Introduction to GraphRAG Workshop

A one hour hands-on introduction to GraphRAG.

Introduction to GraphRAG Workshop → Introduction to GraphRAG

Welcome

Welcome to Introduction to GraphRAG

Welcome to the hands-on **Introduction to GraphRAG** Workshop, where you will learn:

- How to use GraphRAG to improve the quality of LLM-generated content
- How to build a knowledge graph from unstructured data
- How to use LangGraph's prebuilt ReAct agent to connect to Neo4j
- How to provide the agent with access to the knowledge graph using the `langchain-neo4j` library

Get started

The repository, **neo4j-graphacademy/workshop-graphrag-introduction**, has been created for this course. It contains any starter code and resources you need.

You can use a **GitHub codespace** as an online IDE and workspace for this course. It will automatically clone the course repository and set up your environment.

> **GitHub Codespaces**
>
> You will need to login with a GitHub account. The **GitHub Codespaces free monthly usage** will cover the duration of this course.

Develop on your local machine

To follow along locally, you will need:

- **Python**.
- **Visual Studio Code**.
- **Jupyter extension for Visual Studio Code**.
- The ability to install packages using `pip`.

You may want to set up a virtual environment using `venv` or `virtualenv` to keep your dependencies separate from other projects.

Clone the repository

Clone the github.com/neo4j-graphacademy/workshop-genai repository:

bash:

```bash
git clone https://github.com/neo4j-graphacademy/workshop-genai
```

Install the required packages using `pip` and download the required data:

bash:

```bash
cd workshop-genai

pip install -r requirements.txt
```

You do not need to create a Neo4j database as you will use the provided instance.

The instance uses Neo4j's GenAI functions, you can find out more about how to configure them in the **Neo4j GenAI integration documentation**.

Configure the environment

Today we will need an API key to use OpenAI's LLMs and embedding models. You can use the `OPENAI_API_KEY` provided, or use your own API key generated from **platform.openai.com**.

We have created a Neo4j instance for you to use during this workshop.

Create a file called `.env` and copy the following code into it.

env: **env**

```
OPENAI_API_KEY=sk-...

NEO4J_URI=bolt://{instance-ip}:{instance-boltPort}

NEO4J_USERNAME={instance-username}

NEO4J_PASSWORD={instance-password}
```

Test your setup

You can test your setup by running `workshop-genai/test_environment.py` - this will attempt to connect to the Neo4j sandbox and the OpenAI API.

You will see an `OK` message if you have set up your environment correctly. If any tests fail, check the contents of the `.env` file.

Keep your codespace running

To avoid creating a new environment for each challenge, you can keep your environment running for the duration of the workshop.

The environment will automatically pause after a period of inactivity.

Are you ready?

When you are ready, you can move on to the next lesson.

What is GraphRAG?

Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a technique that improves the responses of LLMs by providing them with relevant, up-to-date information retrieved from external sources.

RAG

Retrieval-Augmented Generation (RAG) with Vectors

This typically involves converting text into **vector embeddings** that encodes the semantic meaning of the text, in a format that the user understands, and then using similarity search to find relevant information.

RAG

Vector-based RAG

Vectors work well for:

- Contextual or Meaning Based Questions
- Synonyms or Paraphrasing
- Fuzzy or Vague queries
- Broad or Open-Ended questions
- Complex queries with multiple concepts

> *What does Paul Graham think about Generative AI?*

Vectors are ineffective for:

- Highly Specific or Fact-Based Questions
- Numerical or Exact-Match Queries
- Boolean or Logical Queries
- Ambiguous or Unclear Queries without Context
- Specialised Knowledge

> *How many Generative AI Startups has Paul Graham invested in?*

Unstructured data as vectors

Text	Vector embedding
She’s just a small town girl	[0.12, -0.34, 0.56, 0.78, ..., -0.91]
Living in a lonely world	[0.22, 0.45, -0.67, 0.11, ..., 0.33]
She took the midnight train	[-0.55, 0.89, 0.12, -0.44, ..., 0.67]
Going anywhere	[0.78, -0.23, 0.45, 0.91, ..., -0.12]

Unstructured data as a graph

Text	Vector embedding
She’s just a small town girl	[0.12, -0.34, 0.56, 0.78, ..., -0.91]
Living in a lonely world	[0.22, 0.45, -0.67, 0.11, ..., 0.33]
She took the midnight train	[-0.55, 0.89, 0.12, -0.44, ..., 0.67]
Going anywhere	[0.78, -0.23, 0.45, 0.91, ..., -0.12]

Don’t stop believin' as a graph

Knowledge Graphs

GraphRAG involves creating a **knowledge graph** of nodes and relationships contained in unstructured data.

RAG

Nodes

- Nodes represent **things**
- Nodes are grouped by **labels**
- Nodes are described by **properties** as key-value pairs

Nodes represent things

> *A boy with the `description` of **City Boy***
> *A Location with the `name` **South Detroit**.*

Relationships

- Any two nodes can be connected by a **relationship**
- Each relationship has a **type** and a **direction**
- Relationships have **properties** as key-value pairs

Relationships describe how things are connected

> *The City Boy* ***took*** *the Midnight Train* ***going*** *Anywhere.*

Cypher

Cypher is a *A Powerful & Expressive Query Language* for querying graphs.

a cypher statement and equivalent graph model

Steps of GraphRAG

The term Graph RAG encapsulates the process of **extracting nodes and relationships** from unstructured text, which sit along the vector embeddings in the knowledge graph.

The knowledge graph structure can be **enriched with additional features** derived from graph algorithms in the Neo4j Graph Data Science library, providing deeper insights into the data's patterns and connections.

Then **querying the resulting knowledge graph**, sometimes in combination with vector search, to retrieve the necessary information for the task.

We will explore these points in more detail as we progress through the course.

Building a Graph

Introduction

In this lesson, we will explore how to transform unstructured documents into knowledge graphs using Neo4j's GraphRAG Library.

The **Neo4j GraphRAG Library** provides developers with methods to build and query knowledge graphs.

The library allows you to build pipelines to:

- Chunk text documents and PDFs and create vector embeddings
- Extract entities and relationships from the chunks and store them in a knowledge graph
- Query vector embeddings and return the surrounding nodes and relationships

sh:

```sh
pip install "neo4j_graphrag[openai]"
```

The Source: EDGAR SEC Filings

EDGAR (Electronic Data Gathering, Analysis, and Retrieval) SEC filings are official documents that public companies must submit to the U.S. Securities and Exchange Commission (SEC).

These documents are the authoritative source for company information, containing details about:

- Financial performance and metrics
- Business operations and strategy
- Risk factors and challenges
- Executive leadership and compensation
- Corporate governance

The Challenge:
How do you extract structured insights from these comprehensive but text-heavy regulatory documents that contain crucial business intelligence?

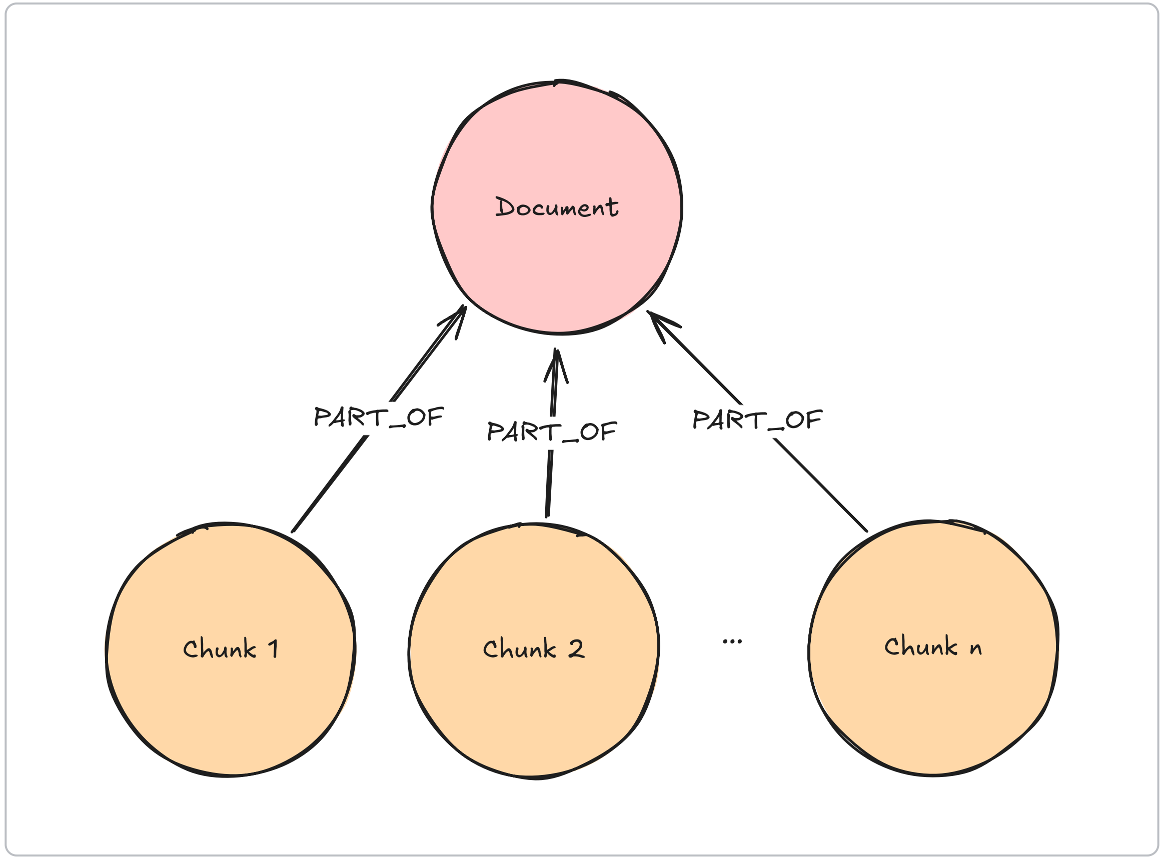
a screenshot of an Apple SEC filing PDF document.

Step 1: Documents and Chunks

Documents in your knowledge graph are the original PDF files that were processed.

Chunks are smaller, semantically meaningful segments of text extracted from each document.

This is known in GraphRAG as a **Lexical graph**.

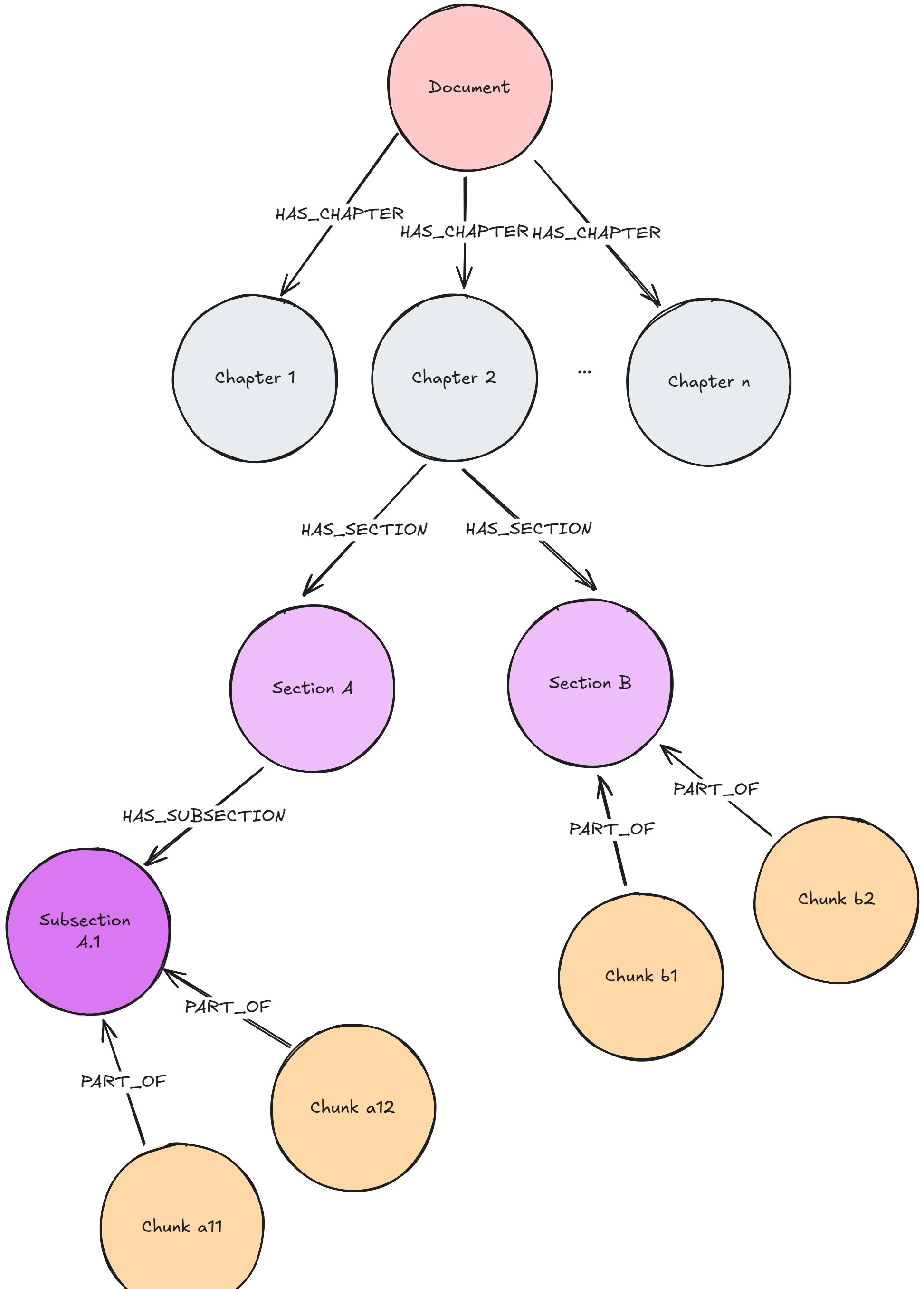


Lexical graphs with hierarchical structure

Documents are also inherently **hierarchical**.

A book will contain chapters, which are in turn a collection of sections, which are in turn a collection of paragraphs.

If the given documents have pre-defined structures, it is useful to persist them into the chunk structure.


Document
HAS_CHAPTER
HAS_CHAPTER
HAS_CHAPTER
Chapter 1
Chapter 2
...
Chapter n
HAS_SECTION
HAS_SECTION
Section A
Section B
HAS_SUBSECTION
PART_OF
PART_OF
Subsection A.1
Chunk b2
Chunk b1
PART_OF
PART_OF
Chunk a12
Chunk a11

Step 2: Guided Extraction Prompts

The extraction process uses **carefully crafted prompts** to ensure quality and accuracy. The following principles are crucial for achieving high-quality results:

- **Company Validation:** Only extract approved companies from predefined lists
- **Context Resolution:** Resolve generic references like "the Company" to actual names
- **Schema Enforcement:** Strict adherence to defined entity types and properties
- **Quality Control:** Validate all extracted relationships against schema

This schema + prompt combination acts as the blueprint - telling the LLM exactly what to look for and how to connect entities in the knowledge graph you'll explore.

An example prompt

Here's how to create effective extraction prompts:

python:

```python
from neo4j_graphrag.experimental.pipeline.kg_builder import ERExtractionTemplate


company_instruction = (
    "You are an expert in extracting company information from SEC filings. "
    "When extracting, the company name must match exactly as shown below. "
    "ONLY USE THE COMPANY NAME EXACTLY AS SHOWN IN THE LIST. "
    "If the text refers to 'the Company', 'the Registrant', or uses a pronoun, "
    "you MUST look up and use the exact company name from the allowed list. "
    "UNDER NO CIRCUMSTANCES should you output 'the Company' or generic phrases. "
)
# Combine with default template
custom_template = company_instruction + ERExtractionTemplate.DEFAULT_TEMPLATE
prompt_template = ERExtractionTemplate(template=custom_template)
```

Step 2: Schema-Driven Extraction

Entity Types:

- Company
- Executive
- Product
- FinancialMetric
- RiskFactor
- StockType
- Transaction
- TimePeriod

python:

```python
entities = [
  {
    "label": "Company",
    "properties": [
      {"name": "name", "type": "STRING"}
    ]
  },
  {
    "label": "Executive",
    "properties": [
      {"name": "name", "type": "STRING"}
    ]
  },
  # ... more entities
]
```

Step 2: Defining the Relationships

Relationships:

- Company **HAS_METRIC** FinancialMetric
- Company **FACES_RISK** RiskFactor
- Company **ISSUED_STOCK** StockType
- Company **MENTIONS** Product

python:

```python
relations = [
  {
    "label": "HAS_METRIC",
    "source": "Company",
    "target": "FinancialMetric"
  },
  {
    "label": "FACES_RISK",
    "source": "Company",
    "target": "RiskFactor"
  },
  # ... more relationships
]
```

Step 3: The GraphRAG Pipeline

The complete pipeline defines the transformation from PDF to knowledge graph using LLM-powered extraction.

The GraphRAG Pipeline:

Diagram showing the Neo4j GraphRAG pipeline process from PDF documents to knowledge graph

Step 3: SimpleKGPipeline Example

python: Connect to the Neo4j database

```python
from neo4j import GraphDatabase

NEO4J_URI = "{instance-scheme}://{instance-host}:{instance-boltPort}"
NEO4J_USERNAME = "{instance-username}"
NEO4J_PASSWORD = "{instance-password}"

driver = GraphDatabase.driver(
  NEO4J_URI,
  auth=(NEO4J_USERNAME, NEO4J_PASSWORD)
)
```

- Use the Neo4j Python Driver to connect to a Neo4j database

Step 3: SimpleKGPipeline Example

python:

```python
from neo4j_graphrag.embeddings import OpenAIEmbeddings
from neo4j_graphrag.llm import OpenAILLM


# Create embedder and LLM
embedder = OpenAIEmbeddings(
  model="text-embedding-3-large"
)


llm = OpenAILLM(
    model_name="gpt-4o",
    model_params={
        "max_tokens": 2000,
        "response_format": {"type": "json_object"},
        "temperature": 0,
    },
)
```

- Choose an embedding service to create embeddings from the text
- Choose an LLM to use for the entity and relationship extraction

Step 3: SimpleKGPipeline Example

Create an extraction pipeline with `SimpleKGPipeline`.

python: **Create the pipeline**

```python
from neo4j_graphrag.experimental.pipeline.kg_builder import SimpleKGPipeline


# Instantiate the SimpleKGPipeline with full configuration
kg_builder = SimpleKGPipeline(
  driver=driver,
  llm=llm, embedder=embedder,       # LLM and embedding service
  prompt_template=prompt_template,  # Custom prompt from previous step
  entities=entities,                # Entity schema
  relations=relations,              # Relationship schema
  from_pdf=False,                   # Set to True for PDF processing
)
```

Step 3: SimpleKGPipeline Example

python:

```python
text = """
Neo4j is developed by Neo4j, Inc., based in San Mateo, California, United States.
In November 2016, Neo4j secured $36M in Series D Funding led by Greenbridge Partners.
In November 2018, Neo4j secured $80M in Series E Funding led by One Peak Partners.
"""

result = await kg_builder.run_async(text=text)
```

The function returns a summary of nodes to extract and the number of nodes created.

```
{
  'resolver': {'number_of_nodes_to_resolve': 42, 'number_of_created_nodes': 18}
}
```

Step 3: SimpleKGPipeline Example

What happened during `pipeline.run()`:

1. **PDF Text Extraction:** Extracted raw text from PDF documents
2. **Document Chunking:** Broke text into semantically meaningful chunks
3. **Entity Extraction:** Used LLM to identify companies, metrics, risks, etc.
4. **Relationship Extraction:** Found connections between entities
5. **Graph Storage:** Saved structured entities and relationships to Neo4j
6. **Vector Embeddings:** Generated embeddings for chunks and stored them

Step 3: Verify Entity Extraction

Verify Entity Extraction:

cypher:

```cypher
// Count what entities were extracted by type
MATCH (e)
WHERE NOT e:Document AND NOT e:Chunk
RETURN labels(e) as entityType, count(e) as count
ORDER BY count DESC
```

Step 4: Enriching the Graph with Structured Data

PDF extraction is only part of the story. The knowledge graph built by entity extraction can be enhanced with structured data loaded from CSV files.

Structured Data Sources:

- **Asset Manager Holdings:** Ownership information connecting asset managers to companies
- **Company Filing Information:** Metadata linking companies to their PDF documents

Why Both Data Types?

- **Unstructured (PDFs):** Rich content about companies, risks, metrics
- **Structured (CSVs):** Precise ownership data and document relationships

This creates a complete picture: detailed company information from PDFs **plus structured ownership and filing relationships**.

Step 4: Sample Structured Data

Asset Manager Holdings (Sample Data):

managerName	companyName	ticker	Value	shares
ALLIANCEBERNSTEIN L.P.	AMAZON COM INC	AMZN	$6,360,000,000	50,065,439
ALLIANCEBERNSTEIN L.P.	APPLE INC	AAPL	$4,820,000,000	28,143,032
AMERIPRISE FINANCIAL INC	ALPHABET INC	GOOG	$4,780,000,000	36,603,757
BlackRock Inc.	AMAZON COM INC	AMZN	$78,000,000,000	613,380,364
FMR LLC	MICROSOFT CORP	MSFT	$68,200,000,000	215,874,152

Step 4: Sample Structured Data

Company Filing Information (Sample Data):

name	ticker	cusip	cik	form10KUrls
AMAZON	AMZN	23135106	1018724	0001018724-23-000004.pdf
NVIDIA Corporation	NVDA	067066G104	1045810	0001045810-23-000017.pdf
APPLE INC	AAPL	3783310	1490054	0001096906-23-001489.pdf
PAYPAL	PYPL	1633917	1633917	0001633917-23-000033.pdf
MICROSOFT CORP	MSFT	594918954	789019	0000950170-23-035122.pdf

Loading Structured Data

1. **Neo4j Data Importer** to map CSV files to the existing knowledge graph
2. **AssetManager nodes** were created from holdings data
3. **OWNS relationships** connected asset managers to companies with holding values
4. **FILED relationships** linked companies to their PDF documents

Exploring What Was Created

Now that we've seen how to build a knowledge graph, let's explore one. You have access to a knowledge graph that contains:

The Complete Data Model:

- **500+ Company entities** extracted from SEC filings
- **Asset Manager entities** with ownership information
- **2,000+ Financial metrics and risk factors** as structured nodes
- **Clear entity relationships** connecting business concepts
- **Document links** bridging structured and unstructured data

Visualize the Complete Schema:

cypher:

```cypher
CALL db.schema.visualization()
```

Explore a Complete Company Profile

Explore a Complete Company Profile:

cypher: See how all three data types connect for one company

```cypher
MATCH (c:Company {name: 'APPLE INC'})
RETURN c.name,
  COUNT { (c)-[r1]->(extracted) WHERE NOT extracted:Chunk AND NOT extracted:Document } AS extractedEntities,
  COUNT { (:AssetManager)-[:OWNS]->(c) } AS assetManagers,
  COUNT { (c)<-[:FROM_CHUNK]->(chunk:Chunk) } AS textChunks
```

Key Takeaways

✅ **Unstructured → Structured:** PDF text will be transformed into business entities and relationships
✅ **Schema-Driven:** Clear entity definitions will guide accurate extraction
✅ **AI-Powered:** LLMs will identify and extract meaningful business concepts
✅ **Relationship-Aware:** Connections between entities will be preserved and made explicit
✅ **Data Model Ready:** Clean, structured data will be prepared for the knowledge graph you'll explore

A structured data model is the foundation for everything that follows. Without it, you would still have inflated, unprecise, unstructured text rather than a repository of facts that the text attempts to convey!

GraphRAG Retrievers

Introduction

You are going to explore how to retrieve information from the knowledge graph using *retrievers*.

This lesson covers the three main types of retriever that are available in the Neo4j GraphRAG library.

What is a Retriever?

A **retriever** is a component that searches and returns relevant information from your knowledge graph to answer questions or provide context to language models.

The Three Types:

- **Vector Retriever:** Semantic search across text chunks
- **Vector + Cypher Retriever:** Semantic search + graph traversal
- **Text2Cypher Retriever:** Natural language to Cypher queries

Vector Retriever

How it works:

- Converts your question into a vector embedding using the `embedder`
- Searches the `chunkEmbeddings` vector index for similar content
- Returns semantically related text chunks based on cosine similarity
- Pure semantic search - no graph traversal

python:

```python
from neo4j_graphrag.retrievers import VectorRetriever


vector_retriever = VectorRetriever(
    driver=driver,
    index_name='chunkEmbeddings',
    embedder=embedder,
    return_properties=['text']
)
```

Vector Retriever

Best for:

- Finding conceptually similar information across all documents
- Semantic search when exact keywords don’t match
- Broad exploratory questions about topics
- When you don’t know specific entity names

Example Query: *"What are the risks that Apple faces?"*

Limitations:

- Returns only text chunks, no entity relationships
- May miss entity-specific context
- Cannot aggregate information across multiple entities

Vector + Cypher Retriever

How it works:

- **Step 1:** Vector search finds semantically relevant text chunks
- **Step 2:** Custom Cypher query traverses from each chunk to related entities
- **Step 3:** Returns enriched context including entities, relationships, and metadata
- Combines semantic relevance with graph intelligence

python:

```python
from neo4j_graphrag.retrievers import VectorCypherRetriever

detailed_context_query = """
WITH node
MATCH (node)-[:FROM_DOCUMENT]-(doc:Document)-[:FILED]-(company:Company)-[:F
RETURN company.name AS company,  node.text AS context, collect(DISTINCT ris
"""

vector_cypher_retriever = VectorCypherRetriever(
    driver=driver,
    index_name='chunkEmbeddings',
    embedder=embedder,
    retrieval_query=detailed_context_query
)
```

Vector + Cypher Retriever

Best for:

- Getting both content and rich contextual information
- Understanding relationships between entities mentioned in chunks
- Questions requiring entity-specific aggregations
- Comprehensive answers that need multiple connected data points

Example Query: *"Which asset managers are most affected by cryptocurrency policies?"*

Why "Apple" Queries Can Fail in Vector-Cypher

The Challenge:

When you ask *"What are the risks that Apple faces?"* using Vector + Cypher, you may not get Apple-specific results.

Why this happens:

- Vector search finds chunks semantically similar to your query
- If those chunks aren’t about Apple, the Cypher query won’t reach Apple entities
- **The chunk is the anchor** - you can only traverse from what you retrieve

Key Insight:

Vector-Cypher works best when your question naturally surfaces relevant chunks about the entities you’re interested in.

Good Vector-Cypher Query Example

Query: *"Which asset managers are most affected by banking regulations?"*

Why this works well:

- Vector search finds chunks about "banking regulations"
- Cypher query traverses to asset managers connected to those companies
- Returns both the regulatory context AND the asset manager entities

Cypher pattern:

cypher:

```cypher
WITH node
MATCH (node)-[:FROM_DOCUMENT]-(doc:Document)-[:FILED]-(company:Company)-[:OWNS]-(manager:AssetManager)
RETURN company.name AS company, manager.managerName AS AssetManager, node.text AS context
```

Text2Cypher Retriever

How it works:

- Uses an LLM to convert natural language questions into Cypher queries
- Leverages the graph schema to understand available entities and relationships
- Executes the generated Cypher query directly against Neo4j
- Returns structured, precise results from the graph

python:

```python
from neo4j_graphrag.retrievers import Text2CypherRetriever


text2cypher_retriever = Text2CypherRetriever(
    driver=driver,
    llm=llm,
    neo4j_schema=schema
)
```

Text2Cypher Retriever

Example Query: *"What are the company names of companies owned by Berkshire Hathaway Inc?"*

Generated Cypher:

cypher:

```
MATCH (am:AssetManager {managerName: 'Berkshire Hathaway Inc'})-[:OWNS]->(c:Company)
RETURN c.name AS company_name
```

Text2Cypher Retriever

Best for:

- Precise, entity-centric questions
- When you need exact data (numbers, dates, counts, names)
- Aggregations and analytical questions
- Direct graph queries without semantic search

Limitations:

- Requires good graph schema understanding
- May struggle with ambiguous natural language
- Less effective for open-ended or exploratory questions

Choosing the Right Retriever

Use Vector Retriever when:

- You want semantic similarity search
- Question is conceptual or broad
- You need to find related topics

Use Vector + Cypher when:

- You want both content and relationships
- Need comprehensive context
- Question involves multiple entities

Use Text2Cypher when:

- You need precise, structured data
- Question asks for specific facts or numbers
- You want to leverage graph relationships directly

Try it yourself

In the next lessons, we will work through the notebooks and see how to use the retrievers in practice.

Introduction to GraphRAG Workshop → Building GraphRAG Agents

Congratulations!

Congratulations!

You've completed the GraphRAG Introduction workshop!

In this hands-on workshop, you've learned the fundamentals of GraphRAG - a technique that combines graph databases with generative AI to enhance LLM-generated content. You've explored how GraphRAG addresses limitations of traditional vector-based RAG by providing additional context through graph relationships and structured data.

What you’ve learned

You’ve gained practical experience with:

- **GraphRAG fundamentals** - Understanding how knowledge graphs enhance retrieval-augmented generation
- **Knowledge graph construction** - Extracting structured information from unstructured data and storing it effectively
- **Three types of retrievers** - Vector, Vector + Cypher, and Text2Cypher retrievers, each with specific strengths and use cases
- **LangGraph integration** - Using prebuilt ReAct agents to connect seamlessly with Neo4j
- **Real-world applications** - Working with financial documents and exploring practical GraphRAG implementations

You now have the foundational knowledge and hands-on experience needed to start building GraphRAG applications that use graph databases to enhance AI responses.

Deepen Your Graph Database Knowledge

Continue learning with additional courses on GraphAcademy that build upon what you've learned:

- **Neo4j Fundamentals** - Learn the core concepts of Neo4j, including database architecture, data modeling principles, and graph database design practices
- **Cypher Fundamentals** - Learn Cypher, Neo4j's query language, covering basic pattern matching through advanced query optimization
- **Graph Data Modeling Fundamentals** - Learn how to design graph schemas, model relationships, and optimize graph structure for performance
- **Importing Data Fundamentals** - Learn techniques for loading data from various sources into Neo4j, including CSV files, APIs, and data streams

Advanced GraphRAG Topics

Continue your GraphRAG journey with specialized courses:

- **Building Knowledge Graphs** - Learn techniques for constructing knowledge graphs from data sources using LLMs and NLP pipelines
- **Using Neo4j with LangChain** - Build conversational AI applications that use Neo4j's graph capabilities through LangChain integrations
- **Vectors and Unstructured Data** - Learn about vector embeddings, semantic search, and hybrid search strategies combining vector similarity with graph traversal

Start Building with Neo4j

Get hands-on experience with your own Neo4j environment. You can use **Neo4j Aura**, Neo4j's managed cloud service, which offers a free tier that includes up to 200,000 nodes and 500,000 relationships. With Aura, there is no setup required, allowing you to deploy instances quickly. It also provides automatic scaling, backups, and security updates, making it suitable for both prototyping and production applications.

Alternatively, you can use **Neo4j Desktop**, a local Neo4j development environment. Neo4j Desktop provides graph visualization tools, supports multiple database instances, and includes built-in monitoring and performance analysis tools. This makes it ideal for development, testing, and learning.

Neo4j & Model Context Protocol (MCP)

Enhance your development workflow with AI-powered tools. The course **Developing with Neo4j MCP Tools** teaches you how to use the Model Context Protocol to create intelligent AI applications with Neo4j's MCP server and tools for natural language database interaction.

The **Neo4j MCP Server** enables AI assistants like GitHub Copilot and Claude to interact with your Neo4j database using natural language. With MCP Server, you can query your database using conversational language, generate Cypher queries from descriptions, explore graph schemas and relationships, and build GraphRAG applications with AI assistance.

You can also connect MCP tools to your development environment for seamless AI-assisted coding with Neo4j, thanks to IDE integration.