Implementing Global Search in Knowledge Graph RAG: A Neo4j and Python Architecture

Executive Summary

The rapid integration of Large Language Models (LLMs) into enterprise data systems has given rise to Retrieval-Augmented Generation (RAG) as a dominant paradigm for grounding AI responses in proprietary data. While traditional RAG architectures—predicated on vector similarity search—have proven highly effective for retrieving specific facts (Local Search), they exhibit significant limitations when tasked with holistic, corpus-wide reasoning (Global Search). This "sensemaking gap" prevents systems from answering high-level questions about trends, themes, and overarching narratives within a dataset, such as a university course curriculum.

To address this, Microsoft Research has introduced GraphRAG, a methodology that superimposes a hierarchical community structure onto a knowledge graph. By clustering entities into semantic communities and pre-generating summaries for these clusters, GraphRAG enables an LLM to reason over the entire dataset at varying levels of abstraction. This report investigates the implementation of this Global Search mechanism within a Neo4j environment, utilizing the graphdatascience (GDS) Python library.

The analysis establishes that the **Leiden algorithm** is theoretically and practically superior to the Louvain algorithm for this application due to its guarantee of connected communities, which is critical for coherent generative summarization. We define a robust implementation strategy that leverages Neo4j GDS to execute hierarchical clustering, extracting a multi-level index of community IDs (Level 0, 1, 2, etc.). Furthermore, we detail the architecture of a "Map-Reduce" summarization engine and a "Dynamic Community Selection" mechanism that optimizes token usage by up to 77% compared to exhaustive search methods. Finally, an automation strategy using asynchronous task orchestration is proposed to trigger this pipeline upon course upload, ensuring that educational content is immediately transformed into a structured, queryable knowledge asset.

---

1. The Evolution of Retrieval-Augmented Generation

The field of Natural Language Processing (NLP) has shifted from purely generative models, which hallucinate when forced to recall specific facts, to retrieval-augmented architectures that ground generation in retrieved evidence. However, the mechanism of retrieval dictates the capabilities of the system. To understand the necessity of GraphRAG and Global Search,

one must first analyze the inherent structural limitations of the prevailing vector-based paradigms.

1.1 The Baseline: Vector-Based Retrieval and Its Limits

Standard RAG architectures, often termed "Naive RAG," rely primarily on vector embeddings to map user queries to relevant text chunks. In this workflow, a document (e.g., a course syllabus or transcript) is sliced into discrete segments. These segments are passed through an embedding model to generate dense vector representations, which are stored in a vector database. When a user queries the system, the query is similarly embedded, and a K-Nearest Neighbor (KNN) search identifies the text chunks with the highest cosine similarity.

This approach constitutes a "Local Search" mechanism. It is highly effective for "needle-in-a-haystack" queries where the answer is explicitly contained within a few contiguous sentences. For instance, if a student asks, "What is the prerequisite for the Advanced Algorithms module?", a vector-based system can easily retrieve the specific paragraph mentioning prerequisites.[1] The semantic overlap between the query terms and the source text is high, ensuring accurate retrieval.

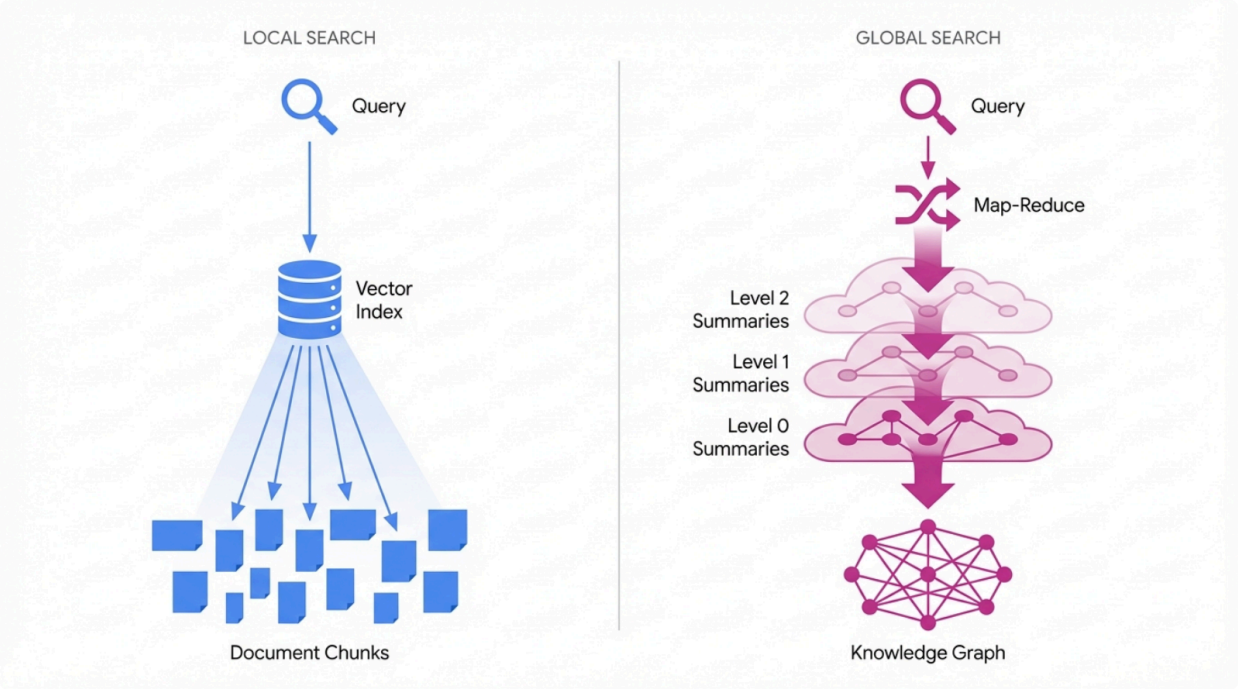
However, this architecture collapses when the user's intent is exploratory, summative, or holistic. Global queries—such as "What are the primary pedagogical themes evolving across the entire Computer Science curriculum?" or "How does the course's approach to ethics contradict its technical instruction?"—do not map to a single text chunk. The answer is not explicit in the data; it is emergent.[2] It requires the synthesis of information scattered across hundreds of documents, lectures, and assignments. A vector search will simply retrieve the top-k chunks that mention "themes" or "curriculum," resulting in a fragmented, repetitive, and often irrelevant set of contexts that fails to capture the "forest" because it is designed to find only specific "trees".[1]

1.2 The "Sensemaking Gap" in Unstructured Data

This limitation is formally described as the "sensemaking gap." It arises because vector search operates on the assumption of **semantic proximity**, not **structural connectivity**. In a large dataset, relevant information is often topologically distant but logically connected.

For example, a concept introduced in Lecture 1 (e.g., "Gradient Descent") might be critiqued in Lecture 12 ("Overfitting") and applied in a Final Project ("Neural Network Optimization"). A user asking for a summary of "Optimization Techniques" needs a system that can traverse these temporal and conceptual distances. Naive RAG, restricted by a limited context window and a local retrieval mechanism, cannot "read" the entire dataset to form this summary. It lacks a pre-computed index of high-level abstractions.[4]

Architectural Divergence: Local Fact Retrieval vs. Global Holistic Synthesis



The Local Search architecture (left) utilizes a direct vector lookup against raw text chunks, limiting context to the 'top-k' results. The Global Search architecture (right) introduces an intermediate 'Community Layer.' The query interacts with pre-summarized community nodes arranged hierarchically, allowing the system to synthesize answers from across the entire dataset structure before referencing specific entities.

1.3 The GraphRAG Paradigm Shift

To bridge this gap, Microsoft Research proposed GraphRAG, a method that restructures the retrieval process around a **Knowledge Graph (KG)**. In this paradigm, the fundamental unit of retrieval shifts from the "text chunk" to the "Community Summary".[2]

The methodology involves three distinct phases:

1. **Indexing (Graph Construction):** The system extracts entities (nodes) and relationships (edges) from the raw text, creating a structured representation of the data.[1]
2. **Hierarchical Semantic Clustering:** Algorithms partition the graph into communities of closely related entities. These communities are detected at multiple levels of granularity—from broad, high-level topics (Level 0) to specific sub-clusters (Level 2 or 3).[6]
3. **Community Summarization:** An LLM generates a comprehensive summary for *each* detected community. These summaries act as a "pre-read" of the dataset, effectively

compressing the semantic information of the graph into a retrievable format.[6]

When a global query is received, the system utilizes a **Map-Reduce** framework. It maps the query against the hierarchy of community summaries (rather than raw text), selects the relevant high-level themes, and reduces them into a coherent global answer. This allows the LLM to reason over the structure of the data, providing answers that are both comprehensive and grounded in the global context of the uploaded course materials.[4]

---

2. Theoretical Foundations of Community Detection

The efficacy of Global Search is entirely dependent on the quality of the communities detected within the graph. If the clusters are incoherent, containing unrelated entities, the resulting summaries will be flawed, leading to hallucinations in the final answer. Therefore, the selection and configuration of the community detection algorithm are critical architectural decisions.

2.1 Modularity Maximization and the Louvain Algorithm

For over a decade, the **Louvain algorithm** has been the standard for community detection in large networks. It operates by optimizing "modularity"—a scalar value between -1 and 1 that measures the density of links inside communities compared to links between communities.[7]

The Louvain method is a greedy, hierarchical optimization algorithm. It proceeds in two steps that are repeated iteratively:

1. **Modularity Optimization:** The algorithm iterates through nodes, moving each node to the community of its neighbors if that move increases the global modularity score.
2. **Community Aggregation:** Once local modularity is optimized, the communities are collapsed into single "super-nodes," creating a new, coarser network. The process then repeats on this coarser network.[8]

While Louvain is fast and scalable, it suffers from a critical theoretical flaw relevant to generative AI: **it does not guarantee the internal connectivity of communities.** In its quest to maximize the mathematical modularity score, Louvain may merge two distinct, unconnected sub-graphs into a single community. For example, in a "World History" knowledge graph, Louvain might group a cluster of nodes regarding "The Aztec Empire" with a cluster regarding "Feudal Japan" simply because merging them increases the global modularity, even if no direct relationship exists between them.[7]

In the context of RAG, this is disastrous. If an LLM is tasked with summarizing this disconnected community, it will attempt to find a narrative thread connecting the Aztecs and the Japanese shogunate, likely resulting in a hallucinated or nonsensical summary.

2.2 The Leiden Algorithm: Recursive Refinement

The **Leiden algorithm** was developed specifically to address the connectivity defects of Louvain. It modifies the Louvain approach by introducing a **refinement phase** between the modularity optimization and the aggregation steps.[7]

The Leiden process guarantees that **communities are well-connected**. If the algorithm detects that a community is becoming disconnected or loosely coupled during the optimization phase, the refinement step randomly breaks the community down into smaller, well-connected sub-communities before the aggregation step occurs.

- **Why Leiden is Superior for GraphRAG:** The guarantee of connectivity ensures that every entity within a community is reachable from every other entity (or at least highly structurally cohesive). This provides the LLM with a logically consistent set of facts, enabling the generation of coherent "executive summaries" without the need for creative fabrication.[9]

2.3 Hierarchical Clustering in Neo4j

Global Search requires navigating the data at different resolutions, or "zoom levels." A high-level summary (Level 0) provides a broad overview, while lower levels (Level 1, 2) provide granular details. Both Louvain and Leiden are inherently hierarchical algorithms that produce this structure naturally as they iterate.[9]

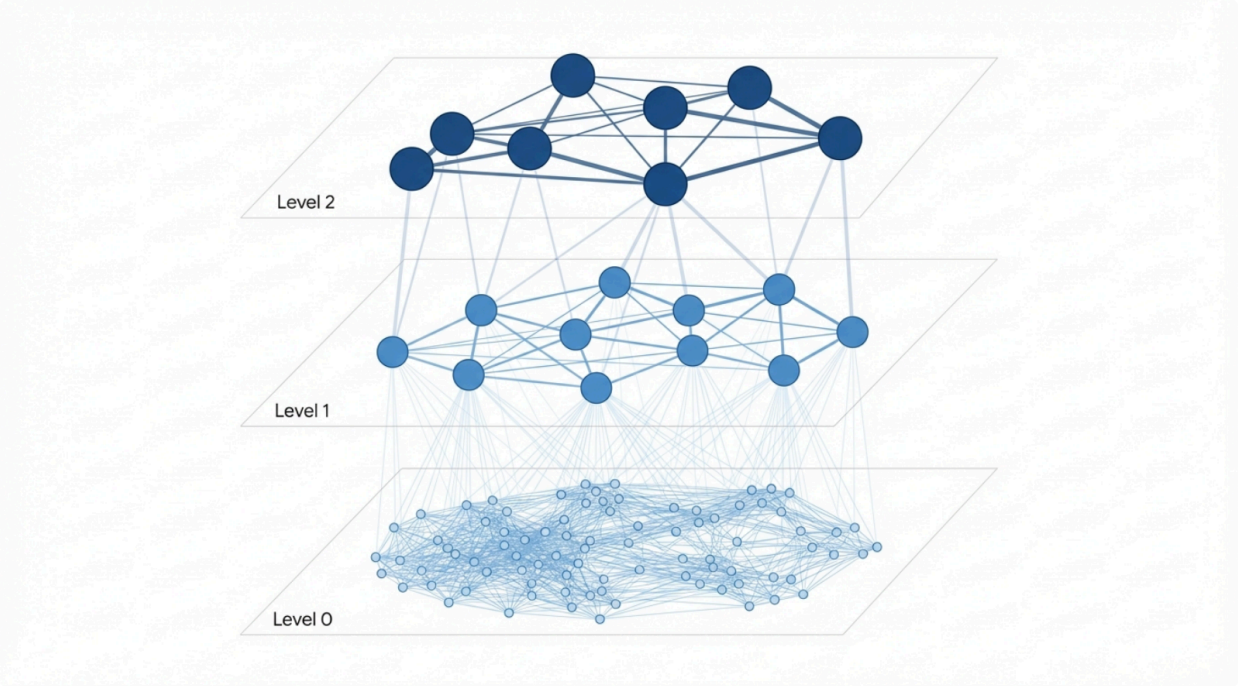
However, standard implementations often return only the *final* partition (the state at convergence). For Global Search, we require the intermediate partitions as well. The Neo4j Graph Data Science (GDS) library supports this via the includeIntermediateCommunities parameter.[9]

When includeIntermediateCommunities=True is set, the algorithm returns a list of community IDs for each node rather than a single integer. A node might have the property leiden_communities: , indicating:

- **Level 0 (Micro):** It belongs to Community 15.
- **Level 1 (Meso):** Community 15 is part of the larger Community 4.
- **Level 2 (Macro):** Community 4 is part of the massive Community 1.[9]

This vector of IDs forms the "address" of the entity within the semantic hierarchy, enabling the system to query the graph at any desired level of abstraction.

Hierarchical Abstraction: From Entities to Community Super-Nodes



The Leiden algorithm processes the graph in iterations. Level 0 represents the raw entity graph. Level 1 groups tightly connected entities into communities. Level 2 collapses these into super-nodes, revealing broader thematic structures. Global Search utilizes summaries from all levels to answer queries of varying granularity.

---

3. Architecture of the Global Search Pipeline

The implementation of Global Search involves a distinct pipeline that transforms raw course content into a summarized semantic index. This pipeline consists of three primary stages: Indexing, Summarization, and Retrieval.

3.1 The Indexing Phase: From Text to Graph

Before community detection can occur, the unstructured text must be converted into a graph. Upon the upload of a course file (e.g., a PDF of a textbook or a transcript of a lecture), the system performs the following:

1. **Chunking:** The document is split into manageable text units (e.g., 600 token chunks).
2. **Entity Extraction:** An LLM processes each chunk to extract entities (Concepts, People, Events) and relationships. For example, from the sentence "Neural Networks use

Backpropagation to minimize error," the LLM creates (:Concept {name: "Neural Networks"})-->(:Concept {name: "Backpropagation"}).[1]
3. **Graph Construction:** These entities and relationships are ingested into Neo4j. The result is a dense, interconnected web of knowledge representing the course content.

3.2 The Community Summarization Phase: Building the Semantic Layer

Once the graph is built, the "Community Detection" and "Summarization" processes begin. This is the core "Map" step of the Map-Reduce architecture.[11]

1. **Detection:** The Leiden algorithm runs on the projected graph, assigning hierarchical community IDs to every node.
2. **Aggregation:** The system iterates through every unique community at every level. For each community, it gathers all member entities and their descriptions.
3. **Summarization:** An LLM is prompted to generate a report for the community. The prompt specifically requests an **Executive Summary**, a list of **Key Findings**, and an **Impact Severity Rating** (a score of 0-10 indicating the community's importance).[11]

This process creates a new layer of "Community" nodes in the graph, each containing a high-level text summary of the underlying entities.

3.3 The Query Phase: Map-Reduce and Dynamic Selection

When a user submits a global query, the system does not search the raw entities. Instead, it targets the Community Summaries.

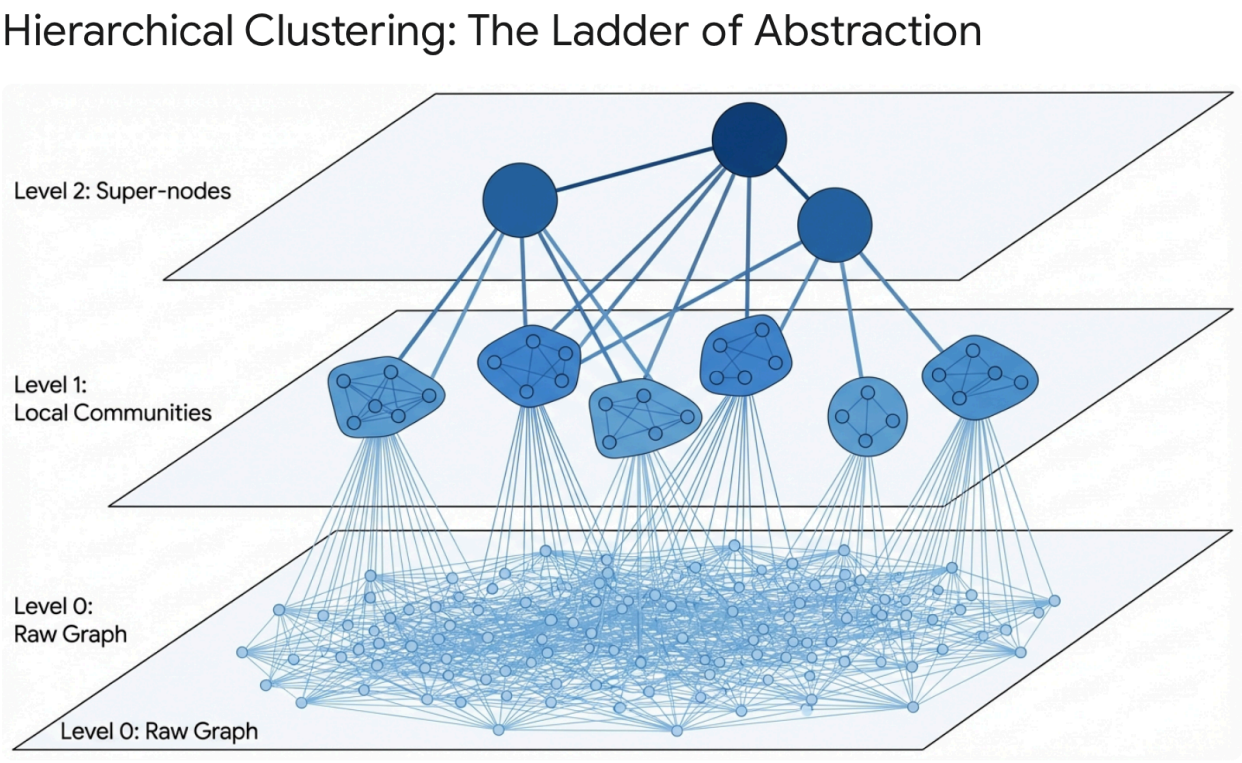
In the original GraphRAG paper, this was done via an exhaustive **Map-Reduce** approach: every community summary at a certain level (e.g., Level 1) was retrieved, processed by the LLM in parallel (Map), and then the results were combined (Reduce).[4] While comprehensive, this is computationally expensive and slow.

To optimize this, Microsoft introduced **Dynamic Community Selection**.[6]

1. **Relevance Rating:** The system first retrieves high-level (Level 2) community summaries. A lightweight LLM (or "Rater") evaluates each summary's relevance to the user's query, assigning a score.
2. **Pruning:** Communities with a score of 0 are discarded.
3. **Drill-Down:** For highly relevant communities, the system retrieves their child communities (Level 1) and repeats the rating process.
4. **Selection:** Only the summaries of the most relevant communities are passed to the final context window.

Data indicates that Dynamic Community Selection reduces the token load by approximately **77%** compared to exhaustive search, making Global Search economically viable for large

datasets.[6]



The Leiden algorithm organizes the Knowledge Graph into a hierarchical structure. 'Level 0' represents the raw entity graph. 'Level 1' groups tightly connected entities into local communities. 'Level 2' aggregates these communities into broader thematic super-nodes. Global Search utilizes summaries from different levels depending on the breadth of the user's query.

---

4. Technical Implementation with Neo4j and Python

This section provides a detailed technical blueprint for implementing the Global Search pipeline using the graphdatascience Python client. This client allows for pure Pythonic interaction with the GDS library, abstracting the complexity of Cypher procedures.[13]

4.1 Environment Setup and Prerequisites

The implementation requires:

- **Neo4j Database:** Version 5.x is recommended for full GDS compatibility.[14]
- **Graph Data Science (GDS) Library:** Installed on the Neo4j server.

- **Python Client:** graphdatascience package (pip install graphdatascience).

The connection is established as follows:

Python

```
from graphdatascience import GraphDataScience

# Initialize GDS connection
# Ideally, retrieve credentials from environment variables
gds = GraphDataScience("bolt://localhost:7687", auth=("neo4j", "password"))

# Verify connection and version
print(f"GDS Version: {gds.version()}")
```

4.2 Graph Projection Strategies for Course Data

In GDS, algorithms run on an in-memory projection of the graph, not the database directly. This improves performance and isolation. For a "Course Upload" scenario, we must project only the relevant subgraph. We utilize **Cypher Projection** to filter the graph by the specific Course ID. This ensures the community detection is context-aware and does not bleed into unrelated courses.[15]

Python

```
def project_course_graph(course_id, project_name):
    """
    Projects a subgraph for a specific course into GDS memory using Cypher Projection.
    """
    # Defensive programming: Drop the graph if it already exists to ensure a fresh state
    if gds.graph.exists(project_name)["exists"]:
        gds.graph.drop(gds.graph.get(project_name))

    # Node Projection Query
    # Selects all entities mentioned in chunks that belong to the target course
    node_query = f"""
    MATCH (n:Entity)-->(:Chunk)-->(:Course {{id: '{course_id}'}})
    RETURN id(n) as id, labels(n) as labels
```

```
    """

    # Relationship Projection Query
    # Selects relationships between valid entities. We treat relationships as UNDIRECTED
    # for community detection to capture bidirectional semantic associations.
    edge_query = f"""
    MATCH (n:Entity)-->(:Chunk)-->(:Course {{id: '{course_id}'}})
    MATCH (n)--(m:Entity)
    WHERE (m)-->(:Chunk)-->(:Course {{id: '{course_id}'}})
    RETURN id(n) as source, id(m) as target, type(r) as type
    """

    # Execute Projection
    G, result = gds.graph.project.cypher(
        project_name,
        node_query,
        edge_query
    )
    print(f"Projected Graph '{project_name}': {G.node_count()} nodes, {G.relationship_count()} edges.")
    return G
```

4.3 Implementing the Leiden Algorithm

Once projected, we execute the Leiden algorithm. We use the .write() mode to persist the results back to the database. The critical parameters are includeIntermediateCommunities for hierarchy and theta/gamma for controlling cluster granularity.[9]

Python

```
def run_leiden_hierarchy(graph_object):
    """
    Runs Leiden algorithm with hierarchical output and writes to DB.
    """
    result = gds.leiden.write(
        graph_object,
        writeProperty="leiden_communities",  # The property added to Neo4j nodes
        includeIntermediateCommunities=True, # CRITICAL: Returns list of IDs [L0, L1, L2...]
        relationshipWeightProperty=None,     # Use weights if available (e.g., semantic similarity score)
        gamma=1.0,                           # Resolution: Higher values = more, smaller communities
        theta=0.01,                          # Randomness: Controls refinement stability
        tolerance=0.0001                     # Convergence criteria
```

```
    )

    # The result object contains statistics about the execution
    print(f"Leiden completed. Max Community Level: {result['ranLevels']}")
    print(f"Modularity Score: {result['modularity']}")
    return result
```

Note on Circular Hierarchies: In some edge cases, Leiden implementations might produce "circular" relationships in the intermediate community IDs due to the way identifiers are reused across iterations.[16] The Python logic parsing these IDs must be robust enough to handle the ordered list index as the definitive "Level" indicator (index 0 is Level 0, index 1 is Level 1) rather than inferring hierarchy solely from the ID values themselves.

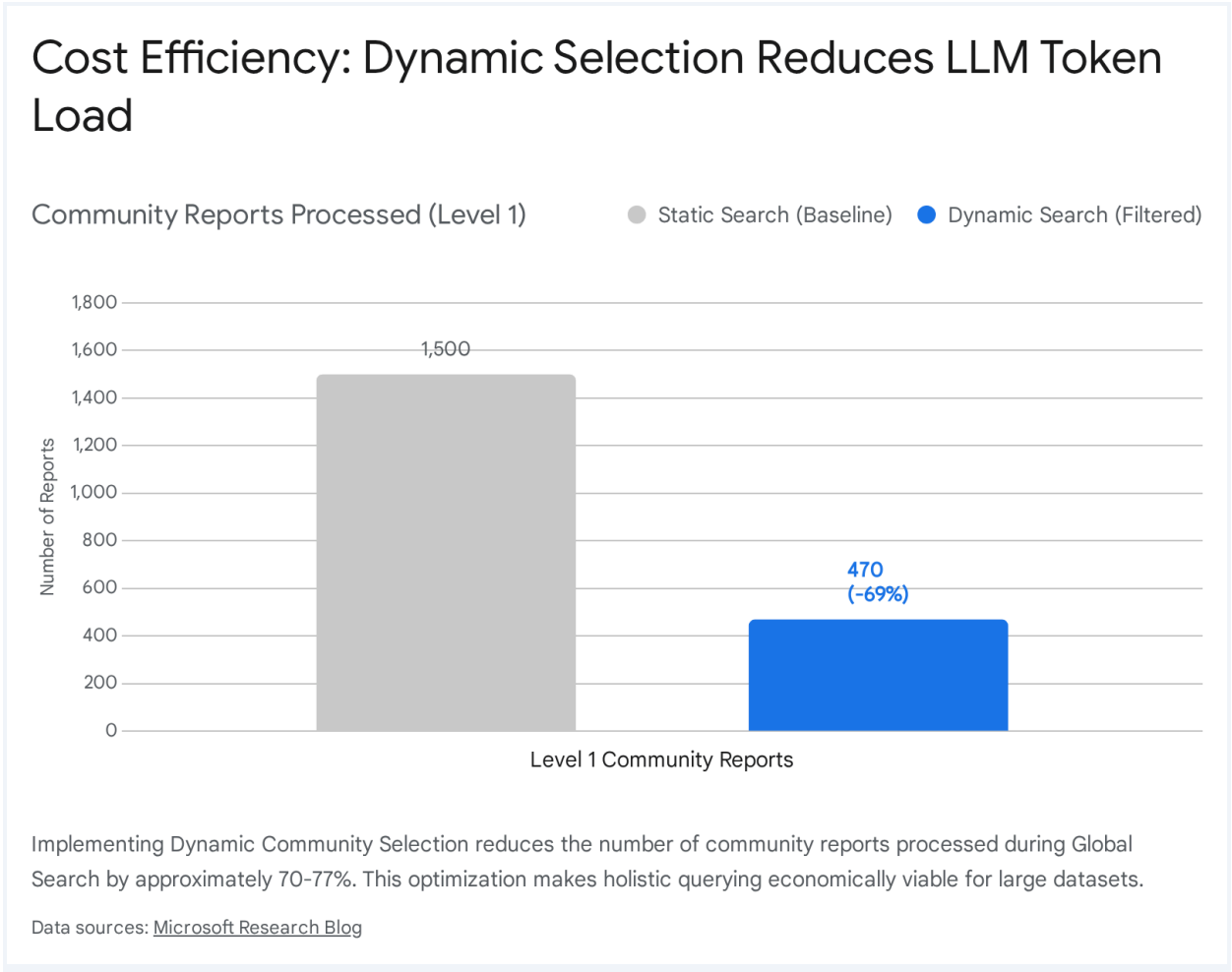
4.4 Extracting and Summarizing Communities (The "Map" Logic)

After writing the leiden_communities property, the graph has the necessary structure. The next step is the "Map" phase: iterating through the communities to generate summaries. This logic is orchestrated in Python, fetching data from Neo4j and sending it to the LLM.

Python

```
def generate_community_summaries(driver, llm_client, max_level):
    """
    Iterates through detected communities, extracts context, and generates summaries.
    """
    # Iterate through each level of the hierarchy
    for level in range(max_level):
        print(f"Summarizing Level {level} communities...")

        # 1. Get all distinct Community IDs for this specific level
        query_distinct = f"""
        MATCH (n:Entity)
        WHERE size(n.leiden_communities) > {level}
        RETURN distinct n.leiden_communities[{level}] as community_id
        """
        records, _, _ = driver.execute_query(query_distinct)

        for record in records:
            comm_id = record["community_id"]

            # 2. Retrieve Context: Get all entities in this community
```

```
        context_query = f"""
        MATCH (n:Entity)
        WHERE n.leiden_communities[{level}] = {comm_id}
        RETURN n.name as name, n.description as description
        """
        nodes = driver.execute_query(context_query)

        # Format context for the LLM
        context_text = "\n".join([f"- {n['name']}: {n['description']}" for n in nodes])

        # 3. Generate Summary via LLM (using Microsoft GraphRAG prompt style)
        prompt = f"""
        Role: Domain Expert.
        Task: Analyze the provided entities and generating a Community Report.

        Requirements:
        1. Title: A concise, descriptive title for this community.
        2. Executive Summary: A high-level synthesis of the common themes.
        3. Impact Rating: A score (0-10) of the severity/importance of this group.

        Input Entities:
        {context_text}
        """
        response = llm_client.generate(prompt) # Assume returns JSON-like object

        # 4. Write the Summary Node to Neo4j
        # We create a new node type :Community to hold the summary
        write_query = f"""
        MERGE (c:Community {{id: {comm_id}, level: {level}}})
        SET c.title = $title,
            c.summary = $summary,
            c.rating = $rating,
            c.course_id = $course_id
        WITH c
        MATCH (n:Entity) WHERE n.leiden_communities[{level}] = {comm_id}
        MERGE (n)-->(c)
        """
        driver.execute_query(write_query,
                             title=response.title,
                             summary=response.summary,
                             rating=response.rating,
                             course_id="CS101") # passed from context
```



---

5. Designing the Global Search Retriever

With the **Community** nodes populated with summaries, the retrieval strategy must now distinguish between "Local" and "Global" intent.

5.1 Vector Indexing of Summaries

To make the summaries searchable, we must create a vector index specifically for them. This is distinct from the index used for raw text chunks.

- **Entity Index:** Stores embeddings of raw text chunks/entities. Used for **Local Search**.
- **Community Index:** Stores embeddings of the Community node's summary text. Used for **Global Search**.

In Neo4j, this is done via:

Cypher

```
CREATE VECTOR INDEX community_summary_index IF NOT EXISTS
FOR (c:Community) ON (c.embedding)
OPTIONS {indexConfig: {
 `vector.dimensions`: 1536,
 `vector.similarity_function`: 'cosine'
}}
```

5.2 Implementing Dynamic Community Selection (The "Reduce" Logic)

When a user asks a global question, we employ the "Dynamic Community Selection" strategy to avoid overwhelming the LLM with too much context.[12]

The Python Workflow:

1. **Broad Search:** Perform a vector search on the community_summary_index to find the top 50 Level 2 (high-level) communities.
2. **Rate Relevance:** Iterate through these 50 summaries. Send each to a lightweight LLM (e.g., gpt-3.5-turbo or a local SLM) with the prompt: *"Rate the relevance of this summary to the user's question on a scale of 0-100."*
3. **Filter & Drill Down:**
    - If score < threshold (e.g., 70), discard.
    - If score >= threshold, check if this community has children (Level 1). If yes, retrieve the children and repeat the rating process for them.
    - If no children, keep the summary.
4. **Final Context Assembly:** Collect the high-scoring summaries. Concatenate them into the final prompt for the reasoning model (e.g., gpt-4o).

This logic ensures that if a user asks about "Ethics," we drill down into the "Ethics" community to find granular details, but we completely ignore the "Linear Algebra" community, saving tokens and reducing noise.

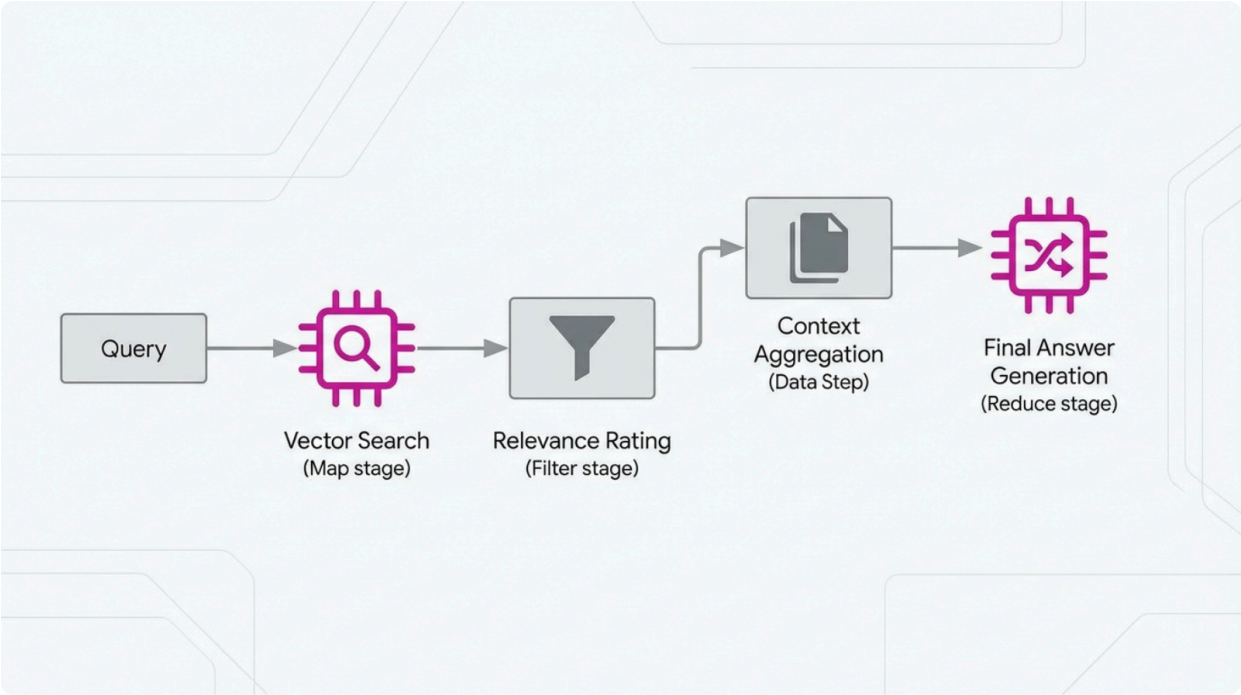
5.3 DRIFT Search: The Hybrid Approach

Recent advancements have introduced **DRIFT Search** (Dynamic Reasoning and Inference with Flexible Traversal), which combines local and global methods.[1] In DRIFT, the system starts with a local search to identify key entities, then expands to the *communities* those entities belong to. This is particularly useful for queries that start specific but require broad context, such as "How does the detailed implementation of the Transformer model relate to the

broader trends in Deep Learning history?"

In our Python implementation, this can be achieved by traversing the (:Entity)-->(:Community) relationship. If a Local Search retrieves the entity "Attention Mechanism," we can traverse up to its Level 1 Community to retrieve the broader context of "Sequence Modeling."

Global Search Workflow: The Map-Reduce Pattern



The Global Search pipeline employs a Map-Reduce pattern. The user query is first 'mapped' against the community index to retrieve candidate summaries. These are then 'filtered' for relevance using a lightweight model or dynamic selection. Finally, the surviving contexts are 'reduced' by a reasoning model to generate the holistic response.

---

6. Automation and Orchestration

To satisfy the requirement of automating this pipeline "upon course upload," we must define an event-driven architecture.

6.1 Architecture: Asynchronous Task DAGs

While Neo4j provides apoc.trigger to execute Cypher when data changes, using DB triggers for this pipeline is **strongly discouraged**.[17] Triggers run synchronously within the transaction.

Executing heavy GDS algorithms and network-bound LLM calls inside a trigger would lock the database and likely cause transaction timeouts.

Instead, we recommend an external orchestration layer using **Apache Airflow** or a Python task queue like **Celery/Redis**.

The Recommended Workflow:

1. **Upload Event:** The course management system uploads the file (PDF/Video) to object storage (S3) and publishes a message to a queue (e.g., RabbitMQ) with the payload {"course_id": "CS101", "file_uri": "s3://..."}.
2. **Task 1 (Ingestion Worker):** Consumes the message. Downloads the file, chunks the text, and writes raw chunks to Neo4j.
3. **Task 2 (Extraction Worker):** Runs the Entity Extraction LLM on the new chunks. Writes Entity and Relationship nodes to Neo4j.
4. **Task 3 (GDS Pipeline):** Once extraction is complete, triggers the Python GDS script.
    - Calls project_course_graph("CS101_graph", "CS101").
    - Calls run_leiden_hierarchy().
5. **Task 4 (Summary Worker):** Triggers the summarization script.
    - Calls generate_community_summaries().
    - Embeds the new summaries and updates the community_summary_index.

6.2 Implementation Detail: Handling Updates

When new content is added to an existing course, re-running the entire pipeline might be expensive. A crucial optimization is **Incremental Updates**.

- **Incremental Leiden:** The Leiden algorithm supports a seedProperty.[9] If we re-run the algorithm, we can use the existing community IDs as "seeds" to speed up convergence and maintain stability, ensuring that adding one lecture does not radically reshape the entire community structure of the course.

7. Conclusion

The implementation of Global Search within a Knowledge Graph RAG pipeline represents a significant leap forward from standard vector-based retrieval. By acknowledging the "sensemaking gap," we move beyond simple fact retrieval to systems capable of reasoning over the holistic structure of data.

The **Leiden algorithm** serves as the mathematical backbone of this architecture, providing the guaranteed connectivity required for coherent summarization. The **Map-Reduce** retrieval pattern, optimized by **Dynamic Community Selection**, allows us to query this structure efficiently.

For the Python developer using Neo4j, the graphdatascience library provides the necessary

tools to project, cluster, and analyze the graph programmatically. By wrapping these operations in an asynchronous automation pipeline, we can ensure that every course uploaded is not just stored, but understood—transformed into a rich, hierarchical knowledge base ready to answer the most complex global inquiries.

Works cited

1. Introducing DRIFT Search: Combining global and local search methods to improve quality and efficiency - Microsoft Research, accessed January 22, 2026, https://www.microsoft.com/en-us/research/blog/introducing-drift-search-combining-global-and-local-search-methods-to-improve-quality-and-efficiency/
2. Global Community Summary Retriever - GraphRAG, accessed January 22, 2026, https://graphrag.com/reference/graphrag/global-community-summary-retriever/
3. GraphRAG: Practical Guide to Supercharge RAG with Knowledge Graphs - Learn OpenCV, accessed January 22, 2026, https://learnopencv.com/graphrag-explained-knowledge-graphs-medical/
4. GraphRAG: New tool for complex data discovery now on GitHub - Microsoft Research, accessed January 22, 2026, https://www.microsoft.com/en-us/research/blog/graphrag-new-tool-for-complex-data-discovery-now-on-github/
5. From Local to Global: A Graph RAG Approach to Query-Focused Summarization - arXiv, accessed January 22, 2026, https://arxiv.org/html/2404.16130v1
6. GraphRAG: Improving global search via dynamic community ..., accessed January 22, 2026, https://www.microsoft.com/en-us/research/blog/graphrag-improving-global-search-via-dynamic-community-selection/
7. Leiden - Neo4j Graph Analytics for Snowflake, accessed January 22, 2026, https://neo4j.com/docs/snowflake-graph-analytics/current/algorithms/leiden/
8. Louvain - Neo4j Graph Data Science, accessed January 22, 2026, https://neo4j.com/docs/graph-data-science/current/algorithms/louvain/
9. Leiden - Neo4j Graph Data Science, accessed January 22, 2026, https://neo4j.com/docs/graph-data-science/current/algorithms/leiden/
10. Graph Data Analytics: A practical guide to process, visualize, and analyze connected data with Neo4j, accessed January 22, 2026, http://103.203.175.90:81/fdScript/RootOfEBooks/E%20Book%20collection%20-%202025%20-%20B/CSE%20%20IT%20AIDS%20ML/Graph%20Data%20Analytics.pdf
11. From Local to Global: A GraphRAG Approach to Query-Focused Summarization - arXiv, accessed January 22, 2026, https://arxiv.org/html/2404.16130v2
12. GraphRAG 2.0 Improves AI Search Results - Search Engine Journal, accessed January 22, 2026, https://www.searchenginejournal.com/graphrag-update-ai-search/533129/
13. A Python client for the Neo4j Graph Data Science (GDS) library - GitHub, accessed January 22, 2026, https://github.com/neo4j/graph-data-science-client
14. Graph Data Science with Neo4j: Learn how to use Neo4j 5 with Graph Data

Science library 2.0 and its Python driver for your project Scifo - Scribd, accessed January 22, 2026, https://www.scribd.com/document/978245531/Graph-Data-Science-with-Neo4j-Learn-how-to-use-Neo4j-5-with-Graph-Data-Science-library-2-0-and-its-Python-driver-for-your-project-Scifo
15. Graph Data Science at scale with Neo4j clusters and Hume - GraphAware, accessed January 22, 2026, https://graphaware.com/blog/graph-data-science-at-scale-with-neo4j-clusters-and-hume/
16. Circular hierarchy from Leiden clustering - Neo4j Community, accessed January 22, 2026, https://community.neo4j.com/t/circular-hierarchy-from-leiden-clustering/69385
17. Triggers - APOC Core Documentation - Neo4j, accessed January 22, 2026, https://neo4j.com/docs/apoc/current/background-operations/triggers/