LangChain Docs

Graph RAG

Graph RAG

Copy page

This guide provides an introduction to Graph RAG. For detailed documentation of all supported features and configurations, refer to the **Graph RAG Project Page**.

Overview

The `GraphRetriever` from the `langchain-graph-retriever` package provides a LangChain **retriever** that combines **unstructured** similarity search on vectors with **structured** traversal of metadata properties. This enables graph-based retrieval over an **existing** vector store.

Integration details

Retriever	Source	PyPI Package	Latest	Projec
GraphRetriever	github.com/datastax/graph-rag	langchain-graph-retriever	v0.8.0	Graph

Benefits

- **Link based on existing metadata:** Use existing metadata fields without additional processing. Retrieve more from an existing vector store!
- **Change links on demand:** Edges can be specified on-the-fly, allowing different relationships to be traversed based on the question.
- **Pluggable Traversal Strategies:** Use built-in traversal strategies like Eager or MMR, or define custom logic to select which nodes to explore.

LangChain Docs

Graph RAG

Setup

Installation

This retriever lives in the `langchain-graph-retriever` package.

pip uv

```
pip install -qU langchain-graph-retriever
```

Instantiation

The following examples will show how to perform graph traversal over some sample Documents about animals.

Prerequisites

Populating the vector store

This section shows how to populate a variety of vector stores with the sample data.

For help on choosing one of the vector stores below, or to add support for your vector store, consult the documentation about **Adapters and Supported Stores**.

AstraDB Apache Cassandra OpenSearch Chroma InMemory

Install the `langchain-graph-retriever` package with the `chroma` extra:

```
pip install "langchain-graph-retriever[chroma]"
```

Then create a vector store and load the test documents:

LangChain Docs

Graph RAG

LangChain Docs

Graph RAG

LangChain Docs

Graph RAG

```python
from langchain_chroma.vectorstores import Chroma
from langchain_graph_retriever.transformers import ShreddingTransformer

vector_store = Chroma.from_documents(
    documents=list(ShreddingTransformer().transform_documents(animals)),
    embedding=embeddings,
    collection_name="animals",
)
```

For help creating an Chroma connection, consult the **Chroma Vector Store Guide**.:::note Chroma doesn't support searching in nested metadata. Because of this it is necessary to use the `ShreddingTransformer` when inserting documents. :::

LangChain Docs

Graph RAG

```
from graph_retriever.strategies import Eager
from langchain_graph_retriever import GraphRetriever

traversal_retriever = GraphRetriever(
    store = vector_store,
    edges = [("habitat", "habitat"), ("origin", "origin")],
    strategy = Eager(k=5, start_k=1, max_depth=2),
)
```

The above creates a graph traversing retriever that starts with the nearest animal (`start_k=1`), retrieves 5 documents (`k=5`) and limits the search to documents that are at most 2 steps away from the first animal (`max_depth=2`).

The `edges` define how metadata values can be used for traversal. In this case, every animal is connected to other animals with the same `habitat` and/or `origin` .

```
results = traversal_retriever.invoke("what animals could be found

for doc in results:
    print(f"{doc.id}: {doc.page_content}")
```

```
capybara: capybaras are the largest rodents in the world and are
heron: herons are wading birds known for their long legs and necks, often see
crocodile: crocodiles are large reptiles with powerful jaws and a long lifesp
frog: frogs are amphibians known for their jumping ability and croaking sound
duck: ducks are waterfowl birds known for their webbed feet and quacking sou
```

Graph traversal improves retrieval quality by leveraging structured relationships in the data. Unlike standard similarity search (see below), it provides a clear, explainable rationale for why documents are selected.

In this case, the documents `capybara` , `heron` , `frog` , `crocodile` , and `newt` all

LangChain Docs

Graph RAG

Comparison to standard retrieval

When `max_depth=0`, the graph traversing retriever behaves like a standard retriever:

```
standard_retriever = GraphRetriever(
    store = vector_store,
    edges = [("habitat", "habitat"), ("origin", "origin")],
    strategy = Eager(k=5, start_k=5, max_depth=0),
)
```

This creates a retriever that starts with the nearest 5 animals (`start_k=5`), and returns them without any traversal (`max_depth=0`). The edge definitions are ignored in this case.

This is essentially the same as:

```
standard_retriever = vector_store.as_retriever(search_kwargs={"k'
```

For either case, invoking the retriever returns:

```
results = standard_retriever.invoke("what animals could be found

for doc in results:
    print(f"{doc.id}: {doc.page_content}")
```

```
capybara: capybaras are the largest rodents in the world and are
iguana: iguanas are large herbivorous lizards often found basking in trees ar
guinea pig: guinea pigs are small rodents often kept as pets due to their ger
hippopotamus: hippopotamuses are large semi-aquatic mammals known for their r
boar: boars are wild relatives of pigs, known for their tough hides and tusks
```

These documents are joined based on similarity alone. Any structural data that existed

Usage

Following the examples above, `invoke` is used to initiate retrieval on a query.

API reference

To explore all available parameters and advanced configurations, refer to the **Graph RAG API reference**.

LangChain Docs

Graph RAG

LangChain Academy

Blog

Trust Center

Powered by mintlify

LangChain Docs

Graph RAG