Agentic-RAG on Mix-typed Knowledge Base: A Comparative Study of SQL and SPARQL Querying

Leonard Heininger
ARDI
TU Berlin
Berlin, Germany
heininger@tu-berlin.de

Nicolas Kohl
ARDI
TU Berlin
Berlin, Germany
nico-kohl@tu-berlin.de

***Abstract*—Retrieval Augmented Generation (RAG) over structured knowledge bases remains limited when relying solely on semantic similarity search, which cannot capture relational structure. We present an agentic RAG system that combines semantic vector search with schema-bound query execution via SQL and SPARQL over the STaRK-Prime biomedical knowledge graph. Using a two-stage approach that decouples entity resolution from structured query execution, we systematically compare four agent configurations (Search-Only, Search+SQL, Search+SPARQL, Search+SQL+SPARQL) across two model scales (GPT-5-mini, GPT-5) on 109 human-generated questions. Structured query access yields a +15.5 percentage-point F1 improvement over pure semantic search within our controlled comparison, and our agentic configurations with GPT-5-class backbones surpass prior STaRK baselines by a wide margin. SQL and SPARQL achieve practically equivalent retrieval quality ($r > 0.9$ item-level correlation, $\leq$ 3 pp F1 difference), though SPARQL exhibits lower error and zero-result rates. Scaling model capacity benefits Search-Only agents but yields diminishing returns for structured-query agents, whose primary bottleneck is entity resolution rather than query generation. We provide a detailed error taxonomy and design recommendations for practitioners building agentic RAG systems over semi-structured databases.**

***Index Terms*—Agentic RAG, Knowledge Graphs, Text-to-SQL, Text-to-SPARQL, Biomedical QA**

I. Introduction

Large Language Models (**LLMs**) have fundamentally changed individual search behavior and information-seeking patterns. Whereas users previously needed to translate natural language into keyword searches tailored to algorithmic patterns, they can now directly employ natural language queries (**NLQs**) to acquire information by interacting with **LLM** systems. This approach better aligns with intuitive user behavior, enables more efficient information synthesis, and is already evident in observed user practices [1], [2].

While **LLMs** can answer **NLQs** directly from their parametric knowledge, this knowledge is static, bounded by the training cutoff, and prone to hallucination, particularly for domain-specific, rapidly evolving, or long-tail factual queries [3]. Retrieval Augmented Generation (**RAG**) addresses these limitations by integrating classical Information Retrieval (**IR**) into the generative workflow: retrieved documents ground the **LLM**'s output in verified sources, reducing hallucination and extending coverage beyond the model's parameters. Historically, **IR** produced ranked document lists from unstructured corpora; **RAG** repurposes this capability by feeding the top-ranked passages directly into the **LLM** context, enabling answer synthesis rather than document ranking [3].

However, standard **RAG** pipelines rely on semantic similarity search over text chunks, which is inherently limited when the underlying data is structured. Chunking tabular or graph-structured data into flat text fragments destroys explicit row/column dependencies, typed relationships, and multi-hop paths; as a result, semantic retrieval struggles with aggregation queries, constraint satisfaction, and compositional relational reasoning [4], [5]. Schema-bound query languages such as Structured Query Language (**SQL**) and SPARQL Protocol and RDF Query Language (**SPARQL**) avoid these failure modes by operating directly on the relational or graph structure, leveraging exact joins, filters, and traversals that cannot be replicated by embedding-based retrieval alone.

As the industry enters what is colloquially termed the "Year of the Agent"[1], agentic Retrieval Augmented Generation (**aRAG**) emerges as the next evolution of **RAG**, moving beyond pre-orchestrated workflows toward dynamic reasoning loops capable of autonomous action [6]. This paper investigates the agent's ability to leverage schema-bound query languages, specifically **SQL** and **SPARQL**, as retrieval tools alongside semantic search, and empirically compares these strategies on the same benchmark. The focus is motivated by two converging observations: first, that semi-structured databases can be effectively constructed from unstructured corpora at scale [7], [8], making structured retrieval broadly applicable; and second, that schema-bound querying consistently improves factual retrieval performance over semantic search alone in agentic systems [6].

II. Related Work

A. STaRK Benchmark

Our evaluation builds on the STaRK benchmark [9], which pairs real-world knowledge graphs, including PrimeKG [10], with natural-language queries whose gold answers are entity sets, enabling set-valued evaluation with standard **IR** metrics (Hit@k, Mean Reciprocal Rank (**MRR**), Recall@k). Wu et al. show that embedding models and **LLM** re-rankers degrade on queries requiring multi-hop relational reasoning, establishing

[1]A widely used industry label; see, e.g., B. Xu, *Why 2025 is the Year of the AI Agent*, Medium, 2025.

a clear ceiling for unstructured retrieval [9]. AvaTaR [11] exploits this gap by employing STaRK in an agentic setting — an **LLM** agent's tool-usage policy is optimized via contrastive reasoning and subsequently achieves a 14% relative improvement on Hit@1 over non-agentic STaRK baselines. Our work extends this direction by adding schema-bound query execution (**SQL** and **SPARQL**) alongside semantic search and comparing these strategies head-to-head on STaRK-Prime.

B. RAG on SQL Tables

Prior work has explored **RAG**-style systems grounded in tabular data. A common baseline is to parse tabular data into a text format, chunk the result, and retrieve top-(k) chunks like an ordinary document corpus. This is often lossy: it weakens explicit row/column dependencies, and chunking fragments surrounding context that is needed for aggregation-style or multi-hop queries [4], [5].

To avoid these failure modes, many recent approaches shift the retrieval step from semantic similarity search over serialized rows to an execution-based access layer over structured relations: a **LLM** synthesizes an executable query (typically SQL) from a **NLQ** which is then executed against a Database Management System (**DBMS**) to compute the relevant subset. In heterogenous document settings, execution-based access avoids a key limitation of chunk retrieval: queries that require reasoning over an entire table (e.g., aggregations or global filters) fail when only a subset of rows is retrieved via top-(k) similarity search [4]. The TAG framework generalizes this view, arguing that (i) pure Text2SQL only covers requests expressible in relational algebra and (ii) "row-retrieval **RAG**" only supports point-lookup style questions; on the corresponding TAG-Bench, vanilla Text2SQL and vanilla RAG each plateau at roughly 20% exact-match accuracy, whereas combined pipelines reach over 55% [5]. These findings are consistent with the broader trend surfaced by large-scale benchmarks such as BIRD, which stresses dirty real-world databases, external knowledge requirements, and SQL efficiency, and on which even frontier models still fall well short of human performance [12].

While **LLMs** have largely taken over the role of earlier specialized neural parsers as the core SQL generation engine, state of the art systems still rely on orchestration layers that ensure the quality of the produced queries. We have identified four such techniques that are widely adopted across recent work:

1) **Schema linking.** A prerequisite for correct **SQL** generation is schema linking. It ensures that mentions in the **NLQ** align to the tables and columns they refer to in the database schema [13], [14]. Errors at this stage cascade irreversibly: if the linker omits a required table or selects a wrong column, no downstream generation step can recover [15]. Early cross-domain systems relied on exact string matching between question tokens and schema item names, which breaks down as soon as synonyms or domain-specific paraphrases appear [16]. A subsequent line of work showed that pre-trained language models already capture schema-linking relations in their representations, enabling robust linking without additional trained parameters [16]. In **LLM**-based pipelines the schema-linking step is now typically implemented as a dedicated prompt or agent call that selects the relevant schema subset before the generation prompt is constructed [17], [18].
2) **Execution-based verification and self-correction.** Early work on execution-guided decoding demonstrated that partially executing candidate **SQL** queries against a **DBMS** during beam search and pruning those that produce runtime failures like syntactic errors, semantic failures such as type mismatches and empty intermediate results, substantially improves output quality [19]. Modern **LLM** pipelines adopt the same principle in a generate–execute–retry loop: the synthesized SQL is run against the database, and if it returns an error or an implausible result the **LLM** is re-prompted with the error trace [13], [18]. This runtime feedback replaces the compile-time guarantees of earlier constrained-decoding methods that enforced syntactic validity by rejecting inadmissible tokens at each autoregressive step [20].
3) **Multi-agent decomposition.** Rather than relying on a single generation call, several recent systems distribute the task across specialised agents: one for schema linking, one for SQL drafting, one for refinement, and one for final selection. This allows each stage to be independently prompted and evaluated [17], [21].
4) **Retrieval-augmented example selection.** Retrieving semantically similar (question, **SQL**) pairs from a curated store and injecting them as few-shot demonstrations has become a standard prompt-construction strategy, with reported execution accuracy of 86.6% on the Spider benchmark attributed in large part to this component [18], [22].

Taken together, these techniques show that while the **LLM** provides the core translation capability, robust Text2SQL still requires an engineered pipeline around it. However the nature of the scaffolding has shifted from grammar-level constraints to runtime orchestration.

C. RAG on Knowledge Graphs

Prior work has also explored **RAG**-style systems grounded in knowledge graphs. When entity descriptions are serialized into text and processed through a standard chunking pipeline, the explicit relational structure of the graph is lost: edge types, multi-hop paths, and typed constraints cannot be recovered from unstructured text chunks [5]. This limitation is especially acute for Knowledge Graph Question Answering (**KGQA**) tasks that require compositional reasoning over graph topology rather than point-lookup of isolated facts.

To preserve relational structure, a growing body of work adopts a graph traversal-based paradigm analogous to Text2SQL: an **LLM** synthesizes a **SPARQL** query from a **NLQ**, which is then executed against a **SPARQL** endpoint to retrieve the answer. However, Text2SPARQL presents distinct challenges relative to Text2SQL. Knowledge graph schemata are typically larger, more heterogeneous, and less standardized than relational database schemata, and the open-ended nature of Resource Description Framework (**RDF**) vocabularies means that entity and predicate identifiers often lack the mnemonic column names that aid SQL generation [23], [24]. On the Spider4SPARQL benchmark, which transposed the complexity tiers of the original Spider benchmark into the **SPARQL** domain, even frontier **LLMs** achieve only up to 51% execution accuracy, considerably below comparable Text2SQL results on the original Spider [23], [24]. This gap confirms that **KGQA** remains substantially harder than its relational counterpart.

As with Text2SQL, state-of-the-art Text2SPARQL systems rely on orchestration layers around the core **LLM** generation step. We identify four widely adopted techniques that parallel those in the relational setting:

1) **Ontology linking.** Domain-specific Knowledge Graphs (**KGs**) with curated schemas such as PrimeKG (10 entity types, 18 relation types) used in the STaRK benchmark [9] present vocabularies comparable in size and regularity to relational databases. For such smaller ontologies, a practical and widely adopted strategy is to inject the full ontology description directly into the **LLM** system prompt. Reif et al. embed the complete ontology together with domain-specific standards into the prompt context, enabling accurate **SPARQL** generation over industrial **KGs** without any fine-tuning [25]. Rasheed and Aguado show that augmenting prompts with discrete vocabulary information extracted from a reduced **KG** ontology yields competitive performance while substantially reducing prompt size, and that this approach enables off-domain users to query domain-specific **KGs** through a domain-agnostic interface [26]. Hernandez-Camero et al. construct prompts that combine the natural-language question with contextual **KG** information (entity types, relation types, and example triples) alongside few-shot examples, improving **SPARQL** accuracy by 6% on an aviation **KG** [27]. This ontology-in-prompt strategy is effective when the schema fits within the **LLM** context window, but does not scale to open-domain **KGs** with thousands of predicates, where more sophisticated ontology linking methods such as graph neural network alignment [28] or learned hybrid prompts [29] become necessary.
2) **Execution-based verification and self-correction.** The generate–execute–retry loop transfers directly to the **SPARQL** setting. FIRESPARQL incorporates a **SPARQL** query correction layer that validates generated queries and repairs structural and semantic errors, achieving 0.90 ROUGE-L for query accuracy on the SciQA benchmark [30]. The Expasy federated **SPARQL** system similarly includes a validation step that corrects generated queries against the **KG** endpoint, reducing hallucinated predicates and malformed triple patterns [31]. GRASP goes further by using the **LLM** to iteratively explore the knowledge graph through strategic **SPARQL** sub-queries, discovering relevant Internationalized Resource Identifiers (**IRIs**) and literals at runtime rather than relying on a static schema description. This agentic exploration achieves state-of-the-art results on multiple Wikidata benchmarks in a zero-shot setting [32].
3) **Retrieval-augmented example selection.** As in Text2SQL, retrieving semantically similar (question, **SPARQL**) pairs for few-shot prompting is a standard strategy. SparqLLM retrieves template-based **SPARQL** examples from a curated store to guide the **LLM** toward structurally correct queries [33]. Avila et al. combine **RAG** with few-shot learning, retrieving both query examples and minimal **KG** subgraphs as context, achieving an F1-score of 0.73 in a zero-shot setting on SciQA which is a significant improvement over the prior score of 0.26 [34]. Kosten et al. systematically evaluate six prompting strategies on Spider4SPARQL and find that a simple prompt combined with an ontology description and five random shots is the most effective configuration [24].
4) **Entity anchoring.** Before a multi-hop graph traversal can begin, the system must identify one or more *anchor entities* — **KG** nodes that correspond to the key concepts mentioned in the user query and serve as entry points into the graph. Shen et al. propose GeAR, which augments a conventional base retriever with a graph expansion mechanism that extracts proximal triples from initially retrieved nodes and maintains a gist memory across reasoning steps, achieving state-of-the-art results on multi-hop Question Answering (**QA**) benchmarks such as MuSiQue while consuming fewer tokens than prior multi-step systems [35]. Xu et al. observe that most **KG**-based **RAG** methods assume anchor entities are given, which breaks down in open-world settings where query terms do not map cleanly to **KG** nodes. Their AnchorRAG framework addresses this with a predictor agent that dynamically identifies candidate anchors by aligning query terms with **KG** nodes and spawns independent retriever agents for parallel multi-hop exploration from each candidate [36]. Robust entity anchoring is particularly important for biomedical **KGs** such as PrimeKG, where synonymy, abbreviations, and cross-ontology identifiers make surface-level matching unreliable.

Taken together, these techniques demonstrate that Text2SPARQL shares many orchestration patterns with Text2SQL. However, the relative difficulty of the two paradigms is schema-dependent: large, heterogeneous **KGs** with thousands of predicates amplify the challenges unique to SPARQL, whereas compact, well-typed ontologies may actually be easier to query as **RDF** triples than through an equivalent relational projection with many tables and foreign-key joins.

D. *Agentic RAG*

Traditional **RAG** pipelines follow a fixed retrieve-then-generate workflow: a query is embedded, the top-k passages are fetched, and a single **LLM** call synthesizes the answer. This static design lacks the adaptability required for multi-step reasoning, heterogeneous data sources, and queries whose complexity is not known in advance [6]. **aRAG** addresses

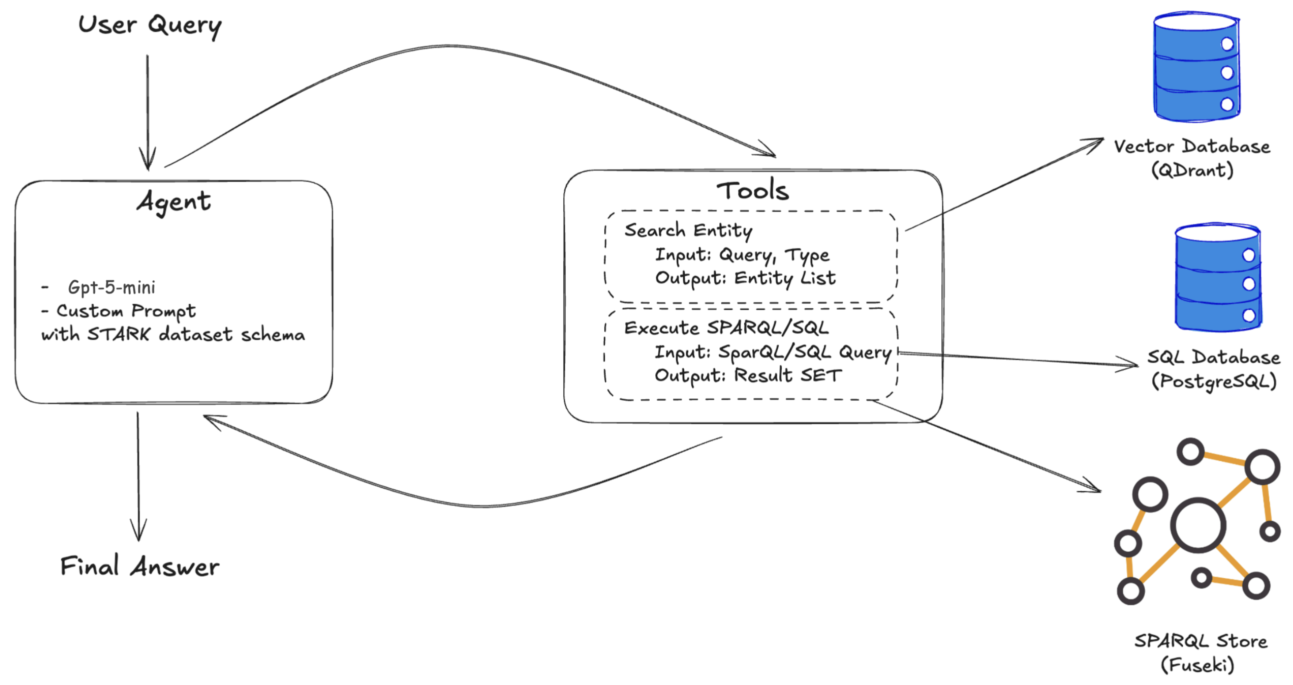

Fig. 1. Agent loop and system components.

these limitations by integrating autonomous agents into the retrieval loop. Singh et al. identify four core agentic design patterns (reflection, planning, tool use, and multi-agent collaboration) that together allow the system to dynamically decide *what* to retrieve, *how* to retrieve it, and *when* to stop [6]. Sapkota et al. draw a broader distinction between task-specific AI agents and full agentic AI systems, characterizing the latter by persistent memory, dynamic task decomposition, and coordinated autonomy across multiple specialized agents [37].

Recent systems instantiate these patterns in different ways. HM-RAG employs a three-tiered hierarchy: a decomposition agent rewrites the query into coherent sub-tasks, modality-specific retrieval agents search vector, graph, and web sources in parallel, and a decision agent fuses answers through consistency voting, yielding a 12.95% accuracy improvement on multimodal **QA** benchmarks [38]. MAO-ARAG takes an adaptive approach, training a planner agent via reinforcement learning to compose per-query workflows from a pool of executor agents (query reformulation, document selection, generation), balancing answer quality against cost and latency [39].

Despite this progress, existing agentic systems typically treat SQL and graph backends as separate, independently managed tools and do not provide controlled experiments comparing when a graph query outperforms a relational query, or how an agent should decide between them. Our work addresses this gap.

III. Approach

We present an agentic retrieval system for the STaRK-Prime knowledge base [9], a biomedical knowledge graph derived from PrimeKG [10] containing over 129,000 entities across 10 types (diseases, drugs, genes/proteins, pathways, biological processes, molecular functions, cellular components, anatomical structures, exposures, and phenotypes) connected by 18 relation types; a detailed per-type entity distribution is provided in the original STaRK paper [9]. Our architecture combines semantic vector search with schema-bound query execution, enabling systematic comparison of retrieval strategies: pure semantic search versus structured querying via **SQL** and **SPARQL**.

A. System Architecture

Our system follows the **aRAG** paradigm, employing an autonomous agent loop as the core reasoning mechanism (see Fig. 1). The agent iteratively reasons about the user's query, selects appropriate tools, executes them, and refines its approach based on intermediate results. This loop continues until the agent determines it has gathered sufficient information to answer the query or reaches a predefined iteration limit of 100 steps, which serves as a safeguard against infinite loops when the agent repeatedly fails to find relevant entities or formulates unsuccessful queries. In practice, the agent is able to surface an answer way before this step count and the hard limit of 100 exists only as a fallback.

The architecture comprises three components:

1) **Agent Component**: An **LLM**-driven reasoning loop that interprets queries, plans tool invocations, and synthesizes final answers. We implement this using

LangChain's[2] ReAct agent framework. LangChain is the most widely adopted agent orchestration library in the research community, offering mature, well-documented abstractions for tool-using agents, composable chains, and callback-based observability [6], [38], [39]. Alternative frameworks pursue different design priorities: AutoGen [40] centers on multi-agent *conversation*, where customizable agents communicate via natural-language message passing. This paradigm is well suited for collaborative reasoning but heavier than needed for our single-agent, multi-tool setting. AutoAgent [41] targets zero-code agent creation through natural language alone, prioritizing accessibility over fine-grained control of the retrieval pipeline. LangChain's explicit tool-calling Application Programming Interface (**API**) and its native integration with observability platforms such as Langfuse[3] make it the most suitable choice for our controlled benchmarking setup, where reproducibility and per-step traceability are essential.

2) **Tool Component**: A set of specialized tools that provide the agent with distinct capabilities for knowledge base access. These tools abstract the complexity of different query interfaces behind a unified function-calling **API**.
3) **Data Component**: Multiple materialized views of the same underlying knowledge graph, each optimized for different query patterns.

B. Two-Stage Query Process

Prior work on the STaRK benchmark [9] evaluated retrieval methods that operate in a single stage: given a natural language query, retrieve the top-k most similar entities based on embedding distance or lexical matching. This approach struggles with multi-hop relational queries because entity descriptions alone do not encode graph structure.

Our key design decision is a **two-stage query approach** that decouples entity resolution from structured query execution:

a) *Stage 1: Entity Resolution:*

Natural language queries reference entities by names, synonyms, or descriptions rather than internal identifiers. The first stage uses semantic vector search to resolve these references to node IDs. This approach leverages embedding similarity to handle:

- Lexical variations (e.g., "heart attack" → "myocardial infarction")
- Partial matches (e.g., "breast cancer" → "invasive breast carcinoma")
- Related concepts (e.g., searching for symptoms may surface related diseases)

Entity embeddings are generated using OpenAI's text-embedding-3-small[4] model and indexed in Qdrant[5], a high-performance vector database. Each entity is embedded as a rich text representation combining its type, name, aliases, and descriptive metadata (e.g., "disease: Type 2 Diabetes Mellitus, Also known as: T2DM, adult-onset diabetes"), enabling type-aware and semantically rich retrieval.

b) *Stage 2: Query Execution:*

Once entity IDs are resolved, the agent formulates structured queries using these identifiers. This separation ensures that:

1) The **LLM** is not burdened with fuzzy string matching in query predicates
2) Queries operate on exact node identifiers, eliminating false negatives from name mismatches
3) The agent can leverage the full expressiveness of **SQL** or **SPARQL** for relational reasoning

C. Tool Design

We provide the agent with three specialized tools:

a) *Semantic Search Tool (search_entities_tool):*

Performs vector similarity search over entity embeddings stored in Qdrant. Parameters include:

- query: Natural language search term
- entity_type: Optional filter to restrict search to specific node types (e.g., "disease", "drug", "gene_protein")
- top_k: Number of results to return (default: 5)

The tool returns ranked entity matches with IDs, types, names, and descriptions.

b) *SQL Query Tool (execute_sql_query_tool):*

Executes read-only **SQL** queries against a PostgreSQL[6] database containing a relational projection of the knowledge graph. The schema comprises:

- **10 entity tables**: One table per node type (disease, drug, gene_protein, pathway, biological_process, molecular_function, cellular_component, anatomy, exposure, phenotype)
- **18 relation tables**: One table per edge type (e.g., indication, contraindication, target, interacts, associated_with)

Each relation table follows a consistent schema with src_id, dst_id, src_type, and dst_type columns, enabling efficient join operations.

c) *SPARQL Query Tool (execute_sparql_query_tool):*

Executes read-only **SPARQL** queries against an Apache Jena Fuseki[7] endpoint containing the knowledge graph in **RDF** format. The **RDF** representation uses a custom namespace (http://stark.stanford.edu/prime/) with:

- Node Uniform Resource Identifiers (**URIs**): sp:node/{id} for entity references
- Edge predicates: sp:{relation_type} for typed relationships
- Node properties: sp:name, sp:type, sp:description for entity attributes

D. Schema Grounding

A critical enabler for structured query generation is providing the **LLM** with comprehensive schema information in its

[2]https://github.com/langchain-ai/langchain
[3]https://github.com/langfuse/langfuse
[4]https://platform.openai.com/docs/models/text-embedding-3-small
[5]https://github.com/qdrant/qdrant
[6]https://github.com/postgres/postgres
[7]https://github.com/apache/jena

system prompt. Without schema knowledge, the model cannot generate syntactically correct queries that reference valid tables, columns, or predicates. Our approach is intentionally general-purpose and does not incorporate benchmark-specific optimizations.

Our system prompt includes:

- **SQL Schema Summary**: Complete listing of all entity tables with their columns, all relation tables with their foreign key structure, and example values for categorical fields. This enables the LLM to formulate valid JOIN conditions and WHERE clauses.
- **SPARQL Vocabulary Summary**: The RDF namespace, available predicates (relation types), node URI patterns, and property definitions. This grounds the LLM's understanding of the graph structure for pattern matching.
- **Query Guidelines**: Best practices for query formulation, including mandatory LIMIT clauses, read-only restrictions, and error handling strategies.

This schema grounding approach instantiates two techniques surveyed in our related work: *schema linking* (Section II.B), which ensures the LLM references only valid tables and columns, and *ontology-in-prompt injection* (Section II.C), which embeds the full KG vocabulary into the generation context. The schema is dynamically generated at agent initialization time, ensuring consistency with the actual data store state.

E. Data Materialization

We materialize the PrimeKG knowledge graph into three complementary data stores, each optimized for different access patterns:

TABLE I
Data stores and their optimized query patterns.

Data Store	Technology	Optimized For
Vector Index	Qdrant	Semantic similarity search, entity resolution
Relational DB	PostgreSQL	Aggregations, complex joins, filtering
Graph Store	Apache Jena Fuseki	Path traversal, pattern matching

This multi-store architecture enables the agent to select the most appropriate query interface for each sub-task. For instance, counting entities is more efficient in SQL, while finding paths between nodes is more natural in SPARQL.

F. Agent Configurations

We implement four agent configurations to systematically isolate the effect of each tool capability. By comparing agents with different tool access, we can measure the marginal value of structured query execution over pure semantic search, and SPARQL's graph-native patterns versus SQL's relational joins:

a) *Search-Only Agent:*

Access to semantic vector search (search_entities_tool) only. This configuration represents pure unstructured retrieval without structured query capabilities. The agent can only find entities by semantic similarity and cannot perform relational reasoning.

Tools: search_entities_tool

Use case: Baseline for evaluating the value of structured query access.

b) *Search+SPARQL Agent:*

Access to semantic search for entity resolution plus SPARQL query execution over the RDF graph. This configuration enables explicit graph traversal via declarative pattern matching.

Tools: search_entities_tool, execute_sparql_query_tool

Use case: Evaluating graph-native query formulation for knowledge graph reasoning.

c) *Search+SQL Agent:*

Access to semantic search plus SQL query execution over the relational projection. This configuration tests whether LLMs can effectively express graph operations as relational joins.

Tools: search_entities_tool, execute_sql_query_tool

Use case: Evaluating relational query formulation and practical deployability with existing SQL infrastructure.

d) *Search+SQL+SPARQL Agent:*

Access to all three tools (semantic search, SPARQL, SQL). This configuration provides maximum flexibility, allowing the agent to choose the most appropriate query language for each sub-task.

Tools: search_entities_tool, execute_sql_query_tool, execute_sparql_query_tool

Use case: Evaluating whether multi-paradigm structured query access yields complementary benefits.

G. Model Selection

We instantiate each agent configuration with two LLM backbones to isolate the effect of model capacity:

- **GPT-5-mini**[8]: A smaller, faster variant optimized for efficiency. Lower cost and latency make it suitable for high-throughput applications.
- **GPT-5**[9]: The full-scale model with enhanced reasoning capabilities. Higher cost but potentially better performance on complex queries.

Both models are accessed via the OpenAI API with temperature set to 0.0 for near-deterministic output (note that API-level non-determinism from batching and floating-point arithmetic may still introduce minor variation across runs). We focus on the GPT family to control for API differences and prompting conventions, enabling cleaner isolation of model scale effects.

H. Observability and Analysis

To enable comprehensive analysis of agent behavior, we integrate observability and benchmarking infrastructure:

- **Langfuse**: Provides trace-level observability of agent reasoning, including tool invocations, intermediate results, and latency breakdowns. Each query execution is logged as a trace with full message history.

[8]https://platform.openai.com/docs/models/gpt-5-mini
[9]https://platform.openai.com/docs/models/gpt-5

TABLE III
Our agentic retrieval performance on STaRK Human QA benchmark (mean over $n = 6$ runs; std in parentheses). Best results per metric are **bolded**; second-best are <u>underlined</u>.

Agent	Model	Precision	Recall	F1	EM	Hit@1	Hit@5	MRR	Latency (s)
Search-Only	GPT-5-mini	28.1% (1.4)	30.9% (1.8)	24.8% (1.4)	11.2% (1.5)	34.1% (1.7)	42.0% (1.8)	37.2% (1.4)	23.8
	GPT-5	37.9% (2.1)	38.2% (2.3)	32.9% (2.1)	17.6% (1.5)	40.3% (2.2)	51.2% (2.4)	44.5% (2.0)	38.6
Search+SPARQL	GPT-5-mini	46.2% (1.0)	<u>47.2%</u> (1.8)	<u>40.3%</u> (1.1)	<u>24.3%</u> (1.7)	46.7% (2.2)	56.7% (1.5)	50.6% (1.9)	42.5
	GPT-5	**47.7%** (1.9)	**47.7%** (1.9)	**40.4%** (1.8)	**24.8%** (1.4)	<u>48.5%</u> (1.8)	<u>58.4%</u> (3.4)	**52.2%** (2.4)	64.4
Search+SQL	GPT-5-mini	44.3% (1.6)	46.5% (1.3)	38.3% (1.2)	20.2% (1.6)	44.9% (1.7)	57.8% (1.7)	49.9% (1.4)	47.6
	GPT-5	45.3% (1.9)	42.5% (2.6)	37.1% (1.8)	22.8% (1.7)	46.8% (2.0)	54.1% (1.7)	49.9% (1.5)	70.9
Search+SQL+SPARQL	GPT-5-mini	<u>46.4%</u> (0.8)	47.0% (1.9)	39.4% (1.1)	20.9% (1.3)	46.9% (0.9)	**59.8%** (2.4)	<u>52.1%</u> (0.9)	39.4
	GPT-5	45.9% (2.4)	43.8% (2.0)	37.8% (1.9)	23.2% (2.1)	**48.6%** (2.2)	56.1% (2.4)	51.8% (2.0)	76.1

- **Custom Benchmark Framework**: Our evaluation harness executes agents across dataset splits, captures per-query metrics (precision, recall, F1, latency, tool call counts), and generates detailed analysis reports with item-level breakdowns and failure mode classification.

This instrumentation enables both real-time debugging during development and systematic post-hoc analysis of experimental results.

IV. Evaluation

A. Experimental Setup

Dataset. We evaluate on the STaRK Human-Generated QA benchmark [9], comprising 109 natural-language questions over PrimeKG [10], a biomedical knowledge graph. Questions require entity retrieval from a heterogeneous graph containing diseases, drugs, genes/proteins, pathways, biological processes, molecular functions, cellular components, anatomical structures, exposures, and phenotypes. Unlike synthetic benchmarks, the human-generated queries exhibit diverse linguistic patterns and varying complexity levels, including simple entity lookups, multi-hop relational queries, and complex constraint satisfaction problems [9].

Agent Configurations. We compare four agentic retrieval strategies as described in Section III.F: Search-Only, Search+SPARQL, Search+SQL, and Search+SQL+SPARQL. Each configuration provides access to different tool combinations, enabling systematic comparison of retrieval strategies. All agents use semantic search; Search-Only relies on it exclusively, while the others add structured query capabilities on top.

Models. Following Section III.G, each agent configuration is instantiated with two model variants: **GPT-5-mini** (smaller and faster) and **GPT-5** (full-scale). This pairing isolates the effect of model capacity on agentic retrieval performance.

Repetitions. To quantify variance from LLM non-determinism, each configuration-model pair is executed 6 times ($n = 6$), yielding 48 total experimental runs and 5,232 individual query evaluations.

B. Evaluation Metrics

Let Y denote the gold entity set and $\hat{Y}$ the predicted entity set. We report standard information retrieval metrics:

- **Precision (P)**: $|\hat{Y} \cap Y| \ / \ |\hat{Y}|$ — fraction of returned entities that are relevant.
- **Recall (R)**: $|\hat{Y} \cap Y| \ / \ |Y|$ — fraction of relevant entities that are returned.
- **F1**: $2 \cdot P \cdot R/(P + R)$ — harmonic mean of precision and recall.
- **Exact Match (EM)**: $\mathbb{1}\left[\hat{Y} = Y\right]$ — indicator of perfect set match.
- **Hit@k**: $\mathbb{1}\left[|\hat{Y}_{1:k} \cap Y| > 0\right]$ — whether any correct entity appears in the top-k results.
- **MRR**: Mean Reciprocal Rank of the first correct entity in the ranked output.
- **Latency**: End-to-end wall-clock time per query (seconds).

C. Main Results

Table III summarizes aggregate performance. We first compare against STaRK's published baselines (Table II), then analyze differences across our agent configurations.

TABLE II
STaRK baseline results on Human QA benchmark [9]. R@20 = Recall at rank 20.

Method	Hit@1	Hit@5	R@20	MRR
AvaTaR (GPT-4-turbo)	33.0%	51.4%	53.3%	41.0%
Claude3 Reranker	28.6%	46.9%	41.6%	36.3%
GPT4 Reranker	28.6%	44.9%	41.6%	34.8%
GritLM-7b	25.5%	41.8%	48.1%	34.3%
multi-ada-002	24.5%	39.8%	47.2%	33.0%
BM25	22.5%	41.8%	42.3%	30.4%

D. Analysis by Retrieval Strategy

a) *Comparison to STaRK Baselines:*

Our agentic approach, combined with GPT-5-class model backbones, achieves substantially higher scores than all published STaRK baselines. The best baseline, AvaTaR (GPT-4-turbo), achieves Hit@1=33.0%, Hit@5=51.4%, and **MRR**=41.0%. In comparison, our Search+SPARQL agent with GPT-5 achieves Hit@1=48.5% (+15.5 pp), Hit@5=58.4% (+7.0 pp), and **MRR**=52.2% (+11.2 pp).

We note that STaRK baselines also report Recall@20 (R@20), ranging from 41.6% to 53.3%. This metric is not directly comparable to our set-based Recall because STaRK baselines return fixed-size ranked lists of 20 candidates, whereas our agents return variable-sized answer sets (typically 1–5 entities); we therefore restrict the comparison to Hit@k and **MRR**.

This comparison is not fully controlled: our system uses GPT-5 / GPT-5-mini, whereas STaRK baselines were evaluated with GPT-4-turbo and earlier models. The observed gains therefore reflect the combined effect of (i) agentic structured-query access, (ii) the two-stage entity-resolution pipeline, and (iii) the stronger model backbone. A fully controlled comparison would require re-running all baselines on identical model backbones. Nevertheless, the gap is large enough to indicate that the combination of agentic retrieval with schema-bound query execution on GPT-5-class models sets a new performance level on STaRK-Prime.

b) *Structured vs. Unstructured Retrieval:*

The most striking result is the substantial performance gap between pure semantic search and structured-query agents. With GPT-5-mini, Search+SPARQL achieves:

- +18.1 pp higher precision (46.2% vs. 28.1%)
- +16.3 pp higher recall (47.2% vs. 30.9%)
- +15.5 pp higher F1 (40.3% vs. 24.8%)
- +13.1 pp higher exact match rate (24.3% vs. 11.2%)

This gap narrows but persists with GPT-5 (F1: 40.4% vs. 32.9%, +7.5 pp). The pattern confirms STaRK's original findings [9]: multi-hop relational queries such as "find diseases treated by drug X that also exhibit phenotype Y" exceed the capabilities of embedding-based retrieval. Semantic search over isolated entity descriptions cannot capture the graph structure needed for relational composition.

c) *SPARQL vs. SQL:*

Search+SPARQL and Search+SQL achieve practically equivalent F1, with SPARQL scoring +2.0 pp higher with GPT-5-mini (40.3% vs. 38.3%) and +3.3 pp with GPT-5. These differences fall within observed standard deviations and should not be interpreted as a statistically confirmed advantage. The overall pattern suggests that modern **LLMs** can effectively translate graph queries into relational operations. As discussed in Section V.E, the two languages differ more in operational characteristics (error rates, zero-result rates) than in end-to-end retrieval quality.

d) *Multi-Tool Agents:*

The Search+SQL+SPARQL agent achieves the highest Hit@5 (59.8% with GPT-5-mini), outperforming both single-structured-query agents. However, its precision and F1 do not consistently exceed Search+SPARQL, and exact match rates are lower. This pattern suggests that multi-tool access improves recall by enabling the agent to recover entities that one query language might miss, but introduces additional decision complexity that can degrade precision.

e) *Item-Level Correlation:*

To understand whether SPARQL and SQL provide complementary or redundant capabilities, we compute per-item F1 correlations between agent configurations (Table IV). High correlation indicates that configurations succeed and fail on the same items.

TABLE IV
Per-item F1 correlation matrix (GPT-5/GPT-5-mini). Values show Pearson r of item-level F1 scores averaged over 6 runs.

Agent	Search-Only	SPARQL	SQL	SQL+SPARQL
Search-Only	1.00	.64/.40	.69/.40	.66/.37
Search+SPARQL	—	1.00	**.92/.90**	.91/.93
Search+SQL	—	—	1.00	.96/.95
Search+SQL+SPARQL	—	—	—	1.00

The correlation between Search+SPARQL and Search+SQL is strikingly high: $r = 0.92$ (GPT-5) and $r = 0.90$ (GPT-5-mini). This indicates that the two query languages succeed and fail on nearly identical items, meaning they are **redundant** rather than complementary. By contrast, Search-Only shows much lower correlation with structured-query agents ($r = 0.40$ to 0.69), confirming that structured queries provide qualitatively different capabilities than pure semantic search.

f) *Effect of Model Scale:*

Scaling from GPT-5-mini to GPT-5 yields heterogeneous effects across agent types:

- **Search-Only**: Large improvement (+9.8 pp precision, +8.1 pp F1). The stronger model better interprets query semantics and generates more effective search terms.
- **Search+SPARQL**: Modest improvement (+1.5 pp precision, +0.1 pp F1). Query generation is already well-handled by the smaller model.
- **Search+SQL**: Marginal or no improvement (+1.0 pp precision, −1.2 pp F1). The larger model does not improve SQL generation quality.
- **Search+SQL+SPARQL**: Mixed results (−0.5 pp precision, −1.6 pp F1). Additional model capacity may introduce suboptimal tool selection.

These findings suggest that the bottleneck for structured-query agents lies in entity resolution and knowledge graph coverage, not **LLM** reasoning capacity.

E. Latency Analysis

Table V presents latency statistics across configurations.

TABLE V
Query latency statistics (seconds). p50 = median, avg = arithmetic mean.

Agent	Model	p50	p90	avg	max
Search-Only	GPT-5-mini	18.9	43.4	23.8	112.9
	GPT-5	29.2	75.1	38.6	213.0
Search+SPARQL	GPT-5-mini	28.6	86.2	42.5	236.0
	GPT-5	52.3	121.6	64.4	590.5
Search+SQL	GPT-5-mini	33.9	99.1	47.6	239.9
	GPT-5	61.7	133.5	70.9	503.4
Search+SQL+SPARQL	GPT-5-mini	26.8	83.5	39.4	206.7
	GPT-5	66.9	141.8	76.1	594.0

Search-Only is fastest due to single-tool invocations. Structured-query agents require additional entity-resolution steps (semantic search to find node IDs, then query formulation and execution), increasing latency by 1.6–2.0×. GPT-5 roughly doubles latency across all configurations due to longer generation times and more complex reasoning traces. The Search+SQL+SPARQL agent with GPT-5 exhibits the highest variance, with tail latencies exceeding 500 seconds on complex multi-tool queries.

F. *Error Taxonomy*

From detailed item-level analysis across all 109 benchmark queries and 48 experimental runs (5,232 query evaluations), we identify four primary failure modes with quantified incidence rates.

a) *Entity Resolution Failure:*

The semantic search does not surface the correct entity, typically because:

- The entity has an unusual or ambiguous name.
- The query mentions the entity indirectly via properties rather than name.
- Multiple entities share similar names, and the wrong one is selected.

This is the most common failure mode: averaged across all 48 runs (8 configurations × 6 repetitions), **34.9%** of queries (38/109) exhibit complete entity resolution failure (mean coverage = 0 across runs), while an additional **22.9%** (25/109) achieve only partial coverage. Only **42.2%** of queries achieve full coverage of gold entities during tool execution. Entity resolution failure is the dominant cause of the **38.5%** zero-recall rate observed across the benchmark. Notably, the semantic search tool itself exhibits **0% error rate** across all configurations: it reliably returns results, but those results may not contain the target entities.

b) *Tool Execution Errors:*

Table VI presents tool-level error rates across agent configurations. We distinguish between two metrics: *error rate per call* (fraction of tool invocations that fail with an exception) and *items affected* (fraction of queries experiencing at least one tool error).

TABLE VI
Tool execution error rates by agent configuration. Error Rate = errors/total calls. Items Affected = queries with $\geq$ 1 error.

Agent	Model	Error Rate	Items Affected
Search-Only	GPT-5-mini	0.00%	0.0%
	GPT-5	0.00%	0.0%
Search+SPARQL	GPT-5-mini	0.39%	2.0%
	GPT-5	0.53%	3.8%
Search+SQL	GPT-5-mini	1.06%	5.8%
	GPT-5	0.96%	6.1%
Search+SQL+SPARQL	GPT-5-mini	0.79%	4.4%
	GPT-5	0.64%	4.3%

Key observations:

- **Search tools are error-free**: Semantic search never fails at the **API** level, making Search-Only agents operationally robust.
- **SQL exhibits higher error rates than SPARQL**: Search+SQL agents experience 1.0% error rate vs. 0.4–0.5% for Search+SPARQL, with errors affecting 5.8–6.1% of items vs. 2.0–3.8%. SPARQL errors are exclusively syntax errors (malformed queries), whereas SQL errors are predominantly schema reference errors (invalid table or column names), indicating **LLM** confusion about the relational schema.
- **Model scale has limited impact on error rates**: GPT-5 does not substantially reduce error rates for SQL agents compared to GPT-5-mini, though SPARQL items affected nearly doubles from 2.0% to 3.8%, suggesting that the larger model may attempt more complex graph patterns that are harder to formulate correctly.

c) *Zero-Result Queries (Semantic Failures):*

Beyond execution errors, queries may succeed syntactically but return empty result sets, indicating semantically incorrect formulations. Table VII shows zero-result rates by tool type.

TABLE VII
Zero-result rates by agent and tool type. Overall rate is weighted by tool usage; per-tool rates show tool-specific query failure.

Agent	Model	Overall Zero-Result	SQL Zero-Result	SPARQL Zero-Result
Search-Only	GPT-5-mini	0.0%	—	—
	GPT-5	0.0%	—	—
Search+SPARQL	GPT-5-mini	14.4%	—	42.3%
	GPT-5	17.2%	—	53.2%
Search+SQL	GPT-5-mini	21.1%	49.6%	—
	GPT-5	21.9%	62.3%	—
Search+SQL+SPARQL	GPT-5-mini	19.1%	50.0%	25.6%
	GPT-5	19.7%	63.3%	28.5%

The most striking finding is that **SQL queries return empty results 50–63% of the time**, compared to 25–53% for SPARQL. This substantial gap indicates that LLMs struggle more with relational query formulation than graph pattern matching. Potential causes include:

- **Schema complexity**: The relational projection requires understanding foreign key relationships and join semantics.
- **Schema navigation**: Expressing a single RDF triple pattern requires an explicit multi-table JOIN in the relational projection, increasing the surface area for semantic errors.
- **Over-constrained queries**: SQL's explicit join syntax may lead to overly restrictive conditions.

Because semantic search always returns results (it retrieves the nearest neighbours regardless of relevance), the overall zero-result rate understates the problem: restricting the count to structured-query tool calls alone raises the failure rate to 50–63% for SQL and 25–53% for SPARQL.

d) *Incomplete Result Selection:*

The agent discovers correct entities during tool calls but fails to include them in the final answer. We report two complementary metrics: (i) the *incidence rate*: **16.5%** of queries (18/109) exhibit at least one missed opportunity (i.e., at least one gold entity appeared in tool output but was absent from the final prediction), and (ii) the *micro-averaged missed opportunity rate*: across all gold entities surfaced by tools, **14.2%** were dropped from the final answer (computed as total missed entities / total found entities across all queries). This represents a reasoning/summarization failure at the answer synthesis step, where the agent incorrectly filters or ranks intermediate results.

V. Discussion

We now interpret the experimental results and derive practical recommendations for Agentic-RAG system design.

A. Structured Query Generation as the Key Advantage

The dominant finding is that agentic access to structured query interfaces (SPARQL or SQL) substantially outperforms pure semantic retrieval in our controlled within-system comparison (Table III). The +15.5 pp F1 improvement of Search+SPARQL over Search-Only (with GPT-5-mini) represents a qualitative capability gap, not merely incremental improvement. In addition, our agentic configurations with GPT-5-class backbones surpass all published STaRK baselines (Table II) by a wide margin, though that comparison confounds query-strategy effects with model-generation effects (see Section IV.D).

The within-system result confirms and extends the central finding from the STaRK benchmark [9]: embedding-based retrieval struggles with multi-hop relational queries that require joins across entity types, path traversals, or constraint filtering. The agentic paradigm—combining iterative reasoning with structured query execution—provides a path forward that pure retrieval methods cannot match.

Importantly, this advantage emerges even though all structured-query agents *depend on* semantic search for entity resolution. The structured query step adds value by enabling explicit relational reasoning over the resolved entities, whereas Search-Only terminates after the initial retrieval step. However, as detailed in Section IV.F, our analysis also reveals **missed opportunities**—a reasoning failure at answer synthesis distinct from retrieval failure.

B. The Entity Resolution Bottleneck

A critical architectural insight from item-level analysis is that entity resolution constitutes the primary failure mode for structured-query agents. The pipeline operates as follows:

1) Semantic search maps natural-language entity mentions to node IDs.
2) The LLM formulates a structured query using these IDs.
3) Query execution retrieves related entities from the graph.

If step (1) fails to surface the correct entity, no subsequent query can recover. We observe cases where agents execute 10+ tool calls across 9 iterations, generating syntactically valid queries, yet fail because the initial entity search returned incorrect node IDs. For example, item 3 in our benchmark ("gene involved in vesicle transport, located in kinetochore, in antigen processing pathway") required finding entity ID 5585, which never appeared in any of 11 search results despite extensive query reformulation.

This bottleneck suggests that improving entity resolution quality—via better embeddings, synonym expansion, fuzzy matching, or hybrid lexical-semantic retrieval—may yield larger gains than improving LLM reasoning capacity for query generation.

C. Diminishing Returns from Model Scale

The observation that GPT-5 substantially improves Search-Only but yields marginal or no improvement for structured-query agents is theoretically significant. We hypothesize two contributing factors:

1) **Query generation saturation**: GPT-5-mini already generates correct SPARQL/SQL for most queries. Additional reasoning capacity provides diminishing returns.
2) **Bottleneck shift**: With structured queries, performance is limited by entity resolution and knowledge graph coverage—factors orthogonal to LLM scale.

This finding aligns with recent text-to-SQL research showing that smaller models can achieve competitive accuracy when provided with adequate schema context [42]. From a practical standpoint, smaller models are preferable for structured-query agents, reserving larger models for pure semantic retrieval where their advantages are realized.

D. Multi-Tool Agents: Redundancy over Complementarity

As noted in Section IV.D, Search+SQL+SPARQL agents achieve the highest Hit@5 but do not dominate on precision or F1. This is surprising: one might expect that providing access to both a graph-native query language (SPARQL) and a relational query language (SQL) would yield complementary benefits, with each excelling on different query types.

However, the item-level correlation analysis (Section IV.D.e, Table IV) reveals that SQL and SPARQL are largely **redundant** rather than complementary. The per-item F1 correlation between Search+SPARQL and Search+SQL is $r = 0.92$ (GPT-5) and $r = 0.90$ (GPT-5-mini)—indicating that these configurations succeed and fail on nearly identical items. Only 12.8% of items show complementary behavior (one language succeeds while the other fails); the remaining 87.2% are redundant (both succeed or both fail).

When both tools are available, the agent does not consistently leverage one for graph patterns and the other for aggregations—instead, it faces additional decision overhead without gaining new expressive power. We attribute the multi-tool agent's failure to dominate to three factors:

- **Tool selection overhead**: Choosing between SPARQL and SQL introduces an additional decision point that can lead to suboptimal selections. Notably, when given both options, the agent uses SQL exclusively 59% of the time despite SPARQL achieving higher overall F1.
- **Redundant tool calls**: Agents sometimes invoke both query tools on the same subproblem, adding latency without improving recall.
- **Prompt complexity**: More tool options increase prompt length and complexity, potentially degrading query formulation quality.

These findings suggest that practitioners should choose **one** structured query language based on infrastructure constraints rather than providing both. Multi-tool agents may still benefit recall-critical applications where exhaustive entity discovery outweighs precision, but the expected complementarity between SQL and SPARQL does not materialize in practice.

E. SPARQL vs. SQL: Practical Equivalence

As shown in Section IV.D, the F1 difference between SPARQL and SQL is small (+2–3 pp) and falls within the observed standard deviations across runs, so we treat the two languages as practically equivalent in retrieval quality. The high item-level correlation ($r > 0.9$, Table IV) confirms that they succeed and fail on largely the same items, indicating broad functional overlap for knowledge graph retrieval. Where the languages differ is in operational characteristics: SQL exhibits roughly 2–3$\times$ higher error rates and substantially higher zero-result rates (50–63% vs.
25–53% for SPARQL), indicating that **SQL**'s multi-table join semantics and explicit foreign-key navigation pose a harder generation target for **LLMs** than **SPARQL**'s triple-pattern syntax. These differences have practical deployment implications:

- Organizations with existing SQL infrastructure can achieve competitive retrieval quality without deploying dedicated SPARQL endpoints, though they should expect somewhat higher query failure rates.
- SPARQL's lower error and zero-result rates make it the safer default when both options are available.
- The choice between SPARQL and SQL should be driven primarily by infrastructure constraints and developer familiarity, with SPARQL preferred when no strong infrastructure reason favors SQL.

F. Design Recommendations for Agentic-RAG Systems

Based on our findings, we offer the following recommendations for practitioners building Agentic-RAG systems over mix-typed knowledge bases:

1) **Always include structured query access** when the knowledge base supports it. The performance gap over pure semantic search is large (+15.5 pp F1) and consistent across model scales.
2) **Invest in entity resolution quality**. This is the primary bottleneck for structured-query agents. Consider:
    - Fine-tuning entity embeddings on domain-specific corpora.
    - Implementing synonym expansion and alias matching.
    - Hybrid lexical-semantic retrieval (e.g., BM25 + dense retrieval fusion).
3) **Use smaller models for structured-query agents**. GPT-5-mini achieves comparable or better F1 than GPT-5 at half the latency and cost.
4) **Avoid redundant tool sets**. In our setup, **SQL** and **SPARQL** overlap almost entirely in expressive power over PrimeKG's compact ontology: the item-level correlation exceeds $r > 0.9$ and the combined agent's Hit@5 improvement (+2 pp) is not statistically significant. Providing both adds decision overhead without meaningful recall gain. When only one language is deployed, **SPARQL**'s lower error and zero-result rates give it an operational advantage, but the end-to-end retrieval quality is comparable. For similar settings, practitioners should select based on infrastructure constraints and team familiarity. This finding is specific to functionally overlapping tools on a small, regular schema and does not generalize to multi-tool agents whose tools serve genuinely different purposes.
5) **Mitigate missed opportunities at the answer-synthesis stage**. Our *missed opportunity rate* (14.2%) shows that correct entities already appear in intermediate tool outputs but are dropped from the final prediction. Several strategies could address this in future work:
    - *Re-ranking pass*: After the agent loop terminates, a dedicated cross-encoder or **LLM** re-ranker scores every entity surfaced during tool execution against the original query and includes all entities above a confidence threshold, preventing premature filtering.
    - *Validation agent*: A second, lightweight agent inspects the candidate answer set alongside the full tool-call history and decides whether any retrieved-but-excluded entities should be reinstated — effectively adding a self-critique step to the pipeline.
    - *Union-by-default aggregation*: Rather than requiring the reasoning loop to explicitly select entities, the system could default to the union of all entities returned by tool calls, shifting the

burden from recall (include enough) to precision (prune irrelevant), which is an easier task for a final verification step.

G. Limitations

Our study has several limitations that suggest directions for future work:

- **Single domain**: Evaluation is limited to the biomedical domain (PrimeKG). Results may differ on other knowledge graphs with different schema complexity, entity distributions, or query patterns.
- **Single benchmark**: We evaluate on STaRK Human QA ($n = 109$). Larger-scale evaluation on diverse benchmarks would strengthen generalizability claims.
- **Single model family**: Only GPT-5 variants are compared. Open-source models (LLaMA, Mistral) or other commercial models (Claude, Gemini) may exhibit different performance characteristics.
- **Entity retrieval only**: We evaluate entity set retrieval, not end-to-end question answering with natural-language generation. Answer synthesis quality is an orthogonal concern.
- **Controlled concurrency**: Latency measurements use concurrency=5, which may not reflect production deployment conditions with varying load patterns.
- **Static knowledge graph**: We do not evaluate performance on evolving knowledge graphs or the impact of knowledge staleness.
- **Prompt sensitivity**: Our system prompt (Section VII.D) is extensively engineered with retry limits, parallelization instructions, and output format constraints. We do not evaluate sensitivity to prompt phrasing; different formulations may yield different performance characteristics.
- **No statistical significance testing**: Despite running 6 repetitions per configuration, we report only means and standard deviations without formal significance tests. Differences of 2–3 pp between SPARQL and SQL fall within observed standard deviations, so the SPARQL advantage may not be statistically significant.

VI. Conclusion

We presented an agentic RAG system that combines semantic vector search with schema-bound query execution (SQL and SPARQL) over the STaRK-Prime biomedical knowledge graph. Through a controlled comparison of four agent configurations across two model scales and six repetitions, we established three main findings. First, structured query access provides a substantial advantage over pure semantic search (+15.5 pp F1 in our within-system comparison), confirming that multi-hop relational queries exceed the capabilities of embedding-based retrieval; combined with GPT-5-class backbones, this approach also surpasses all published STaRK baselines by a wide margin. Second, SQL and SPARQL achieve practically equivalent retrieval quality ($r > 0.9$ item-level correlation, ≤ 3 pp F1 difference), though SPARQL exhibits lower error and zero-result rates; practitioners should choose one query language based on infrastructure constraints rather than providing both. Third, the dominant bottleneck for structured-query agents is entity resolution, not LLM reasoning capacity: scaling from GPT-5-mini to GPT-5 yields minimal improvement when structured query tools are available, while 34.9% of queries fail due to incomplete entity coverage during semantic search.

These results suggest that future work should prioritize improving entity resolution quality—through better embeddings, synonym expansion, or hybrid lexical-semantic retrieval—over more sophisticated query generation or larger language models. Extending this evaluation to additional domains, knowledge graph schemas, and open-source model families would further strengthen the generalizability of our findings.

References

[1] N. Sommerfeld, R. Dave, and D. Webster-Clark, "Marketing's New Middleman: AI Agents," Industry Report, Feb. 2025. [Online]. Available: https://www.bain.com/insights/marketings-new-middleman-ai-agents/

[2] C. Kaiser, J. Kaiser, R. Schallner, and S. Schneider, "A New Era of Online Search? A Large-Scale Study of User Behavior and Personal Preferences during Practical Search Tasks with Generative AI versus Traditional Search Engines," in *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, in CHI EA '25.: Association for Computing Machinery, 2025. doi: 10.1145/3706599.3720123.

[3] Y. Gao *et al.*, "Retrieval-Augmented Generation for Large Language Models: A Survey." [Online]. Available: https://arxiv.org/abs/2312.10997

[4] X. Yu, P. Jian, and C. Chen, "TableRAG: A Retrieval Augmented Generation Framework for Heterogeneous Document Reasoning," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, Eds., Suzhou, China: Association for Computational Linguistics, Nov. 2025, pp. 14063–14082. doi: 10.18653/v1/2025.emnlp-main.710.

[5] A. Biswal *et al.*, "Text2SQL is Not Enough: Unifying AI and Databases with TAG." [Online]. Available: https://arxiv.org/abs/2408.14717

[6] A. Singh, A. Ehtesham, S. Kumar, and T. T. Khoei, "Agentic Retrieval-Augmented Generation: A Survey on Agentic RAG." [Online]. Available: https://arxiv.org/abs/2501.09136

[7] Q. Sun *et al.*, "Docs2KG: A Human-LLM Collaborative Approach to Unified Knowledge Graph Construction from Heterogeneous Documents," in *Companion Proceedings of the ACM on Web Conference 2025*, in WWW '25. Sydney NSW, Australia: Association for Computing Machinery, 2025, pp. 801–804. doi: 10.1145/3701716.3715309.

[8] J. Bai *et al.*, "AutoSchemaKG: Autonomous Knowledge Graph Construction through Dynamic Schema Induction from Web-Scale Corpora." [Online]. Available: https://arxiv.org/abs/2505.23628

[9] S. Wu *et al.*, "STaRK: Benchmarking LLM Retrieval on Textual and Relational Knowledge Bases," in *Advances in Neural Information Processing Systems*, 2024.

[10] P. Chandak, K. Huang, and M. Zitnik, "Building a knowledge graph to enable precision medicine," *Scientific Data*, vol. 10, no. 1, p. 67, 2023.

[11] S. Wu *et al.*, "AvaTaR: Optimizing LLM Agents for Tool Usage via Contrastive Reasoning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[12] J. Li *et al.*, "Can LLM Already Serve as a Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[13] Z. Hong *et al.*, "Next-Generation Database Interfaces: A Survey of LLM-Based Text-to-SQL," *IEEE Transactions on Knowledge and Data Engineering*, vol. 37, pp. 7328–7345, 2025, doi: 10.1109/TKDE.2025.3609486.

[14] X. Liu *et al.*, "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, vol. 37, pp. 5735–5754, 2025, doi: 10.1109/TKDE.2025.3592032.

[15] Z. Cao, Y. Zheng, Z. Fan, X. Zhang, W. Chen, and X. Bai, "RSL-SQL: Robust Schema Linking in Text-to-SQL Generation," in *arXiv preprint*, 2024.

[16] L. Wang *et al.*, "Proton: Probing Schema Linking Information from Pre-trained Language Models for Text-to-SQL Parsing," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2022, pp. 1889–1898. doi: 10.1145/3534678.3539305.

[17] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[18] D. Gao *et al.*, "Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation," *Proceedings of the VLDB Endowment*, vol. 17, no. 5, pp. 1132–1145, 2024, doi: 10.14778/3641204.3641221.

[19] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, "Execution-Guided Neural Program Decoding," in *ICML Workshop on Neural Abstract Machines and Program Induction (NAMPI)*, 2018.

[20] 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 (EMNLP)*, 2021. doi: 10.18653/v1/2021.emnlp-main.779.

[21] H. Li *et al.*, "OmniSQL: Synthesizing High-quality Text-to-SQL Data at Scale," *Proceedings of the VLDB Endowment*, vol. 18, no. 11, pp. 4695–4709, 2025, doi: 10.14778/3749646.3749723.

[22] T. Yu *et al.*, "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 (EMNLP)*, 2018, pp. 3911–3921. doi: 10.18653/v1/D18-1425.

[23] C. Kosten, P. Cudré-Mauroux, and K. Stockinger, "Spider4SPARQL: A Complex Benchmark for Evaluating Knowledge Graph Question Answering Systems," in *IEEE International Conference on Big Data*, 2023. doi: 10.1109/BigData59044.2023.10386182.

[24] C. Kosten, F. Nooralahzadeh, and K. Stockinger, "Evaluating the Effectiveness of Prompt Engineering for Knowledge Graph Question Answering," *Frontiers in Artificial Intelligence*, vol. 7, 2025, doi: 10.3389/frai.2024.1454258.

[25] J. Reif, T. Jeleniewski, M. S. Gill, F. Gehlhoff, and A. Fay, "Chatbot-Based Ontology Interaction Using Large Language Models and Domain-Specific Standards," in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2024. doi: 10.1109/ETFA61755.2024.10711065.

[26] M. H. Rasheed and M. Aguado, "LLM-Based Natural Language to SPARQL Translation over Domain-Specific Knowledge Graph," *Knowledge Organization*, 2025, doi: 10.31083/ko42705.

[27] I.-V. Hernandez-Camero, E. García-López, A. Garcia-Cabot, and S. Caro-Álvaro, "Context-Aware Few-Shot Learning SPARQL Query Generation from Natural Language on an Aviation Knowledge Graph," *Machine Learning and Knowledge Extraction*, vol. 7, no. 2, p. 52, 2025, doi: 10.3390/make7020052.

[28] R. Cai, J. Yuan, B. Xu, and Z. Hao, "SADGA: Structure-Aware Dual Graph Aggregation Network for Text-to-SQL," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[29] L. Jiang, J. Huang, C. Moller, and R. Usbeck, "Ontology-Guided, Hybrid Prompt Learning for Generalization in Knowledge Graph Question Answering," in *IEEE International Conference on Semantic Computing (ICSC)*, 2025. doi: 10.1109/ICSC64641.2025.00010.

[30] X. Pan, V. de Boer, and J. van Ossenbruggen, "FIRESPARQL: A LLM-based Framework for SPARQL Query Generation over Scholarly Knowledge Graphs," in *IC3K*, 2025. doi: 10.5220/0013774000004000.

[31] V. Emonet, J. T. Bolleman, S. Duvaud, T. M. d. Farias, and A. C. Sima, "LLM-based SPARQL Query Generation from Natural Language over Federated Knowledge Graphs." 2024.

[32] S. Walter and H. Bast, "GRASP: Generic Reasoning And SPARQL Generation across Knowledge Graphs," in *The Semantic Web – ISWC 2025*, in LNCS, vol. 16140. 2025, pp. 271–289. doi: 10.1007/978-3-032-09527-5_15.

[33] M. Arazzi, D. Ligari, S. Nicolazzo, and A. Nocera, "Augmented Knowledge Graph Querying leveraging LLMs," in *International Joint Conference on Neural Networks (IJCNN)*, 2025. doi: 10.1109/IJCNN64981.2025.11229060.

[34] C. V. S. Avila, V. Vidal, W. Franco, and M. A. Casanova, "Few-Shot Learning or RAG in LLM-Based Text-to-SPARQL? Why Not Both?," in *IEEE International Conference on Semantic Computing (ICSC)*, 2025. doi: 10.1109/ICSC64641.2025.00016.

[35] Z. Shen *et al.*, "GeAR: Graph-enhanced Agent for Retrieval-Augmented Generation." 2024.

[36] J. Xu, M. Li, Y. Tang, P. Wang, and W. Zhang, "Towards Open-World Retrieval-Augmented Generation on Knowledge Graph: A Multi-Agent Collaboration Framework," in *Proceedings of the ACM Web Conference (WWW)*, 2025. doi: 10.1145/3774904.3792389.

[37] R. Sapkota, K. I. Roumeliotis, and M. Karkee, "AI Agents vs.\ Agentic AI: A Conceptual Taxonomy, Applications and Challenges," *Information Fusion*, 2025, doi: 10.1016/j.inffus.2025.103599.

[38] P. Liu *et al.*, "HM-RAG: Hierarchical Multi-Agent Multimodal Retrieval Augmented Generation." 2025.

[39] Y. Chen *et al.*, "MAO-ARAG: Multi-Agent Orchestration for Adaptive Retrieval-Augmented Generation." 2025.

[40] Q. Wu *et al.*, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation." 2023.

[41] J. Tang, T. Fan, and C. Huang, "AutoAgent: A Fully-Automated and Zero-Code Framework for LLM Agents." 2025.

[42] A. Mohammadjafari, A. S. Maida, and R. Gottumukkala, "From Natural Language to SQL: Review of LLM-based Text-to-SQL Systems." [Online]. Available: https://arxiv.org/abs/2410.01066

VII. Appendix A: Illustrative Examples

The following examples ground the discussion findings in concrete query executions. We present one successful retrieval demonstrating structured query advantages (Section VII.A), one failure case illustrating the entity resolution bottleneck (Section VII.B), and one missed opportunity case (Section VII.C).

A. A.1 Successful Multi-Hop Retrieval

TABLE VIII
Successful multi-hop retrieval requiring entity resolution followed by graph traversal.

Query: *"I have pharyngitis and chemosis. What skin disease might I have?"*
Gold Answer: Toxic epidermal necrolysis (ID: 32299)
Agent Execution (Search+SPARQL, GPT-5):

Iter	Tool	Action & Result
1	search_entities	Query: "pharyngitis" → Pharyngitis (ID: 26964)
1	search_entities	Query: "chemosis" → Chemosis (ID: 89294)
1	search_entities	Query: "skin disease" → skin disease (ID: 36644)
2	execute_sparql	Find diseases that are subtypes of skin disease AND have both phenotypes → **32299**
3	execute_sparql	Retrieve name for ID 32299 → "toxic epidermal necrolysis"

Result: Precision=100%, Recall=100%, F1=100%, Latency=62s

This example illustrates several key findings:

1) **Entity resolution as prerequisite**: The agent first resolves natural language terms ("pharyngitis", "chemosis", "skin disease") to knowledge graph node IDs via semantic search. Without this step, no structured query is possible.
2) **Structured query enables relational composition**: The SPARQL query expresses a constraint that would be impossible with pure semantic search: find entities that are (a) subtypes of skin disease AND (b) linked to both phenotype nodes via phenotypePresent edges. This multi-hop, multi-constraint pattern is precisely where structured queries excel.
3) **Search-Only would fail**: Semantic search over entity descriptions cannot discover "toxic epidermal necrolysis" from the query "pharyngitis and chemosis" because the disease name shares no lexical overlap with the symptoms. The relationship exists only in the graph structure.

B. A.2 Entity Resolution Failure

TABLE IX
Entity resolution failure: extensive tool use cannot compensate for missing the correct entity in initial search.

Query: *"Which gene is involved in vesicle transport and located in kinetochore, and in the pathway of antigen processing?"*
Gold Answer: Gene ID 5585
Agent Execution (Search+SPARQL, GPT-5):

Iter	Tool	Action & Result
1	search_entities	Query: "vesicle transport" → 5 biological processes
1	search_entities	Query: "kinetochore" → 5 cellular components
1	search_entities	Query: "antigen processing" → 5 pathways
2	execute_sparql	Find genes linked to all three → **0 rows**
3	execute_sparql	Relaxed query → **0 rows**
4	search_entities	Expanded search (top_k=15) for all three terms
5	execute_sparql	Query with expanded IDs → **0 rows**
6	execute_sparql	Alternative predicate patterns → **0 rows**
7	execute_sparql	Further relaxation → **0 rows**
8	search_entities	Direct gene search: "kinetochore vesicle transport antigen processing gene"
9	execute_sparql	Final attempt with candidate genes → **0 rows**

Result: Precision=0%, Recall=0%, F1=0%, Latency=146s
Failure Analysis:

- Total tool calls: 17 (11 searches, 6 SPARQL queries)
- All 6 SPARQL queries returned 0 rows
- Gold entity ID 5585 **never appeared** in any of the 21 unique IDs returned by search
- Coverage: 0% (entity resolution failed completely)

This failure case demonstrates critical limitations:

1) **Entity resolution bottleneck**: Despite 11 search queries with various formulations, the correct gene (ID 5585) never appeared in any result set. The agent's SPARQL queries were syntactically valid and logically correct, but operated on the wrong set of candidate entities.
2) **Diminishing returns from iteration**: The agent executed 9 iterations and 17 tool calls over 146 seconds, yet made no progress toward the answer. Additional reasoning effort cannot overcome a fundamental retrieval gap.
3) **Query formulation was correct**: The SPARQL queries correctly expressed the multi-constraint pattern (gene linked to vesicle transport process, kinetochore component, and antigen processing pathway). The failure mode is not **LLM** reasoning but knowledge base access.

4) **Implications for system design**: This example motivates investment in entity resolution quality (better embeddings, synonym expansion, fuzzy matching) rather than more sophisticated query generation or additional LLM reasoning steps.

C. A.3 Missed Opportunity (Found but Not Returned)

TABLE X
Missed opportunity: agent discovers correct entity but fails to include it in final prediction.

Query: *"Mixed mucinous and nonmucinous bronchioloalveolar adenocarcinoma is a subtype of what disease?"*
Gold Answers: IDs 95312 (the disease itself) and 35882 (minimally invasive lung adenocarcinoma)
Agent Execution:

Iter	Tool	Action & Result
1	search_entities	Found ID **95312** (the query disease) ranked #1
2	execute_sparql	Query parentChild relationship → returned ID **35882**

Final Prediction: [35882] (only the parent disease)
Result: Precision=100%, Recall=50%, F1=67% \| Missed opportunity rate: 50%
Analysis: The agent discovered both gold entities (95312 via search, 35882 via SPARQL) but only returned the SPARQL result. Entity 95312 was found but not included in the final answer.

This example illustrates the **incomplete result selection** failure mode:

1) **Full coverage achieved**: Both gold entities were discovered during tool execution (coverage = 100%).
2) **Filtering failure**: The agent correctly identified 95312 as the query entity but interpreted the question as asking only for the *parent* disease, not the disease hierarchy. This represents a reasoning failure at the answer synthesis stage.
3) **Actionable improvement**: Explicit re-ranking or verification of intermediate results—checking whether discovered entities should be included in the final answer—could address this failure mode without improving entity resolution or query generation.

D. A.4 Agent System Prompt

The following is the complete system prompt used for the Search+SQL+SPARQL agent configuration. This prompt is instantiated at runtime with the actual database schema and vocabulary information. The {schema_and_vocab} and {max_rows} placeholders are replaced at runtime with the actual database schema and query limits.

```
You are an expert biomedical knowledge base analyst. Your task is to answer
questions about diseases, drugs, genes/proteins, pathways, molecular functions,
and their relationships using the STaRK-Prime knowledge base.

## Available Tools

You have access to THREE tools:

1. **search_entities_tool** - Semantic search to find entities by name/description
  - Use this FIRST to resolve entity names to their IDs
  - Handles synonyms, partial matches, and related terms
  - Returns entity IDs that you can use in queries

2. **execute_sql_query_tool** - Execute SQL queries (PostgreSQL)
   - Use AFTER finding entity IDs with search_entities_tool
   - Best for aggregations, joins, and filtering

3. **execute_sparql_query_tool** - Execute SPARQL queries (Fuseki)
   - Use AFTER finding entity IDs with search_entities_tool
   - Best for path traversal and pattern matching

## Two-Stage Query Process (IMPORTANT!)

**ALWAYS follow this two-stage approach:**

### Stage 1: Entity Resolution
When a question mentions specific entities (diseases, drugs, genes, etc.):
1. Use `search_entities_tool` to find the entity IDs
2. Note the returned IDs for use in your query

Example: For "What genes are associated with both diabetes and hypertension?"
-> First: search_entities_tool("diabetes", "disease")
        search_entities_tool("hypertension", "disease")
-> Get: diabetes ID 12345, hypertension ID 67890

### Stage 2: Query Execution
Use the resolved entity IDs in your SQL or SPARQL query.

## Exploration Strategy

### Strategy 1: Queries without explicit entity mentions
Use `search_entities_tool` with the full question as the query and `top_k=15`.

### Strategy 2: Queries with explicit entity mentions
1. Resolve each entity with `search_entities_tool` (use `entity_type` when obvious).
2. Proceed to query execution with the resolved IDs.

### Strategy 3: Multi-entity or complex queries
1. Disambiguate all mentioned entities.
2. Explore neighborhoods of key entities with relevant filters.
3. Combine information from multiple exploration paths.

## Query Language Selection

- **SQL** is better for: lookups, aggregations (COUNT, AVG, MAX), joins,
  complex filters, questions asking "how many" or "list all"
- **SPARQL** is better for: path traversal, relationship exploration,
  pattern matching, questions like "what is connected to X through Y"

## Knowledge Base Schema

{schema_and_vocab}

## Query Guidelines

1. **Entity Resolution First**: ALWAYS use search_entities_tool to find entity
   IDs before querying. Do NOT try to match entity names with SQL LIKE or
   SPARQL FILTER - use semantic search instead.
2. **Read-only only**: SQL must be SELECT-only; SPARQL must be read-only.
3. **Limit results**: Always use LIMIT {max_rows}.
4. **Be precise**: Use exact table/column names from the schema above.
5. **Handle errors**: If a query fails, analyze the error and try again.
6. **Node IDs are answers**: The benchmark expects node IDs (integers).

## Answer Format

Output ONLY a JSON object with exactly two fields:
{"ids": [123, 456, 789], "reasoning": "Found 3 genes associated with diabetes"}

- The `ids` field must be an array of integers, empty array [] if no results
- The `reasoning` field must be a string explaining your process
- IDs preserve ranking order for Hit@1 and MRR metrics

## Efficiency Guidelines

### Parallelization
- Resolve multiple entities IN PARALLEL in ONE turn (max 4-5 searches)

### Retry Limits
- Query errors: max 2 corrective attempts per query
- Zero-row results: max 2 query reformulations
- Hard cap: 6 tool rounds total. If reached, answer with best available IDs

### Answer Strategy
- If a query returns 0 rows, adjust and retry (within retry limits)
- Partial results are valuable - report entity IDs even without relationship data

## Query Examples

Example SQL workflow:
1. search_entities_tool("breast cancer", "disease") -> ID: 789
2. execute_sql_query_tool("SELECT dst_id FROM indication WHERE src_id = 789")

Example SPARQL workflow:
1. search_entities_tool("insulin", "gene_protein") -> ID: 456
2. execute_sparql_query_tool("PREFIX sp: <http://stark.stanford.edu/prime/>
   SELECT ?related WHERE { sp:node/456 sp:associatedWith ?related } LIMIT 5")

Now answer the user's question using the two-stage approach.
```