EBOOK

THE DEVELOPER'S GUIDE TO

GraphRAG

Alison Cossette
Zach Blumenfeld
Damaso Sanoja

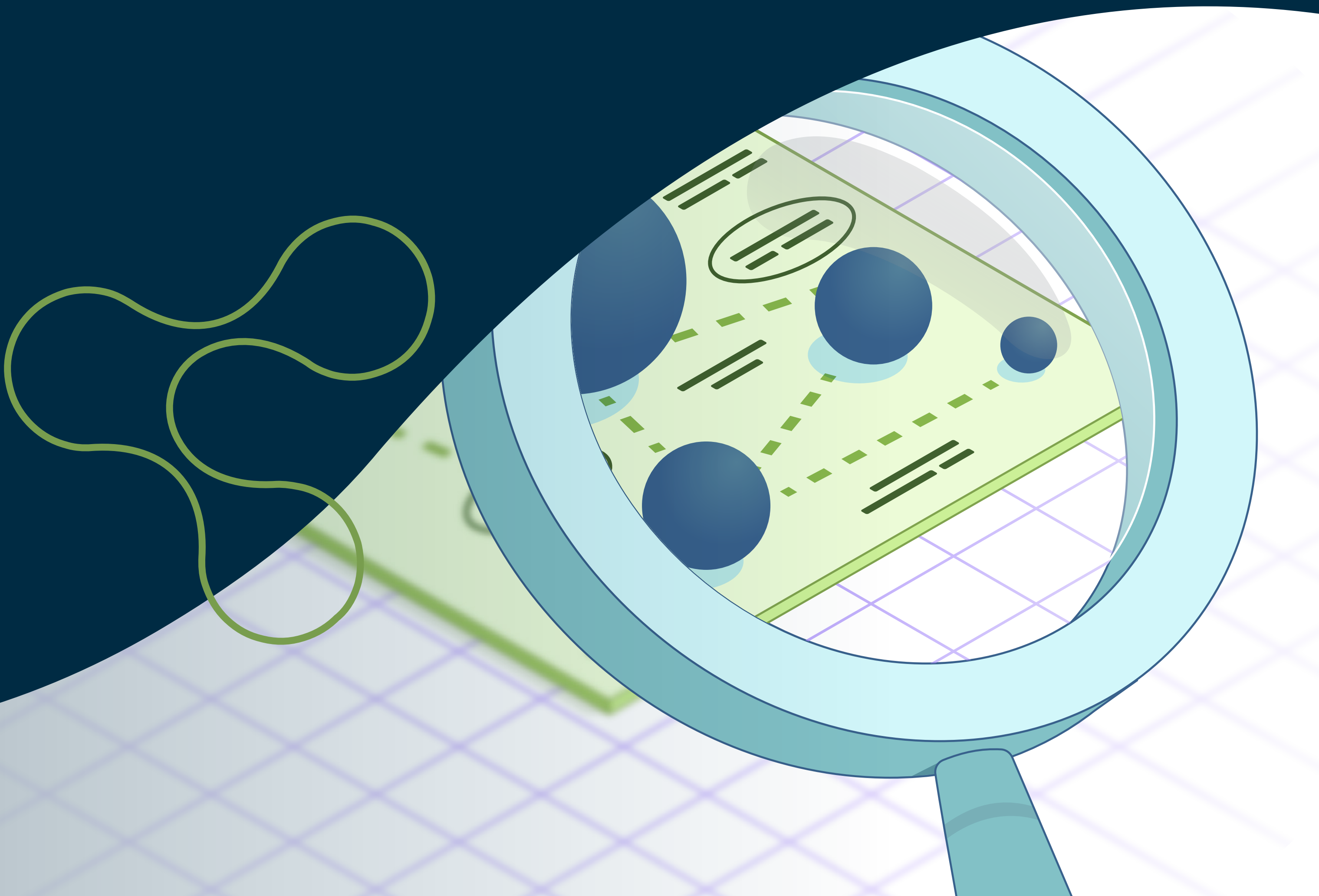


Table of Contents

Table of Contents (continued)

PART I: The Problem With Current RAG

Why chunk-based RAG hits a ceiling — and why developers need more context to answer well

You've built a retrieval-augmented generation (RAG) system. You embedded the docs, connected the vector store, wrapped a prompt around the output, and deployed it. For a minute, it felt like you cracked the code. The model was grounded in your own data, giving answers that sounded smarter than base GPT.

Then reality hit.

The system works — but only under the most forgiving conditions. The moment you ask a question that spans documents, relies on implicit context, or touches anything complex or structured, the cracks start to show. Answers get vague. Sometimes they're just plain wrong. Or worse, the system confidently quotes the right chunk — but misses the point entirely.

Your RAG system isn't broken. It's just blind.

RAG retrieves semantically similar text, but it doesn't know how the pieces fit together.

It has no map of your domain. No memory of what matters. It's like hiring a new developer and giving them a stack of index cards with code snippets from your repo. They can parrot back functions, maybe even modify them, but they don't understand the architecture. They don't know the "why," only the "what."

That's the ceiling of traditional RAG. And that's what this book is here to fix.

Here's the core issue: RAG retrieves based on similarity, not understanding.

You give it a query, it vectorizes that query, and fetches the top-k similar chunks. That's fine if the answer you need lives entirely within isolated chunks. But most real-world questions don't work that way.

Let's say a user asks about a contract clause, but the meaning depends on a sales addendum from three weeks earlier. Or maybe they ask a support question that only makes sense in the context of their infrastructure and license tier. The information is there, but it's scattered across multiple documents, formats, and timelines. Chunk-based retrieval can't bridge that gap.

Traditional RAG doesn't have shared context across documents. That's because it doesn't track relationships. It doesn't know which concepts are upstream, downstream, dependent, or mutually exclusive. It doesn't distinguish between definitions, instructions, timelines, policies, or decision logic.

The bottom line: Traditional RAG treats all chunks as equal, flat, unstructured blobs of text.

Even more problematic is that the system has no mental model for your business. It cannot understand what a "customer" is in your world. Or how a support ticket relates to a contract. Or what a system diagram implies about downstream integrations. The mental model that represents the structure behind your content is absent in RAG.

Without it, RAG can't reason. It can only retrieve, and that isn't enough.

You already know what your RAG system should be able to do. It's the kind of reasoning your team does every day without thinking. Consider this: If a customer reaches out to your support team, the employee will listen to the customer's concern, look up their account and tech stack, check previous service requests, etc. When answering the customer's question, the employee brings context. They may answer differently if the person is a new customer vs. a long-term customer.

You want your RAG application to do what humans do naturally: use context to inform its answer. As examples, you might want the RAG system to:

- Answer a support question and understand the user's tech stack, contract level, and product version.
- Explain a contract term — and know what the sales path looked like, who signed off, and which systems were impacted.

- Interpret a customer review and place it in context with purchase history, usage data, and net promoter score (NPS).

These shouldn't feel like advanced use cases — they're basic context. They're what you, as a human developer, bring into every decision without even realizing it. And that's the problem: Your RAG system has none of that. Sure, it has some document metadata available, but no user metadata, no business logic, no connected data — just isolated chunks in a vector store. But RAG can't use what it can't see. So until you give it structure — until you teach it relationships, timelines, ownership, and dependencies — it will keep retrieving the right words for the wrong reasons.

This isn't a whitepaper. It's a build-it-yourself playbook.

We're going to walk you through:

- Ingesting documents and turning them into a knowledge graph
- Structuring real-world context from messy PDFs, CSVs, and APIs
- Building retrievers that combine vector search and graph traversal
- Using text-to-query generation to run dynamic Cypher queries (a query language for graphs) and pull precise information and calculations from your data

And we're going to do it with code. No fluff. Just the stack, the logic, and the patterns that actually work. If you've built RAG, and you know it's not enough, then this is the guide to take you further.

PART II: What Makes It GraphRAG – Structure, Logic, and Meaning

To understand GraphRAG, let's explore its foundational components — RAG and knowledge graphs — and why they work so well together.

What Is RAG?

Let's start with the well-known problems of large language models (LLMs), which power chatbots such as ChatGPT, Gemini, and Claude. When a user's prompt goes directly to the LLM, it generates a response based on its training data. Due to the probabilistic nature of response generation, LLMs often produce responses that lack accuracy and nuance and don't draw on knowledge specific to your business. In addition, the LLM in question may have limited explainability, which limits its adoption in enterprise settings.

RAG addresses these challenges by intercepting a user's prompt, querying external data, usually a vector store, and passing relevant documents back to the LLM. Adding retrieval to the LLM enables the application to answer questions with knowledge from a specific dataset. This simple technique suddenly makes it possible to build applications for a variety of use cases. As examples:

- Knowledge assistants can tap into company-specific information for accurate, contextual responses.
- Recommendation systems can incorporate real-time data for more personalized suggestions.
- Search APIs can deliver more nuanced and context-aware results.

RAG consists of three key components:

- An LLM that serves as the generator
- A knowledge base or database that stores the information to be retrieved
- A retrieval mechanism to find relevant information from the knowledge base, based on the input query

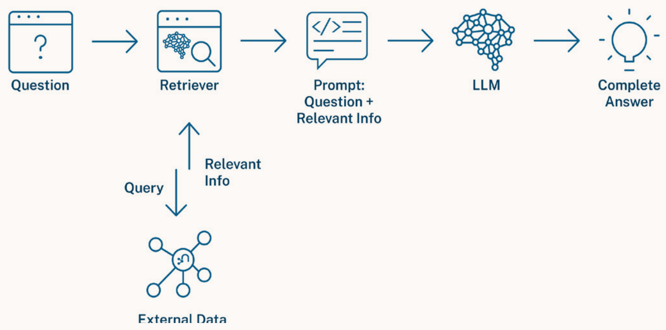


Figure 1. Querying a knowledge graph with an LLM

The quality of a RAG response depends heavily on the database type the information is retrieved from. If you use a vector store (as in traditional RAG), the process goes like this: The user query is turned into a vector, which is then used to retrieve semantically similar text chunks from a vector database. While retrieval based on semantic similarity can work across multiple documents, it often falls short when questions require understanding implicit context or relationships that span those documents. Traditional RAG treats each chunk in isolation, as it lacks a holistic view of the domain.

Retrieval based on semantic similarity can only get you so far. And this is where GraphRAG comes in. GraphRAG gives the LLM a mental model of your domain so that it can answer questions by drawing on the correct context.

What Is GraphRAG?

In GraphRAG, the knowledge base used for retrieval is a knowledge graph. A knowledge graph organizes facts as connected entities and relationships, which helps the system understand how pieces of information relate to each other.

The knowledge graph becomes a mental map of your domain, providing the LLM with information about dependencies, sequences, hierarchies, and meaning. This makes GraphRAG especially effective at answering complex, multi-step questions that require reasoning across multiple sources.

Imagine that a customer calls to request support regarding a recent purchase. Customer Service uses an internal chatbot to troubleshoot the request. A traditional system built on vector-only RAG would retrieve a product name from the customer support ticket:

Service Ticket	Service Ticket Text	Embedding
234381	My new JavaCo coffee maker isn't working.	[.234, .789, .123......]

But that's all the RAG system would surface.

A GraphRAG system, on the other hand, would show not only this service ticket text but also the customer's purchase history, known issues with that product version, related documentation, and prior support conversations.

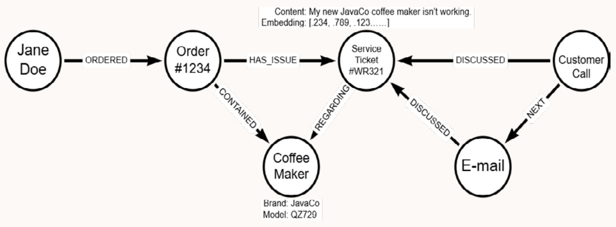


Figure 2. Order issue flow

A knowledge graph holds all related information together across both structured and unstructured data. A RAG system built on a knowledge graph — or GraphRAG — excels at generating context-aware responses.

The main reasons to implement a GraphRAG solution include:

1. **Context-Aware Responses**

   Unlike traditional RAG, which retrieves isolated chunks of text based on similarity, GraphRAG retrieves facts in context. Since the knowledge graph explicitly encodes relationships, GraphRAG returns relevant information, as well as related information. This structured retrieval ensures that application outputs are comprehensive, reducing hallucinations and leading to more accurate, reliable outputs and improving real-world applicability.

2. **Traceability and Explainability**

   LLMs and even standard RAG approaches operate as black boxes, making it difficult to know why and how a certain answer was generated. GraphRAG increases transparency by structuring retrieval paths through the knowledge graph. The knowledge graph will show the sources and relationships that contributed to a response. This makes it easier to audit results, build trust, and meet compliance needs.

3. **Access to Structured and Unstructured Data**

   GraphRAG overcomes a key limitation of vector-only RAG by integrating both structured and unstructured data. It integrates information like whole databases, ontologies, documents, and real-time streams into a single knowledge graph. Richer data means superior AI responses.

How GraphRAG Works

GraphRAG works by using a knowledge graph to retrieve and connect relevant information. It starts with a search — vector, full-text, spatial, or others — to find entry points in the graph, then follows related nodes and relationships to gather more context. The system considers the user's task and filters and ranks the results before passing them to the generation phase.

Think of GraphRAG as a RAG architecture built on a knowledge graph. Using a knowledge graph affects the way you design the entire solution. There are two main steps to creating a GraphRAG application:

1. **Preparing a knowledge graph for GraphRAG**
   - Documents and unstructured text ingestion
   - Structured data source import
2. **Implementing GraphRAG retrieval patterns**

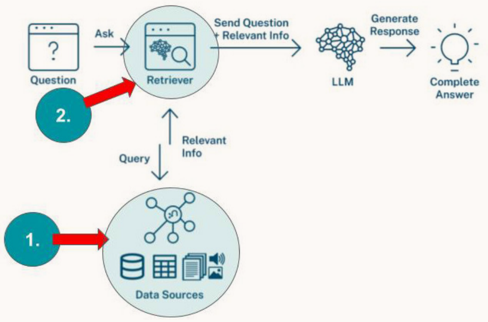


Figure 3. Implementing GraphRAG retrieval patterns flow

The rest of this book walks you through these two critical steps.

Prepare a Knowledge Graph for GraphRAG

Effective retrieval in GraphRAG starts with a well-structured knowledge graph. The data needs to be structured to model the business domain as it relates to the documents. That means having a clear data model that defines both the content you're working with and how it is connected.

There are two aspects to consider when you're modeling a knowledge graph for AI workflows:

1. **The relationships between documents — or how your content is organized and related:**
   - How chunks connect to source documents
   - How sections of a book or catalog are structured
   - How content is grouped or nested
2. **Business entities and logic:**
   - The core entities (i.e., Customers, Products, Companies)
   - How these entities relate to each other
   - The structure and relationships that already exist in your current databases, schemas, or business logic

These two layers — the document structure and the business domain — work together to give GraphRAG its power. GraphRAG is retrieving documents in the context of your business. Consider a customer review in context of their purchase history or a user's question in context of their technical stack.

The first step is to determine where you can access that business domain and how to connect it to your documents. It might be well defined in your structured data (databases, business hierarchies, etc.) or it may be hidden inside your unstructured content (i.e., contract terms, product features). A knowledge graph brings it all together, connecting the dots so your LLM retrieves not just semantic similarity but also relevant facts.

Ground With Unstructured and Structured Data

If you've worked with RAG systems, you're already familiar with vector databases and unstructured content — PDFs, contracts, reports. But the most important context for your data rarely lives in a single format. In fact, most of the time you'll want to use more than just unstructured data. Structured data like CRM exports, product catalogs, and relational databases often contains crucial grounding information for the answers your users need.

To build systems that retrieve the right answer at the right time, you need to **connect two worlds: unstructured and structured.** That's where knowledge graphs come in. By linking unstructured chunks to structured business entities and relationships, you create a semantic network that makes retrieval smarter, safer, and more transparent. So, where do you start? With your documents or your structured schema?

Technically, you can begin from either side. But in practice, most teams start with unstructured data because that's where the buried context usually lives. Think financial disclosures, legal contracts, emails, and support tickets. These contain implicit business logic, risk factors, and decision-making signals that don't show up in structured rows and columns.

But here's the catch: Structure isn't binary. It's a continuum.

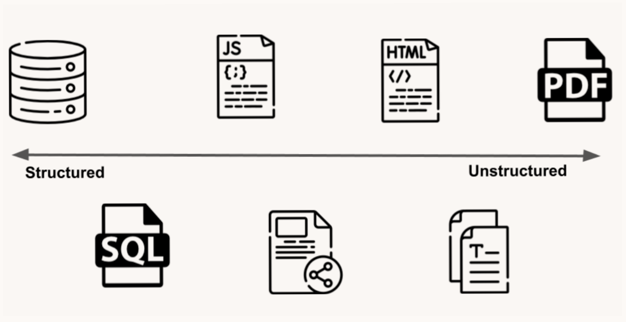


Figure 4. Structured and unstructured data continuum

At one end, you've got relational databases and clean CSV, where entities and relationships are explicitly defined. At the other end, you've got raw text: meaning buried in natural language. In between? A complex middle: XML files, JSON logs, form submissions, and mixed-format documents with both tables and prose.

As you think about your own dataset, ask yourself these questions: Where does the context for your application actually live? **And where on the structure continuum does it fall?** These questions matter because they will help you determine the tools you should use to build the knowledge graph. For this guide, you'll use:

- Neo4j Data Importer (Neo4j Aura Platform) for structured data
- Knowledge Graph Builder Pipeline (Neo4j GraphRAG Python Package) for extracting implicit relationships from natural language

If you find that your dataset has more complex data structures, you can consider adding tools to your workflow. This is an ever-evolving field, and many are working on building tools for these scenarios. A few to consider:

Took	Description	Resource
Unstructured.io	Extracts structured data (tables, lists, key-value pairs) from unstructured documents like PDFs, HTML, and email	Neo4j Integration Guide
Boundary's Annotation Modeling Language (BAML)	Declarative language for extracting structured data from unstructured sources, demonstrated with Neo4j	BAML to Neo4j Tutorial by Jason Koo
pdfplumber	Parses tables and text from PDF files, ideal for extracting structured data from documents	GitHub Repository
LangChain	Framework for developing applications powered by language models, with support for Neo4j integration	Neo4j Integration

For this exercise, you'll start with unstructured financial documents. Using an LLM-powered pipeline to extract entities like Company and Risk Factor, you'll look for relationships such as `FACES_RISK` to build a knowledge graph in Neo4j. This process mirrors what many teams face: extracting meaning from dense reports, contracts, or disclosures.

You'll then use Neo4j's Data Importer to load structured datasets — the kind of CSVs or database connectors most companies already have — further enriching the graph with known entities and relationships.

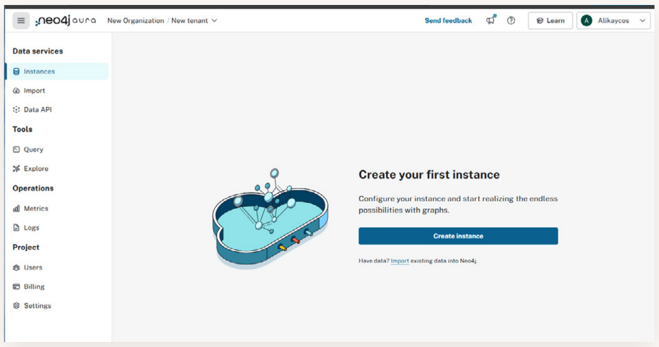
Finally, you'll test retrieval strategies, from vector search to graph-enhanced queries, to dynamic Cypher generation with Text2Cypher. The same process can be applied to your own PDFs, internal databases, and business domain to build a semantic layer over enterprise knowledge, making it accessible to GenAI systems with precision, transparency, and context.

PART III: Constructing the Graph

Create a Neo4j Database

Begin by choosing a Neo4j database solution that fits your needs. Options include a free instance of AuraDB or a free trial of AuraDB Professional. Neo4j is also available on all the major cloud partner marketplaces. When you navigate to https://console.neo4j.io and log in, you'll see the following screen, inviting you to create your first instance.

> **Tip:** *Download your AuraDB credentials (URI, username, password) immediately after creating the instance. They will not be available for download later. Store them securely, as you'll need them to connect your application to Neo4j.*



Create your first instance

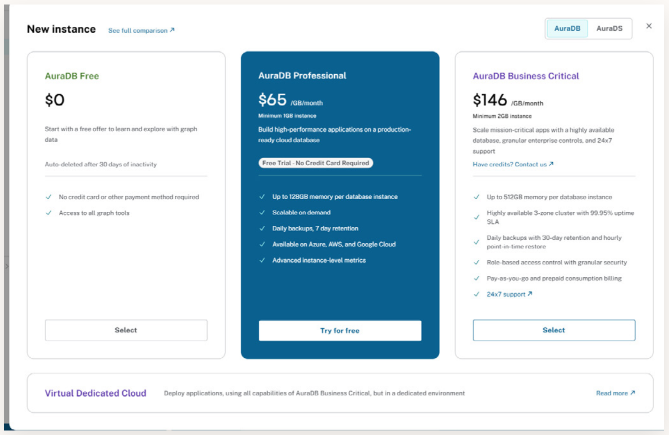
Configure your instance and start realizing the endless possibilities with graphs.

Have data? Import existing data into Neo4j.

Figure 5. Create your first instance screen

You then have three choices of instances to choose from:

- **AuraDB Free,** a small database (2 GB) that will always be free, though it will be deleted after 30 days of no activity.
- **AuraDB Professional** offers up to 128 GB of memory and a free 14-day trial.
- **AuraDB Business Critical** is the most robust and offers up to 512 GB of memory and pay-as-you-go billing.



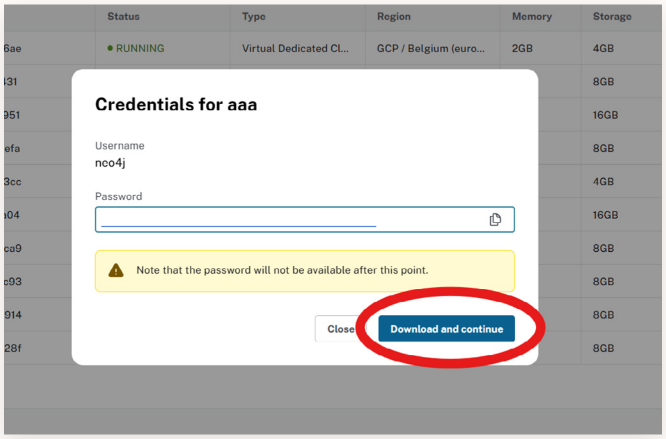
AuraDB Free	AuraDB Professional	AuraDB Business Critical
$0	$65 /GB/month	$146 /GB/month
	Minimum 1GB instance	Minimum 2GB instance
Start with a free offer to learn and explore with graph data	Build high-performance applications on a production-ready cloud database	Scale mission-critical apps with a highly available database, granular enterprise controls, and 24x7 support
Auto-deleted after 30 days of inactivity	Free Trial - No Credit Card Required	Have credits? Contact us ↗
✓ No credit card or other payment method required	✓ Up to 128GB memory per database instance	✓ Up to 512GB memory per database instance
✓ Access to all graph tools	✓ Scalable on demand	✓ Highly available 3-zone cluster with 99.95% uptime SLA
	✓ Daily backups, 7 day retention	✓ Daily backups with 30-day retention and hourly point-in-time restore
	✓ Available on Azure, AWS, and Google Cloud	✓ Role-based access control with granular security
	✓ Advanced instance-level metrics	✓ Pay-as-you-go and prepaid consumption billing
		✓ 24x7 support ↗

Virtual Dedicated Cloud — Deploy applications, using all capabilities of AuraDB Business Critical, but in a dedicated environment

Figure 6. New instance tiers

If you're just getting started, you'll do well with AuraDB Free or AuraDB Professional trial.

Be sure to download the credentials when you set up the database because they won't be available later on.



	Status	Type	Region	Memory	Storage
6ae	● RUNNING	Virtual Dedicated Cl...	GCP / Belgium (euro...	2GB	4GB
431					8GB
951					16GB
efa					8GB
3cc					4GB
04					16GB
ca9					8GB
c93					8GB
914					8GB
28f					8GB

Credentials for aaa

Username
nco4j

Password

Note that the password will not be available after this point.

Figure 7. Credential download and continue screen

Ingest Unstructured Data

As you begin to build your knowledge graph, you can use the Neo4j GraphRAG Python library. This package offers specialized functionalities that streamline and enhance the process of building a knowledge graph from unstructured data, such as PDFs. Capabilities include document chunking, embedding generation, and knowledge graph construction.

```
pip install neo4j-graphrag
```

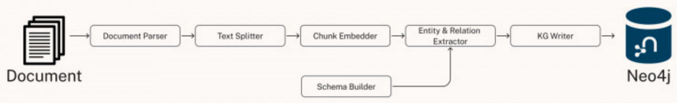


Figure 8. Document flow

Key Features of Neo4j GraphRAG Package

- **Knowledge Graph Construction Pipeline:** Automates the extraction of entities and relationships from unstructured text and structures them into a Neo4j graph.
- **Vector Indexing and Retrieval:** Facilitates the creation of vector indices for efficient semantic search within the graph.
- **Integration with LLMs:** Seamlessly integrates with LLMs for tasks like entity extraction and relation identification.
- **Document Chunking and Storage:** The package uses the SimpleKGPipeline class to automate chunking and storage. This class handles the parsing of documents, the chunking of text, and storage of chunks as nodes in Neo4j.

```
from neo4j import GraphDatabase

from neo4j_graphrag.experimental.pipeline.kg_
builder import SimpleKGPipeline

from neo4j_graphrag.llm import OpenAILLM

from neo4j_graphrag.embeddings import
OpenAIEmbeddings

from neo4j_graphrag.generation.prompts import
ERExtractionTemplate

from dotenv import load_dotenv

import os, time, asyncio, glob, csv
```

- `neo4j`: **Official Python** driver for interacting with a Neo4j database.
- `GraphDatabase`: Connects to **Neo4j** to interact with the graph database.
- `SimpleKGPipeline`: Automates **chunking, entity recognition, and storage** in Neo4j.
- `OpenAILLM`: Integrates **GPT-4** for text-based processing and knowledge extraction.
- `OpenAIEmbeddings`: Handles **vector embeddings** to enable **semantic search** in Neo4j.
- `ERExtractionTemplate`: Supplies **prompt templates** for entity-relation extraction.

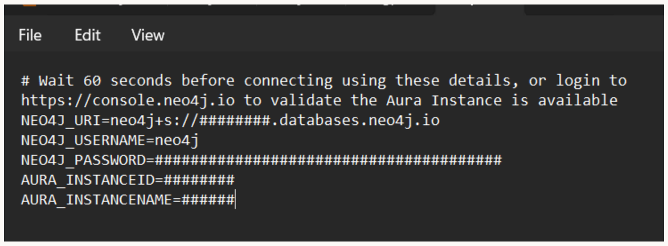
The LLM does the thinking by extracting meaningful concepts from text. The embedder turns the text into vectors, which lets your system perform semantic search later.

Neo4j Connection

You'll use `GraphDatabase` from the Neo4j Python driver to connect to Neo4j Graph Database.

```
driver = GraphDatabase.driver(NEO4J_URI,
auth=(NEO4J_USER, NEO4J_PASSWORD))
```

Note that the required credentials can be found in the .txt file you downloaded when you created the instance.



```
# Wait 60 seconds before connecting using these details, or login to
https://console.neo4j.io to validate the Aura Instance is available
NEO4J_URI=neo4j+s://########.databases.neo4j.io
NEO4J_USERNAME=neo4j
NEO4J_PASSWORD=###########################################
AURA_INSTANCEID=########
AURA_INSTANCENAME=######
```

Figure 9. Credentials from .txt file

- `NEO4J_URI:` The database URL (e.g., `“neo4j+s://ef123456.database.neo4j.io”`)
- `auth=(NEO4J_USER, NEO4J_PASSWORD)`: Credentials to authenticate

Initialize the LLM and Embeddings

```
llm = OpenAILLM(model_name=”gpt-4o”, api_key=openai_api_key)
dimensions = 1536
embedder = OpenAIEmbeddings(api_key=openai_api_key)
```

- llm: Uses **GPT-4o** to extract entities, relationships, and summarize text.
- embedder: Generates **vector embeddings** to **enable semantic** search and **contextual retrieval.**

Define Node Labels and Relationship Types

```
entities = [
    {“label”: “Executive”, “properties”: [{“name”: “name”, “type”: “STRING”}]},
    {“label”: “Product”, “properties”: [{“name”: “name”, “type”: “STRING”}]},
    {“label”: “FinancialMetric”, “properties”: [{“name”: “name”, “type”: “STRING”}]},
    {“label”: “RiskFactor”, “properties”: [{“name”: “name”, “type”: “STRING”}]},
    {“label”: “StockType”, “properties”: [{“name”: “name”, “type”: “STRING”}]},
    {“label”: “Transaction”, “properties”: [{“name”: “name”, “type”: “STRING”}]},
    {“label”: “TimePeriod”, “properties”: [{“name”: “name”, “type”: “STRING”}]},
    {“label”: “Company”, “properties”: [{“name”: “name”, “type”: “STRING”}]}
]
relations = [
    {“label”: “HAS_METRIC”, “source”: “Company”, “target”: “FinancialMetric”},
    {“label”: “FACES_RISK”, “source”: “Company”, “target”: “RiskFactor”},
    {“label”: “ISSUED_STOCK”, “source”: “Company”, “target”: “StockType”},
    {“label”: “MENTIONS”, “source”: “Company”, “target”: “Product”}
]
```

Defining your nodes and relationships in two lists is a key moment in the knowledge graph construction process. This is when you determine the data model. These lists control what the SimpleKGBuilder will look for in the text and how it will organize that information in your graph. To understand how you might want to construct these lists, let’s take a look at some general ideas.

Entities = Nouns

What are the real-world concepts you’re trying to capture?

Company, Executive, RiskFactor, Product — whatever matters to your domain.

Relationships = Verbs or Connectors

How do those concepts relate?

Perhaps a Company → FACES_RISK → RiskFactor, or Company → ISSUED_STOCK → StockType.

If you aren’t sure which entities and relationships to include in your first project, ask yourself: What information would help my chunk provide a better answer? Alternatively, what information connects various chunks? Ultimately, you want to think through the application’s use case and start with the entities and relationships that will move the needle the most on your project. This step isn’t just configuration; it’s your chance to define the mental model of your data.

Initialize and Run the Pipeline

```
pipeline = SimpleKGPipeline(
        driver=driver,
        llm=llm,
        embedder=embedder,
        entities=entities,
        relations=relations,
        enforce_schema=”STRICT”)
```

The `SimpleKGPipeline` sets up a structured pipeline for extracting and storing knowledge from unstructured text into a graph database. It starts with the `driver`, which is the Neo4j connection used to write data into the graph. The `llm` parameter specifies the language model that will interpret and extract meaningful entities and relationships from the input text. The `embedder` is the embedding model used to vectorize text, which supports similarity-based retrieval alongside structured querying.

The `entities` and `relations` define the schema: what kinds of objects (like Customers, Contracts, Products) and relationships (like `HAS_CONTRACT`, `CONTAINS`, `REFERENCES`) the pipeline should look for. Finally, `enforce_schema=True` ensures that only the entity and relationship types that have been explicitly defined in those lists are allowed into the graph. This prevents schema drift and keeps the resulting knowledge graph clean and reliable.

Process the PDF Document

Running the pipeline involves I/O-heavy operations:

- Calling the LLM to extract structured meaning from text
- Generating embeddings via an external API
- Writing data into Neo4j

All of these are network-bound and would block the main thread in a normal synchronous setup. That's why the pipeline is designed to be asynchronous – so these operations can run concurrently and efficiently. To execute it, you need to use Python's `async` / `await syntax`: The `await` keyword tells Python:

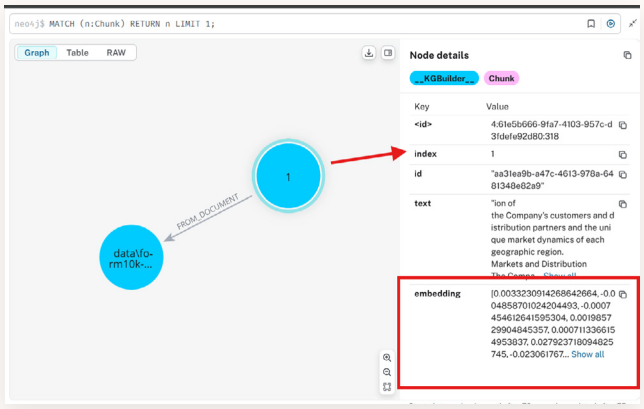
"Pause this function while we wait on an external operation, but don't freeze the whole program."

```
async def run_pipeline_on_file(file_path, pipeline):
    await pipeline.run_async(pdf_path=file_path)
```

If you're calling this inside another async function, it will work by itself. If you're in a regular script or notebook, you'll need to run it inside an event loop. If you're unfamiliar with it, don't worry — you can treat `await pipeline.run_async()` like a normal function call, as long as it's inside an `async` context.

```
for pdf_file in pdf_files:
    asyncio.run(run_pipeline_on_file(pdf_file, pipeline))
```

As you can see in the image below, the document and chunk nodes have been created and written to the database. Note that there is now a property on the node called `embedding`, which represents the vector of the chunk text. This is how your retriever finds the relevant chunk in your application: by comparing the embedding of the query and the embeddings in your data store.


```
neo4j$ MATCH (n:Chunk) RETURN n LIMIT 1;
```

Node details

__KGBuilder__ Chunk

Key	Value
<id>	4:61e5b666-9fa7-4103-957c-d3fdefe92d80:318
index	1
id	"aa31ea9b-a47c-4613-978a-6481348e82a9"
text	"ion of the Company's customers and distribution partners and the unique market dynamics of each geographic region. Markets and Distribution The Compa... Show all
embedding	[0.0033230914268642664, -0.004858701024204493, -0.0007454612641595304, 0.001985729904845357, 0.0007113366154953837, 0.027923718094825745, -0.023061767... Show all

Figure 10. Node details

Create the Vector Index

A vector index is a type of database index that enables fast similarity search over high-dimensional vectors, such as embeddings from models like OpenAI's. Unlike traditional indexes that look for exact matches, vector indexes retrieve items most similar to a query vector using metrics like cosine similarity or Euclidean distance.

In the context of Neo4j and RAG, here's what you need to know:

1. Each node (e.g., a Chunk) stores an embedding, a numeric representation of its semantic content.
2. The vector index organizes these embeddings so that, given a new query embedding, Neo4j Graph Database can quickly retrieve the most similar nodes.
3. This capability is essential for semantic search, question answering, and other AI-powered applications where meaning and context matter more than exact keywords.

By using a vector index, Neo4j enables scalable, real-time retrieval of relevant knowledge from large and complex graphs.

```
from neo4j_graphrag.indexes import create_vector_index

create_vector_index(driver, name="chunkEmbeddings",
label="Chunk",
                    embedding_property="embedding",
dimensions=1536, similarity_fn="cosine")
```

Ingest Structured Data

Getting Started With Data Importer

Neo4j Data Importer provides a streamlined process for bringing structured data into your graph database. Here's how to use this powerful tool. The Neo4j Aura console includes a dedicated Data Importer feature that allows you to transform tabular data into graph structures without writing code. This tool works well in quickly populating your knowledge graph with data from existing datasets.

Import Structured Data

To import your data:

1. Navigate to **Import** > **Data Importer** in the Neo4j Aura console.

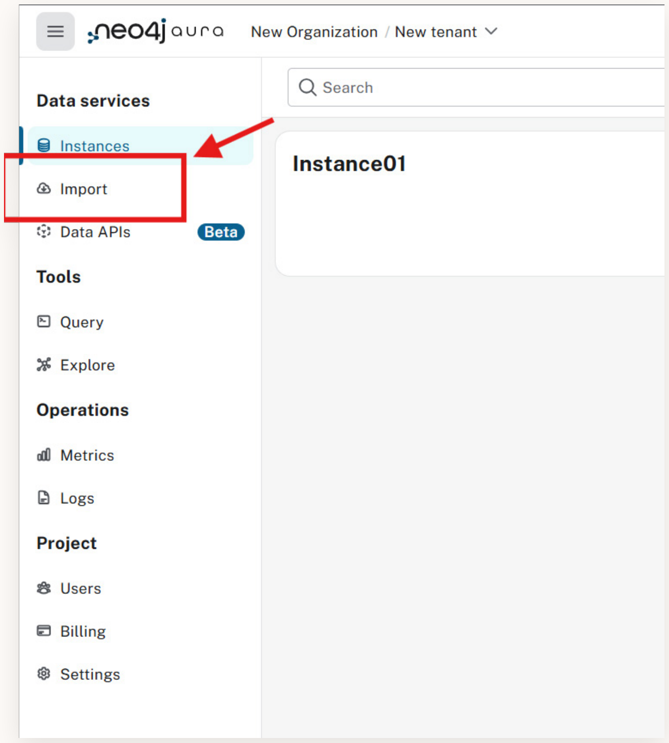
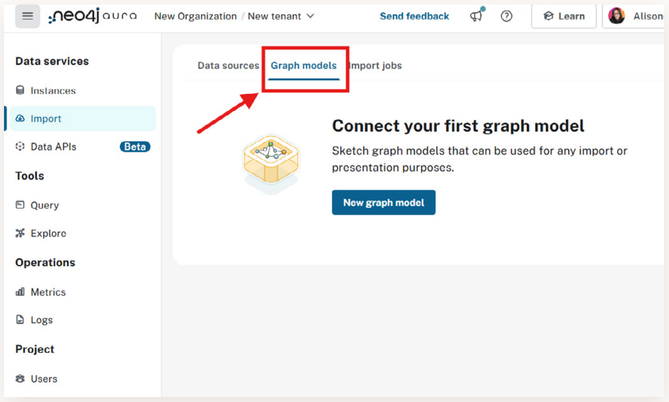


Figure 11. Neo4j Aura Data Importer

2. Create a new graph model.

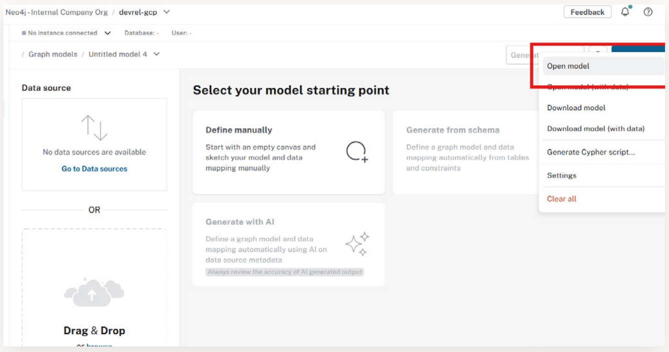


Connect your first graph model

Sketch graph models that can be used for any import or presentation purposes.

Figure 12. New graph model screen

3. A graph data model has been provided for your convenience. **Note**: Due to pathway differences between operating systems, please choose either Mac or Windows data models.


Neo4j - Internal Company Org / devrel-gcp
Feedback
No instance connected
Database: -
User: -
Graph models / Untitled model 4
Data source
No data sources are available
Go to Data sources
OR
Drag & Drop
Open model
Open model (with data)
Download model
Download model (with data)
Generate Cypher script...
Settings
Clear all

Select your model starting point

Define manually — Start with an empty canvas and sketch your model and data mapping manually

Generate from schema — Define a graph model and data mapping automatically from tables and constraints

Generate with AI — Define a graph model and data mapping automatically using AI on data source metadata

Always review the accuracy of AI generated output

Figure 13. Selecting model starting point screen

4. Once you've loaded the provided data model, click **Browse** and navigate to the data folder in your repository, selecting both the Asset_Manager_Holdings.csv file and the Company_Filings.csv files.

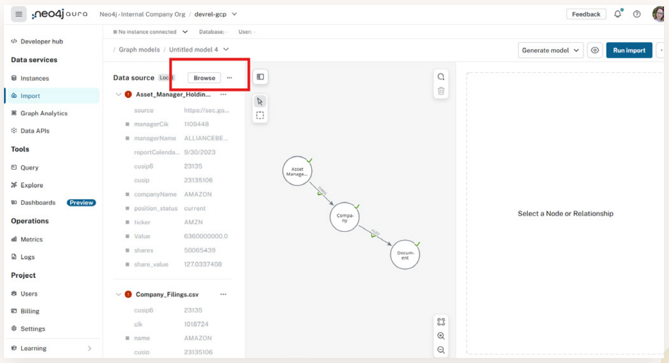


Figure 14. Browse to .csv files screen

5. Once the files are connected, you'll see that the data model has check marks for each entity and relationship. Click **Run Import** in the upper right-hand corner.

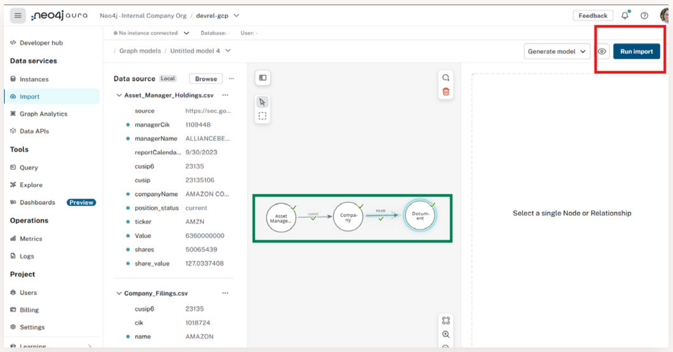

Generate model
Run import
Data source
Asset_Manager_Holdings.csv
Company_Filings.csv
Select a single Node or Relationship

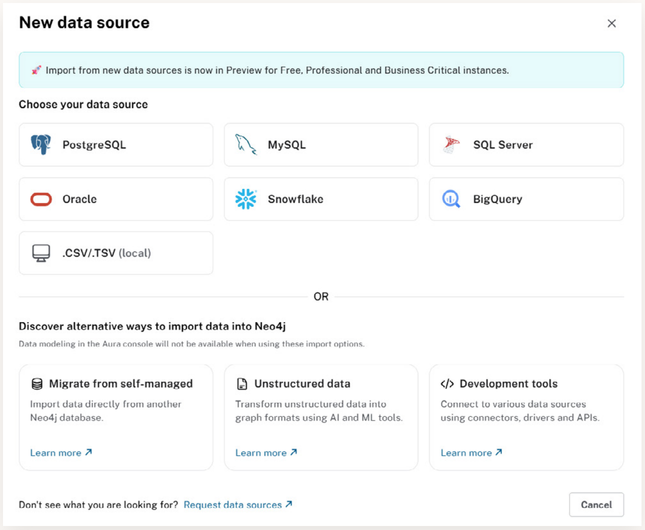
Figure 15. Run import screen

Mapping Your Data to Graph Structures

To get you started, we've given you a full, completed data model for this exercise. When working with your own data, you'll create these data model maps yourself.

If you'd like to work with your own dataset, here's how to get started.The Aura console provides a unified experience where you can manage your database instances, connect to diverse data sources, [import](#) structured data, model graphs visually, [query](#) your data with Cypher, [explore](#) your graph, and more.

When navigating to **Import > New Data Sources,** you're presented with many possible connectors. For our case, there are two CSVs in this dataset: [Asset_Manager_Holdings.csv](#) and [Company_Filings.csv](#).



New data source

Import from new data sources is now in Preview for Free, Professional and Business Critical instances.

Choose your data source

PostgreSQL | MySQL | SQL Server
Oracle | Snowflake | BigQuery
.CSV/.TSV (local)

OR

Discover alternative ways to import data into Neo4j

Data modeling in the Aura console will not be available when using these import options.

Migrate from self-managed — Import data directly from another Neo4j database. Learn more ↗

Unstructured data — Transform unstructured data into graph formats using AI and ML tools. Learn more ↗

Development tools — Connect to various data sources using connectors, drivers and APIs. Learn more ↗

Don't see what you are looking for? Request data sources ↗ Cancel

Figure 16. New data source connectors screen

Once you've uploaded these CSV files, you'll be given a choice as to how to proceed. Click **Define Manually** to begin building your data model.

First, you'll see a blank node, and on the right-hand side, you'll see the parameters for that node, including **Label, Table, Properties.**

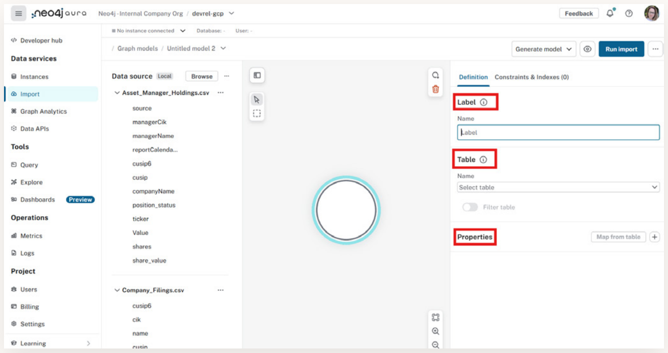


Figure 17. Node parameters options screen

Label refers to the type of node. **Table** points to the data source where the information is sourced (the tables you uploaded will appear on the left). **Properties** refer to the values you want associated with that node. Let's start with the [Company_Filings.csv.](#)

Company Node

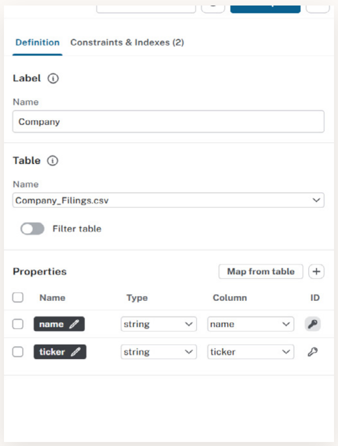
Label: `Company`

Table: `Company_Filings.csv`

Properties: `name, ticker`

ID(key): `name`

You'll also need to identify the unique ID property for that node, akin to the primary key, which in this case is the name of the company. This is done by clicking the key icon next to the property name.



Definition | Constraints & Indexes (2)

Label
Name: Company

Table
Name: Company_Filings.csv
Filter table

Properties — Map from table

Name	Type	Column	ID
name	string	name	🔑
ticker	string	ticker	

Figure 18. Company node screen

Document Node

Label: Document

Table: Company_Filings.csv

Properties: path *(this must match exactly - read below)*

ID(key): path

CRITICAL STEP: Rename Your Path Column to path

The kg_builder has already created Document nodes using a path property. To correctly link companies to their documents, your imported data must use the exact same property name: path.

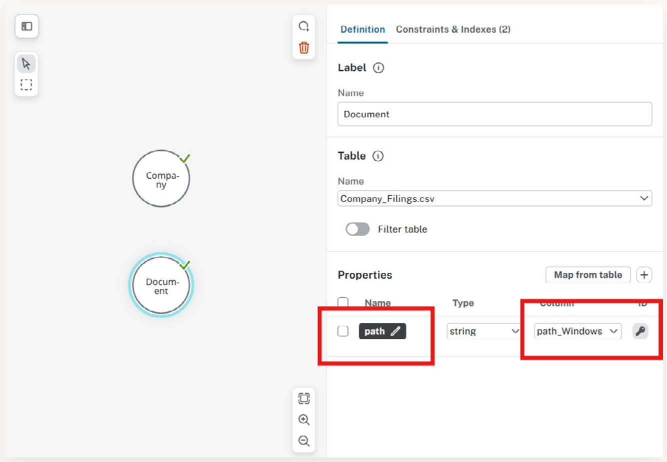


Figure 19. Path property screen

⚠️ If you skip this renaming step, the relationship will NOT connect and your graph will be incomplete.

The CSV includes two columns with OS-specific paths:

- path_Windows for **Windows users**
- path_Mac_ix for **macOS/Linux users**
- Choose the appropriate column based on your operating system and **rename it** to path during import

Pick the column for your system:

1. Rename that column to exactly: path (lowercase, no quotes).
2. Even though Document nodes already exist, we're now creating relationships between each Company and its corresponding Document. This connection bridges **structured** (Company) and **unstructured** (Document) data, enabling advanced retrieval and reasoning across your graph.

Asset Manager Node

Label: AssetManager

Table: Asset_Manager_Holdings.csv

Properties: managerName

ID(key): managerName

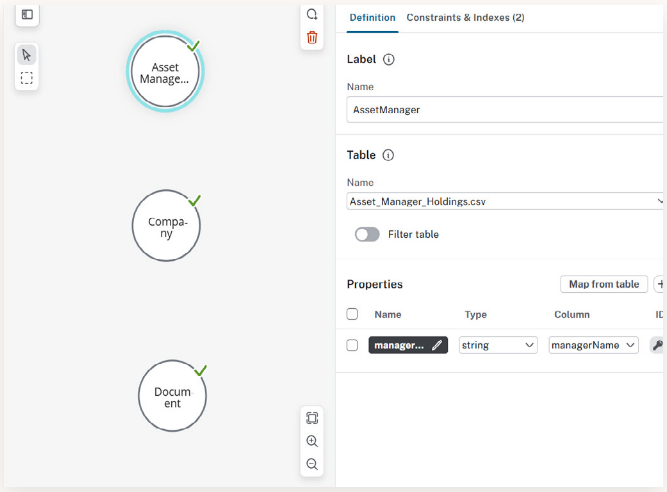


Figure 20. AssetManager node screen

Mapping Relationships

Relationships are created with the following criteria:

- **Relationship Label:** Describes the type of connection between the entities. It is common practice in knowledge graphs for the relationships to be in **ALL_CAPS** with no spaces.
- **Table:** Has identifiers for each node type contained in it. It is the way we connect the two nodes.
- **Node ID Mapping:** Maps the columns in the relevant table to the IDs of the pertinent nodes.
- **Properties:** Adds information to a relationship or entity.

Next, let's create connections between and among these entities. In our domain, the Asset Managers own stock in various companies. Here's a sample from the Asset_Manager_Holdings.csv:

managerName	companyName	ticker	shares
ALLIANCEBERNSTEIN L.P.	AMAZON	AMZN	50065439
ALLIANCEBERNSTEIN L.P.	APPLE INC	AAPL	28143032
ALLIANCEBERNSTEIN L.P.	INTEL CORP	INTC	5735993
ALLIANCEBERNSTEIN L.P.	MCDONALDS CORP	MCD	1201960
ALLIANCEBERNSTEIN L.P.	MICROSOFT CORP	MSFT	46541943

In a knowledge graph, we want to map the domain knowledge of structured data, which in this case is the Asset Managers' ownership of stock in a given company. If entities are nouns, then relationships are verbs. So let's create the relationship `OWNS` that goes from Asset Manager to Company.

1. Click on the AssetManager node. You'll see a blue outline of the node:

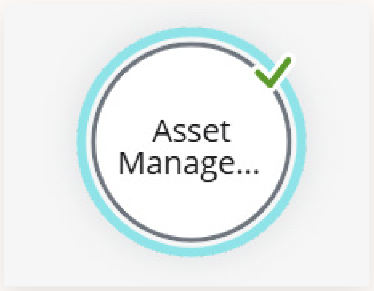


Figure 21. AssetManager blue outline

2. Hover over the outline until it turns gray:

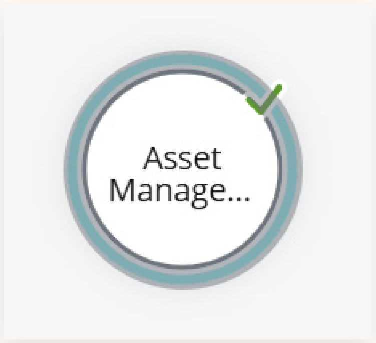


Figure 22. AssetManager gray outline

3. Drag the outline of the AssetManager node to cover the Company node. When you release, you'll see a new relationship arrow between them:

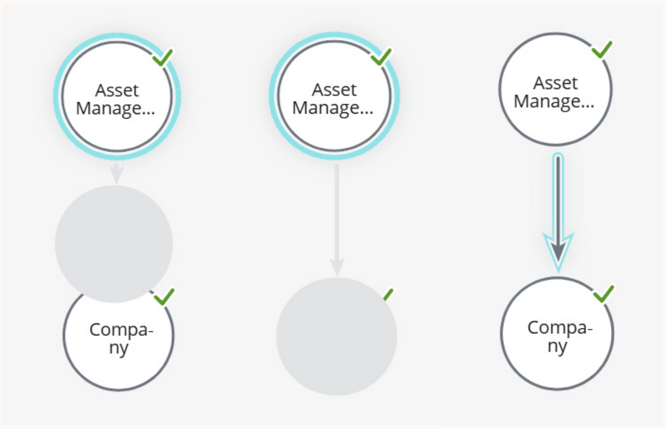
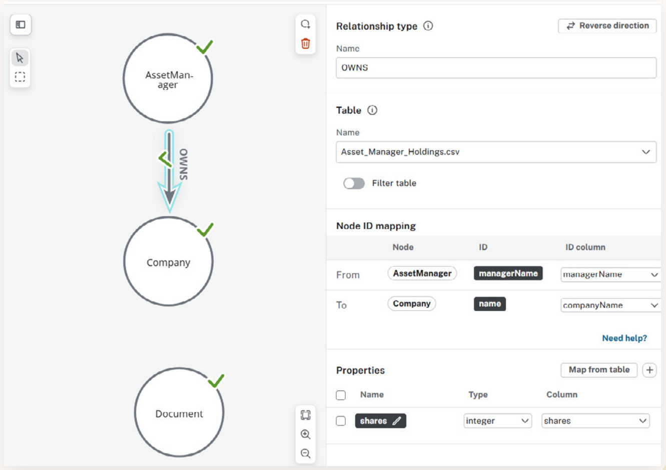


Figure 23. Drag and release for new relationship

Clicking on this arrow allows you to edit the parameters of the relationship.

OWNS Relationship

Relationship Type: `OWNS`
Table: `Asset_Manager_Holdings.csv`
Node ID Mapping
From:
    Node: `AssetManager`
    ID: `managerName`
    ID column: `managerName`
To:
    Node: `Company`
    ID: `name`
    ID column: `companyName`
Properties: `shares`



Relationship type
Reverse direction
Name: OWNS

Table
Name: Asset_Manager_Holdings.csv
Filter table

Node ID mapping

	Node	ID	ID column
From	AssetManager	managerName	managerName
To	Company	name	companyName

Need help?

Properties
Map from table

Name	Type	Column
shares	integer	shares

Figure 24. OWNS relationship

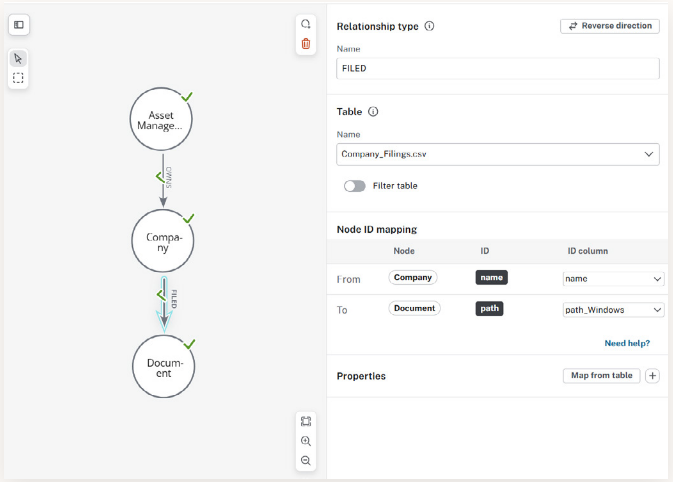
The property shares represents the number of shares of the Company owned by the Asset Manager and for this book is an optional inclusion. Additional columns such as value or sharevalue are optional, as well. When working with your own data, it's best to consider if that property will have value to your use case. Will you be asking to rank based on shares owned? Does the total value of the holding have relevance to your application? Additional information on data modeling can be found at GraphAcademy.

FILED Relationship

Note that the relationship between **Company** and **Document** is the linchpin that connects the structured and the unstructured data in this GraphRAG application.

Relationship Type: FILED
Table: Company_Filings.csv

Node ID Mapping
From:
    Node: Company
    ID: name
    ID column: companyName
To:
    Node: Document
    ID: path
    ID column: path_Windows or path_Mac_ix



Relationship type
Name: FILED

Table
Name: Company_Filings.csv
Filter table

Node ID mapping

	Node	ID	ID column
From	Company	name	name
To	Document	path	path_Windows

Properties

Figure 25. FILED relationship

As you see in the diagram above, each entity and relationship will have a green check mark when it has been properly mapped. Now you're ready to run the import. Click the blue **Run import** button in the upper right corner of the screen.

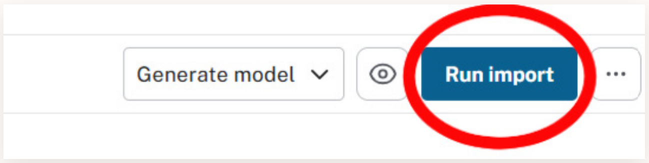


Figure 26. Run import button

Now that your unstructured and structured data is loaded, you can use the Explore and Query functions to refine your graph structure and data to accurately represent your business domain. Use **Explore** to visualize and navigate your graph with Neo4j Bloom and **Query** to investigate the graph.

For a detailed walkthrough of graph data modeling, see The Developer's Guide: How to Build a Knowledge Graph.

PART IV: Implementing GraphRAG Retrieval Patterns

GraphRAG retrieval patterns are practical mechanisms that define how the LLM in your GraphRAG solution accesses the context and connections in your knowledge graph.

Let's examine some of the most common GraphRAG patterns and how to use them.

Import Libraries

```
from neo4j import GraphDatabase

from neo4j_graphrag.llm import OpenAILLM

from neo4j_graphrag.embeddings import
OpenAIEmbeddings

from neo4j_graphrag.retrievers import
VectorRetriever, VectorCypherRetriever,
Text2CypherRetriever

from neo4j_graphrag.generation import GraphRAG

from neo4j_graphrag.schema import get_schema

from detenv import load_dotenv
```

This notebook imports the core libraries required for building and querying RAG pipelines with Neo4j and GraphRAG:

- `neo4j.GraphDatabase:` The official Python driver for connecting to and querying a Neo4j database.
- `neo4j_graphrag.llm.OpenAILLM`: Integrates OpenAI language models for generating and processing natural language queries.
- `neo4j_graphrag.embeddings.OpenAIEmbeddings`: Provides access to OpenAI's embedding models for generating vector representations of text.
- `Neo4j_graphrag.retrievers`: Different retriever classes for semantic and hybrid search over graph data using vector similarity and Cypher queries:
    - `VectorRetriever`
    - `VectorCypherRetriever`
    - `Text2CypherRetriever`
- `neo4j_graphrag.generation.GraphRAG`: The main class for orchestrating RAG workflows over a Neo4j knowledge graph.
- `neo4j_graphrag.schema.get_schema`: Utility to introspect and retrieve the schema of your Neo4j database.
- `dotenv.load_dotenv`: Loads environment variables (such as credentials and API keys) from an .env file for secure configuration.

These imports enable advanced semantic search, retrieval, and GenAI capabilities directly on your Neo4j knowledge graph.

Load Environment Variables and Initialize Neo4j Driver

```
load_dotenv()
NEO4J_URI = os.getenv('NEO4J_URI')
NEO4J_USER = os.getenv('NEO4J_USERNAME')
NEO4J_PASSWORD = os.getenv('NEO4J_PASSWORD')
OPERNAI_API_KEY = os.getenv('OPENAI_API_KEY')

driver = GraphDatabase.driver(NEO4J_URI, auth=(NEO4J_
USER, NEO4J_PASSWORD))
```

Here, you load sensitive configuration values (such as database credentials and API keys) from environment variables, ensuring that secrets aren't hardcoded in your notebook. The steps include:

- `load_dotenv()`: Loads environment variables from an .env file into your Python environment.
- `os.getenv()`: Fetches the Neo4j connection URI, username, and password, as well as your OpenAI API key.
- `GraphDatabase.driver()`: Initializes the Neo4j database driver with the provided credentials, allowing your notebook to connect and interact with your Neo4j instance securely.

TIP: *Make sure your .env file contains the correct values for* `NEO4J_URI, NEO4J_USERNAME, NEO4J_PASSWORD,` *and* `OPENAI_API_KEY` *before running this code. This approach keeps your credentials secure and makes your codebase easier to share and maintain.*

Initialize the LLM and Embedder

Just as you selected a specific LLM and embedding model when processing your PDFs, you should do the same when generating embeddings for your text data. It's important to keep track of the language model and embedding tools that you use during this process.

For the retrievers to work correctly, the embedding model used during retrieval must match the one used to generate the dataset's embeddings. This ensures accurate and meaningful search results.

```
llm = OPENAILLM (model_name='gpt-4o', api_key=OPENAI_API_KEY)

embedder = OPENAIEmbeddings(api_key=OPENAI_API_KEY)
```

The Basic Retriever Pattern

The basic retriever uses vector embeddings to find nodes that are semantically similar based on content. This retriever is useful only for handling specific information requests about topics contained in just one or a few chunks. It's a starting point for more complex graph-based retrievals, and it's easy to implement if you're familiar with RAG but new to GraphRAG.

There are two components in the process:

- **Chunks as nodes:** The pattern uses the already chunked data to create a graph, where each chunk becomes a node in the graph.
- **Retrieval**: When a query is performed, the basic retriever pattern searches through these chunk nodes to find the most relevant information.

Let's look at how you would implement this pattern using the SEC dataset.

You can now execute vector similarity searches to retrieve a company's current challenges based on certain text in their filing. The retriever compares a query vector generated from the search prompt (i.e., the numeric representation of the question) against the indexed text embeddings of the chunks. Vector similarity searches work well for simple queries with a narrow focus, such as: "What are the risks around cryptocurrency?"

```
from neo4j-graphrag.retrievers import VectorRetriever

# Initialize the retriever
retriever = VectorRetriever(
     driver,
     index_name= "text_embeddings",
     embedder=embedder,
     return_properties=["text"]
)


query = "What are the main risks around cryptocurrency?"
result = vector_retriever.search(query_text=query, top_
k=10)
```

Be sure to review your retrieval results before generating any text output. This step helps you confirm that your retriever is functioning as intended and returning relevant data from your knowledge graph. For example, in the query above, a sample of the retrieved content is displayed for inspection:

```
result_table=pd.DataFrame([(item.metadata['score'], item.
content [10:80],
item.metadata['id']) for item in result.items],
columns=['Score', 'Content', 'ID']
```

Score	Content	ID
0.913177	cryptocurrency assets could be treated as a general unsecured claim ag..	6064a2f775a8:1724
0.908264	agency offerings could subject us to additional regulations, licensing r...	6064a2f775a8:1723
0.903259	cyberextortion, distributed denial-of-service attacks, ransomware, spe...	6064a2f775a8:1718
0.898422	While we maintain insurance policies intended to help offset the\ nfina...	6064a2f775a8:1720
0.896942	financing, and branded credit card products; branded debit card and\ ns...	6064a2f775a8:1731
0.89476	our customers unimpaired and unconstrained access to our online servic...	6064a2f775a8:1731
0.894135	changes in cryptocurrencies, government cryptocurrency policies and ,,,	6064a2f775a8:1251
0.893723	ct our reputation and revenue. Actual or perceived vulnerabilities may...	6064a2f775a8:2508
0.893539	the past and could in the future have a material adverse effect on our...	6064a2f775a8:1254
0.893402	may continue to result\nin, disruption of and volatility in global fin...	6064a2f775a8:1260

You should review the results, or at least check the number of returned items. If there's an error in your retriever and you proceed directly to natural language generation, your application may produce a generic LLM response that isn't grounded in your data. This validation step ensures your outputs remain accurate and reflect the content of your underlying dataset.

To get the natural language output, use the following code:

```
rag = GraphRAG(llm=llm, retriever=vector_retriever)

print(rag.search(query).answer)
```

The basic retriever will cause the LLM to generate a result like this:

> The main risks around cryptocurrency, as highlighted in the context, include:
>
> 1. **Regulatory Risks:** The regulatory status of certain cryptocurrencies is unclear, which could subject businesses to additional licensing and regulatory obligations. If cryptocurrencies are deemed securities, this might necessitate securities broker-dealer registration under federal securities laws. Non-compliance could lead to regulatory actions, fines, and other consequences.
> 2. **Custodial Risks:** Cryptocurrency assets held through a third-party custodian are susceptible to various risks, such as inappropriate access, theft, or destruction. Inadequate insurance coverage by custodians and their potential inability to maintain effective controls can expose customers to losses. In the event of a custodian's bankruptcy, the treatment of custodial holdings in proceedings remains uncertain, which could delay or prevent the return of assets.
> 3. **Third-Party Partner Risks:** Dependence on third-party custodians and financial institutions means exposure to operational disruptions, inability to safeguard holdings, and financial defaults by these partners, which could harm business operations and customer trust.
>
> These risks underscore the need for robust regulatory compliance, secure custodial arrangements, and the management of third-party relationships to mitigate potential negative impacts on businesses offering cryptocurrency products.

While the vector search provided useful information about cryptocurrency risks, it did not answer deeper, more actionable questions, such as:

- Which specific companies are exposed to these risks?
- What other risks may be occurring concurrently?
- Which asset managers are associated with the affected companies? (e.g., multi-hop relationships from risk to company to asset manager)

In other words, the approach demonstrated here retrieves relevant text fragments. However, it doesn't use the graph's structure to connect the risks to companies or asset managers, nor does it show related or concurrent risks. There's no traversal or multi-hop reasoning, so you miss out on the rich, contextual insights that a knowledge graph can provide.

To answer these more complex, relationship-driven questions, you need to combine vector search with graph-powered Cypher queries that can traverse and analyze connections between entities. This is where graph-enhanced retrieval patterns come in.

The Graph-Enhanced Vector Search Pattern

The basic retriever pattern typically relies on text-based embeddings, capturing only the semantic meaning of content. While this method is effective in identifying similar chunks, it leaves the LLM in the dark as to how those items interact in the real world.

The **Graph-Enhanced Vector Search Pattern**, also known as augmented vector search, overcomes this limitation by drawing on the graph structure (i.e., using not just what items are but also how they connect). By embedding node positions and relationships within a graph, this approach generates contextually relevant nodes, integrating both:

- **Unstructured data**: Product descriptions, customer reviews, and other text content via semantic similarity
- **Structured data**: Purchase patterns, category relationships, and transaction records via explicit instructions

The `VectorCypherRetriever` uses the full graph capabilities of Neo4j by combining vector-based similarity searches with graph traversal techniques. The retriever completes the following actions:

1. Processes a query embedding to perform a similarity search against a specified vector index.
2. Retrieves relevant node variables.
3. Executes a Cypher query to traverse the graph based on these nodes.

To set up this particular query, you need to tell the graph where and how to traverse from the semantic nodes. In this example, the query is:

"What are the risk factors for companies discussing cryptocurrency in their filings?"

The following code creates a retriever to answer this query:

```
company_risk_list_query = """
WITH node
MATCH (node)-[:FROM_DOCUMENT]-(d:Document)-[:FILED]-
(c:Company)-[:FACES_RISK]-(rf:RiskFactor)
RETURN c.name AS company,  node.text AS context,
collect(DISTINCT r.name) AS risks
"""
```

Let's start by looking at the parts of the graph that help to answer this query. We start by identifying the `Chunk` that is semantically similar to the cryptocurrency query. Then we need to traverse the graph to identify the `Document` the `Chunk` comes from, the `Company` that `FILED` the `Document` and collect the other `RiskFactors` for that `Company`. Once this information is retrieved, it's converted to Cypher and set as the retrieval query.

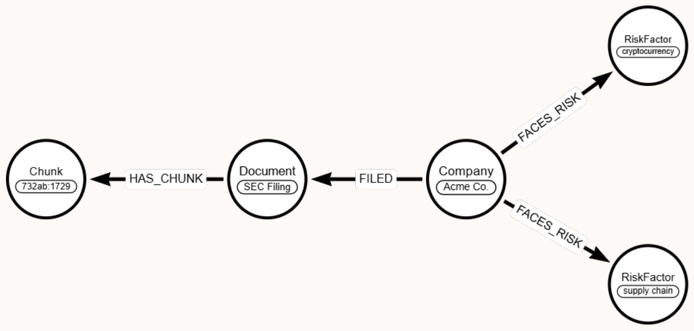


Figure 27. VectorCypherRetriever example 1

Next, let's add this new retrieval query to the `VectorCypherRetriever` parameters:

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

`VectorCypherRetriever` parameters:

- `Driver`: The Neo4j database connection
- `Index_name`: The name of the vector index (here, `chunkEmbeddings`) used for semantic search
- `Embedder`: The embedding model used to generate/query vector representations
- `Retrieval_query`: The Cypher query (defined above) that tells Neo4j how to traverse the graph from the semantically matched nodes

This setup enables you to start with a semantic search (e.g., for "cryptocurrency risk") and automatically traverse your knowledge graph to reveal which companies are involved and what other risks they face. The resulting responses are both semantically relevant and graph-aware.

VectorCypher Retriever in Practice

The power of the **Graph-Enhanced Vector Search Pattern** lies in its flexibility. While the example above focuses on linking companies to risk factors in financial filings, the approach can be applied to any domain or vertical by customizing the graph schema and Cypher queries.

How might this look for other industries?

- **Healthcare**: Retrieve patient records, diagnoses, and treatment plans by combining semantic search of clinical notes with graph traversal across relationships like doctor-patient, medication-prescribed, or symptom-diagnosis.
- **Ecommerce**: Connect customer reviews or product descriptions (unstructured text) to

purchase behavior, category hierarchies, or supplier relationships (a structured graph), enabling recommendations and/or supply chain insights.
- **Law**: Link case law or legal opinions to statutes, precedents, and involved parties, surfacing not just relevant text but also the legal context and network of citations.
- **Cybersecurity**: Combine threat intelligence reports (text) with the graph relationships between vulnerabilities, affected assets, and mitigation strategies to provide a holistic view of your security posture.
- **Education**: Map student essays or discussion posts to learning objectives, course materials, and assessment outcomes for personalized education analytics.

Let's summarize the major tasks from this example so you can apply it to your domain:

- **Adapt the Pattern Model Your Domain**: Define the node types, relationships, and key properties relevant to your vertical (e.g., Patient, Diagnosis, Product, Supplier, Case, Asset, etc.).
- **Index the Right Data**: Create vector indexes on the appropriate text or document nodes for semantic retrieval.
- **Craft Domain-Specific Cypher Queries**: Write Cypher queries that traverse from the retrieved nodes to related entities and/or relationships that matter in your context.
- **Integrate With VectorCypherRetriever**: Use the VectorCypherRetriever with your custom query to combine semantic and structural search.

The result: You can ask complex, context-aware questions about entities in your own industry. The GraphRAG retriever will surface relevant information that connects context across structured and unstructured data to drive real-world understanding.

With this in mind, let's look at another `VectorCypherRetriever` example.

VectorCypher Retrieval: A Working Example

Which Asset Managers are most affected by reseller concerns?

Let's again start with the `Chunks` semantically similar to "reseller concerns," and then traverse through the `Document` to the `Company` through `OWNS` to identify the `AssetManagers` relevant to the query. We'll also include the property `shares` from the relationship `OWNS` and order by largest holdings.

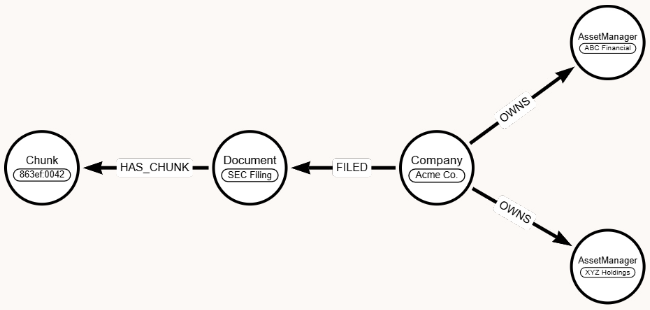


Figure 28. VectorCypherRetriever example 2

```
chunk_to_asset_manager_query = """
WITH node
MATCH
(node)-[:FROM_DOCUMENT]-(doc:Document)-[:FILED]-
(company:Company)-[owns:OWNS]-(manager:AssetManager)
RETURN distinct company.name AS company, manager.managerName AS
AssetManager, owns.shares AS shares order by shares desc
"""
```

Next, add this new retrieval query to the `VectorCypherRetriever` parameters:

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

`VectorCypherRetriever` parameters:

- `Driver`: The Neo4j database connection
- `Index_name:` The name of the vector index (here, `chunkEmbeddings`) used for semantic search

- `Embedder:` The embedding model used to generate/query vector representations
- `Retrieval_query:` The Cypher query (defined above) that tells Neo4j how to traverse the graph from the semantically matched nodes

```
result = vector_cypher_retriever.search(query_text=query,
top_k=10)
for item in result.items:
    print(item.content[:100])
```

Let's look at the results:

```
<Record company='APPLE INC'
AssetManager='BlackRock Inc.'
shares=1031407553>

<Record company='APPLE INC'
AssetManager='Berkshire Hathaway Inc'
shares=915560382>

<Record company='AMAZON'
AssetManager='BlackRock Inc.'
shares=613380364>

<Record company='APPLE INC'
AssetManager='STATE STREET CORP'
shares=569291690>

<Record company='MICROSOFT CORP'
AssetManager='BlackRock Inc.'
shares=533634606>

<Record company='AMAZON'
AssetManager='STATE STREET CORP'
shares=332449318>

<Record company='AMAZON'
AssetManager='FMR LLC' shares=302101441>

<Record company='APPLE INC'
AssetManager='FMR LLC' shares=298321726>

<Record company='APPLE INC'
AssetManager='GEODE CAPITAL MANAGEMENT,
LLC' shares=296103070>
```

Since these results look as expected, we proceed to the natural language output:

```
result = GraphRag(llm=llm,retriever=vector_cyper_retriever)
print(rag.search(query_text=query_text).answer)
```

The Asset Managers most affected by cryptocurrency concerns are:

```
1. BlackRock Inc.

2. FMR LLC

3. STATE STREET CORP

4. GEODE CAPITAL MANAGEMENT, LLC

5. MORGAN STANLEY

6. NORTHERN TRUST CORP

7. BANK OF AMERICA CORP /DE/

8. Bank of New York Mellon Corp

9. ALLIANCEBERNSTEIN L.P.

10. AMUNDI

11. WELLINGTON MANAGEMENT GROUP LLP

12. Capital World Investors

13. AMERIPRISE FINANCIAL INC

14. WELLS FARGO & COMPANY/MN
```

This is where GraphRAG really shines. You may be wondering how to construct the `retrieval query` that traverses the graph. In this example, you can see that the retrieval_query is a string of Cypher code, the language of graph querying. Now let's look at one last retriever pattern found in the Neo4j library: the `Text2CypherRetriever`.

Text2CypherRetriever

You can use `Text2CypherRetriever` to seamlessly generate Cypher queries from natural language questions. Instead of manually crafting each Cypher statement, the retriever uses an LLM to translate your plain-English queries into Cypher based on its understanding of your Neo4j schema.

The process begins with a natural language question, such as:

> *"What are the names of companies owned by BlackRock Inc.?"*

The retriever then uses the schema, described as a string outlining the main node types and relationships in your graph (for example, companies, risk factors, and asset managers), to guide the LLM in generating an appropriate Cypher query. While you could pass a hard-coded schema to the retriever, it's best practice to access the schema as it currently exists in your instance. Here's a sample of the full schema:

```
result = get_schema (driver)
```

```
Node properties:

Document {id: STRING, path: STRING,
createdAt: STRING}

Chunk {id: STRING, index: INTEGER, text:
STRING, embedding: LIST}

Company {id: STRING, name: STRING, chunk_
index: INTEGER, ticker: STRING}

Product {id: STRING, name: STRING, chunk_
index: INTEGER}

. . .

Relationship properties:

OWNS {position_status: STRING, Value:
FLOAT, shares: INTEGER, share_value:
FLOAT}

The relationships:

....

(:Executive)-[:FROM_CHUNK]->(:Chunk)

(:StockType)-[:FROM_CHUNK]->(:Chunk)

(:AssetManager)-[:OWNS]->(:Company)
```

Now that you've defined the schema, you have everything you need to set the `Text2CypherRetriever.`

```
query="What are the names of the companies owned by BlackRock
Inc.?"

text2cypher_retriever = Text2CypherRetriever(

    driver=driver,

    llm=llm,

    neo4j_schema= schema

)


cypher_query = text2cypher_retriever.get_search_results(query)

cypher_query.metadata["cypher"]
```

```
MATCH (a:AssetManager {managerName:
'BlackRock Inc.'})-[:OWNS]->(c:Company)

RETURN c.name AS company_name
```

This approach has several advantages. It removes the need to write Cypher by hand for each query, making graph data accessible even to those without technical expertise. It's ideal for rapid prototyping, exploratory analysis, and building natural language interfaces to your knowledge graph, enabling a broader range of users to interact with complex graph data.

Now you can pass that Cypher query directly to the driver to get the results:

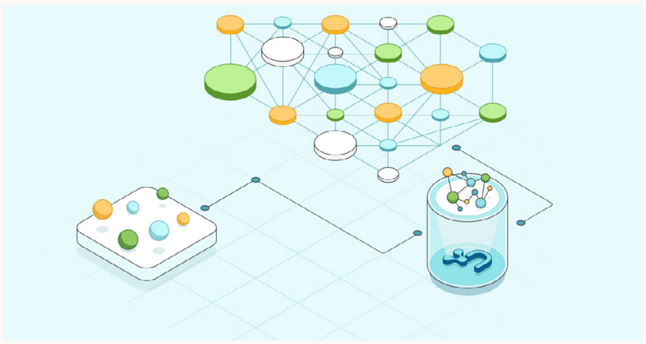
```
result = driver.execute_query(cypher_query.metadata["cypher"])

for record in result.records:

    print(record)
```

```
<Record companyName='APPLE INC'>

<Record companyName='MICROSOFT CORP'>

<Record companyName='INTEL CORP'>

<Record companyName='AMAZON'>

<Record companyName='PG&E CORP'>

<Record companyName='NVIDIA CORPORATION'>
```

While the Text2Cypher functionality in the Neo4j GraphRAG library offers a powerful way to translate natural language queries into Cypher, there are important considerations to keep in mind when using it.

First, because Text2Cypher relies on an LLM to generate queries dynamically, the same input may not always yield identical results. The model's responses can vary depending on context, training data, and even minor changes in phrasing. While the flexibility of Text2Cypher allows for more natural interactions, it can also introduce inconsistencies when precise, repeatable queries are required.

Additionally, query optimization remains an important factor. While LLMs are capable of generating complex Cypher queries, they may not always produce the most efficient ones. Without human intervention or performance tuning, these queries might not be optimized for speed or resource consumption, which could potentially slow application performance.



Finally, high-stakes applications — such as those requiring strict reproducibility, financial computations, or regulatory compliance — may require standardized, manually crafted Cypher queries instead. In such cases, relying entirely on an AI-generated query could introduce risks, especially if the generated query structure does not fully align with business logic or data constraints.

Despite these limitations, Text2Cypher is a valuable tool for making Neo4j more accessible, particularly for applications where flexibility, adaptability, and user-driven query formulation are more important than absolute precision. Understanding these trade-offs will help you integrate Text2Cypher effectively while ensuring that it is used in scenarios where its strengths outweigh its potential drawbacks.

Check out the Text2Cypher Crowdsourcing App to explore Text2Cypher applications and contribute to development projects.

Community Summary Pattern

You may have heard the term GraphRAG and thought of the pattern popularized by Microsoft, where the text is used to summarize community or other knowledge (i.e., forum posts). This type of retriever is often called the Community Summary Pattern.

While a Microsoft-style GraphRAG emphasizes summarization and community Q&A, Neo4j's approach focuses on domain-specific schema control and composable query generation. This focus expands GraphRAG from summarization into structured reasoning, decision tracing, and dynamic compliance use cases.

Concluding Thoughts and Next Steps

Integrating a knowledge graph with RAG gives GenAI systems structured context and relationships, improving the relevance and quality of generated results.

This guide has equipped you with the foundational skills needed to implement GraphRAG. You learned how to use Neo4j's cloud-based graph database service, Neo4j Aura, to prepare a knowledge graph for GraphRAG, Data Importer, and the GraphRAG Python library to create a knowledge graph from unstructured data. You also learned how to implement foundational GraphRAG retrieval patterns, including the **basic retriever, graph-enhanced vector search**, and **Text2Cypher.**

Like other AI technologies, GraphRAG is rapidly evolving. A few trends to watch:

- More advanced, dynamic Cypher queries and sophisticated retrieval patterns that use graph algorithms and machine learning techniques are pushing the boundaries of what's possible in information retrieval and generation.
- Deeper integration with other AI technologies, such as knowledge graph embeddings and graph neural networks, promises to enhance the semantic understanding and reasoning capabilities of GraphRAG systems.
- Integrating GraphRAG with agentic systems and other multi-tool, multi-step RAG chains can result in more autonomous and intelligent systems capable of handling complex, multifaceted tasks with greater efficiency and accuracy.
- Incorporating semantic layers in GraphRAG systems can provide even more nuanced understanding and context awareness in information retrieval and generation tasks.

Build on what you learned in this guide:

- The Neo4j for GenAI use case page offers guides, tutorials, and best practices about GraphRAG implementation.
- The GraphRAG site contains explanations of GraphRAG principles and step-by-step guides for various implementation scenarios.
- Neo4j GraphAcademy offers free, hands-on online courses.

Explore GenAI With Neo4j

Neo4j uncovers hidden relationships and patterns across billions of data connections deeply, easily, and quickly, making graph databases an ideal choice for building your first GraphRAG application.

Learn More

Appendix

Technical Resources in Workflow Order

Stage	Resource	Why It's Useful
1. Data Modeling	Designing a Graph Data Model for GenAI (Neo4j Blog)	Helps you define entity-relationship schemas (ontology) that power GraphRAG context.
2. Data Modeling	Neo4j Data Modeling Guide	Foundation for understanding how to structure both unstructured and structured data into a graph.
3. Environment Setup	Neo4j Aura Free Tier	Spin up a secure cloud instance instantly – perfect for prototyping.
4. Data Ingestion (Structured)	Neo4j Data Importer Tool	Visual UI for mapping CSVs and relational data to graph nodes and relationships.
5. Data Ingestion (Unstructured)	Neo4j GraphRAG Python Library	Convert PDFs and text to a knowledge graph using LLM-powered entity + relationship extraction.
6. Data Ingestion (Unstructured)	KGBuilder Tutorial – SEC Filings Example	Walkthrough for turning dense financial disclosures into structured graph nodes and edges.
7. Embeddings + Vector Indexing	Neo4j Vector Indexing Docs	Build and manage vector embeddings inside Neo4j for hybrid retrieval.
8. Retrieval: Basic + Vector	Neo4j GraphRAG Basic Retriever Pattern	First step: combine chunked content and embedding for basic semantic retrieval.
9. Retrieval: Graph-Enhanced	Graph-Enhanced Vector Search with Neo4	Augment vector search with traversal logic to improve contextual accuracy.
10. Test2Cypher Automation	Text2Cypher Documentation & Examples	Translate user queries into Cypher automatically using LLMs – ideal for dynamic GraphRAG.
11. Agentic & Multi-Step Use	GraphRAG + NeoConverse + Agents	Build multi-tool agents that query graphs autonomously across task chains.
12. Semantic Enhancement	Topic Extraction for Semantic RAG	Use LLMs to extract topics and themes into your graph to add interpretability.
13. Deployment + Ops	Neo4j Deployment Best Practices	Tips for scaling and monitoring GraphRAG in production environments.