



Report : Hybrid Graph RAG System

Data Ingestion Pipeline

Overview

This report outlines the architecture and workflow of the Hybrid Graph RAG data ingestion pipeline. The pipeline is engineered to process complex PDF documents (specifically budget reports) and transform them into a dual data representation: a structured **Knowledge Graph** for explicit facts and a semantic **Vector Index** for contextual understanding. This hybrid approach provides a robust foundation for an advanced Retrieval-Augmented Generation (RAG) system capable of answering both precise and nuanced queries.

The entire process is orchestrated by `run_ingestion.py` and is segmented into three primary stages:

1.  **Document Parsing & Enrichment**
2.  **Knowledge Graph Construction**
3.  **Vector Index Construction**

Orchestrator: `run_ingestion.py`

This script serves as the master controller for the entire pipeline. It defines the sequence of operations, manages the overall workflow, handles errors, and logs the process from start to finish.

The core of the script is the `IngestionPipeline` class, which executes the following steps in a controlled sequence:

1. **Initialization:** It begins by identifying the source data directory and the chosen vectorization strategy (`simple`, `enhanced`, or `hybrid`).
2. **Run Document Parsing:** It invokes `document_parser.py` to load, parse, chunk, and enrich the raw PDF files.
3. **Run Graph Construction:** It takes the processed documents and uses `graph_builder.py` to extract entities and relationships, building a knowledge graph in a Neo4j database. This stage is designed to be optional; the pipeline can continue even if graph construction fails.

4. **Run Vector Construction:** It utilizes `vectorizer.py` to create semantic vector embeddings from the documents and store them in a vector database. This stage is critical for the pipeline's success.
5. **Validation & Finalization:** It performs health checks on database connections, validates data integrity, and logs a comprehensive summary of the operation, including duration, files processed, and any errors encountered.

Stage 1: Document Parsing (`document_parser.py`)

This is the foundational stage where unstructured PDF documents are converted into a structured and enriched format suitable for the subsequent stages. The process is designed to be highly robust and intelligent.

Multi-Strategy Parsing

The pipeline does not rely on a single parsing tool. The `MultiModalDocumentParser` class attempts several strategies to ensure the best possible text extraction from complex PDFs that may contain tables, images, and varied layouts.

1. **Adaptive Parsing:** It sequentially tries different parsers: `DoclingReader` (best for complex layouts), `UnstructuredReader` (for high-resolution OCR), `PyMuPDFReader` (for fast text extraction), and `SimpleDirectoryReader` (as a reliable fallback).
2. **Quality Evaluation:** After each attempt, it scores the quality of the extracted text using the `_evaluate_parsing_quality` method. This function scores based on content length, the presence of structural markers (e.g., "Table", "Figure"), financial terms, and overall text cleanliness.
3. **Best Result Selection:** The parsing result with the highest score is chosen for the next step. This adaptive approach maximizes the quality of the extracted data from diverse PDF formats.

Intelligent Splitting

Once a document's text is extracted, it is too large to be processed effectively. The `split_documents_intelligently` function breaks it down into smaller, meaningful chunks.

- **Primary Method (Semantic Splitting):** It first attempts to use `SemanticSplitterNodeParser`. This advanced splitter uses an embedding model to understand the semantic meaning of the text, ensuring that chunks are split between topics rather than in the middle of a sentence or idea.
- **Fallback Method (Sentence Splitting):** If semantic splitting fails, it falls back to a `SentenceSplitter`. The chunk size is set to **2048 tokens**, which is large enough to hold significant context while remaining manageable for language models.

Metadata Enrichment

This is a crucial step where each text chunk is enhanced with valuable metadata. The `apply_metadata_extraction` function applies a series of LlamaIndex extractors and a custom extractor:

- **Standard Extractors:**
 - `SummaryExtractor`: Generates a brief summary for each chunk.
 - `QuestionsAnsweredExtractor`: Generates a list of questions that the chunk can answer.
 - `KeywordExtractor`: Pulls out the most relevant keywords.
- **Custom `BudgetDocumentExtractor`:** This is a domain-specific extractor tailored for budget documents. Using **regular expressions (regex)**, it scans the text for specific patterns to extract:
 - `budget_year`: Identifies fiscal years (e.g., 2024, 2025).
 - `department`: Finds department or ministry names.
 - `budget_amount`: Extracts monetary values (e.g., "\$1.5 billion").
 - `document_type`: Classifies the chunk as a summary, detailed report, analysis, etc.

By the end of this stage, the raw PDFs have been transformed into a list of text chunks, each packed with rich, contextual metadata.

Stage 2: Knowledge Graph Construction (`graph_builder.py`)

This stage takes the enriched text chunks and builds a structured knowledge graph, which allows for precise, explicit queries about relationships between different entities.

Predefined Schema

The `BudgetSchema` class defines the backbone of the knowledge graph. It specifies the allowed **entity types** (e.g., `DEPARTMENT`, `FISCAL_YEAR`, `PROGRAM`) and the **relationship types** that can connect them (e.g., `ALLOCATED_TO`, `FUNDS`, `BELONGS_TO`). This schema guides the LLM to extract information in a consistent and structured format.

Multi-Extractor Graph Building

Similar to the parsing stage, the `EnhancedGraphBuilder` uses multiple LLM-based extractors to identify entities and relationships (triplets) within the text:

1. **SchemaLLMPathExtractor**: Strictly follows the predefined **BudgetSchema** to find known patterns.
2. **DynamicLLMPathExtractor**: Has the flexibility to discover new or unexpected entities and relationships not defined in the schema.
3. **SimpleLLMPathExtractor**: Serves as a general-purpose fallback.

This combination ensures both high-quality, structured data extraction and the flexibility to capture novel information. The extracted triplets are then used to build the graph using **PropertyGraphIndex.from_documents**, which also creates vector embeddings for the graph nodes themselves (**embed_kg_nodes=True**).

Graph Enhancement & Post-Processing

After the initial graph is built, a series of custom **Cypher queries** are executed to refine and enrich its structure. This is a powerful feature of the pipeline.

- **_create_document_nodes**: Creates a specific **BUDGET_DOCUMENT** node for each source file.
- **_create_budget_hierarchy**: Establishes a master hierarchy, linking all **FISCAL_YEAR** nodes to a central "Federal Budget" node.
- **_add_cross_year_relationships**: This is critical for comparative analysis. It creates **PRECEDES** and **FOLLOWS** relationships between consecutive fiscal year nodes (e.g., FY2024 -> PRECEDES -> FY2025), enabling time-series queries.

The final output is a rich, interconnected knowledge graph stored in a Neo4j database, representing the factual backbone of the documents.

Stage 3: Vector Index Construction (**vectorizer.py**)

This stage creates the semantic vector index, which is essential for similarity-based search and answering nuanced, context-dependent questions.

Intelligent Processing Pipeline

The **BudgetVectorProcessor** class manages the creation of the vector index. It uses a LlamaIndex **IngestionPipeline** to apply a series of transformations:

1. **Metadata Cleaning**: Before processing, it cleans the metadata attached to each document chunk, truncating long values to prevent them from overwhelming the actual text content during embedding.
2. **Adaptive Chunking**: It uses a **SentenceSplitter** but intelligently calculates a **safe_chunk_size** based on the size of the metadata. This prevents errors where the

combined size of the text and metadata exceeds the embedding model's context window.

3. **Further Metadata Extraction:** It applies more extractors like `TitleExtractor` and `SummaryExtractor`. This metadata is embedded *along with the text*, providing richer semantic context to the resulting vectors.

Index Creation and Hybrid Strategy

The processed nodes are then passed to `VectorStoreIndex`, which uses a configured embedding model to generate vectors and store them.

The pipeline also supports a `hybrid` strategy via the `HybridVectorStrategy` class. If selected, it creates **multiple, specialized vector indices**:

- **full**: An index of all document chunks for broad queries.
- **financial**: A smaller, targeted index containing only chunks identified as having financial data.
- **departmental**: An index focused on chunks containing departmental information.

This multi-index approach allows a RAG system to route queries to the most relevant index, significantly improving the precision of search results. For example, a query about "funding for the Department of Energy" could be directed to both the `financial` and `departmental` indices.

The output of this stage is one or more vector indices ready for efficient semantic retrieval.

Retrieval Pipeline

Overview

This report details the architecture and workflow of the retrieval pipeline for the Hybrid Graph RAG system. The pipeline is designed to answer complex questions about government budget documents by intelligently combining semantic vector search with structured knowledge graph traversal. Its primary goal is to provide accurate, context-rich, and verifiable answers.

The retrieval process is managed by two key files:

1. **engine.py**: The central factory for creating and configuring different types of sophisticated query engines.
2. **retrievers.py**: Defines the custom logic for fetching relevant data from the hybrid storage system (vector store and graph database).

Core Component: The Hybrid Graph Retriever (**retrievers.py**)

The heart of the retrieval strategy is the **HybridGraphRetriever** class. It implements a powerful pattern known as "**Vector-First, Graph-Expansion**" to ensure that the final answer is built from both broad context and precise facts.

This process unfolds in three distinct steps:

Step 1: Vector Search (Broad Context)

When a query is received, the retriever first performs a semantic search against the vector index.

- **Purpose**: To quickly find text chunks from the original documents that are contextually relevant to the user's question. This is ideal for understanding the "why" and "how" behind budget decisions.
- **Mechanism**: It uses the `vector_index.as_retriever()` to fetch the top `k` most similar document chunks based on their vector embeddings.

Step 2: Entity Linking (Connecting the Dots)

The text from the retrieved vector nodes is then processed to identify key entities.

- **Purpose**: To extract specific, factual nouns (like department names, fiscal years, or program titles) that can serve as entry points into the structured knowledge graph.

- **Mechanism:** An LLM is prompted (`_get_entities_from_text`) to read the text and list the key entities it contains. This acts as a bridge between the unstructured text and the structured graph.

Step 3: Graph Traversal (Factual Enrichment)

Using the entities extracted in the previous step, the retriever queries the knowledge graph.

- **Purpose:** To enrich the initial context with hard, factual data and explicit relationships. This is perfect for finding precise figures, connections between departments, and funding allocations.
- **Mechanism:** The `graph_index.as_retriever()` is called with a query that combines the original question and the extracted entities. This traverses the graph from the entity nodes to find connected information.

The final result is a consolidated list of nodes from both the vector search and the graph traversal, providing a rich, hybrid context for the final answer synthesis.

Query Engine Architecture (`engine.py`)

The `engine.py` script acts as a factory, capable of constructing different query engines to handle various types of questions, from simple lookups to complex, multi-part analyses.

The Base Engine: `create_base_query_engine`

This function assembles the fundamental components:

1. **Indices:** It loads the existing `VectorStoreIndex` and `PropertyGraphIndex`.
2. **Retriever:** It initializes the `HybridGraphRetriever` described above.
3. **Response Synthesizer:** This is a critical component that takes the retrieved context and synthesizes a final, human-readable answer. It is configured with a custom, advanced prompt template (`SYNTHESIS_PROMPT_TEMPLATE`).

The Synthesis Prompt: A Key Innovation

The `SYNTHESIS_PROMPT_TEMPLATE` is engineered to guide the LLM's reasoning process. It explicitly instructs the AI to:

- Act as an expert financial analyst.
- Use **only** the provided context (from the retriever).
- Combine facts from both unstructured text and structured graph data.
- Perform calculations if necessary.
- **Cite every fact** with its source document, ensuring verifiability.

This "reasoned synthesis" approach dramatically improves the quality and trustworthiness of the final answers.

Handling Complex Queries

For queries that require more than a simple lookup, the pipeline offers two advanced decomposition strategies:

1. **create_multistep_query_engine:**
 - **Best for:** Sequential, multi-step questions (e.g., "Find the budget for the Department of Energy in 2024, then calculate its percentage change from 2023.").
 - **How it works:** It uses a `DecomposeQueryTransform` to break the complex query into a series of simpler, sequential steps. It solves the first step, uses the result to inform the second, and so on.
2. **create_subquestion_query_engine:**
 - **Best for:** Comparative questions with parallel components (e.g., "Compare the funding for cybersecurity and renewable energy research.").
 - **How it works:** It breaks the main query into multiple independent sub-questions that can be executed in parallel. The final answers are then gathered and synthesized into a single, comprehensive response.

By offering these different engine types, the retrieval pipeline can dynamically adapt its strategy to best fit the complexity and structure of the user's question, ensuring efficient and accurate information retrieval.

API & Query Orchestration Pipeline

Overview

This report covers the final layer of the Hybrid Graph RAG system: the API and Query Orchestration layer, managed by `main.py`. This component exposes the powerful retrieval and synthesis capabilities of the system through a robust, high-performance REST API. It acts as the "front door" for user queries, intelligently processing them to select the best retrieval strategy and ensuring a reliable and informative user experience.

The primary responsibilities of this layer are:

1. **Providing a clear API** for interacting with the RAG engine.
2. **Managing the lifecycle** of different query engines for efficiency.
3. **Intelligently analyzing and routing** user queries to the most appropriate processing pipeline.

API Architecture (`main.py`)

The entire API is built using **FastAPI**, a modern Python web framework chosen for its high performance and automatic interactive documentation.

Key API Endpoints

The application exposes several well-defined endpoints to manage interaction:

- **POST /query**: The main endpoint for submitting complex questions. It accepts a JSON body specifying the query and the desired decomposition strategy.
- **GET /query/simple**: A simplified endpoint for quick, straightforward queries via URL parameters.
- **GET /health**: A health check endpoint that confirms the API is running and reports the status of the cached query engines (e.g., `loaded` or `not_loaded`).
- **GET /engines/preload**: An administrative endpoint that preloads all available query engines into memory, minimizing latency for first-time queries.
- **GET /engines/info**: A discovery endpoint that provides detailed information about the available query engines and their ideal use cases.

Intelligent Query Orchestration

The true innovation of this layer lies in how it processes incoming queries. It doesn't just pass the query along; it analyzes it and makes smart decisions to optimize the retrieval process.

1. On-Demand Engine Loading & Caching

To conserve memory and resources, query engines are not all loaded at startup. The `get_query_engine` function acts as a cached factory:

- When a request for a specific engine type (e.g., `multistep`) arrives, the function first checks if an instance of that engine already exists in a cache (`_engines_cache`).
- If it exists, the cached instance is returned immediately.
- If not, it creates the engine, stores it in the cache, and then returns it.

This **lazy-loading** approach ensures that memory is only used for engines that are actively needed, while the cache guarantees that subsequent requests are processed without the overhead of re-initialization.

2. Automatic Query Complexity Analysis

Before processing, every query is passed through the `determine_query_complexity` function.

- **Mechanism:** This function uses a simple but effective heuristic, scanning the query for keywords that typically indicate a complex question (e.g., `compare`, `analysis`, `percentage change`, `trend`).
- **Purpose:** It classifies each query as "low," "medium," or "high" complexity.

3. Automatic Decomposition Strategy

The complexity analysis directly informs the query strategy.

- If a user submits a "high" complexity query without specifying a decomposition method, the system **automatically defaults to the `multistep` engine**.
- This ensures that complex analytical questions are automatically handled by the more powerful sequential decomposition engine, improving the quality of the answer without requiring the user to be an expert on the system's internal workings.

4. Response & Citation Formatting

After the query engine generates a response, the API layer performs the final formatting:

- It extracts the textual answer.
- It calls `extract_sources_from_response` to pull the source document filenames (e.g., `budget_2024.pdf`) from the response metadata. This is crucial for **verifiability and trust**.
- It packages everything into a clean `QueryResponse` JSON object, including the answer, sources, processing time, and the final decomposition method used.

This orchestration transforms a simple user question into a sophisticated, multi-step process that leverages the best retrieval strategy for the task at hand, delivering a reliable, verifiable, and context-rich answer.