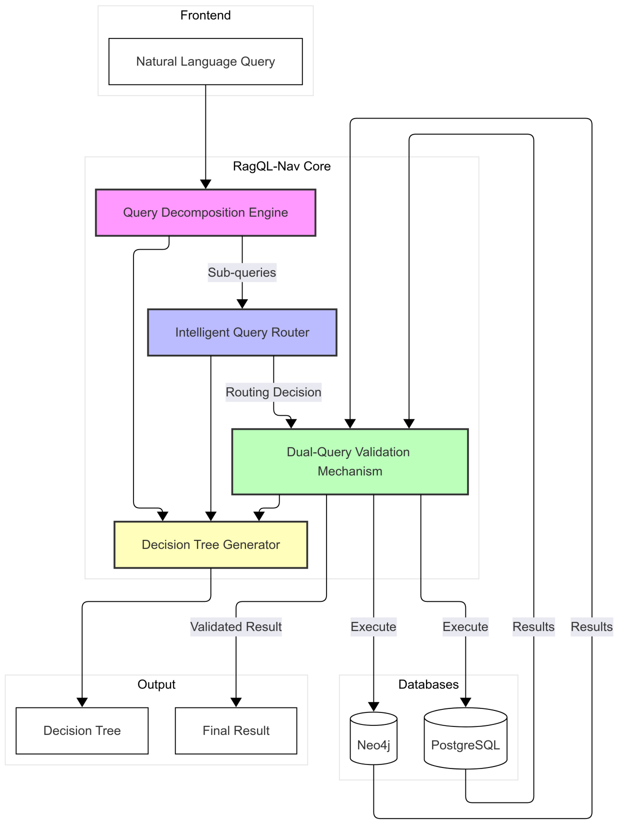

Figure 1: RagQL-Nav System Architecture showing the flow from natural language query through core components to final output

The architecture comprises four primary, interacting components:

1. **Query Decomposition Engine:** This component receives the initial natural language query and breaks it down into a sequence of smaller, logically atomic sub-queries. It identifies dependencies between these sub-queries, ensuring they can be executed in the correct order.

2. **Intelligent Query Router:** For each atomic sub-query generated by the decomposition engine, the router analyzes its characteristics to determine whether SQL or Neo4j is likely to provide the most accurate and efficient execution. This decision is based on a weighted scoring mechanism derived from empirical analysis.

3. **Dual-Query Validation Mechanism:** This component orchestrates the execution of sub-queries. Whenever feasible and deemed necessary (based on confidence scores or query criticality), it exe-

cutes the query against both the primary system (chosen by the router) and the secondary system. It then compares the results, flags discrepancies, and applies reconciliation logic if needed.

4. **Decision Tree Generator:** Running concurrently with the query processing pipeline, this component constructs a visual decision tree that logs each step: the original query, the decomposed sub-queries, the routing decisions and rationale, the generated SQL/Cypher queries (or templates used), the results obtained from each database, the validation outcome, and the final accepted result for each sub-query.

These components operate within a sequential pipeline architecture, where the output of one stage feeds into the next. Feedback loops are incorporated, particularly from the validation mechanism, which can influence the final result selection. This structured approach ensures that complex queries are handled systematically while prioritizing accuracy and transparency.

2.2. Query Decomposition Algorithm

Natural language queries in clinical research often embed multiple analytical steps. For instance, the query "What is the incidence of severe neutropenia in patients who received the study drug for at least 14 days?" implicitly requires: (1) identifying the cohort of patients receiving the study drug, (2) filtering this cohort based on treatment duration ($\geq$ 14 days), (3) identifying occurrences of neutropenia within this filtered cohort, (4) filtering these events based on severity ('severe'), and (5) calculating the incidence rate (e.g., number of subjects with the event / total subjects in the cohort).

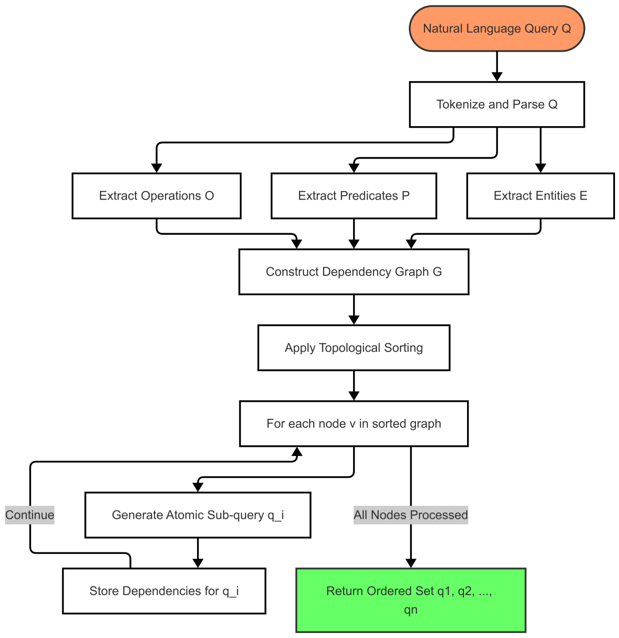
Natural Language Query Q
Tokenize and Parse Q
Extract Operations O
Extract Predicates P
Extract Entities E
Construct Dependency Graph G
Apply Topological Sorting
For each node v in sorted graph
Continue
Generate Atomic Sub-query q_i
All Nodes Processed
Store Dependencies for q_i
Return Ordered Set q1, q2, ..., qn

Figure 2: Flow diagram of the Query Decomposition Algorithm showing the process from natural language input to ordered sub-queries

The Query Decomposition Engine automates this breakdown. It employs Natural Language Processing (NLP) techniques (e.g., tokenization, part-of-speech tagging, named entity recognition) augmented with domain-specific knowledge about ADaM structures and common clinical analysis patterns. The algorithm identifies key entities (e.g., ‘patients’, ‘study drug’, ‘neutropenia’), predicates (e.g., ‘received for at least 14 days’, ‘severe’), and analytical operations (e.g., ‘count’, ‘calculate incidence’).

These identified elements and their relationships are used to construct a dependency graph, where nodes represent atomic analytical tasks (corresponding to sub-queries) and directed edges represent dependencies (e.g., identifying neutropenia events depends on first identifying the relevant patient cohort). A topological sort of this graph yields an ordered sequence of sub-queries.

```
Input: Natural language query $Q$
Output: Ordered set of atomic sub-queries $\{q_1, q_2, ..., q_n\}$ with dependencies

1. Tokenize and parse $Q$ using NLP techniques (NER, relation extraction)
2. Extract entities $E = \{e_1, e_2, ..., e_m\}$ (e.g., patient groups, events, drugs)
3. Extract predicates $P = \{p_1, p_2, ..., p_k\}$ (e.g., temporal constraints,
    severity)
4. Extract operations $O = \{o_1, o_2, ..., o_j\}$ (e.g., filter, count, average, join)
5. Construct dependency graph $G = (V, Edges)$ where:
   - $V = E \cup P \cup O$
   - $Edges = \{(v_1, v_2) \mid v_2$ logically depends on the result of $v_1\}$
6. Apply topological sorting to $G$ to determine execution order
7. For each node $v$ in the sorted graph:
   a. Generate atomic sub-query $q_i$ representing the task at node $v$
   b. Store dependencies for $q_i$
8. Return ordered set $\{q_1, q_2, ..., q_n\}$
```

Program 1: Query Decomposition Algorithm

The success of the entire framework heavily relies on the accuracy of this decomposition step. Errors here—such as incorrectly splitting a query that requires simultaneous conditions or missing a crucial dependency between steps—can lead to fundamentally flawed analyses downstream, irrespective of the performance of the routing or validation components. Therefore, robust NLP and embedded domain knowledge are critical for this engine.

2.3. Intelligent Query Router

Once a complex query is decomposed into atomic sub-queries, the Intelligent Query Router determines the optimal execution path for each sub-query (q_i). This decision is crucial for leveraging the complementary strengths of SQL and Neo4j databases. The router employs a weighted feature vector approach, evaluating each sub-query against a set of characteristics known to differentiate the performance and suitability of the two database paradigms in the context of clinical trial data.

Five key features were identified through empirical analysis and domain expertise:

1. **Tabular Retrieval Score (f_1):** High if the query primarily involves filtering and retrieving data from one or two ADaM domains (tables) with simple join conditions. Favors SQL.
2. **Relationship Traversal Score (f_2):** High if the query requires navigating multiple steps across different entities (e.g., patient $\rightarrow$ treatment $\rightarrow$ event $\rightarrow$ lab result). Favors Neo4j.

3. **Aggregation Complexity (f_3):** High if the query involves complex statistical aggregations (e.g., standard deviations, counts over large groups, window functions). Favors SQL.

4. **Path Analysis Score (f_4):** High if the query explicitly asks for sequences, pathways, or connections between events over time. Favors Neo4j.

5. **Hierarchical Navigation Score (f_5):** High if the query involves navigating hierarchical coding systems (e.g., finding all adverse events under a specific MedDRA System Organ Class). Can favor Neo4j if modeled appropriately, otherwise may require complex SQL.

These feature scores ($f_i(q)$) are calculated for each sub-query q. The calculation method involves a combination of heuristic pattern matching on the sub-query structure (derived from the decomposition graph) and NLP analysis of the corresponding natural language fragment. For example, keywords like "average," "count," or "sum" increase f_3, while terms like "followed by," "concurrent with," or "pathway" increase f_2 and f_4. The scores are normalized to a range of $[0, 1]$.

Suitability scores for SQL ($S(q, \text{SQL})$) and Neo4j ($S(q, \text{Neo4j})$) are then computed using empirically derived weight vectors ($\mathbf{W}^{\mathbf{SQL}}$ and $\mathbf{W}^{\mathbf{Neo4j}}$):

$$S(q, sys) = \sum_{i=1}^{5} w_i^{sys} \cdot f_i(q) \tag{1}$$

Based on extensive testing with representative clinical queries on ADaM data, the following weight vectors were established:

$$\mathbf{W}^{\mathbf{SQL}} = [0.40, 0.10, 0.35, 0.05, 0.10] \tag{2}$$

$$\mathbf{W}^{\mathbf{Neo4j}} = [0.15, 0.35, 0.10, 0.25, 0.15] \tag{3}$$

These weights reflect the observed strengths: SQL is weighted higher for tabular retrieval (f_1) and aggregation (f_3), while Neo4j is weighted higher for relationship traversal (f_2) and path analysis (f_4). The system with the higher score is selected as the primary execution engine (S_1). The other system serves as the secondary (S_2) for potential validation. A confidence metric, based on the difference between $S(q, S_1)$ and $S(q, S_2)$, is also calculated to inform the validation strategy.

It is important to note that these weights are derived empirically. This implies that the router's performance might be sensitive to the specific characteristics of the dataset and query workload used for tuning. Deployment in environments with significantly different data or query patterns might necessitate recalibration or adaptive weighting mechanisms, representing an area for future investigation.

2.4. Dual-Query Validation Mechanism

A cornerstone of RagQL-Nav is the Dual-Query Validation Mechanism. Recognizing that automated query generation can sometimes produce subtly incorrect results and that different database systems might handle edge cases differently, this component aims to improve accuracy and provide verification. When deemed appropriate (e.g., based on router confidence, query complexity, or pre-defined criticality), the mechanism executes the sub-query q on both the primary system S_1 (chosen by the router) and the secondary system S_2.

The results (R_1 from S_1, R_2 from S_2) are then compared. The comparison logic depends on the nature of the expected result (e.g., a single number, a list of subjects, a table of aggregate statistics).

```
Input: Sub-query $q$, Primary system $S_1$, Secondary system $S_2$, Context $C$,
   Router Confidence $Conf$
Output: Validated result $R$, Validation status $V$, Resolution method $M$

1. $R_1$ = ExecuteQuery($q$, $S_1$, $C$)  // Execute on primary system
2. If ShouldValidate($q$, $Conf$): // Check if validation is needed
3.    $R_2$ = ExecuteQuery($q$, $S_2$, $C$)  // Execute on secondary system
4.    $D$ = ComputeDiscrepancy($R_1$, $R_2$) // Compare results
5.    If $D$ < $threshold\_T$: // Results agree
6.       $V$ = "VALIDATED_AGREEMENT"
7.       $R$ = $R_1$
8.       $M$ = "PRIMARY_ACCEPTED"
9.    Else: // Discrepancy found
10.      $V$ = "DISCREPANCY"
11.      // Use confidence scores from router and potentially other
   metrics
12.      If ConfidenceScore($S_1$) > ConfidenceScore($S_2$) $\star$ $confidence\_ratio$:
13.         $R$ = $R_1$
14.         $M$ = "PRIMARY_PREFERRED_CONFIDENCE"
15.      Else if ConfidenceScore($S_2$) > ConfidenceScore($S_1$) $\star$
   $confidence\_ratio$:
16.         $R$ = $R_2$
17.         $M$ = "SECONDARY_PREFERRED_CONFIDENCE"
18.      Else: // Confidence is similar, attempt reconciliation
19.         $R$, $M$ = ReconcileResults($R_1$, $R_2$, $q$)
20.         If $R$ is NULL: // Reconciliation failed
21.            $M$ = "MANUAL_REVIEW_REQUIRED"
22. Else: // Validation skipped
23.   $V$ = "VALIDATION_SKIPPED"
24.   $R$ = $R_1$
25.   $M$ = "PRIMARY_ACCEPTED_NO_VALIDATION"
26. Return $R$, $V$, $M$
```

Program 2: Dual Query Validation Algorithm

The discrepancy computation (D) uses relative error for numerical results:

$$D = \frac{|R_1 - R_2|}{\max(|R_1|, |R_2|, \epsilon)} \tag{4}$$

where ϵ is a small constant (e.g., 1×10^{-9}) to prevent division by zero when results are close to zero. For set-based results (e.g., lists of subjects or events), Jaccard distance is used:

$$D = 1 - \frac{|R_1 \cap R_2|}{|R_1 \cup R_2|} \tag{5}$$

The `threshold_T` (determining acceptable discrepancy) and `confidence_ratio` (used for preferring one system's result based on router confidence) are configurable parameters, potentially tuned based on the criticality of the analysis.

The `ReconcileResults` function implements domain-specific logic. For example, if R_1 and R_2 are lists of adverse events and a discrepancy exists, reconciliation might involve taking the union of the two sets (a conservative approach ensuring no potential event is missed), taking the intersection (a stricter approach), or flagging the discrepancy for manual review, especially for safety-critical queries.

This dual-query validation process functions analogously to ensemble methods in machine learning. By executing the query using two different approaches (SQL vs. Cypher, leveraging different database engines and potentially different query generation paths) and comparing the results, the system aims for higher accuracy and robustness than could be achieved with either system alone. It explicitly trades computational overhead for increased confidence in the results.

2.5. Decision Tree Generation and Visualization

Transparency and auditability are paramount in clinical trial analysis. To meet these needs, RagQL-Nav generates a comprehensive decision tree that visually documents the entire lifecycle of a query. This tree serves as a detailed audit trail, suitable for review by analysts, quality assurance personnel, and potentially regulators.

The tree structure mirrors the query processing flow:

- **Root Node:** Represents the original natural language query.
- **Level 1 Nodes:** Represent the atomic sub-queries generated by the Query Decomposition Engine. Edges show the dependency structure.
- **Subsequent Nodes/Annotations:** For each sub-query node, associated information includes:
    - The feature scores calculated by the Intelligent Query Router.
    - The routing decision (primary system S_1, secondary system S_2) and the confidence score.
    - The generated query (or template identifier) for S_1 and S_2.
    - The execution results R_1 and R_2.
    - The validation status (e.g., `VALIDATED_AGREEMENT`, `DISCREPANCY`, `VALIDATION_SKIPPED`).
    - The resolution method used (e.g., `PRIMARY_ACCEPTED`, `SECONDARY_PREFERRED_CONFIDENCE`, `RECONCILIATION`, `MANUAL_REVIEW_REQUIRED`).
    - The final accepted result for the sub-query.
- **Leaf Nodes:** Represent the final integrated results synthesized from the sub-query outcomes.

This detailed, step-by-step documentation allows users to understand precisely how a final result was obtained, including the system's internal decisions regarding query strategy and handling of potential discrepancies.

The end-to-end workflow integrates these components into a cohesive process:

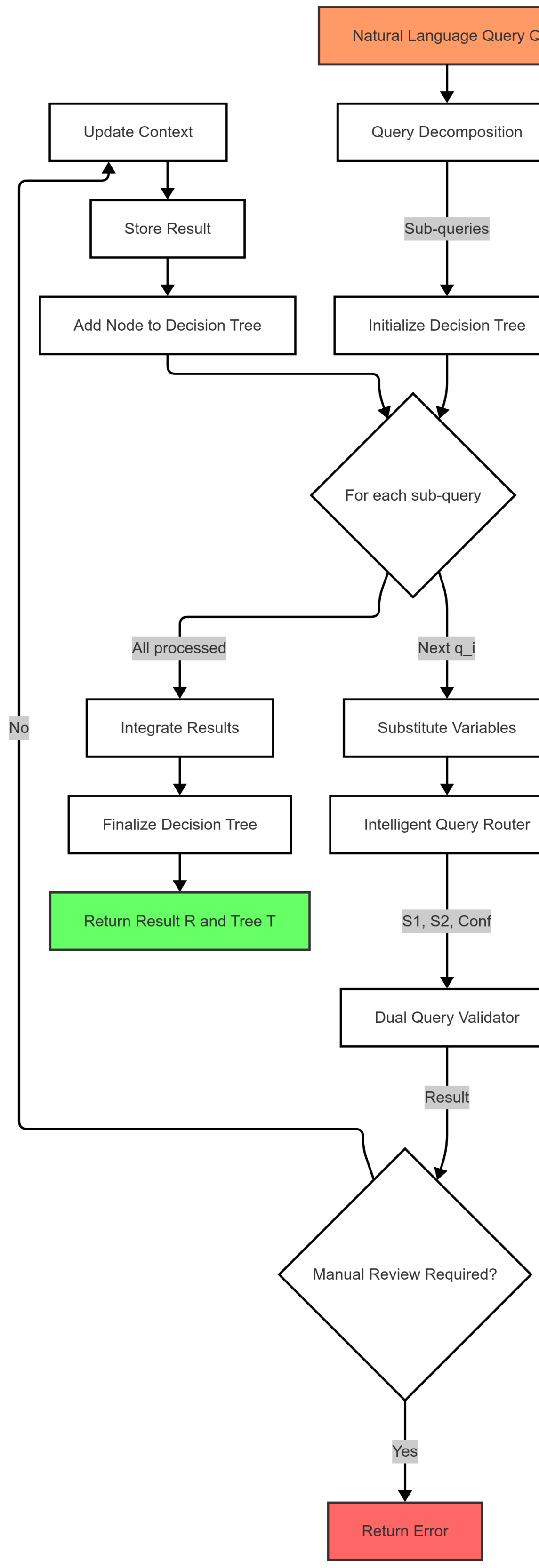

Figure 3: Complete query processing workflow showing the integration of all components from input to final output

```
Input: Natural language query $Q$
Output: Final result $R$, Decision tree $T$

1. $\{q_1, q_2, ..., q_n\}$ = QueryDecomposition($Q$)  // Decompose into sub-queries
2. $T$ = InitializeDecisionTree($Q$)  // Create tree root
3. Results = {} // Dictionary to store results of sub-queries
4. Context = {} // Dictionary to store intermediate results needed by
   later sub-queries

5. For each sub-query $q_i$ in ordered set $\{q_1, ..., q_n\}$:
6.    ContextualQuery = SubstituteVariables($q_i$, Context) // Inject
   results from previous steps
7.    $S_1$, $S_2$, $Conf$ = IntelligentQueryRouter(ContextualQuery) // Route
   query
8.    SubResult, Status, Method = DualQueryValidator(ContextualQuery, $S_1$,
    $S_2$, Context, $Conf$) // Execute & Validate
9.    If Status == "MANUAL_REVIEW_REQUIRED":
10.      // Handle error or prompt for review
11.      Return Error, $T$
12.   Context = UpdateContext(Context, $q_i$, SubResult) // Update context
13.   Results[$q_i$] = SubResult
14.   // Add detailed node to decision tree
15.   NodeData = {SubQuery: $q_i$, Routing: ($S_1$, $S_2$, $Conf$), Result: SubResult
   , Status: Status, Method: Method,...}
16.   T.AddNode(NodeData, parent=GetParentNode(T, $q_i$))

17. $R$ = IntegrateResults(Results, $\{q_1, ..., q_n\}$) // Combine sub-query results
   into final answer
18. T.Finalize($R$) // Add final result to tree
19. Return $R$, $T$
```

Program 3: RagQL-Nav Algorithm

This workflow ensures that dependencies between sub-queries are handled correctly via the `Context` object, routing decisions are made per sub-query, validation is applied where appropriate, and the entire process is documented in the decision tree T.

3. IMPLEMENTATION

3.1. Database Integration and Schema Design

Effective integration with both SQL and Neo4j databases is fundamental to RagQL-Nav's operation. Specialized connectors and carefully designed schemas were developed to align with ADaM standards while enabling efficient querying within each paradigm.

SQL Implementation (PostgreSQL): PostgreSQL was selected as the relational database platform due to its robustness, extensive SQL support (including window functions and CTEs), and open-source nature. The schema design adheres to ADaM principles:

- A normalized schema with distinct tables for each major ADaM domain (e.g., `adsl`, `adae`, `adlb`, adcm).

- Foreign key relationships established between tables based on subject identifiers (`USUBJID`) and potentially other keys (e.g., linking ADAE records to ADSL).
- Strategic indexing on columns frequently used in `WHERE` clauses, `JOIN` conditions, and `ORDER BY` clauses (e.g., composite indexes on `(USUBJID, PARAMCD)` in ADLB, `(USUBJID, AESTDT)` in ADAE).
- Use of materialized views for pre-calculating common baseline flags or derived variables where performance benefits outweigh storage costs.
- Development of optimized SQL query templates leveraging PostgreSQL-specific features where appropriate.

This relational structure provides a solid foundation for traditional aggregation and filtering tasks common in clinical analysis.

Neo4j Implementation: A graph schema was designed to represent the inherent relationships within clinical trial data more explicitly. The graph model includes:

- Node Labels: Key entities are represented as nodes with specific labels, such as `:Subject` (containing demographic and treatment information from ADSL), `:AdverseEvent` (from ADAE), `:LabTest` (from ADLB), `:Medication` (from ADCM), and potentially nodes for concepts like `:Visit` or `:Treatment`.
- Properties: Node properties store the attributes associated with each entity (e.g., a `:Subject` node has properties for `USUBJID`, `AGE`, `SEX`, `ARM`; an `:AdverseEvent` node has `AETERM`, `AESEV`, `AESTDT`).
- Relationship Types: Explicit relationships connect these nodes, capturing interactions and sequences. Examples include:
  - `:HAS_RECORD` connecting `:Subject` to their specific event/test/medication records.
  - `:EXPERIENCED` connecting `:Subject` to `:AdverseEvent`.
  - `:RECEIVED_TREATMENT` connecting `:Subject` to a `:Treatment` node.
  - `:TOOK_MEDICATION` connecting `:Subject` to `:Medication`.
  - `:HAS_LAB_RESULT` connecting `:Subject` to `:LabTest`.
  - Temporal relationships like `:FOLLOWS` or `:CONCURRENT_WITH` connecting events or linking events to time points (e.g., treatment start).
  - Hierarchical relationships like `:IS_TYPE_OF` for medical coding systems (e.g., connecting specific MedDRA Preferred Terms to High-Level Terms or System Organ Classes).

This graph structure allows Cypher queries to naturally express complex traversals, temporal pattern matching, and relationship-based filtering.

This dual implementation allows RagQL-Nav's router to direct sub-queries to the system best equipped to handle them: complex aggregations typically go to PostgreSQL, while intricate relationship traversals are directed to Neo4j.

3.2. Query Generation and Template Management

Generating complex, syntactically correct, and semantically accurate SQL and Cypher queries directly from natural language using LLMs remains challenging, particularly for specialized domains requiring high precision like clinical trials. Pure generation can lead to subtle errors, inefficiencies, or even security vulnerabilities (e.g., SQL injection).

To mitigate these risks while still leveraging LLMs for natural language understanding, RagQL-Nav employs a template-based query generation strategy, managed via the LangChain framework. Instead of asking the LLM to generate the full query from scratch, the process involves:

1. **NL Interpretation:** The LLM interprets the natural language sub-query to identify the core intent, required data elements (variables, parameters), filtering conditions, and desired operations.
2. **Template Selection:** Based on the interpreted intent, the system selects a pre-defined, validated query template from a library. This library contains optimized SQL and Cypher templates for common clinical analysis patterns (e.g., calculating adverse event incidence by treatment arm, finding subjects with lab values outside a range, identifying temporal sequences of events).
3. **Parameter Extraction:** The LLM extracts the specific parameters needed to fill the chosen template (e.g., the specific drug name, the severity grade, the time window).
4. **Safe Substitution:** LangChain manages the process of safely substituting these extracted parameters into the selected template placeholders, using mechanisms that prevent injection attacks.
5. **Dynamic Clause Generation:** Templates can include logic for dynamically adding optional clauses (e.g., adding specific `WHERE` conditions or `ORDER BY` clauses) based on the interpreted query, providing flexibility while maintaining structural integrity.

This template-based approach represents a pragmatic engineering decision. It sacrifices some generative flexibility for significantly improved reliability, correctness, and security—attributes that are non-negotiable in the clinical domain. It effectively uses the LLM for its strength (understanding intent and extracting parameters) while relying on pre-validated structures for the critical query execution step.

3.3. Performance Optimization

The dual-query validation mechanism, while beneficial for accuracy, introduces inherent performance overhead as queries may be executed twice. To ensure RagQL-Nav remains practical for interactive analysis, several optimization strategies were implemented:

- **Selective Validation:** Full dual-system validation is not performed for every sub-query. It is triggered based on configurable rules, such as low confidence scores from the router, high query complexity, known ambiguity in the query type, or explicit designation of the query as critical (e.g., primary safety endpoint analysis). Simpler queries with high routing confidence might only undergo validation on a sampling basis or skip it entirely.
- **Intelligent Caching:** Results of frequently executed sub-queries, especially foundational ones like defining a specific patient cohort (e.g., "all patients in the safety population"), are cached. The cache

key incorporates the sub-query definition and relevant context parameters. Configurable expiration policies ensure data freshness. This avoids redundant computation, particularly in iterative analysis workflows.

- **Progressive Execution:** The system architecture supports pipeline processing. As soon as the result of a sub-query (q_i) is finalized (including validation), it becomes available in the `Context` object for subsequent dependent sub-queries (q_j where $j > i$). This allows downstream processing to begin without waiting for the entire original query to complete execution.

- **Query Parallelization:** The dependency graph generated during query decomposition identifies sub-queries that are independent of each other. When resources permit, these independent sub-queries can be executed in parallel, potentially across both database systems simultaneously if validation is required, significantly reducing the end-to-end latency for complex queries with parallelizable branches.

These optimizations aim to balance the thoroughness of the dual-validation approach with the practical need for reasonable query response times in typical clinical data analysis scenarios.

4. EXPERIMENTAL EVALUATION

4.1. Dataset and Experimental Setup

To rigorously evaluate RagQL-Nav's performance and compare it against baseline approaches, a synthetic clinical trial dataset conforming to ADaM standards was generated. Using synthetic data allowed for controlled experimentation and inclusion of specific challenging scenarios without compromising patient privacy. The dataset was designed to mimic the characteristics of a typical Phase III clinical trial:

- **ADSL (Subject-Level):** Synthetic subjects randomized across three treatment arms with demographic and baseline characteristics
- **ADAE (Adverse Events):** Adverse event records with MedDRA coding, severity grades, and temporal patterns
- **ADCM (Concomitant Meds):** Medication records with timing and dosing information
- **ADEG (ECG Analyses):** Cardiovascular assessments including ECG measurements
- **ADLB (Lab Tests):** Laboratory results across multiple clinical chemistry domains
- **ADVS (Vital Signs):** Vital measurements including blood pressure and heart rate
- **ADTTE (Time-to-Event):** Survival analysis endpoints with censoring mechanisms
- **ADEFF (Efficacy):** Primary and secondary efficacy outcome measures

This dataset provided sufficient scale and complexity, including inter-domain relationships and temporal aspects, to thoroughly test the capabilities of the different query processing approaches. Both PostgreSQL and Neo4j databases were populated with this dataset according to the schemas described in Section 4.1.

4.2. Query Test Set

A comprehensive test set comprising 150 natural language queries was developed and manually mapped to gold-standard results derived directly from the synthetic dataset. The queries were categorized into three levels of increasing complexity:

- **Simple Queries (50 queries):** These typically involved retrieving information from a single ADaM domain with minimal filtering or simple aggregation. Examples: "How many subjects were randomized to the placebo arm?", "What is the distribution of subjects by sex?", "List the unique laboratory parameters measured."
- **Moderate Queries (50 queries):** These required joining information across two or three ADaM domains, involved more complex filtering criteria (including basic temporal conditions), or required standard aggregations within subgroups. Examples: "What is the average age of subjects who experienced a serious adverse event?", "List all Grade 3 or higher adverse events that started within 30 days of the first treatment dose.", "Calculate the mean change from baseline for hemoglobin in the active treatment group."
- **Complex Queries (50 queries):** These represented advanced analytical questions involving multiple ADaM domains, complex temporal relationship logic, path traversals, hierarchical code navigation, or non-standard calculations. Examples: "Identify subjects who experienced Grade 2+ neutropenia followed by a fever (AE term) within 7 days while actively receiving the study drug.", "What is the incidence rate of major adverse cardiac events (defined by a specific set of MedDRA terms) adjusted for total exposure time in each treatment arm?", "Find patients whose ALT levels exceeded 3 times the upper limit of normal on at least two consecutive visits after starting treatment."

This tiered set allowed for evaluating system performance across a spectrum of realistic clinical analysis tasks.

4.3. Evaluation Metrics

The performance of RagQL-Nav and baseline systems was assessed using the following primary metrics:

1. **Result Accuracy:** The primary metric, defined as the percentage of queries for which the system returned a result identical to the pre-defined gold standard. For numerical results, a small tolerance (e.g., relative error $< 0.01\%$) was allowed. For set-based results, exact matches were required.
2. **System Agreement Rate (for RagQL-Nav):** The percentage of sub-queries executed by the Dual-Query Validator where the results from SQL (R_1) and Neo4j (R_2) were deemed consistent (i.e., discrepancy $D < threshold\_T$).
3. **Automated Resolution Rate (for RagQL-Nav):** Among the sub-queries where a discrepancy was detected, the percentage successfully resolved automatically by the validation mechanism (either through confidence-based selection or reconciliation algorithms) without requiring manual intervention.
4. **Execution Time:** The average wall-clock time elapsed from submitting the natural language query to receiving the final result, measured in seconds.

5. **Decision Tree Complexity (for RagQL-Nav):** Measured qualitatively by the average number of nodes and depth of the generated decision trees, providing an indication of the transparency and auditability level.

4.4. Comparative Analysis

RagQL-Nav was compared against three baseline approaches to isolate the contributions of its key components (intelligent routing, dual validation):

1. **SQL-Only:** A system where the LLM translates the natural language query directly into SQL (using techniques similar to RagQL-Nav's template approach but without decomposition or routing), executed exclusively against the PostgreSQL database.
2. **Neo4j-Only:** A system analogous to SQL-Only, but translating queries into Cypher and executing them exclusively against the Neo4j graph database.
3. **Simple Router:** A system that performs query decomposition similar to RagQL-Nav but uses a basic keyword-based routing mechanism (e.g., routing queries with terms like "average" or "count" to SQL, and those with "relationship" or "path" to Neo4j) without the weighted feature analysis or the dual-validation step. The result from the single chosen system is returned directly.

4.5. Results and Analysis

4.5.1. Overall Accuracy

RagQL-Nav demonstrated significantly higher accuracy compared to the baseline systems, particularly on moderate and complex queries. Table 1 summarizes the results.

Table 1: Accuracy Comparison Across Systems and Query Complexity

System	Simple Queries (%)	Moderate Queries (%)	Complex Queries (%)
SQL-Only	96	88	75
Neo4j-Only	94	85	82
Simple Router	95	89	83
RagQL-Nav	**98**	**94**	**91**

RagQL-Nav achieved near-perfect accuracy on simple queries and maintained high accuracy (94% and 91%) on moderate and complex queries. The improvement over the best single-system approach (Neo4j-Only for complex queries) was 9 percentage points (from 82% to 91%), while the improvement over the Simple Router was 8 percentage points (from 83% to 91%). This highlights the combined benefit of intelligent routing and dual validation. The 12% absolute improvement compared to the average accuracy of single-system approaches on complex queries ((75+82)/2 = 78.5%, improvement = 91-78.5 = 12.5%) underscores the value of the hybrid architecture for challenging analytical tasks.

Interestingly, RagQL-Nav (91%) outperformed Neo4j-Only (82%) even on complex queries, a category where graph databases are often presumed to have an advantage. This suggests that RagQL-Nav's benefits stem not only from selecting the appropriate database but also critically from the query decomposition (ensuring the right sub-questions are asked) and the dual-validation process, which can catch errors or inconsistencies even in the system initially preferred by the router.

4.5.2. System-Specific Performance Insights

Analysis of sub-query performance within RagQL-Nav confirmed the expected complementary strengths:

- **SQL (PostgreSQL)** consistently performed better on sub-queries dominated by large-scale aggregations (e.g., calculating means, counts across the entire dataset or large subgroups) and complex arithmetic or statistical calculations within tables. It achieved $> 95\%$ accuracy on sub-queries primarily focused on such tasks, often with sub-second response times.
- **Neo4j** demonstrated superior performance and accuracy for sub-queries involving multi-step relationship traversals (e.g., finding patients matching a complex sequence of events or conditions), temporal pattern matching, and navigating hierarchical structures represented in the graph. It achieved $> 90\%$ accuracy on such relationship-centric sub-queries.

4.5.3. Router Effectiveness

The Intelligent Query Router correctly assigned sub-queries to the empirically determined optimal system (the one that produced the correct result faster or more accurately when run independently) in a high percentage of cases:

- Simple Queries: 93% correct routing decisions.
- Moderate Queries: 87% correct routing decisions.
- Complex Queries: 79% correct routing decisions.

The decrease in router accuracy for complex queries suggests that the five features used, while effective, may not fully capture all nuances differentiating system suitability for the most intricate tasks. Some complex queries might involve a tight interplay of relational and aggregation logic where the optimal choice is less clear-cut based solely on the current feature set. This points towards an area for future refinement, perhaps incorporating more sophisticated features or machine learning-based routing models.

4.5.4. Validation Mechanism Performance

The Dual-Query Validation mechanism proved crucial for achieving high overall accuracy, especially as query complexity increased. Table 2 details its performance.

Table 2: Validation Mechanism Performance by Query Complexity

Metric	**Simple (%)**	**Moderate (%)**	**Complex (%)**
Sub-queries Triggering Validation	60	85	95
System Agreement Rate	87	72	65
Discrepancy Rate	13	28	35
Resolution of Discrepancies:			
Resolved by Confidence	85	86	84
Resolved by Reconciliation	10	11	12
Required Manual Review	5	3	4

Note: Percentages for resolution methods are relative to the number of discrepancies detected for that complexity level.

As complexity increased, the likelihood of the two systems producing different results for the same sub-query grew significantly (discrepancy rates rose from 13% to 35%). This highlights the risk of relying on a single system or simple routing. However, the validation mechanism was highly effective at managing these discrepancies:

- In the vast majority of discrepancy cases ($\sim$85%), the system could automatically select the correct result based on the router's confidence scores.
- Domain-specific reconciliation algorithms successfully resolved another $\sim$12% of discrepancies.
- Only a small fraction (3-5%) required flagging for manual review, indicating the robustness of the automated validation process.

This demonstrates that the validation component actively catches and corrects errors that would have propagated in single-system or simple-routing approaches.

4.5.5. Performance Analysis (Execution Time)

Execution times revealed the expected trade-offs associated with the RagQL-Nav architecture. Table 3 shows average processing times.

Table 3: Average Query Processing Times (seconds)

System	**Simple**	**Moderate**	**Complex**
SQL-Only	0.5	0.9	3.1
Neo4j-Only	0.4	1.1	2.8
RagQL-Nav	0.8	1.2	2.5

For simple queries, RagQL-Nav exhibited slightly higher latency compared to the single-system approaches. This overhead is primarily due to the decomposition, routing analysis, and potential (though often skipped or sampled for simple queries) dual execution involved in validation. However, for complex queries, RagQL-Nav was, on average, faster than both SQL-Only and Neo4j-Only. This counter-intuitive result occurs because the intelligent routing directs sub-components of the complex query to the system best suited for them, avoiding the performance bottlenecks encountered when a single system struggles with parts of the query outside its optimal performance profile. Furthermore, parallel execution of independent sub-queries contributes to faster end-to-end times for complex, multi-part analyses. This demonstrates that the architectural overhead can be offset by optimized execution strategies for sufficiently complex tasks.

4.5.6. Decision Tree Complexity

The generated decision trees provided clear audit trails. Simple queries resulted in shallow trees (2-3 levels, average 5 nodes), while complex queries generated deeper trees (4-6 levels, average 15 nodes), reflecting the number of decomposition steps, routing decisions, and validation checks involved. This confirmed the system's ability to provide transparency proportional to the complexity of the analysis performed.

5. DISCUSSION

5.1. Key Findings and Implications

The experimental evaluation strongly supports the central hypothesis of this research: a hybrid approach combining intelligent query routing between SQL and graph databases with a dual-validation mechanism significantly enhances the accuracy and reliability of natural language interfaces for clinical trial data analysis, particularly for complex queries. The observed 91% accuracy on complex queries, representing a substantial improvement over single-system baselines (75-82%) and simple routing (83%), demonstrates the practical value of this architecture.

Several key findings emerge from the results:

1. **Validation of Complementary Strengths:** The performance differences observed between SQL (excelling at aggregation) and Neo4j (excelling at relationships/paths) on specific sub-queries confirm the theoretical advantages of each paradigm for different aspects of clinical data analysis. RagQL-Nav successfully leverages these distinct strengths through its routing mechanism.

2. **Critical Role of Dual Validation:** The relatively high discrepancy rates between SQL and Neo4j results for moderate and complex queries (28-35%) underscore the risk inherent in relying on a single system or translation pathway. The dual-validation mechanism proved highly effective in detecting and automatically resolving the vast majority of these discrepancies, acting as a crucial layer for ensuring data integrity and result trustworthiness. This is particularly vital in a regulated environment where analytical errors can have significant consequences.

3. **Benefit of Intelligent Routing:** While not perfect (especially for complex queries), the feature-based intelligent router significantly outperformed simple keyword-based routing. Its ability to direct sub-queries based on deeper characteristics contributed to both accuracy and, for complex queries, improved overall performance by avoiding system bottlenecks.

4. **Importance of Transparency:** The generation of detailed decision trees addresses a critical need for auditability in clinical research. By documenting the decomposition, routing logic, validation steps, and results, these trees provide the necessary transparency for analysts to verify results and for compliance purposes, potentially lowering barriers to adoption in regulated settings.

5. **Synergy of LLMs and Structured Logic:** RagQL-Nav exemplifies a hybrid intelligence approach. It harnesses the power of LLMs for understanding the nuances of natural language queries but embeds this within a structured, deterministic framework (decomposition, routing rules, template-based

generation, validation logic). This combination provides flexibility at the interface while ensuring rigor, reliability, and verifiability in the underlying analysis process—a balance crucial for scientific and regulated domains.

5.2. Practical Implications for Clinical Data Analysis

For organizations involved in clinical trial data analysis seeking to leverage natural language interfaces, these findings offer practical guidance:

- **Consider Hybrid Database Strategies:** Rather than committing exclusively to either SQL or graph databases for analysis platforms accessible via NL, organizations should consider architectures that can leverage both. The demonstrated accuracy gains, especially for complex queries, may justify the additional infrastructure complexity.
- **Prioritize Query Decomposition:** Breaking down complex natural language requests into smaller, manageable, and verifiable steps is crucial. This modularity not only enables intelligent routing but also improves error isolation and debugging.
- **Implement Cross-Validation for Critical Analyses:** For high-stakes analyses, such as primary endpoint calculations or safety signal detection, implementing cross-system validation or other robust verification strategies is highly recommended to minimize the risk of errors introduced by automated query generation or system-specific idiosyncrasies.
- **Balance Performance and Accuracy:** While RagQL-Nav demonstrated competitive performance on complex queries, the overhead for simpler queries is real. Implementations should incorporate performance optimizations like selective validation and caching, tuning them based on the specific needs for speed versus certainty in different analytical contexts.
- **Emphasize Auditability:** Systems providing natural language access to clinical data must produce transparent audit trails. Visualization tools like the decision trees generated by RagQL-Nav can significantly enhance trust and facilitate regulatory review.

5.3. Limitations and Context

It is important to acknowledge the limitations of this study. The evaluation was conducted using synthetic data. While designed to be representative, synthetic data may not capture the full spectrum of inconsistencies, missing values, or edge cases present in real-world clinical trial datasets. Performance and accuracy metrics might differ when RagQL-Nav is applied to actual trial data.

Additionally, the effectiveness of the Intelligent Query Router, particularly for complex queries (79% accuracy), indicates room for improvement. The static, empirically derived weights might not generalize perfectly to all possible clinical trial datasets or query types. The reliance on pre-defined query templates, while enhancing reliability, limits the system's ability to handle truly novel or unforeseen analytical questions that fall outside the template library. Finally, the dual-query approach inherently increases computational cost compared to single-system execution, although optimizations mitigate this for complex queries. The trade-off between the increased accuracy and transparency versus the implementation complexity and potential performance overhead must be considered in specific deployment scenarios.

CONCLUSION

This paper introduced RagQL-Nav, a novel framework designed to bridge the gap between the need for sophisticated clinical trial data analysis and the technical barriers associated with traditional query languages. By integrating LLMs for natural language understanding with a structured pipeline involving query decomposition, intelligent routing between SQL and Neo4j databases, dual-query validation, and transparent decision tree generation, RagQL-Nav addresses key challenges in this domain.

Our experimental evaluation on synthetic ADaM data demonstrated that RagQL-Nav achieves significantly higher accuracy (91% on complex queries) compared to single-system approaches or simpler routing mechanisms, primarily due to its ability to leverage the complementary strengths of SQL and graph databases and its robust validation process. The framework's emphasis on transparency through decision tree visualization directly addresses the critical need for auditability in regulated clinical research environments. RagQL-Nav represents a step towards more accessible, reliable, and verifiable analysis of complex clinical trial data, offering concrete architectural principles for developing next-generation natural language interfaces in this field.

FUTURE WORK

Based on the findings and limitations of this work, several avenues for future research are apparent:

1. **Adaptive and Learning-Based Routing:** To improve router accuracy, especially for complex queries, future work could explore adaptive routing mechanisms. This might involve incorporating feedback from the validation step to dynamically adjust routing weights or developing machine learning models trained to predict the optimal system based on a richer set of query features.

2. **Enhanced Query Decomposition:** Improving the robustness and nuance of the Query Decomposition Engine is crucial, as errors here impact the entire downstream process. Integrating deeper domain knowledge, perhaps through clinical ontologies or knowledge graphs, could enhance the engine's ability to interpret complex clinical questions accurately.

3. **Expansion of Query Template Library:** The range of queries RagQL-Nav can handle is currently limited by its template library. Expanding this library to cover more advanced statistical analyses, survival analysis patterns, or even generating data visualizations would increase its utility.

4. **User Studies:** Evaluating the usability and effectiveness of RagQL-Nav with its intended users—clinical researchers and data analysts—is essential. User studies can provide qualitative feedback on the natural language interface, the clarity of the decision trees, and the overall impact on the data analysis workflow.

5. **Exploration of Other Database Technologies:** While this work focused on PostgreSQL and Neo4j, future research could investigate integrating other database types (e.g., time-series databases, document stores) that might offer advantages for specific kinds of clinical data or queries.

Addressing these areas will help mature RagQL-Nav from a promising framework into a robust, production-ready tool capable of significantly improving the efficiency and reliability of clinical trial data analysis.

REFERENCES

[1] Quanticate. *A Guide to CDISC ADaM Standards in 2024*. Blog Post, 2015 (Referenced as 2024 guide). URL: `https://www.quanticate.com/blog/bid/90417/exploring-the-analysis-data-model-adam-datasets`. Accessed: 2025-04-27.

[2] Zhou, X., et al. Statistical Analysis in Clinical Trials Using the Study Data Tabulation Model and ADaM. *International Journal of Clinical Biostatistics and Biometrics*, 9:052, 2023. URL: `https://clinmedjournals.org/articles/ijcbb/international-journal-of-clinical-biostatistics-and-biometrics-ijcbb-9-052.pdf`.

[3] Yuan, M., et al. Insights into ADaM. In *Proceedings of PharmaSUG 2010*. Paper HW06, 2010. URL: `https://www.lexjansen.com/pharmasug/2010/HW/HW06.pdf`.

[4] Ogbuji, K. O. *Graphical Database Architecture For Clinical Trials*. PhD thesis, North Carolina Agricultural and Technical State University, 2017. URL: `https://digital.library.ncat.edu/cgi/viewcontent.cgi?article=1321&context=theses`.

[5] Qualio. *21 CFR Part 11: A guide for clinical trial compliance*. Blog Post, 2022. URL: `https://www.qualio.com/blog/21-cfr-part-11-clinical-trials`.

[6] Yan, J. ENHANCING CLINICAL TRIAL DATA QUERIES WITH LLMS AND NEO4J: A FLEXIBLE FRAMEWORK FOR ADAM DATASET MANAGEMENT. In *Proceedings of PHUSE Connect US 2025 Conference*. Paper PAP_DH03, Orlando, FL, USA, March 16-19, 2025. URL: `https://phuse.s3.eu-central-1.amazonaws.com/Archive/2025/Connect/US/Orlando/PAP_DH03.pdf`.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Jaime Yan
Merck & Co., Inc.
Rahway, NJ, USA
mingyu.yan1@merck.com