Lab 3: Vector Embeddings & Hybrid Search

Using the neo4j-graphrag Python Package

The GenAI Promise and Its Limits

LLMs excel at pattern recognition and language fluency:

- **Text generation**: Creating human-like responses, summaries, explanations
- **Language understanding**: Parsing intent, extracting meaning
- **Pattern completion**: Continuing sequences, generating variations

But they have fundamental limitations that require solutions.

LLM Limitations

Limitation	Problem	Example
Hallucination	Generates probable, not accurate responses	Fabricated citations
Knowledge cutoff	No access to your private data	"What did our CEO say?"
Relationship blindness	Can't reason across connections	"Which customers want renewable energy?"

Solution: Provide context through Retrieval-Augmented Generation (RAG).

Traditional RAG vs GraphRAG

Traditional RAG:

- Treats documents as isolated blobs
- Retrieves similar text chunks
- Misses relationships between entities

GraphRAG:

- Extracts structure from documents
- Preserves entities and relationships
- Enables graph traversal + semantic search

What Traditional RAG Sees

Traditional RAG treats documents as isolated, unstructured blobs:

```
Chunk 1: "James is interested in renewable energy investments..."
Chunk 2: "Customer C0001 holds technology stocks in account A001..."
Chunk 3: "The renewable energy sector shows strong growth..."
```

What traditional RAG misses:

- Which customer is James?
- What stocks does James actually hold?

The GraphRAG Advantage

GraphRAG connects chunks to the knowledge graph:

```
(:Chunk {text: "James wants renewable energy..."})
    ↓ FROM_DOCUMENT
(:Document {title: "Customer Profile"})
    ↓ DESCRIBES
(:Customer {name: "James Anderson"})
    ↓ HAS_ACCOUNT
(:Account) → HAS_POSITION → (:Stock) → IN_SECTOR → (:Sector)
```

Now you can traverse from semantic search results to structured data.

Chunks as Graph Anchors

The key insight: **Chunks become anchors for graph traversal.**

1. **Vector search** finds relevant chunks by meaning
2. **Graph traversal** follows relationships to connected entities
3. **Combined result** provides content + structured context

```
"renewable energy" → finds chunk → traverses to Customer →
    gets Account → gets Stock holdings → gets Sectors
```

Enriched Context for the LLM

Traditional RAG returns:

> "James is interested in renewable energy investments..."

GraphRAG returns:

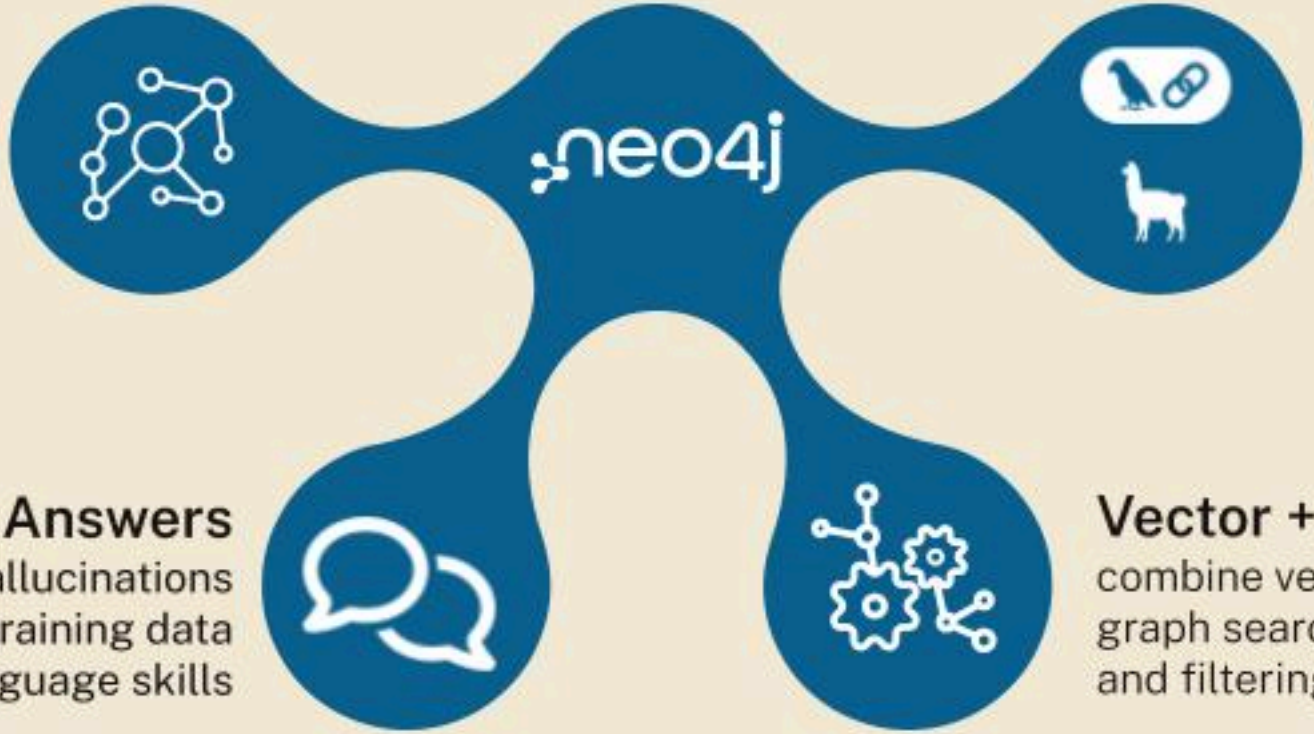
> "James is interested in renewable energy investments..."
> **Customer:** James Anderson (C0001)
> **Current Holdings:** TCOR (Technology), SMTC (Technology), MOBD (Technology)
> **Sectors:** 100% Technology, 0% Renewable Energy

The LLM now has the context to identify the gap.

GraphRAG Impact

Query knowledge graph within your GenAI application

Rich Context
graph entities
graph algorithms for
enhancement &
clustering

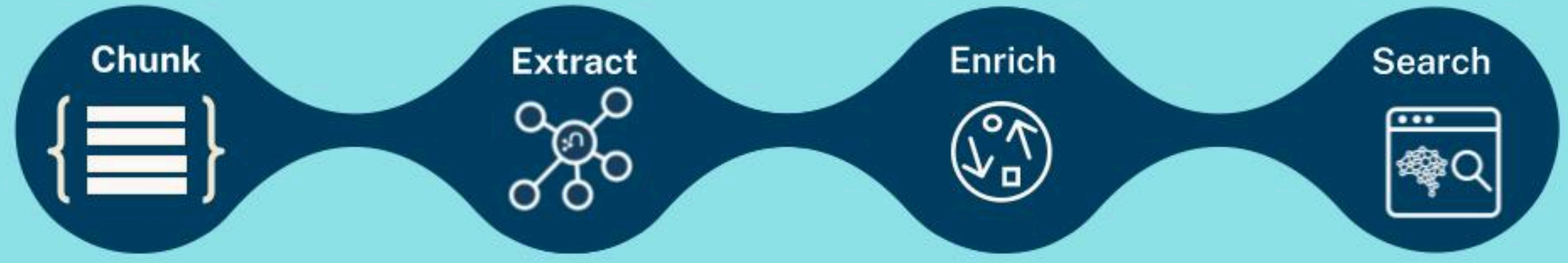


Orchestration & Flow
only fetch relevant
information from reliable
data sources and integrate in
architecture

Accurate & Relevant Answers
avoid hallucinations
deprioritize LLM training data
only use language skills

Vector + Graph
combine vector and
graph search for retrieval
and filtering

GraphRAG Phases



Substructure, **chunk and embed** documents into a **lexical graph**

Extract **entities and relationships** with descriptions, facts, and claims

Compute clusters of topical structures, generate **community summaries**
Enrich with other graph features from algorithms

Enhanced vector search with graph context
Local search over entities and their patterns
Global search over community summaries

The neo4j-graphrag Python Package

The official Neo4j GenAI package for Python provides first-party support for building GraphRAG applications.

The package provides:

- **Text splitters** for chunking documents with configurable overlap
- **Embedding integrations** for OpenAI, Vertex AI, Sentence Transformers, and more
- **Index management** for creating and populating vector and full-text indexes
- **Retrievers** for vector, hybrid, and graph-aware search patterns
- **LLM integrations** for OpenAI, Anthropic, Vertex AI, MistralAI, Cohere, and Ollama
- **GraphRAG orchestration** combining retrieval, augmentation, and generation

Components Used in This Lab

Component	Module	Purpose
`FixedSizeSplitter`	`text_splitters`	Text chunking with overlap
`SentenceTransformerEmbeddings`	`embeddings`	Local embedding generation
`create_vector_index`	`indexes`	Vector index creation
`create_fulltext_index`	`indexes`	Full-text index creation
`VectorRetriever`	`retrievers`	Semantic similarity search
`HybridRetriever`	`retrievers`	Combined vector + keyword search

What is a Vector?

Vectors are lists of numbers representing data in high-dimensional space.

Embeddings encode text meaning as vectors (often 384-1024 dimensions).

Key property: Similar meanings produce similar vectors.

- "interested in renewable energy" ↔️ "wants green investments" → vectors close together
- "interested in renewable energy" ↔️ "quarterly earnings report" → vectors far apart

Why Vectors Matter

Without vectors:

- Need exact keyword matches
- "What are James's investment interests?" won't find "renewable energy goals"

With vectors:

- Questions and chunks become embeddings
- Find content with similar *meaning*, regardless of exact words
- "Investment interests" finds content about "goals" and "preferences"

The Document Graph Schema

```
(:Document)
    - document_id, filename, document_type, title

(:Chunk)
    - chunk_id, text, embedding, index

(:Document)<-[:FROM_DOCUMENT]-(:Chunk)
(:Chunk)-[:NEXT_CHUNK]->(:Chunk)
(:Document)-[:DESCRIBES]->(:Customer)
```

Chunks store embeddings for semantic search while maintaining graph structure.

Chunking Strategy

Documents must be broken into **chunks** that fit within processing limits.

Trade-off:

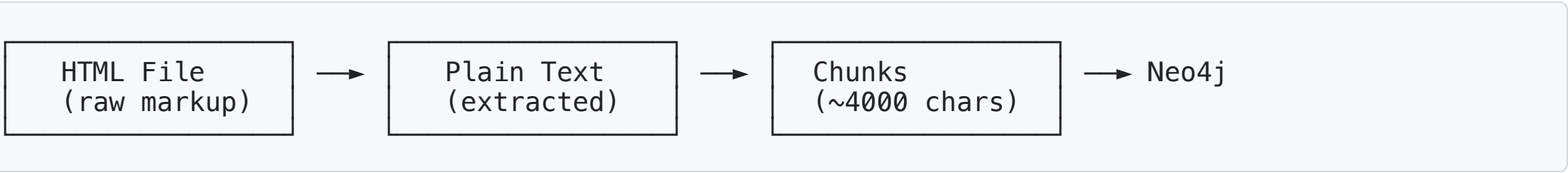
- **Larger chunks**: Better context, less precise retrieval
- **Smaller chunks**: More precise retrieval, less context

```
from neo4j_graphrag.experimental.components.text_splitters import FixedSizeSplitter

splitter = FixedSizeSplitter(chunk_size=4000, chunk_overlap=200)
```

Document Processing Pipeline

The pipeline transforms HTML documents into searchable chunks:



Step 1: Parse HTML, strip tags, extract clean text

Step 2: Split text into overlapping chunks

Step 3: Store chunks as `(:Chunk)` nodes with `text` property

Both vector and full-text search query the **chunks**, not the original HTML.

What Gets Indexed

Node	Indexed Property	Index Type	Purpose
`Chunk`	`embedding`	Vector	Semantic similarity search
`Chunk`	`text`	Full-text (Lucene)	Keyword matching

The `(:Document)` node stores only **metadata** (title, filename, type).
The actual searchable content lives in `(:Chunk)` nodes.

Creating Indexes

Vector Index for semantic similarity search:

```
from neo4j_graphrag.indexes import create_vector_index

create_vector_index(
    driver, "chunk_embedding_index",
    label="Chunk", embedding_property="embedding",
    dimensions=384, similarity_fn="cosine"
)
```

Full-text Index for keyword search:

```
from neo4j_graphrag.indexes import create_fulltext_index

create_fulltext_index(driver, "chunk_text_index", label="Chunk", node_properties=["text"])
```

What is a Retriever?

A **retriever** searches your knowledge graph and returns relevant information.

The GraphRAG flow:

```
User Question
     ↓
Retriever finds relevant context
     ↓
Context passed to LLM
     ↓
LLM generates grounded answer
```

The retriever's job is finding the right context. The LLM's job is generating a coherent answer.

Retriever Types in neo4j-graphrag

Retriever	What It Does
VectorRetriever	Semantic similarity search across text chunks
HybridRetriever	Combines vector + keyword search
VectorCypherRetriever	Semantic search + graph traversal
HybridCypherRetriever	Hybrid search + graph traversal
Text2CypherRetriever	Natural language → Cypher query

Each pattern excels at different question types.

Vector Retriever

Find semantically similar content:

```
from neo4j_graphrag.retrievers import VectorRetriever

retriever = VectorRetriever(
    driver=driver,
    index_name="chunk_embedding_index",
    embedder=embedder,
)

results = retriever.get_search_results(
    query_text="investment strategies for moderate risk",
    top_k=5,
)
```

Graph-Aware Search (VectorCypherRetriever)

Combine semantic search with graph traversal using a custom retrieval query:

```
from neo4j_graphrag.retrievers import VectorCypherRetriever

retrieval_query = """
WITH node, score
MATCH (node)-[:FROM_DOCUMENT]->(d:Document)
OPTIONAL MATCH (d)-[:DESCRIBES]->(customer:Customer)
RETURN node.text AS text, score,
       d.title AS document_title,
       customer.name AS customer_name
"""

retriever = VectorCypherRetriever(
    driver=driver, index_name="chunk_embedding_index",
    retrieval_query=retrieval_query, embedder=embedder,
)
```

The retrieval query runs **after** vector search, allowing graph traversal from found chunks.

Why Hybrid Search?

Search Type	Strengths	Weaknesses
Vector	Semantic similarity, synonyms	Misses exact terms
Keyword	Exact matches, proper nouns	Misses meaning
Hybrid	Best of both	More complex

Example: Query "renewable energy"

- **Vector** finds "green investments", "sustainable portfolios"
- **Keyword** finds exact phrase "renewable energy"
- **Hybrid** finds both, then fuses results

How Hybrid Search Works (neo4j-graphrag)

The library performs result fusion entirely in Cypher at the database level:

1. **Vector search** runs → scores normalized: `score / max_vector_score`
2. **Fulltext search** runs → scores normalized: `score / max_fulltext_score`
3. Results combined via **UNION**
4. **Deduplication** via aggregation:
   - **NAIVE**: `max(score)` — best score from either index wins
   - **LINEAR**: `alpha × vector + (1−alpha) × fulltext` — weighted blend
5. Final `ORDER BY score DESC LIMIT top_k`

No Python-side deduplication needed — the library handles it in Cypher.

NAIVE vs LINEAR Ranker

Aspect	NAIVE	LINEAR
Formula	`max(vector, fulltext)`	`α × vector + (1−α) × fulltext`
Parameters	None	`alpha` (0.0 to 1.0)
Behavior	Best single score wins	Weighted combination
Use when	Either index could be best	You want to tune balance

NAIVE example: Doc appears in vector (0.9) and fulltext (0.6) → final score: **0.9**

LINEAR example (α=0.7): Same doc → `0.7 × 0.9 + 0.3 × 0.6` = **0.81**

When to use each:

- **NAIVE**: Good default, discovers results from either index equally
- **LINEAR**: When you need fine control (e.g., prioritize semantic meaning with α=0.8)

Hybrid Retriever

Combine vector similarity with keyword matching:

```
from neo4j_graphrag.retrievers import HybridRetriever

retriever = HybridRetriever(
    driver=driver,
    vector_index_name="chunk_embedding_index",
    fulltext_index_name="chunk_text_index",
    embedder=embedder,
)

# NAIVE ranker (default): max(vector_score, fulltext_score)
results = retriever.get_search_results(query_text="renewable energy", top_k=5)

# LINEAR ranker: alpha * vector + (1-alpha) * fulltext
results = retriever.get_search_results(
    query_text="renewable energy", top_k=5,
    ranker="linear", alpha=0.7  # 70% vector, 30% fulltext
)
```

Graph-Aware Hybrid Search (HybridCypherRetriever)

Combine hybrid search with graph traversal:

```
from neo4j_graphrag.retrievers import HybridCypherRetriever

retrieval_query = """
WITH node, score
MATCH (node)-[:FROM_DOCUMENT]->(d:Document)-[:DESCRIBES]->(c:Customer)
MATCH (c)-[:HAS_ACCOUNT]->(:Account)-[:HAS_POSITION]->(:Position)
      -[:OF_SECURITY]->(s:Stock)-[:OF_COMPANY]->(company:Company)
RETURN node.text, score, c.name AS customer, collect(DISTINCT company.name) AS companies
"""

retriever = HybridCypherRetriever(
    driver=driver,
    vector_index_name="chunk_embedding_index",
    fulltext_index_name="chunk_text_index",
    retrieval_query=retrieval_query,
    embedder=embedder,
)
```

The Complete Picture

After Lab 3, your knowledge graph contains:

- **Structured entities**: Customers, Accounts, Stocks, Companies
- **Document graph**: Documents → Chunks with embeddings
- **Connections**: Documents linked to Customers they describe
- **Search capabilities**: Vector, full-text, and hybrid search

This enables questions that require both semantic understanding and graph traversal.

Summary

In this lab, you learned:

- **Embeddings** encode text meaning as vectors for semantic search
- **neo4j-graphrag** provides official tools for building GraphRAG applications
- **Chunking** breaks documents into searchable pieces
- **Vector indexes** enable fast similarity search
- **Hybrid search** combines vector and keyword matching
- **Graph-aware retrievers** traverse relationships from search results

Next: Export graph data to Databricks for agent-based analysis.