Graph Retrieval-Augmented Generation (Graph RAG) with Neo4j and Python

Graph RAG is a technique that combines Retrieval-Augmented Generation (RAG) with a knowledge graph. Traditional RAG retrieves text snippets via vector search to ground an LLM's answer. In Graph RAG, the system uses a **graph database** (a knowledge graph) as the retrieval source. In other words, Graph RAG **retrieves connected subgraphs** as context for the LLM [1] [2]. This richer, structured context allows the model to capture **relationships and multi-hop reasoning** that flat text alone can miss [3] [4]. In practice, the LLM may extract entities/relations from text and store them as nodes/edges in Neo4j, then use Cypher queries to fetch a relevant subgraph given a user query. These graph-structured facts are then provided to the LLM as background knowledge before answer generation.

Graphs are useful in RAG because they **explicitly represent entities and relationships**. This adds *relational context*, supports *multi-hop reasoning*, and represents hierarchical or complex data more naturally than a flat document store [3]. For example, Graph RAG can follow a chain of edges (e.g. author → works_at → company → has_project) to answer complex queries that require linking multiple facts. A knowledge graph can also index heterogeneous data (structured and unstructured) in one view, enabling more precise retrieval. In summary, Graph RAG differs from traditional vector-based RAG by integrating a graph traversal or query step into retrieval [5] [6], which often yields more accurate, context-aware results.

Neo4j Overview

Neo4j is a popular native graph database that implements the *labeled property graph* model [7]. In Neo4j, data is stored as **nodes** (entities) and **relationships** (edges), each of which can have *labels* and *properties* [7]. For example, you might have `(:Person {name: "Alice"})` and `(:Company {name: "Acme Inc."})`, connected by a `[:WORKS_FOR]` edge. This flexibility makes Neo4j **schema-free** – you can add new node or relationship types on the fly and change the ontology in real time [8]. Neo4j is fully ACID-compliant and optimized for graph traversals (its native storage model uses *index-free adjacency* for fast graph queries). It also offers built-in graph data science libraries and full-text/vector indexing for advanced analytics. Crucially for Graph RAG, Neo4j can store text embeddings on nodes and support k-NN similarity search using its vector index [9]. The combination of a flexible graph schema and powerful query engine makes Neo4j ideal as the knowledge graph backend in RAG.

Neo4j's **Cypher** query language is a concise, SQL-like way to read and write graph data. For example, `MATCH (a:Person)-[:KNOWS]->(b) RETURN a, b` finds all connected persons. The official Neo4j Python Driver lets you run Cypher queries from Python apps [10]. (Neo4j also has community tools like Py2neo, though the official driver is now the recommended approach.) Using these libraries, you can create nodes/relationships (`CREATE`/`MERGE`), run complex `MATCH` queries, and even call Neo4j procedures. Neo4j's integration with Python makes it straightforward to build ingestion and query pipelines in Python code [10] [11].

Key Neo4j Features for Graph RAG

Neo4j provides several features that support Graph RAG workflows:

- **Flexible schema** – Neo4j's graph model is *schema-optional*. You can define new node labels or relationship types at any time without downtime [8]. This "whiteboard-friendly" model means you can easily evolve the knowledge graph as your domain knowledge grows (for example, adding a new entity type or relationship).
- **Cypher query language** – Cypher is a powerful declarative language for graphs. It can express path queries, pattern matching, graph projections, and even full-text or vector index lookups. For example, you can use `CALL db.index.vector.queryNodes(...)` to find nearest-neighbor nodes by embedding [12] [13]. Since Cypher is tightly integrated, you can combine graph traversal with filtering or similarity search in one query, which is very useful for Graph RAG retrieval.
- **Native graph processing** – Neo4j is built for graph traversals. It stores relationships as direct pointers (index-free adjacency), so multi-hop traversals are very fast. Neo4j also offers the Graph Data Science (GDS) library with advanced algorithms (PageRank, community detection, path finding, etc.) which can be used to enrich queries or reasoning steps in a RAG system.
- **Python integration** – Neo4j provides an official Python driver (installable via `pip install neo4j`) that lets you connect to any Neo4j instance, send Cypher queries, and process results in Python [10]. There are also higher-level Python libraries (e.g. LangChain integrations, neo4j-graphql bindings, etc.) that streamline creating and querying knowledge graphs. The community library Py2neo is also available for graph operations, though Neo4j now recommends the official driver for new projects.
- **Vector and full-text indexing** – Modern Neo4j versions support hybrid retrieval. You can create a **vector index** on a node property (holding an embedding) and run KNN queries [9]. You can also define full-text search indexes to quickly find nodes by keywords. This means a Graph RAG system can use both semantic search (via embeddings in Neo4j) and graph queries to retrieve context.

Collectively, these features make Neo4j a strong choice for Graph RAG: it can store complex, structured domain knowledge and serve up relevant subgraphs on demand.

Implementation Guide

Below is a step-by-step outline for building a Graph RAG application with Neo4j and Python. Each step includes example Python code where helpful.

1. Set Up Neo4j

Install and run a Neo4j database. You have two main options:

- **Neo4j Aura (Cloud)**: Sign up for a free Neo4j AuraDB instance (cloud) to get an instant Neo4j 5.x database (no local install needed). Aura handles the server for you. See Neo4j Aura for a free trial.
- **Neo4j Desktop or Docker**: Download Neo4j Desktop (https://neo4j.com/download) and create a local database, or run Neo4j in Docker. For example, you can use `docker run -p 7474:7474 -p 7687:7687 neo4j:latest` to start a local Neo4j. Ensure you note the **Bolt URI** (e.g. `bolt://localhost:7687`) and credentials (default user "neo4j" and your password).

Once Neo4j is running (you can open its browser UI at http://localhost:7474), you can create constraints or indexes if desired (though not required for a small test). For example, in Neo4j Browser you might run `CREATE CONSTRAINT ON (p:Person) ASSERT p.id IS UNIQUE` to enforce unique IDs.

2. Connect Neo4j with Python

In your Python environment, install the Neo4j driver (and optionally Py2neo):

```
pip install neo4j        # official driver
pip install py2neo       # community library (optional)
```

Using the **official Neo4j driver**:

```
from neo4j import GraphDatabase

# Replace with your URI and credentials
uri = "neo4j://localhost:7687"      # or neo4j+s://<your-
instance>.databases.neo4j.io
username = "neo4j"
password = "your_password"

# Create a driver instance
driver = GraphDatabase.driver(uri, auth=(username, password))
# Verify connectivity (optional)
driver.verify_connectivity()
```

Using **Py2neo** (older style):

```
from py2neo import Graph

graph = Graph("bolt://localhost:7687", auth=("neo4j", "your_password"))
# Test a simple query
result = graph.run("RETURN 1 AS number")
print(result.data())  # should show [{'number': 1}]
```

Both drivers let you run Cypher queries against Neo4j. The official driver uses `driver.session()` and session-run methods, while Py2neo provides a simpler `Graph` object with `.run()`. In the examples below we will use the official driver.

3. Ingest and Structure Data as a Graph

Prepare your data so that it can be represented as nodes and relationships. For example, if you have JSON or CSV data about people and their jobs, you would create `(:Person)` and `(:Company)` nodes and `(:Person)-[:WORKS_FOR]->(:Company)` relationships.

In Python, you can use Cypher `CREATE` or `MERGE` commands. For instance, to create two people and a relationship:

```
with driver.session() as session:
    session.run("""
        MERGE (a:Person {name: $nameA})
        MERGE (b:Person {name: $nameB})
        MERGE (a)-[:KNOWS]->(b)
        """,
        nameA="Alice", nameB="Bob"
    )
```

This code uses parameterized Cypher (avoiding injection) to ensure nodes are created if they don't already exist. You can ingest any data: text documents can be pre-processed to extract entities/relations and then stored as nodes/edges in the graph. The Neo4j Graph Data Science (GDS) or apoc procedures can help with complex ingestion (e.g. splitting PDF text, calling NLP models, etc.).

For a simple example, using the official driver's `execute_query()`:

```
# Example: create a small graph
summary = driver.execute_query("""
    CREATE (p:Person {name: $person})
    CREATE (c:Company {name: $company})
    CREATE (p)-[:WORKS_FOR]->(c)
    """,
    person="Carol", company="Acme Inc",
    database_="neo4j"
).summary
print(f"Nodes created: {summary.counters.nodes_created}")
```

[11] (This example illustrates using parameters to create nodes and relationships.)

You can ingest larger datasets by batching these queries, using `UNWIND` on lists, or using `LOAD CSV` if your data is in CSV format. Once your data is loaded, Neo4j will maintain the graph of entities and relationships which you can query next.

4. Query the Graph for Relevant Context

When a user query arrives, we need to retrieve relevant facts from the graph. This is done with Cypher queries that match nodes/paths relevant to the question. For example, if the user query is about a person, you might find that person's node and then expand nearby relationships:

```
query = """
MATCH (p:Person {name: $personName})
```

```
OPTIONAL MATCH (p)-[:WORKS_FOR]->(c:Company)
OPTIONAL MATCH (p)-[:KNOWS]->(friend:Person)
RETURN p, collect(c.name) AS companies, collect(friend.name) AS friends
"""
result = driver.execute_query(query, personName="Alice", database_="neo4j")
for record in result.records:
    print(record["p"]["name"], "works at", record["companies"], "and knows",
record["friends"])
```

This kind of multi-`MATCH` Cypher query fetches a subgraph around the entity. You can adapt queries to follow multiple hops or to focus on certain relationship types. For more sophisticated retrieval (e.g. semantic similarity), Neo4j's **vector index** can be used: you store text embeddings as node properties and then run a KNN query:

```
// Neo4j Cypher example (run via driver)
CALL db.index.vector.queryNodes('myIndex', 5, $queryVector)
YIELD node, score
RETURN node, score;
```

Here `myIndex` is a vector index previously created on a node property, and `$queryVector` is the query embedding (e.g. from OpenAI's text-embedding-ada-002). This finds the top-5 most similar nodes. That retrieved list of nodes (and their text) can then be added to the LLM context. In practice, many Graph RAG implementations use a **hybrid** approach: first use vector or full-text search to find a few candidate entities/documents, then run graph traversals on those to fetch related facts. The Neo4j docs example below shows a combined approach using full-text lookup plus multi-hop MATCHes [14]:

```
// Example Graph RAG Cypher (from Neo4j blog)
CALL db.index.fulltext.queryNodes('entity', 'Palantir', {limit:1}) YIELD node
AS org
OPTIONAL MATCH (org)-[:HAS_INVESTOR]->(inv:Person)
OPTIONAL MATCH (org)-[:HAS_COMPETITOR]-(comp:Company)
WITH org.name AS company, COLLECT(DISTINCT inv.name) AS investors,
COLLECT(DISTINCT comp.name) AS competitors
RETURN company, investors, competitors;
```

This query finds the "Palantir" organization node and collects its related people and companies [14]. You can adapt similar queries to your domain.

5. Integrate with an LLM for Answer Generation

Once the relevant graph context is retrieved, you feed it into an LLM to generate the answer. In a simple setup, you would concatenate the user query and the retrieved facts into a prompt. For example (using OpenAI's Python library):

```
import openai
openai.api_key = "YOUR_API_KEY"
context_facts = "Alice works at Acme Inc. Alice knows Bob."
user_query = "Where does Alice work?"
prompt = f"Use the following facts to answer the question:\n{context_facts}\nQ:
{user_query}\nA:"
response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[{"role": "user", "content": prompt}]
)
answer = response.choices[0].message.content
print(answer)
```

More robust applications might use **LangChain** or other frameworks to structure this pipeline. For instance, LangChain provides a `Neo4jGraph` wrapper and chain utilities to automate the retrieval and prompting steps [15]. You could set up a `GraphQL` retrieval chain or a custom pipeline that first queries Neo4j and then calls `OpenAI` under the hood. The key idea is the same: the LLM's context includes the *knowledge graph subgraph* (in text form) found by Neo4j.

Example Python Snippets

Below are concise code snippets illustrating the key steps:

- **Connecting to Neo4j (Python driver):**

```
from neo4j import GraphDatabase
driver = GraphDatabase.driver("bolt://localhost:7687",
auth=("neo4j","password"))
with driver.session() as session:
    result = session.run("RETURN 1 AS number")
    print(result.single()["number"])
```

(*This uses the official driver* [16].)

- **Creating nodes/relationships:**

```
with driver.session() as session:
    session.run("""
        MERGE (p:Person {name: $name})
        MERGE (c:Company {name: $company})
        MERGE (p)-[:WORKS_FOR]->(c)
        """, name="Carol", company="Acme Inc.")
```

(*Example of ingesting data into the graph.*)

- **Querying the graph:**

```
query = """
  MATCH (p:Person {name:$name})
  OPTIONAL MATCH (p)-[:WORKS_FOR]->(c:Company)
  RETURN p.name AS person, collect(c.name) AS companies
"""
records = driver.execute_query(query, name="Carol", database_="neo4j")
for record in records:
    print(record["person"], "works at", record["companies"])
```

(*Retrieves Carol's companies. Adapt MATCH patterns as needed.*)

- **Feeding context to an LLM:**

```
# (After retrieving context from Neo4j)
facts = "Carol works at Acme Inc."
question = "Where does Carol work?"
import openai
openai.api_key = "..."
completion = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[{"role": "user", "content": facts + "\nQ: " + question}]
)
print(completion.choices[0].message.content)
```

(*Simple LLM call with ground-truth facts.*)

These examples sketch how to wire together Neo4j and an LLM. In a full application, you'd loop over user queries, perform the graph queries each time, and possibly store chat context or logs. But the core pattern remains: **Graph query -> format context -> LLM call**.

Architecture Diagram of a Graph RAG Pipeline

Figure: Example Graph RAG pipeline architecture. Neo4j holds the knowledge graph, which is queried based on user input. Retrieved subgraphs (entities and their relations) are passed as context to an LLM for answer generation. The diagram above illustrates a typical flow: the user submits a query (or uploads data), the system queries Neo4j (using Cypher or vector search) to fetch relevant facts, and then an LLM (e.g. via a chat interface) uses those facts to generate a grounded response. In this way, Neo4j serves as the dynamic knowledge base (grounding) that augments the LLM's output [2].

References: Neo4j documentation and blogs (e.g. Neo4j Developer Blog) provide detailed guides on setting up Neo4j, using Cypher, and integrating with Python [10] [1] [11]. The official Neo4j Python driver manual is a good reference for connecting and running queries [16] [11]. For Graph RAG concepts and examples, see Neo4j's Generative AI blog posts [1] [14] and related resources. These sources detail how knowledge graphs can enrich RAG pipelines and demonstrate code patterns similar to the ones above.

---

[1] [15] Using a Knowledge Graph to implement a RAG application
https://neo4j.com/blog/developer/knowledge-graph-rag-application/

[2] Graph RAG: Better Context for GenAI Apps | DataStax
https://www.datastax.com/guides/graph-rag

[3] [5] GraphRAG vs RAG. Retrieval-Augmented Generation (RAG) | by Praveen Raj | Medium
https://medium.com/@praveenraj.gowd/graphrag-vs-rag-40c19f27537f

[4] [6] Graph RAG vs traditional RAG: A comparative overview
https://www.ankursnewsletter.com/p/graph-rag-vs-traditional-rag-a-comparative

[7] Neo4j is a Graph Database - Overview of Neo4j 4.x
https://neo4j.com/graphacademy/training-overview-40/01-overview40-neo4j-graph-database/

[8] Flexible Graph Database Model | Graph Data Flexibility | Neo4j
https://neo4j.com/product/neo4j-graph-database/flexibility/

[9] [12] [13] Neo4j Vector Index and Search - Neo4j Labs
https://neo4j.com/labs/genai-ecosystem/vector-search/

[10] [11] [16] Build applications with Neo4j and Python - Neo4j Python Driver Manual
https://neo4j.com/docs/python-manual/current/

[14] GraphRAG and Agentic Architecture: Practical Experimentation with Neo4j and NeoConverse - Graph Database & Analytics
https://neo4j.com/blog/developer/graphrag-and-agentic-architecture-with-neoconverse/