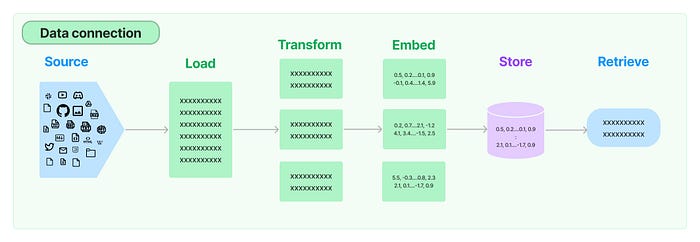
**Retrievers in RAG**

**What are Retrievers?**



Reference: LangChain docs <https://python.langchain.com/docs/modules/data_connection/>

So in these series of articles we are going to cover a comprehensive overview of the LangChain framework. We will cover all the aspects of the LangChain framework so that you don’t need to read the docs.

**Why do we need LangChain?**

So the first and foremost question that we should address is, that why should one use the LangChain framework to make llm apps. We could probably just make one using the openAI API. Here are a list of reasons why you should use LangChain:

1. LangChain doesn’t restrict you to one platform. It has made the effort to incorporate all sorts of service providers at every stage of the LLM application ranging from the LLM models to vector stores. The users have a myriad of options to choose from depending upon their needs
2. LangChain provides a layer of abstraction over repetitive tasks while building LLM apps.

So in a nutshell it is possible to create LLM apps even without LangChain. However that would take too much time, effort and energy, and probably the outcome wouldn’t be really impressive.

The LangChain framework has multiple modules which provide various functionalities. You can take advantages of these features to build complex LLM apps. In the following article we are going to cover the idea of retrieval.

**Retrieval**

The biggest hurdle faced when while using LLMs is the fact that they perform awful when used with external data. The solution to the problem is (RAG) or Retrieval Augmented Generation. According to this the data is retrieved from a datastore and then passed to the LLM model for generation.

Now there are several components of the Retrieval module, which as a whole solve the problem.

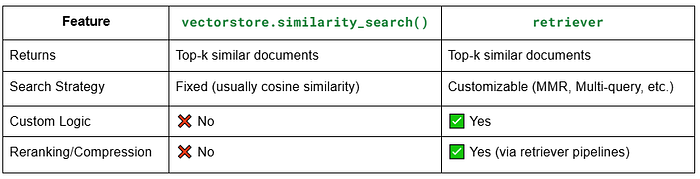
1. **Document loaders:** LangChain has incorporated 100 plus services to load all types of documents (pdf, text, markdown so and so forth), from all sorts of sources (local system, s3 bucket etc)
2. **Document Transformers:**Once the data has been loaded, it is important to transform the data to suit the needs of our application. These operations may include splitting the data, converting them into different languages, removing redundant parts, so and so forth. This helps us store relevant pieces of information
3. **Text Embedding models:**LangChain provides a high level interface to interact with a variety of Embedding models. The embedding algorithms represent text in the form of vectors. This makes searching for relevant text a lot faster
4. **Vector Stores:**With the introduction of Text Embedding models the vectors need to be stored efficiently. We need a database that provides efficient storage and searching of data. LangChain has incorporated 50 plus databases on the framework.
5. **Retrievers:**Once we have stored the data, it is time to retrieve it and feed it to the llm. LangChain provides a lot of out of the box algorithms to search relevant data, ranging from basic semantic search to very complex algorithms to improve performance

**🚀 What Are Retrievers?**

A **Retriever** is a LangChain component used to **fetch relevant chunks of data** based on a user query.  
Think of it as a **search engine** for your embedded documents.

Yes, we can use vectorstore.similarity\_search() directly, but **retrievers give us more flexibility** and control over how results are selected and returned.

Press enter or click to view image in full size



In short: retrievers are a **wrapper around vector store search**, adding **intelligence and flexibility**.

Code Example:

**Code Example.**

Here is an example implementation.

from langchain.chains import RetrievalQA  
from langchain.llms import OpenAI  
from langchain.document\_loaders import TextLoader  
from langchain.indexes import VectorstoreIndexCreator  
  
loader = TextLoader('./state\_of\_the\_union.txt', encoding="utf-8")  
  
index = VectorstoreIndexCreator().from\_loaders([loader])  
  
query = "What did the president say about Ketanji Brown Jackson"  
ans = index.query(query)  
print(ans)

The above piece of code reads the contents in the text file “state \_of\_the\_union.txt” to answer the following question.

The *VectorstoreIndexCreator()*does a bunch of things under the hood. Lets break it down one by one.

1. First we fetch the documents from the text file

documents = loader.load()

2. After that we split the documents into chunks. Here the size of each chunk is 100 words and we don’t have any overlap.

from langchain.text\_splitter import CharacterTextSplitter  
text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)  
texts = text\_splitter.split\_documents(documents)

3. We will then select which embeddings we want to use.

from langchain.embeddings import OpenAIEmbeddings  
embeddings = OpenAIEmbeddings()

4. We now create the vector store to use as the index.

from langchain.vectorstores import Chroma  
db = Chroma.from\_documents(texts, embeddings)

5. Then, we expose this index in a retriever interface. After that we we create a chain and use it to answer questions!

retriever = db.as\_retriever()  
qa = RetrievalQA.from\_chain\_type(llm=OpenAI(), chain\_type="stuff", retriever=retriever)

All the above tasks have been just wrapped inside one complete function, i.e. *VectorstoreIndexCreator().*

**what is the difference between vector store and vector database**

people often confuse **vector stores** and **vector databases** in LangChain and general LLM workflows. Let’s break it down clearly:

**🔹 Vector Store**

* A **vector store** is a simple abstraction / library for **storing and retrieving vectors (embeddings)**.
* It doesn’t necessarily manage scalability, persistence, or distributed storage.
* Typically runs in **memory** or in a lightweight local backend (e.g., FAISS, Chroma running in-memory).
* Provides:
  + Store documents + embeddings
  + Retrieve nearest neighbors (similarity search, cosine similarity, dot product, etc.)
  + Basic metadata filtering

👉 Example:

* **FAISS**, **Chroma (in-memory mode)**, **SKLearn NearestNeighbors**
* Used when you just want to load, embed, and query documents locally in your Python app.

**🔹 Vector Database**

* A **vector database** is a **production-grade database system** designed to handle **billions of embeddings**, often distributed and persistent.
* They provide:
  + **Persistence** (data survives restarts)
  + **Scalability** (clustered / distributed indexing)
  + **Advanced filtering** (metadata, hybrid queries with text/keywords + vectors)
  + **CRUD operations** (insert, update, delete, search)
  + APIs for integration with enterprise apps

👉 Examples:

* **Chroma (with persistent storage)**
* **Weaviate**
* **Pinecone**
* **Milvus**
* **Qdrant**

**🔑 Key Difference**

| **Feature** | **Vector Store** | **Vector Database** |
| --- | --- | --- |
| **Scale** | Small → fits in memory | Large scale → millions/billions of vectors |
| **Persistence** | Often not persistent (unless manually saved) | Persistent by default |
| **Deployment** | Local / library-based | Runs as a service (cloud or self-hosted) |
| **Features** | Basic similarity search | Metadata filtering, hybrid search, CRUD |
| **Use Case** | Quick prototyping, experiments | Production-grade RAG, real-time search |

**📌 Example with LangChain**

from langchain\_community.vectorstores import FAISS, Chroma

from langchain\_openai import OpenAIEmbeddings

# Embeddings model

embeddings = OpenAIEmbeddings()

# Vector Store (FAISS in-memory, simple)

vectorstore = FAISS.from\_texts(["hello world", "machine learning is cool"], embedding=embeddings)

# Vector Database (Chroma persistent)

vectordb = Chroma(

collection\_name="my\_docs",

embedding\_function=embeddings,

persist\_directory="./chroma\_store" # persistence → makes it a DB

)

# Add documents

vectordb.add\_texts(["This is a new document"])

vectordb.persist()

# Search

results = vectordb.similarity\_search("machine learning", k=2)

print(results)

✅ So:

* **Vector store** = lightweight, in-memory, simple, often for prototyping.
* **Vector database** = scalable, persistent, production-ready.

let’s go step by step. I’ll give you **full code** that:

1. Loads a .txt file.
2. Splits it into chunks using a text splitter.
3. Stores chunks into a **Chroma vector store** with embeddings.
4. Performs **CRUD operations** (Create, Read, Update, Delete) on the stored documents using **LangChain retrievers & chains with prompts**.

We’ll use **LangChain + OpenAI + Chroma**.

**✅ Full Example Code**

from langchain\_openai import OpenAIEmbeddings, ChatOpenAI

from langchain\_text\_splitters import CharacterTextSplitter

from langchain\_community.document\_loaders import TextLoader

from langchain\_chroma import Chroma

from langchain.schema import Document

from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

import os

# -------------------------------

# 1. Setup LLM + Embeddings

# -------------------------------

os.environ["OPENAI\_API\_KEY"] = "your-openai-key" # put your key here

llm = ChatOpenAI(model="gpt-4o-mini")

embeddings = OpenAIEmbeddings()

# -------------------------------

# 2. Load & Split Text File

# -------------------------------

loader = TextLoader("sample.txt") # Replace with your text file

docs = loader.load()

splitter = CharacterTextSplitter(chunk\_size=200, chunk\_overlap=50)

splits = splitter.split\_documents(docs)

# -------------------------------

# 3. Create Chroma Vector Store

# -------------------------------

vectorstore = Chroma(

collection\_name="my\_docs",

embedding\_function=embeddings,

persist\_directory="./chroma\_store" # saves locally

)

# Add documents to vectorstore (Create)

vectorstore.add\_documents(splits)

print("✅ Documents inserted into Chroma DB")

# -------------------------------

# 4. CRUD OPERATIONS

# -------------------------------

# ---- C: Create (already done by add\_documents) ----

new\_doc = Document(page\_content="LangChain makes AI easy with LLMs and retrievers.", metadata={"id": "extra1"})

vectorstore.add\_documents([new\_doc])

print("✅ Created new document.")

# ---- R: Read (query with retriever + chain) ----

retriever = vectorstore.as\_retriever(search\_type="similarity", search\_kwargs={"k": 2})

query = "What is LangChain?"

retrieved\_docs = retriever.get\_relevant\_documents(query)

print("\n🔍 Retrieved Docs:")

for d in retrieved\_docs:

print("-", d.page\_content[:100])

# Use prompt + chain to get nice answer

prompt = PromptTemplate(

input\_variables=["context", "question"],

template="Answer the question based on context:\nContext: {context}\nQuestion: {question}"

)

chain = LLMChain(llm=llm, prompt=prompt)

context\_text = " ".join([doc.page\_content for doc in retrieved\_docs])

answer = chain.run(context=context\_text, question=query)

print("\n🤖 LLM Answer:", answer)

# ---- U: Update (delete old + add new version) ----

vectorstore.delete(ids=["extra1"]) # remove old

updated\_doc = Document(page\_content="LangChain is a framework for building AI apps using LLMs, retrievers, and chains.", metadata={"id": "extra1"})

vectorstore.add\_documents([updated\_doc])

print("✅ Document updated.")

# ---- D: Delete ----

vectorstore.delete(ids=["extra1"])

print("✅ Document deleted.")

# -------------------------------

# 5. Persist & Reload Store

# -------------------------------

vectorstore.persist() # Save to disk

print("💾 Chroma store persisted at ./chroma\_store")

**🔑 Key Points**

* **Loader**: Loads .txt into Document objects.
* **Splitter**: Breaks text into chunks for embeddings.
* **Chroma**: Vector DB to store & retrieve embeddings.
* **CRUD**:
  + **Create** → add\_documents()
  + **Read** → retriever.get\_relevant\_documents()
  + **Update** → delete old doc + add new version
  + **Delete** → delete(ids=[...])
* **Chain with Prompt**: Converts retrieved docs into a final LLM answer.

👉 Do you want me to also show you how to **list all documents** currently in Chroma DB (like a "Read all" operation) along with CRUD?

**How to configure a database (like Chroma) in LangChain**

**and How to perform CRUD (Create, Read, Update, Delete) operations on that** **database.**

**1. Configuring Chroma in LangChain**

Chroma is an open-source **vector database** that stores documents + embeddings. In LangChain, you typically:

* Install Chroma
* Pick an embedding model (e.g., OpenAI, HuggingFace)
* Insert documents (with metadata)
* Query via similarity search

**Install**

pip install chromadb langchain

**Example Setup**

from langchain\_community.vectorstores import Chroma

from langchain\_openai import OpenAIEmbeddings

# Step 1: Initialize embeddings

embedding\_function = OpenAIEmbeddings(model="text-embedding-3-small")

# Step 2: Create Chroma DB (persistent or in-memory)

vectorstore = Chroma(

collection\_name="my\_collection",

embedding\_function=embedding\_function,

persist\_directory="./chroma\_db" # optional, for persistence

)

**2. CRUD Operations in Chroma**

Chroma stores data as (id, document, embedding, metadata).  
Here’s how you do each CRUD:

**✅ Create (Insert Documents)**

docs = [

{"id": "doc1", "text": "LangChain makes LLM orchestration easier.", "metadata": {"topic": "AI"}},

{"id": "doc2", "text": "Chroma is a vector database optimized for embeddings.", "metadata": {"topic": "DB"}},

]

# Add to vectorstore

vectorstore.add\_texts(

texts=[d["text"] for d in docs],

metadatas=[d["metadata"] for d in docs],

ids=[d["id"] for d in docs]

)

vectorstore.persist() # Save to disk

**✅ Read (Query / Retrieve)**

# Similarity search

results = vectorstore.similarity\_search("What is LangChain?", k=2)

for r in results:

print(r.page\_content, r.metadata)

**✅ Update (Replace or Add New Version)**

Chroma does not support in-place updates directly.  
You **delete by ID** then **add again with new text/metadata**.

# Delete old doc

vectorstore.delete(ids=["doc1"])

# Re-add with updated content

vectorstore.add\_texts(

texts=["LangChain helps build intelligent applications using LLMs."],

metadatas=[{"topic": "AI"}],

ids=["doc1"]

)

vectorstore.persist()

**✅ Delete**

# Delete by IDs

vectorstore.delete(ids=["doc2"])

vectorstore.persist()

**3. Using Chroma as a Retriever**

Chroma can be wrapped in a Retriever for LangChain pipelines:

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 3})

# Now you can plug into RetrievalQA or chains

docs = retriever.get\_relevant\_documents("Tell me about databases.")

for d in docs:

print(d.page\_content, d.metadata)

**🔑 Summary**

* **Chroma setup** → initialize embeddings + collection.
* **CRUD** → add\_texts() for create, similarity\_search() for read, delete()+add\_texts() for update, delete() for delete.
* **Retriever** → wrap vectorstore.as\_retriever() for LangChain integration.

**List all different retrievers in vector store and vector database**

LangChain provides a wide variety of **pre-built retrievers**, each tailored to a specific **data source**.  
These retrievers abstract away the complexities of querying from structured or external sources like APIs, databases, or platforms

— retrievers are the "search API" layer in **LangChain**, which sit on top of **vector stores** or **vector databases**.

Here’s the breakdown:

**🔹 1. Vector Store Retrievers**

These retrievers are wrappers around *simple vector stores* (like Chroma, FAISS, Weaviate’s in-memory mode, etc.). They usually support only *basic similarity search*.

* **VectorStoreRetriever**
  + Standard retriever wrapping any vector store (Chroma, FAISS, Pinecone, etc.).
  + Uses cosine similarity / dot product / Euclidean distance.
  + Example: retriever = vectorstore.as\_retriever()
* **MultiVectorRetriever**
  + Supports multiple embeddings per document.
  + Useful for documents with structured fields (title + content embeddings).
* **ParentDocumentRetriever**
  + Stores child chunks in the vector store.
  + Retrieves parent documents when a child chunk matches.
  + Helps with preserving context.
* **ContextualCompressionRetriever**
  + Wraps another retriever.
  + First fetches docs → compresses context (with LLM or embeddings).
  + Useful for long documents where you want only the most relevant snippets.
* **EnsembleRetriever**
  + Combines multiple retrievers (e.g., vector + BM25 keyword retriever).
  + Can use weighted score fusion.
* **TimeWeightedVectorStoreRetriever**
  + Prioritizes *recent documents* along with similarity.
  + Good for chatbots with temporal memory.

**🔹 2. Vector Database Retrievers**

These are retrievers that leverage *database-level features* of dedicated vector DBs (Pinecone, Weaviate, Milvus, Qdrant, etc.).  
They typically provide **richer search methods** beyond similarity.

* **Similarity Search Retriever**
  + Default method in most vector DBs.
  + Finds nearest neighbors to query embedding.
* **MMR Retriever (Maximal Marginal Relevance)**
  + Reduces redundancy in results.
  + Balances similarity + diversity.
* **Hybrid Search Retriever**
  + Combines **vector similarity** with **keyword search** (BM25).
  + Example: Weaviate + HybridRetriever, Pinecone hybrid.
* **Metadata Filtering Retriever**
  + Query includes metadata conditions (e.g., only docs from last week, or category = "finance").
  + Most vector DB retrievers (Pinecone, Weaviate, Qdrant) support this.
* **Self-Query Retriever**
  + Uses an LLM to **translate a natural language query** into:
    - embedding query
    - metadata filters
  + Very powerful when user asks: *“find research papers about AI published after 2020”*.
* **Weaviate BM25 + Vector Retriever**
  + Weaviate has native keyword BM25 retriever + vector retriever → can be combined.
* **Milvus Hybrid Retriever**
  + Supports scalar filtering + ANN vector search.
  + Used for structured + unstructured queries.

✅ **In short:**

* **Vector Store Retrievers** = simple similarity search, chunk management, and hybrids built on top of FAISS/Chroma/etc.
* **Vector Database Retrievers** = advanced retrievers leveraging **DB features** like hybrid search, filtering, metadata queries, temporal weighting, etc.

Would you like me to also create a **code example** where we use **Chroma (vector store retriever)** and **Weaviate (vector database retriever)** side-by-side, so you can see the practical difference?

**🔹 1. Vector Store Retrievers**

different **retrievers** available in vector stores and when/why to use each. Let’s go step by step:

**🔎 1. VectorStoreRetriever**

* **What it is**: The most basic retriever. Wraps a vector store like **Chroma, FAISS, Pinecone, Weaviate, etc.**
* **How it works**: Uses **cosine similarity, dot product, or Euclidean distance** to fetch k nearest neighbors.
* **When to use**: When you just want **simple semantic search**.
* ✅ **Pros**: Fast, simple, widely supported.
* ❌ **Cons**: No advanced logic (e.g., parent-child doc relationships, recency weighting).

**Example**

from langchain\_chroma import Chroma

from langchain\_openai import OpenAIEmbeddings

# Create embeddings and vectorstore

embeddings = OpenAIEmbeddings()

vectorstore = Chroma(persist\_directory="./chroma\_store", embedding\_function=embeddings)

# Basic retriever

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 3})

query = "What is LangChain?"

docs = retriever.get\_relevant\_documents(query)

for d in docs:

print(d.page\_content)

**🔎 2. MultiVectorRetriever**

* **What it is**: Stores **multiple embeddings per document**.
* **When to use**: For **structured data** like *title + abstract + body* (each gets its own embedding). Helps when **titles are short but important**.
* ✅ **Pros**: Captures multiple views of the same doc.
* ❌ **Cons**: More storage & compute (multiple vectors per doc).

**Example**

from langchain.retrievers.multi\_vector import MultiVectorRetriever

from langchain\_core.documents import Document

from langchain.storage import InMemoryStore

import uuid

# Vectorstore + storage for mapping doc\_ids

vectorstore = Chroma(embedding\_function=embeddings, persist\_directory="./chroma\_multi")

store = InMemoryStore()

id\_key = "doc\_id"

retriever = MultiVectorRetriever(vectorstore=vectorstore, docstore=store, id\_key=id\_key)

# Example doc with title + content

doc\_id = str(uuid.uuid4())

retriever.vectorstore.add\_documents([

Document(page\_content="LangChain is a framework for LLM applications", metadata={id\_key: doc\_id}),

Document(page\_content="Title: LangChain Overview", metadata={id\_key: doc\_id})

])

retriever.docstore.mset([(doc\_id, Document(page\_content="Full LangChain doc"))])

docs = retriever.get\_relevant\_documents("Tell me about LangChain")

print(docs[0].page\_content) # Full parent doc

**🔎 3. ParentDocumentRetriever**

* **What it is**: Stores **child chunks** in vector store but returns the **parent doc**.
* **When to use**: For **large docs** where context matters (e.g., legal contracts, research papers).
* ✅ **Pros**: Keeps full context without embedding huge docs.
* ❌ **Cons**: Larger retrievals (since parent can be big).

**Example**

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

from langchain.retrievers import ParentDocumentRetriever

splitter = RecursiveCharacterTextSplitter(chunk\_size=100, chunk\_overlap=20)

vectorstore = Chroma(embedding\_function=embeddings, persist\_directory="./chroma\_parent")

docstore = InMemoryStore()

retriever = ParentDocumentRetriever(

vectorstore=vectorstore,

docstore=docstore,

child\_splitter=splitter,

)

# Add parent doc

retriever.add\_documents([Document(page\_content="LangChain lets you build apps with LLMs in a modular way...")])

docs = retriever.get\_relevant\_documents("What is LangChain?")

print(docs[0].page\_content)

**🔎 4. ContextualCompressionRetriever**

* **What it is**: Wraps another retriever. Retrieves → compresses docs with LLM or embeddings.
* **When to use**: For **very long docs**, where only a few **snippets** matter.
* ✅ **Pros**: Saves token cost, improves relevance.
* ❌ **Cons**: Extra LLM calls → slower.

**Example**

from langchain.retrievers import ContextualCompressionRetriever

from langchain.retrievers.document\_compressors import LLMChainFilter

from langchain\_openai import ChatOpenAI

base\_retriever = vectorstore.as\_retriever(search\_kwargs={"k": 5})

llm = ChatOpenAI(model="gpt-4o-mini")

compressor = LLMChainFilter.from\_llm(llm)

retriever = ContextualCompressionRetriever(base\_compressor=compressor, base\_retriever=base\_retriever)

docs = retriever.get\_relevant\_documents("How does LangChain support retrievers?")

for d in docs:

print(d.page\_content)

**🔎 5. EnsembleRetriever**

* **What it is**: Combines **multiple retrievers** (e.g., semantic + keyword search).
* **When to use**: When you want **hybrid search** (semantic + lexical).
* ✅ **Pros**: Covers weaknesses of a single retriever.
* ❌ **Cons**: More compute, needs tuning of weights.

**Example**

from langchain.retrievers import EnsembleRetriever

from langchain.retrievers import BM25Retriever

# BM25 keyword retriever

bm25 = BM25Retriever.from\_documents([Document(page\_content="LangChain is powerful for LLM apps")])

# Vector retriever

vector\_retriever = vectorstore.as\_retriever(search\_kwargs={"k": 3})

# Combine

retriever = EnsembleRetriever(

retrievers=[bm25, vector\_retriever],

weights=[0.3, 0.7] # more weight to vector search

)

docs = retriever.get\_relevant\_documents("LangChain apps")

print(docs[0].page\_content)

**🔎 6. TimeWeightedVectorStoreRetriever**

* **What it is**: Prioritizes **recent docs** along with similarity.
* **When to use**: Chatbots that need **short-term memory** (recent chats more relevant).
* ✅ **Pros**: Mimics human recency bias.
* ❌ **Cons**: Older but important docs may get ignored.

**Example**

from langchain.retrievers import TimeWeightedVectorStoreRetriever

retriever = TimeWeightedVectorStoreRetriever(

vectorstore=vectorstore,

decay\_rate=0.01, # how fast old docs lose importance

k=3

)

retriever.add\_documents([Document(page\_content="User asked about LangChain retrievers.")])

docs = retriever.get\_relevant\_documents("Tell me about retrievers")

print(docs[0].page\_content)

**✅ Summary Table**

| **Retriever** | **Best For** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **VectorStoreRetriever** | Simple semantic search | Fast, simple | No context handling |
| **MultiVectorRetriever** | Structured data (title + body) | Captures multiple aspects | Higher storage cost |
| **ParentDocumentRetriever** | Large docs | Returns full context | Parent docs can be large |
| **ContextualCompressionRetriever** | Long docs, token saving | Precise snippets | Slower (extra LLM calls) |
| **EnsembleRetriever** | Hybrid search | Combines strengths | Needs weight tuning |
| **TimeWeightedVectorStoreRetriever** | Chatbots, recency | Mimics memory | Ignores old docs |

⚡ Now — I can give you a **realistic end-to-end project** where we:

* Load a dataset (say FAQs or support tickets),
* Store in **Chroma**,
* Try **all retrievers** with the same query and compare results.

👉 Do you want me to prepare that **full runnable code** for you as a single script?

**🔹 2. Vector Database Retrievers**

This is the **second category of retrievers** — the ones that leverage **database-level features of dedicated vector DBs** (like Pinecone, Weaviate, Milvus, Qdrant).

**🔹 1. Similarity Search Retriever**

**What it is:**

* The most common retriever.
* Finds documents most similar to a query using vector similarity (cosine, dot product, Euclidean).

**When to use:**

* Baseline search (most RAG systems start here).
* When you only need semantic similarity without extra logic.

**Pros:**

* Fast and simple.
* Works out of the box with any vector DB.

**Cons:**

* Can return redundant or overly narrow results.
* Doesn’t enforce diversity.

✅ **Example with Pinecone**

from langchain\_openai import OpenAIEmbeddings

from langchain\_pinecone import PineconeVectorStore

# Setup

embeddings = OpenAIEmbeddings()

index\_name = "my-index"

vectorstore = PineconeVectorStore(index\_name=index\_name, embedding=embeddings)

# Retriever

retriever = vectorstore.as\_retriever(search\_type="similarity", search\_kwargs={"k": 3})

results = retriever.get\_relevant\_documents("What is generative AI?")

for r in results:

print(r.page\_content)

**🔹 2. MMR Retriever (Maximal Marginal Relevance)**

**What it is:**

* Returns diverse but relevant results.
* Balances similarity with diversity (avoids duplicates).

**When to use:**

* For Q&A where you want coverage from different angles.
* When documents may be repetitive.

**Pros:**

* Reduces redundancy.
* Covers broader context.

**Cons:**

* Slightly slower than plain similarity.
* Might return less strictly relevant docs.

✅ **Example**

retriever = vectorstore.as\_retriever(

search\_type="mmr",

search\_kwargs={"k": 5, "lambda\_mult": 0.5} # lambda controls diversity

)

results = retriever.get\_relevant\_documents("Explain reinforcement learning")

for r in results:

print(r.page\_content)

**🔹 3. Hybrid Search Retriever**

**What it is:**

* Combines **vector search (semantic)** with **keyword search (BM25, sparse vectors)**.
* Example: Weaviate, Pinecone hybrid mode.

**When to use:**

* When queries may contain **keywords that embeddings miss**.
* Best for technical domains (code search, medical terms).

**Pros:**

* Best of both worlds (semantic + keyword).
* Improves recall for rare or OOV terms.

**Cons:**

* More complex to configure.
* Requires DB support (not all DBs support hybrid natively).

✅ **Example with Weaviate Hybrid**

import weaviate

client = weaviate.Client("http://localhost:8080")

result = client.query.get("Document", ["title", "content"]) \

.with\_hybrid(query="deep learning vs machine learning", alpha=0.5) \

.with\_limit(5) \

.do()

print(result)

**🔹 4. Metadata Filtering Retriever**

**What it is:**

* Adds filters on metadata during search.
* Example: Only retrieve docs where category = finance or date > 2022-01-01.

**When to use:**

* Enterprise RAG systems with structured metadata.
* When you want contextual filtering (author, time range, tags).

**Pros:**

* Very powerful for multi-tenant systems.
* Reduces irrelevant results.

**Cons:**

* Requires good metadata hygiene.
* Too restrictive filters may miss useful context.

✅ **Example with Qdrant**

from qdrant\_client import QdrantClient

from langchain\_qdrant import Qdrant

client = QdrantClient(":memory:") # in-memory example

qdrant = Qdrant(client=client, collection\_name="docs", embedding=embeddings)

retriever = qdrant.as\_retriever(

search\_kwargs={"k": 3, "filter": {"must": [{"key": "category", "match": {"value": "finance"}}]}}

)

results = retriever.get\_relevant\_documents("Explain inflation trends")

for r in results:

print(r.page\_content, r.metadata)

**🔹 5. Self-Query Retriever**

**What it is:**

* Uses an **LLM to interpret natural language queries** into:
  + embedding queries
  + metadata filters

**When to use:**

* Natural queries with filters:
  + “Find research papers about AI published after 2020.”
  + “Show me legal cases in California about copyright.”

**Pros:**

* Very user-friendly.
* Lets end-users query without knowing schema.

**Cons:**

* Relies on LLM correctness.
* More expensive (extra LLM calls).

✅ **Example with LangChain**

from langchain.chains import SelfQueryRetriever

from langchain\_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini")

retriever = SelfQueryRetriever.from\_llm(

llm, vectorstore,

document\_content\_description="Research papers",

metadata\_field\_info=[

{"name": "year", "type": "int", "description": "Year of publication"},

{"name": "category", "type": "string", "description": "Domain of research"}

],

verbose=True

)

results = retriever.get\_relevant\_documents("Find AI papers published after 2020")

for r in results:

print(r.page\_content, r.metadata)

**🔹 6. Weaviate BM25 + Vector Retriever**

**What it is:**

* Weaviate has **built-in BM25 keyword retriever + vector retriever**.
* You can fuse results.

**When to use:**

* When you want native hybrid search.
* For **Weaviate-powered RAG** projects.

✅ **Example**

result = client.query.get("Document", ["title", "content"]) \

.with\_bm25(query="transformer models") \

.with\_near\_text({"concepts": ["transformer models"]}) \

.with\_limit(5) \

.do()

print(result)

**🔹 7. Milvus Hybrid Retriever**

**What it is:**

* Milvus supports **scalar filtering + ANN vector search**.
* Example: Retrieve “AI articles from 2023 sorted by relevance”.

**When to use:**

* If you want **structured + unstructured** search.
* Great for analytics-heavy + RAG pipelines.

✅ **Example**

from pymilvus import connections, Collection

connections.connect("default", host="localhost", port="19530")

collection = Collection("documents")

results = collection.search(

data=[embeddings.embed\_query("AI in healthcare")],

anns\_field="embedding",

param={"metric\_type": "COSINE", "params": {"nprobe": 10}},

limit=5,

expr="year >= 2023"

)

for r in results[0]:

print(r)

**📝 Summary Table**

| **Retriever** | **Best For** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Similarity** | General semantic search | Simple, fast | Redundant, narrow |
| **MMR** | Diverse context | Reduces duplication | May lose precision |
| **Hybrid** | Mixed keyword + semantic | Best of both | More setup |
| **Metadata Filtering** | Structured data filtering | Precise results | Needs clean metadata |
| **Self-Query** | Natural filters from user queries | User-friendly | Expensive, LLM errors |
| **Weaviate BM25+Vector** | Native hybrid in Weaviate | Optimized | Tied to Weaviate |
| **Milvus Hybrid** | Structured + unstructured | Very powerful | Setup complexity |

👉 Do you want me to also prepare a **side-by-side full runnable demo** where I implement the same query using **Pinecone, Weaviate, Milvus, and Qdrant** so you can compare retrievers in action?

**Some Other Retrievers in vector database:-**

A **Vector Store Retriever** is the most common retriever in LangChain. It fetches documents from a vector store (like **FAISS**, **Chroma**, **Weaviate**) based on **semantic similarity** using **vector embeddings**.

**How It Works:**

* Store your documents in a **vector store**
* Each document is converted into a **dense vector** using an **embedding model**
* When the user submits a query:
* *It’s also embedded into a vector*
* *The retriever performs a similarity comparison between the****query vector****and****stored vectors***
* *It returns the****top-k most similar vectors/documents***

**Code Example:**

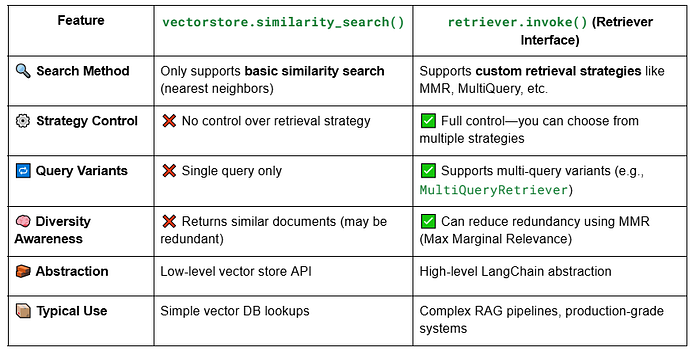
from langchain\_community.vectorstores import Chroma  
from langchain\_ollama import OllamaEmbeddings  
from langchain\_core.documents import Document  
  
# Step 1: Your source documents  
documents = [  
 Document(page\_content="LangChain helps developers build LLM applications easily."),  
 Document(page\_content="Chroma is a vector database optimized for LLM-based search."),  
 Document(page\_content="Embeddings convert text into high-dimensional vectors."),  
 Document(page\_content="OpenAI provides powerful embedding models."),  
]  
  
# Step 2: Initialize embedding model  
embedding\_model = OllamaEmbeddings(  
 model="llama2",  
)  
  
# Step 3: Create Chroma vector store in memory  
vectorstore = Chroma.from\_documents(  
 documents=documents,  
 embedding=embedding\_model,  
 collection\_name="my\_collection"  
)  
  
# Step 4: Convert vectorstore into a retriever  
retriever = vectorstore.as\_retriever(search\_kwargs={"k": 2})  
  
query = "What is Chroma used for?"  
results = retriever.invoke(query)  
  
for i, doc in enumerate(results):  
 print(f"\n--- Result {i+1} ---")  
 print(doc.page\_content)

*Alternative (Direct Search) Example:*

results = vectorstore.similarity\_search(query, k=2)

**🔍 Vector Store vs Retriever**

Press enter or click to view image in full size



📌 Example Comparison

1. **Using Vector Store Directly**

results = vectorstore.similarity\_search("climate change", k=3)

* Very fast
* Basic similarity
* No re-ranking or redundancy control

2. **Using the Retriever with Strategy**

from langchain.retrievers import MultiQueryRetriever  
  
retriever = MultiQueryRetriever.from\_llm(  
 retriever=vectorstore.as\_retriever(),  
 llm=ChatOpenAI()  
)  
docs = retriever.invoke("climate change")

* Generates multiple query formulations using an LLM
* Improves recall and coverage
* Better suited for production-grade semantic search

**🎯 Summary**

* Use similarity\_search() When you just want quick and simple lookups.
* Use Retriever If you want smart control over how results are found, ranked, or diversified.

**🔀 Retrievers Based on Search Strategies**

LangChain allows us to use **different search strategies** via retrievers, offering much more control than raw similarity search.

**1. 🎯 Maximal Marginal Relevance (MMR)**

*“How can we pick results that are not only relevant but also different from each other?”*

MMR is an information retrieval algorithm designed to reduce redundancy in the retrieved results while maintaining high relevance to the query.

**Why Use MMR Retriever?**

Regular search may return:

* Similar documents
* Repeated content
* Lack of diverse perspectives

**MMR helps by:**

1. Picking the most relevant document first
2. Then choosing documents that are both relevant and **least similar** to already selected docs
3. And so on…

**Ideal for**:

* RAG pipelines where diverse but relevant context is essential
* Especially useful when documents are semantically overlapping.

**Code Example**:

# Sample documents  
docs = [  
 Document(page\_content="LangChain makes it easy to work with LLMs."),  
 Document(page\_content="LangChain is used to build LLM based applications."),  
 Document(page\_content="Chroma is used to store and search document embeddings."),  
 Document(page\_content="Embeddings are vector representations of text."),  
 Document(page\_content="MMR helps you get diverse results when doing similarity search."),  
 Document(page\_content="LangChain supports Chroma, FAISS, Pinecone, and more."),  
]  
  
from langchain\_community.vectorstores import FAISS  
  
# Initialize OpenAI embeddings  
embedding\_model = OllamaEmbeddings(  
 model="llama2",  
)  
  
# Step 2: Create the FAISS vector store from documents  
vectorstore = FAISS.from\_documents(  
 documents=docs,  
 embedding=embedding\_model  
)  
  
# Enable MMR in the retriever  
retriever = vectorstore.as\_retriever(  
 search\_type="mmr", # <-- This enables MMR  
 search\_kwargs={"k": 3, "lambda\_mult": 0.5} # k = top results, lambda\_mult = relevance-diversity balance  
)  
  
query = "What is langchain?"  
results = retriever.invoke(query)  
  
for i, doc in enumerate(results):  
 print(f"\n--- Result {i+1} ---")  
 print(doc.page\_content)

*Lambda\_mult is between 0 and 1, if 1 then it behaves just like other normal similarity search and 0 it will give more diverse results*

**2. 🔁 Multi-Query Retriever**

Sometimes, a single query might not capture all the ways information is phrased in your documents, or if the user’s query is ambiguous, a single vector search might return irrelevant results.

For Example:

Query :

How can I stay health?

it could mean :

* What should I eat?
* How often should I exercise?
* How can I manage stress?

A single similarity search might miss documents that talk about those things but don’t use the word “healthy”.

**Solution**:

1. Pass the original query to an LLM
2. LLM generates **multiple variations** of the query
3. Each variation is used to search the vector store
4. Combine results, **deduplicate**, and return top results

**Visual Flow**:

query → LLM → [q1, q2, q3, q4, q5]  
 ↓ ↓ ↓  
 search search ...  
 ↓ ↓  
 results combined → deduped → final docs

Code Example:

from langchain.retrievers.multi\_query import MultiQueryRetriever  
from langchain\_community.vectorstores import FAISS  
from langchain\_ollama import OllamaEmbeddings  
from langchain\_ollama import ChatOllama  
  
# Relevant health & wellness documents  
all\_docs = [  
 Document(page\_content="Regular walking boosts heart health and can reduce symptoms of depression.", metadata={"source": "H1"}),  
 Document(page\_content="Consuming leafy greens and fruits helps detox the body and improve longevity.", metadata={"source": "H2"}),  
 Document(page\_content="Deep sleep is crucial for cellular repair and emotional regulation.", metadata={"source": "H3"}),  
 Document(page\_content="Mindfulness and controlled breathing lower cortisol and improve mental clarity.", metadata={"source": "H4"}),  
 Document(page\_content="Drinking sufficient water throughout the day helps maintain metabolism and energy.", metadata={"source": "H5"}),  
 Document(page\_content="The solar energy system in modern homes helps balance electricity demand.", metadata={"source": "I1"}),  
 Document(page\_content="Python balances readability with power, making it a popular system design language.", metadata={"source": "I2"}),  
 Document(page\_content="Photosynthesis enables plants to produce energy by converting sunlight.", metadata={"source": "I3"}),  
 Document(page\_content="The 2022 FIFA World Cup was held in Qatar and drew global energy and excitement.", metadata={"source": "I4"}),  
 Document(page\_content="Black holes bend spacetime and store immense gravitational energy.", metadata={"source": "I5"}),  
]  
  
  
# Initialize OpenAI embeddings  
embedding\_model = OllamaEmbeddings(  
 model="llama2",  
)  
# Create FAISS vector store  
vectorstore = FAISS.from\_documents(documents=all\_docs, embedding=embedding\_model)  
  
# Create retrievers  
similarity\_retriever = vectorstore.as\_retriever(search\_type="similarity", search\_kwargs={"k": 5})  
  
  
model = ChatOllama(  
 model="mistral",  
 temperature=0  
)  
multiquery\_retriever = MultiQueryRetriever.from\_llm(  
 retriever=vectorstore.as\_retriever(search\_kwargs={"k": 5}),  
 llm=model  
)  
  
query = "What is langchain?"  
  
# Retrieve results  
similarity\_results = similarity\_retriever.invoke(query)  
multiquery\_results= multiquery\_retriever.invoke(query)  
  
for i, doc in enumerate(similarity\_results):  
 print(f"\n--- Result {i+1} ---")  
 print(doc.page\_content)  
  
print("\*"\*150)  
  
for i, doc in enumerate(multiquery\_results):  
 print(f"\n--- Result {i+1} ---")  
 print(doc.page\_content)

**3. 📦 Contextual Compression Retriever**

The contextual compression Retriever in LangChain is an advanced retriever that improves retrieval quality by compressing documents after retrieval — keeping only the relevant content based on the user’s query.

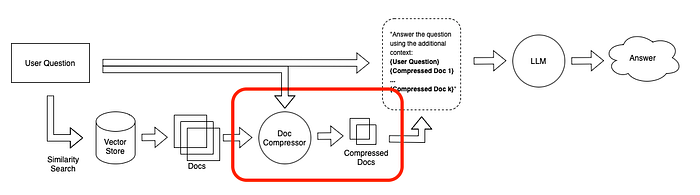
**How It Works**:

1. Retrieve N documents using a base retriever (e.g., FAISS)
2. Apply a **compressor** (typically an LLM) to each document
3. Retain only relevant sentences
4. Discard the rest

**When to Use**:

* Your documents are long and contain mixed information
* You want to reduce **LLM context length**
* You want more **accurate answers** in RAG pipelines

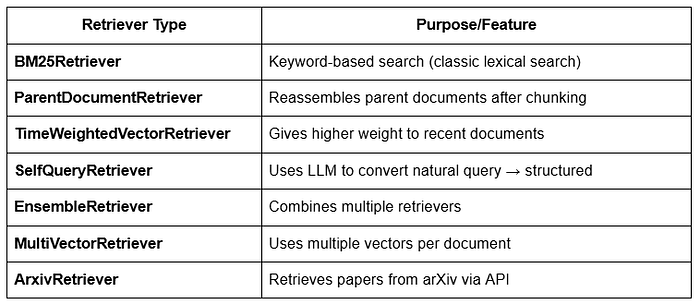
Press enter or click to view image in full size



**🧩 More Retriever Types in LangChain**

LangChain supports several other specialized retrievers for different use cases:

Press enter or click to view image in full size



**🧠 Types of Retrievers (Based on Data Source)**

.

Below are the key types:

📚 1. **Wikipedia Retriever**

A **Wikipedia Retriever** is a retriever that queries the **Wikipedia API** to fetch relevant content for a given query.

**How It Works:**

* You provide a query (e.g., *“Albert Einstein”*)
* It sends this query to Wikipedia’s API
* The API returns the **most relevant articles**
* These articles are returned as LangChain Document objects

**Best Use Cases:**  
✅ General-purpose Q&A  
✅ Research assistants  
✅ Projects with no private knowledge base

**Code Example:**

from langchain\_community.retrievers import WikipediaRetriever  
  
# Initialize the retriever (optional: set language and top\_k)  
retriever = WikipediaRetriever(top\_k\_results=2, lang="en")  
  
  
# Define your query  
query = "the geopolitical history of india and pakistan from the perspective of a chinese"  
  
# Get relevant Wikipedia documents  
docs = retriever.invoke(query)  
  
# Print retrieved content  
for i, doc in enumerate(docs):  
 print(f"\n--- Result {i+1} ---")  
 print(f"Content:\n{doc.page\_content}...") # truncate for display

Output :

📦2. **ArxivRetriever**

Retrieves scientific papers and abstracts from **arXiv** (research database for academic papers in CS, physics, etc.)

**How it works:**

* Uses arXiv’s API to find top-matching papers
* Returns abstracts and metadata for LLMs to work with.

**Best Use Cases:**  
✅ Research assistant  
✅ Academic paper summarization  
✅ AI/NLP RAG bots

**Code Example:**

from langchain.retrievers import ArxivRetriever  
  
# Create the retriever  
retriever = ArxivRetriever(load\_max\_docs=3) # limits number of papers retrieved  
  
# Define a query  
query = "Transformer models for time series forecasting"  
  
# Perform the retrieval  
docs = retriever.get\_relevant\_documents(query)  
  
# Print the titles and abstracts  
for doc in docs:  
 print(f"Title: {doc.metadata.get('Title')}")  
 print(f"Summary: {doc.page\_content}\n")  
 print("=" \* 80)

Advanced:-

**Introduction**

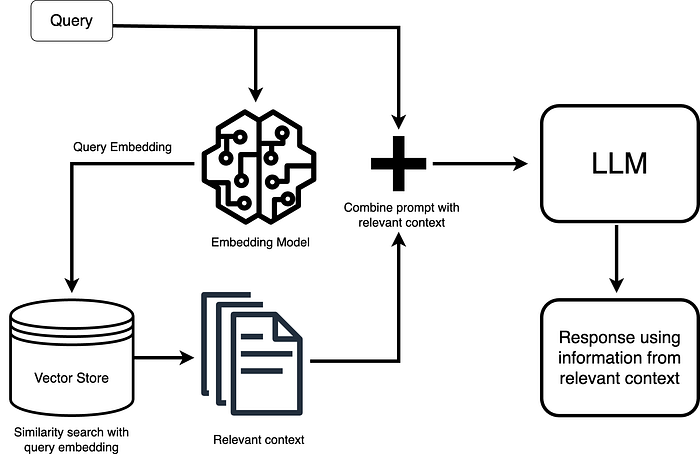
Retrieval-Augmented Generation (RAG) is a system which combines the strengths of information retrieval systems and large language models (LLMs) to enhance response accuracy. In a RAG system, relevant documents are first retrieved from a knowledge base and then used to generate responses. This hybrid approach ensures that the generated content is backed by factual information, enhancing the reliability of the outputs.

Despite its advantages, RAG faces significant challenges, particularly hallucinations where models produce plausible but incorrect information. These hallucinations stem from inefficiencies in traditional retrieval mechanisms, which often struggle with precision and relevance when managing large and diverse datasets. This can lead to broad and misaligned information retrieval, ultimately affecting the quality of the generated responses.

In this article we will see how can we apply advance retrieval techniques using langchain so that the context passed on to the LLM is relatable and accurate.

**Basic RAG Implementation using LangChain**

Press enter or click to view image in full size



Step1 : Installing required dependencies

pip install langchain langchain-openai langchain\_community langchain\_chroma

Step 2 : Importing necessary libraries

(note :- Here, I am using Chroma as a vector database but choose whichever database which suits you)

from langchain import hub  
from langchain\_chroma import Chroma  
from langchain\_community.document\_loaders import WebBaseLoader  
from langchain\_core.output\_parsers import StrOutputParser  
from langchain\_core.runnables import RunnablePassthrough  
from langchain\_openai import OpenAIEmbeddings  
from langchain\_text\_splitters import RecursiveCharacterTextSplitter

Step3 : Extracting data from a data source and storing the embeddings in our chromadb

3.1 Extracting the data

loader = WebBaseLoader(  
 web\_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/")  
)  
docs = loader.load()

3.2 Spliting the data into chunks and storing it into our vectorDB

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)  
splits = text\_splitter.split\_documents(docs)  
vectorstore = Chroma.from\_documents(documents=splits, embedding=OpenAIEmbeddings())

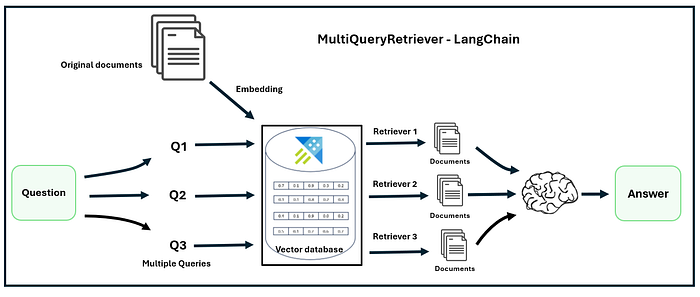
3.3 Retrieve data and generate answers to the questions using the relevant snippets of the knowledge base

retriever = vectorstore.as\_retriever()  
prompt = hub.pull("rlm/rag-prompt")  
  
  
def format\_docs(docs):  
 return "\n\n".join(doc.page\_content for doc in docs)  
  
  
rag\_chain = (  
 {"context": retriever | format\_docs, "question": RunnablePassthrough()}  
 | prompt  
 | llm  
 | StrOutputParser()  
)  
  
rag\_chain.invoke("What is Task Decomposition?")

Now Let’s dicuss some advance methods which make the process of retrieving data more efficient.

**1. MultiQuery Retriever**

Press enter or click to view image in full size



MultiQuery Retriever

MultiQueryRetriever involves LLM to generate multiple queries from different perspectives based on a given user input.

For each query, it retrieves a set of relevant documents and takes the unique union across all queries to get a larger set of potentially relevant documents. By generating multiple perspectives on the same question, the MultiQueryRetriever might be able to overcome some of the limitations of the distance-based retrieval and get a richer set of results.

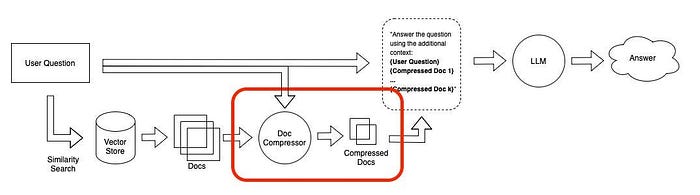
from langchain.retrievers.multi\_query import MultiQueryRetriever  
from langchain\_openai import ChatOpenAI  
  
question = "What are the approaches to Task Decomposition?"  
llm = ChatOpenAI(temperature=0)  
retriever\_from\_llm = MultiQueryRetriever.from\_llm(  
 retriever=vectordb.as\_retriever(), llm=llm  
)

unique\_docs = retriever\_from\_llm.invoke(question)

note :- Langchain internally uses an in-built prompt to generate multiple queries from the user query.

**2. Contextual compression**

Press enter or click to view image in full size



Contextual compression

Contextual compression in LangChain is a technique used to compress and filter documents based on their relevance to a given query. It aims to extract only the relevant information from documents, reducing the need for expensive language model calls and improving response quality.

Contextual compression is achieved by using a base retriever and a document compressor -

The base retriever retrieves the initial set of documents based on the query, and the document compressor processes these documents to extract the relevant content. You can use contextual compression when you have a document storage system and want to improve retrieval performance by returning only the most relevant information. It is particularly useful when the relevant information is buried within documents containing a lot of irrelevant text.

By using contextual compression, you can enhance the efficiency and effectiveness of your document retrieval process, resulting in better user experiences and optimized resource utilization.

from langchain.retrievers import ContextualCompressionRetriever  
from langchain.retrievers.document\_compressors import LLMChainExtractor  
from langchain\_openai import OpenAI  
  
llm = OpenAI(temperature=0)  
compressor = LLMChainExtractor.from\_llm(llm)  
compression\_retriever = ContextualCompressionRetriever(  
 base\_compressor=compressor, base\_retriever=retriever  
)  
  
compressed\_docs = compression\_retriever.invoke(  
 "What are the approaches to Task Decomposition?"  
)  
print(compressed\_docs)

**Adding Embeddings filter**

Embeddings filteris a effective compressor that uses an embedding similarity threshold to decide which of the initially retrieved documents to filter out and which ones to return, without manipulating the document contents.

from langchain.retrievers.document\_compressors import EmbeddingsFilter  
from langchain\_openai import OpenAIEmbeddings  
  
embeddings = OpenAIEmbeddings()  
embeddings\_filter = EmbeddingsFilter(embeddings=embeddings, similarity\_threshold=0.76)  
compression\_retriever = ContextualCompressionRetriever(  
 base\_compressor=embeddings\_filter, base\_retriever=retriever  
)  
  
compressed\_docs = compression\_retriever.invoke(  
 "What are the approaches to Task Decomposition?"  
)  
print(compressed\_docs)

**DocumentCompressorPipeline**

The DocumentCompressorPipeline is a feature in LangChain that allows you to combine multiple compressors and document transformers in sequence.

It helps in compressing and transforming documents in a contextual manner. The pipeline can include compressors like EmbeddingsRedundantFilter to remove redundant documents based on embedding similarity, and EmbeddingsFilter to filter documents based on their similarity to the query. Document transformers like TextSplitter can be used to split documents into smaller pieces.

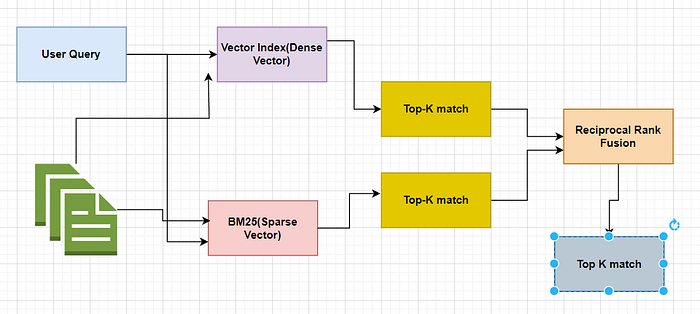
You may need to use the DocumentCompressorPipeline when you want to perform multiple compression and transformation steps on your documents.

from langchain.retrievers.document\_compressors import DocumentCompressorPipeline  
from langchain\_community.document\_transformers import EmbeddingsRedundantFilter  
  
redundant\_filter = EmbeddingsRedundantFilter(embeddings=embeddings)  
relevant\_filter = EmbeddingsFilter(embeddings=embeddings, similarity\_threshold=0.76)  
pipeline\_compressor = DocumentCompressorPipeline(  
 transformers=[splitter, redundant\_filter, relevant\_filter]  
)

compression\_retriever = ContextualCompressionRetriever(  
 base\_compressor=pipeline\_compressor, base\_retriever=retriever  
)  
  
compressed\_docs = compression\_retriever.invoke(  
 "What are the approaches to Task Decomposition?"  
)  
print(compressed\_docs)

**3. Ensemble Retriever**

Press enter or click to view image in full size



Ensemble Retriever

The EnsembleRetriever takes a list of retrievers as input and ensemble the results of their get\_relevant\_documents() methods and rerank the results based on the [Reciprocal Rank Fusion](https://plg.uwaterloo.ca/~gvcormac/cormacksigir09-rrf.pdf) algorithm.

By leveraging the strengths of different algorithms, the EnsembleRetriever can achieve better performance than any single algorithm.

The most common pattern is to combine a sparse retriever (like BM25) with a dense retriever (like embedding similarity), because their strengths are complementary. It is also known as “hybrid search”. The sparse retriever is good at finding relevant documents based on keywords, while the dense retriever is good at finding relevant documents based on semantic similarity.

The EnsembleRetriever in LangChain is a retrieval algorithm that combines the results of multiple retrievers and reranks them using the Reciprocal Rank Fusion algorithm.

It is used to improve the performance of retrieval by leveraging the strengths of different algorithms. You may need to use the EnsembleRetriever when you want to achieve better retrieval performance than any single algorithm can provide. It is particularly useful when combining a sparse retriever (e.g., BM25) with a dense retriever (e.g., embedding similarity) because their strengths are complementary.

The sparse retriever is good at finding relevant documents based on keywords, while the dense retriever is good at finding relevant documents based on semantic similarity.

To use the EnsembleRetriever, you need to initialize it with a list of retrievers and their corresponding weights. The retrievers can be instances of different retrieval algorithms, such as BM25Retriever and ChromaRetriever. The weights determine the importance of each retriever in the ensemble. The EnsembleRetriever then combines the results of the retrievers and reranks them based on the Reciprocal Rank Fusion algorithm.

If you have multiple retrievers that perform well on different aspects of the task, combining them using the EnsembleRetriever can lead to improved performance.

Additionally, if you have a combination of sparse and dense retrievers, the FusionRetriever can help leverage their complementary strengths.

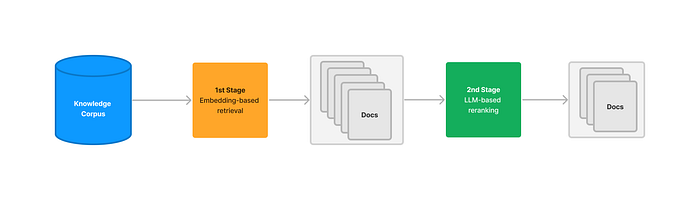
!pip install rank\_bm25

from langchain.retrievers import EnsembleRetriever  
from langchain\_community.retrievers import BM25Retriever  
from langchain\_community.vectorstores import Chroma  
from langchain\_openai import OpenAIEmbeddings

doc\_list\_1 = [  
 "I like apples",  
 "I like oranges",  
 "Apples and oranges are fruits",  
]  
  
# initialize the bm25 retriever and faiss retriever  
bm25\_retriever = BM25Retriever.from\_texts(  
 doc\_list\_1, metadatas=[{"source": 1}] \* len(doc\_list\_1)  
)  
bm25\_retriever.k = 2  
  
doc\_list\_2 = [  
 "You like apples",  
 "You like oranges",  
]  
  
embedding = OpenAIEmbeddings()  
chroma\_vectorstore = Chroma.from\_texts(documents=splits, embedding=OpenAIEmbeddings(),metadatas=[{"source": 2}] \* len(doc\_list\_2))  
  
chroma\_retriever = chroma\_vectorstore.as\_retriever(search\_kwargs={"k": 2})  
  
# initialize the ensemble retriever  
ensemble\_retriever = EnsembleRetriever(  
 retrievers=[bm25\_retriever, chroma\_retriever], weights=[0.5, 0.5]  
)  
  
docs = ensemble\_retriever.invoke("apples")

**4. Long Context Re-Order**

Press enter or click to view image in full size



Re-ranking

[A study](https://arxiv.org/abs/2307.03172) observed that the best performance typically arises when crucial data is positioned at the start or conclusion of the input context. Additionally, as the input context lengthens, performance drops notably, even in models designed for long contexts. This is because LLMs tend to ignore the data present in the middle portion of your prompt and This problem is called as “Lost in the middle”

Hence When you have large number of retrieved documents and try to fit it into your prompt all at once, there is a substantial performance degradation. To solve this issue what we do is we keep the highest-score (most relevant) context at the begining, with lower-scored documents in the middle.

from langchain\_chroma import Chroma  
from langchain\_community.document\_transformers import (  
 LongContextReorder,  
)  
from langchain\_community.embeddings import HuggingFaceEmbeddings  
from langchain\_openai import OpenAI  
  
# Get embeddings.  
embeddings = HuggingFaceEmbeddings(model\_name="all-MiniLM-L6-v2")  
  
texts = [  
 "Basquetball is a great sport.",  
 "Fly me to the moon is one of my favourite songs.",  
 "The Celtics are my favourite team.",  
 "This is a document about the Boston Celtics",  
 "I simply love going to the movies",  
 "The Boston Celtics won the game by 20 points",  
 "This is just a random text.",  
 "Elden Ring is one of the best games in the last 15 years.",  
 "L. Kornet is one of the best Celtics players.",  
 "Larry Bird was an iconic NBA player.",  
]  
  
# Create a retriever  
retriever = Chroma.from\_texts(texts, embedding=embeddings).as\_retriever(  
 search\_kwargs={"k": 10}  
)  
query = "What can you tell me about the Celtics?"  
  
# Get relevant documents ordered by relevance score  
docs = retriever.invoke(query)

reordering = LongContextReorder()  
reordered\_docs = reordering.transform\_documents(docs)

**5. LOTR Retriever**

Lord of the Retrievers (LOTR), also known as MergerRetriever, takes a list of retrievers as input and merges the results of their get\_relevant\_documents() methods into a single list. it then de-duplicate the results and then rank the results of the different retrievers, which can help to ensure that the most relevant documents are returned first.

import os  
import chromadb  
from langchain.retrievers import (  
 ContextualCompressionRetriever,  
 DocumentCompressorPipeline,  
 MergerRetriever,  
)  
from langchain\_chroma import Chroma  
from langchain\_community.document\_transformers import (  
 EmbeddingsClusteringFilter,  
 EmbeddingsRedundantFilter,  
)  
from langchain\_huggingface import HuggingFaceEmbeddings  
from langchain\_openai import OpenAIEmbeddings

Let’s create a Merge Retriever which have two retrievers combined together

# Get 3 diff embeddings.  
all\_mini = HuggingFaceEmbeddings(model\_name="all-MiniLM-L6-v2")  
multi\_qa\_mini = HuggingFaceEmbeddings(model\_name="multi-qa-MiniLM-L6-dot-v1")  
filter\_embeddings = OpenAIEmbeddings()  
  
ABS\_PATH = os.path.dirname(os.path.abspath(\_\_file\_\_))  
DB\_DIR = os.path.join(ABS\_PATH, "db")  
  
# Instantiate 2 diff chromadb indexes, each one with a diff embedding.  
client\_settings = chromadb.config.Settings(  
 is\_persistent=True,  
 persist\_directory=DB\_DIR,  
 anonymized\_telemetry=False,  
)  
db\_all = Chroma(  
 collection\_name="project\_store\_all",  
 persist\_directory=DB\_DIR,  
 client\_settings=client\_settings,  
 embedding\_function=all\_mini,  
)  
db\_multi\_qa = Chroma(  
 collection\_name="project\_store\_multi",  
 persist\_directory=DB\_DIR,  
 client\_settings=client\_settings,  
 embedding\_function=multi\_qa\_mini,  
)  
  
# Define 2 diff retrievers with 2 diff embeddings and diff search type.  
retriever\_all = db\_all.as\_retriever(  
 search\_type="similarity", search\_kwargs={"k": 5, "include\_metadata": True}  
)  
retriever\_multi\_qa = db\_multi\_qa.as\_retriever(  
 search\_type="mmr", search\_kwargs={"k": 5, "include\_metadata": True}  
)  
  
# The Lord of the Retrievers will hold the output of both retrievers and can be used as any other  
# retriever on different types of chains.  
lotr = MergerRetriever(retrievers=[retriever\_all, retriever\_multi\_qa])

We can remove redundant results from both retrievers using yet another embedding. Using multiples embeddings in diff steps could help reduce biases.

filter = EmbeddingsRedundantFilter(embeddings=filter\_embeddings)  
reordering = LongContextReorder()  
pipeline = DocumentCompressorPipeline(transformers=[filter,reordering])  
compression\_lotr\_retriever = ContextualCompressionRetriever(  
 base\_compressor=pipeline, base\_retriever=lotr  
)

**Conclusion**

In this blog we explored various complex retrieval techniques which can improve the output of your RAG system drastically. With these advance techniques,we can effectively reduce the hallucination in LLM responses and ensure more precise, contextually relevant answers.

**4. Self-Query Retrievers: Querying Metadata and Content**

A self-querying retriever is like having an AI assistant that can refine your questions before searching for answers. Instead of just relying on keyword matching, it can understand the underlying structure of your query and use metadata to filter information more accurately. Think of it as adding precision to your search.

* **Index Type:** Vectorstore
* **Uses an LLM:** Yes
* **Use Cases**: Ideal when your queries require filtering by metadata (like author, date, or category) rather than just relying on semantic content matching.

If a user’s question relates more to document metadata than the content, Self Query is the way to go.

Press enter or click to view image in full size

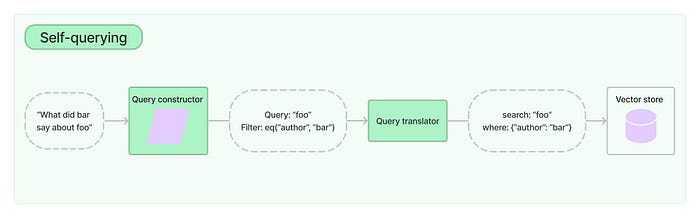


Photo from web: <https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/self_query/>

# Example - Self Query Retrievers  
  
# Load and split documents  
loaders = [  
 TextLoader("../../Tortoise and the Hare.txt"),  
 TextLoader("../../ลูกหมูสามตัว.txt"),  
]  
docs = []  
for loader in loaders:  
 docs.extend(loader.load())  
  
# Create documents with metadata  
docs\_with\_metadata = [  
 Document(  
 page\_content=docs[0].page\_content,  
 metadata={"title": "ลูกหมูสามตัว", "genre": "children", "year": 1990, "language": "Thai"}  
 ),  
 Document(  
 page\_content=docs[1].page\_content,  
 metadata={"title": "Tortoise and the Hare", "genre": "children", "year": 1995, "language": "English"}  
 )  
]  
  
# Define metadata fields  
metadata\_field\_info = [  
 AttributeInfo(  
 name="title",  
 description="The title of the story",  
 type="string",  
 ),  
 AttributeInfo(  
 name="genre",  
 description="The genre of the story, usually 'children'",  
 type="string",  
 ),  
 AttributeInfo(  
 name="year",  
 description="The year the story was published",  
 type="integer",  
 ),  
 AttributeInfo(  
 name="language",  
 description="The language of the story",  
 type="string",  
 ),  
]  
  
# Create VectorStore  
vectorstore = Chroma.from\_documents(docs\_with\_metadata, OpenAIEmbeddings(api\_key=OPENAI\_API\_KEY))  
  
# Initialize SelfQueryRetriever  
document\_content\_description = "A brief summary of the story"  
llm = ChatOpenAI(temperature=0, api\_key=OPENAI\_API\_KEY)  
  
retriever = SelfQueryRetriever.from\_llm(  
 llm,  
 vectorstore,  
 document\_content\_description,  
 metadata\_field\_info,  
 # enable\_limit=True, # Filter K (optional)  
  
)  
  
result = retriever.invoke("I want a children story in Thai")  
  
# Example of output of result  
"""  
Story 1:  
ลูกหมูสามตัว  
กาลครั้งหนึ่งนานมาแล้ว มีแม่หมูอยู่กับลูกหมูสามตัว ลูกหมูทั้งสามโตหมดแล้ว ถึงเวลาที่พวกเขาต้องสร้างบ้านของตัวเองและออกไปผจญภัยในโลกกว้าง  
ลูกหมูโต ชื่อ ข้าวสวย- พี่ชายคนโต เป็นหมูที่ค่อนข้างขี้เกียจ เขาตัดสินใจสร้างบ้านอย่างรวดเร็ว เจอกองฟางข้าวใกล้ๆ นึกในใจว่า "แค่นี้ก็พอแล้ว!" จากนั้นเขาก็รีบๆ สร้างบ้านฟางฟางที่กางออกง่ายๆ  
ลูกหมูรอง ชื่อ ข้าวหอม- น้องชายคนที่สอง ไม่ได้ขี้เกียจเท่าพี่ชาย เขาตัดสินใจสร้างบ้านที่แข็งแรงกว่า เขาเก็บกิ่งไม้และก้านไม้จากป่ามาสร้างบ้านไม้ มันไม่ใช่บ้านที่แข็งแรงที่สุด แต่เขาคิดว่ามันก็น่าจะเพียงพอ  
ลูกหมูเล็กชื่อ ข้าวสุก - น้องชายคนสุดท้อง ขึ้นชื่อเรื่องการทำงานหนักและวางแผน เขาต้องการบ้านที่จะทำให้เขามีความปลอดภัยอย่างแท้จริง เขาใช้เวลาหลายวันในการเก็บอิฐที่แข็งแรงและสร้างบ้านอิฐที่แข็งแรงอย่างระมัดระวัง เขายังสร้างปล่องไฟและประตูที่เหมาะสมพร้อมกับกุญแจที่แข็งแรง  
วันแดดสดใสวันหนึ่ง มีหมาป่าตัวใหญ่ใจร้ายมีนามว่า แกงส้ม เดินเลาะเลียบไปตามทาง มันได้กลิ่นหอมกรุ่นของหมูย่างและเดินตามกลิ่นนั้นไปจนถึงบ้านฟางของลูกหมูตัวแรก  
"ลูกหมู ๆ เปิดประตูให้ข้าเข้าไปสิ!" หมาป่าคำราม  
"ไม่! ข้าจะไม่ให้เจ้าเข้ามา!" ลูกหมูตัวเล็กตกใจร้องเสียงแหลม  
หมาป่าสูดลมพ่นออกแรง ๆ เป่าบ้านฟางจนพังทลาย! ลูกหมูตัวเล็กกรี๊ดร้องและวิ่งหนีเร็วที่สุดเท่าที่จะวิ่ง  
เขาวิ่งไปที่บ้านไม้ของพี่ชาย หมาป่าวิ่งตามมาติดๆ และเมื่อมาถึงบ้านไม้ มันก็ร้องขอ "ลูกหมู ๆ เปิดประตูให้ข้าเข้าไปสิ!"  
"ไม่! พวกเราจะไม่ให้เจ้าเข้ามา!" ลูกหมูทั้งสองตัวร้องออกมาพร้อมกัน  
หมาป่าสูดลมพ่นออกแรง ๆ เป่าบ้านไม้จนพังทลาย! ลูกหมูทั้งสองตัวกรี๊ดร้องและวิ่งหนีเร็วที่สุดเท่าที่จะวิ่ง ไปที่บ้านอิฐของน้องชายคนสุดท้อง  
หมาป่าตอนนี้หิวโซกว่าเดิม วิ่งตามพวกมันไป เมื่อมาถึงบ้านอิฐ มันตะโกนว่า "ลูกหมู ๆ เปิดประตูให้ข้าเข้าไปสิ!"  
"ไม่! พวกเราจะไม่ให้เจ้าเข้ามา!" ลูกหมูทั้งสามตัวตะโกนออกมาพร้อมกัน ตอนนี้พวกเขารู้สึกกล้าหาญมากขึ้นเพราะพวกเขาปลอดภัยอยู่ข้างใน  
หมาป่าสูดลมพ่นออกแรง ๆ เป่าบ้านอิฐจนสุดแรงเกิด มันเป่าแรงจนหน้าแดง แต่บ้านอิฐก็ไม่ขยับเขยื้อน หมาป่าหงุดหงิดและพ่ายแพ้ ยอมแพ้และเดินโซซัดเซไปในป่า  
ลูกหมูทั้งสามตัว ปลอดภัยอยู่ในบ้านอิฐ พวกเขาได้เรียนบทเรียนอันมีค่าในวันนั้น ความขยันหมั่นเพียรและการวางแผนนั้นสำคัญ และรากฐานที่แข็งแกร่งสามารถช่วยให้คุณปลอดภัยจากอันตราย พวกเขาอยู่ด้วยกันอย่างมีความสุขตลอดไปในบ้านอิฐที่แข็งแรง ไม่เคยลืมความสำคัญของการเตรียมตัว  
"""

**5. Contextual Compression: Focused and Relevant Retrieval**

Imagine you’re searching through a huge book for a specific quote. You wouldn’t want to read the entire book, just the relevant parts, right? That’s what contextual compression does for AI. It narrows down the information returned by retrievers to the most relevant bits, leading to faster and more focused responses.

* **Index Type:** Flexible, can be used with various types of indices
* **Uses an LLM:** Optional
* **Use Cases**: When retrieved documents contain too much irrelevant information, potentially distracting LLMs.

The idea is simple: instead of immediately returning retrieved documents as-is, you can compress them using the context of the given query, so that only the relevant information is returned. “Compressing” here refers to both compressing the contents of an individual document and filtering out documents completely.

**5.1) Vanilla Vector Store Retriever**

First, we will show what the return data from the normal retriever look like.

# Example - Parent Document Retriever - Vanilla Vector Store Retriever  
  
# Load and split documents  
loaders = [  
 TextLoader("../../Tortoise and the Hare.txt"),  
 TextLoader("../../ลูกหมูสามตัว.txt"),  
]  
documents = []  
for loader in loaders:  
 documents.extend(loader.load())  
  
# Helper function for printing docs  
def pretty\_print\_docs(docs):  
 print(  
 f"\n{'-' \* 100}\n".join(  
 [f"Document {i+1}:\n\n" + d.page\_content for i, d in enumerate(docs)]  
 )  
 )  
  
# Split documents into smaller chunks  
text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)  
texts = text\_splitter.split\_documents(documents)  
  
# Create retriever  
retriever = FAISS.from\_documents(texts, OpenAIEmbeddings(api\_key=OPENAI\_API\_KEY)).as\_retriever()  
  
# Get documents from query  
def pretty\_print\_docs(docs):  
 print(  
 f"\n{'-' \* 100}\n".join(  
 [f"Document {i+1}:\n\n" + d.page\_content for i, d in enumerate(docs)]  
 )  
 )  
  
docs = retriever.invoke("What did the tortoise do in the race")  
pretty\_print\_docs(docs)  
  
# Example of output of result  
"""  
Document 1:  
Tortoise and the Hare  
Deep within a sun-dappled clearing of the DC Forest, lived a hare named Piti, famed for his lightning speed. He would streak past the other animals, a blur of brown fur that left them breathless in his wake. One crisp morning, Piti was bragging about his agility, puffing out his chest and flicking his tail with unconcealed pride.  
"There's no creature in this entire forest faster than me!" he declared, his voice echoing through the trees.  
A slow, rumbling voice came from behind a nearby thicket. It was Donato, a tortoise known for his steady pace and unwavering determination.  
"Speed isn't everything, Piti," Donato rumbled. "Even the slowest can achieve victory, if they set their mind to it."  
Piti burst into laughter. "You? Win a race against me? Donato, that's the most ludicrous notion I've ever heard!"  
----------------------------------------------------------------------------------------------------  
Document 2:  
He drifted off to sleep, picturing himself crossing the finish line first to a chorus of cheers. But time crawled by for the sleeping hare, while for Donato, it marched on relentlessly.  
Donato, inch by inch, made his way towards the finish line. The animals, who had initially mocked him, now cheered him on, their voices echoing through the forest. They were impressed by his unwavering perseverance.  
Finally, Donato, with a triumphant plod, crossed the finish line. Piti woke up with a start, his ears twitching in disbelief. He saw, to his utter humiliation, the crowd celebrating Donato's victory.  
The hare had lost the race, not to speed, but to slow and steady determination. The cheers of the animals resonated through the DC Forest, a testament to the fact that slow and steady truly does win the race.  
----------------------------------------------------------------------------------------------------  
Document 3:  
To everyone's surprise, Donato challenged Piti to a race. The other animals gathered around, buzzing with excitement at the prospect of such an unequal competition. Even the wise old owl hooted in amusement, his amber eyes twinkling with anticipation.  
The race began. Piti shot off like a furry bullet, leaving Donato in a cloud of dust. The animals cheered for the hare, certain of his victory. But Piti, brimming with overconfidence, spotted a patch of wildflowers bursting with color. He darted off the track, unable to resist the temptation of a tasty treat.  
Meanwhile, Donato plodded on steadily, never stopping, never wavering. He may have been slow, but his determination burned bright.  
Back on the track, Piti, feeling sluggish from his snack, decided to take a nap under the shade of a towering oak. "Old Donato won't catch up to me anyway," he thought arrogantly.  
----------------------------------------------------------------------------------------------------  
Document 4:  
"ลูกหมู ๆ เปิดประตูให้ข้าเข้าไปสิ!" หมาป่าคำราม  
"ไม่! ข้าจะไม่ให้เจ้าเข้ามา!" ลูกหมูตัวเล็กตกใจร้องเสียงแหลม  
หมาป่าสูดลมพ่นออกแรง ๆ เป่าบ้านฟางจนพังทลาย! ลูกหมูตัวเล็กกรี๊ดร้องและวิ่งหนีเร็วที่สุดเท่าที่จะวิ่ง  
เขาวิ่งไปที่บ้านไม้ของพี่ชาย หมาป่าวิ่งตามมาติดๆ และเมื่อมาถึงบ้านไม้ มันก็ร้องขอ "ลูกหมู ๆ เปิดประตูให้ข้าเข้าไปสิ!"  
"ไม่! พวกเราจะไม่ให้เจ้าเข้ามา!" ลูกหมูทั้งสองตัวร้องออกมาพร้อมกัน  
หมาป่าสูดลมพ่นออกแรง ๆ เป่าบ้านไม้จนพังทลาย! ลูกหมูทั้งสองตัวกรี๊ดร้องและวิ่งหนีเร็วที่สุดเท่าที่จะวิ่ง ไปที่บ้านอิฐของน้องชายคนสุดท้อง  
หมาป่าตอนนี้หิวโซกว่าเดิม วิ่งตามพวกมันไป เมื่อมาถึงบ้านอิฐ มันตะโกนว่า "ลูกหมู ๆ เปิดประตูให้ข้าเข้าไปสิ!"  
"ไม่! พวกเราจะไม่ให้เจ้าเข้ามา!" ลูกหมูทั้งสามตัวตะโกนออกมาพร้อมกัน ตอนนี้พวกเขารู้สึกกล้าหาญมากขึ้นเพราะพวกเขาปลอดภัยอยู่ข้างใน  
หมาป่าสูดลมพ่นออกแรง ๆ เป่าบ้านอิฐจนสุดแรงเกิด มันเป่าแรงจนหน้าแดง แต่บ้านอิฐก็ไม่ขยับเขยื้อน หมาป่าหงุดหงิดและพ่ายแพ้ ยอมแพ้และเดินโซซัดเซไปในป่า  
"""

**5.2) Contextual Compression with an LLMChainExtractor**

This method uses an LLM chain to extract only the content from each retrieved document that is most relevant to the query, filtering out the rest.

# Example - Parent Document Retriever - Contextual Compression Retriever  
  
# Load and split documents  
loaders = [  
 TextLoader("../../Tortoise and the Hare.txt"),  
 TextLoader("../../ลูกหมูสามตัว.txt"),  
]  
documents = []  
for loader in loaders:  
 documents.extend(loader.load())  
  
# Split documents  
text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)  
texts = text\_splitter.split\_documents(documents)  
  
# Create retriever  
retriever = FAISS.from\_documents(texts, OpenAIEmbeddings(api\_key=OPENAI\_API\_KEY)).as\_retriever()  
  
# Create Contextual Compression  
llm = OpenAI(temperature=0, api\_key=OPENAI\_API\_KEY)  
compressor = LLMChainExtractor.from\_llm(llm)  
  
# Initialize ContextualCompressionRetriever  
compression\_retriever = ContextualCompressionRetriever(  
 base\_compressor=compressor, base\_retriever=retriever  
)  
  
# Set the question  
question = "What did the tortoise do in the race"  
compressed\_docs = compression\_retriever.invoke(question)  
pretty\_print\_docs(compressed\_docs)  
  
# Example of output of result  
"""  
Document 1:  
Tortoise and the Hare  
Donato, a tortoise known for his steady pace and unwavering determination.  
"Speed isn't everything, Piti," Donato rumbled. "Even the slowest can achieve victory, if they set their mind to it."  
----------------------------------------------------------------------------------------------------  
Document 2:  
Donato, inch by inch, made his way towards the finish line. The animals, who had initially mocked him, now cheered him on, their voices echoing through the forest. They were impressed by his unwavering perseverance.  
Finally, Donato, with a triumphant plod, crossed the finish line.  
----------------------------------------------------------------------------------------------------  
Document 3:  
- Donato challenged Piti to a race.  
- The race began.  
- Piti shot off like a furry bullet, leaving Donato in a cloud of dust.  
- Piti, brimming with overconfidence, spotted a patch of wildflowers bursting with color.  
- He darted off the track, unable to resist the temptation of a tasty treat.  
- Meanwhile, Donato plodded on steadily, never stopping, never wavering.  
- Piti, feeling sluggish from his snack, decided to take a nap under the shade of a towering oak.  
"""

More built-in compressors: filters[​](https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/contextual_compression/#more-built-in-compressors-filters)

5.3) **Contextual Compression with an LLMChainFilter**[**​**](https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/contextual_compression/#llmchainfilter)

This compressor is a simpler approach, using an LLM to decide which of the initially returned documents to filter out, without altering the document contents.

# Example - Parent Document Retriever - LLMChainFilter  
  
# Load and split documents  
loaders = [  
 TextLoader("../../Tortoise and the Hare.txt"),  
 TextLoader("../../ลูกหมูสามตัว.txt"),  
]  
documents = []  
for loader in loaders:  
 documents.extend(loader.load())  
  
# Split documents  
text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)  
texts = text\_splitter.split\_documents(documents)  
  
# Create retriever  
retriever = FAISS.from\_documents(texts, OpenAIEmbeddings(api\_key=OPENAI\_API\_KEY)).as\_retriever()  
  
# Create LLMChainFilter  
llm = OpenAI(temperature=0, api\_key=OPENAI\_API\_KEY)  
\_filter = LLMChainFilter.from\_llm(llm)  
  
# Initialize ContextualCompressionRetriever  
compression\_retriever = ContextualCompressionRetriever(  
 base\_compressor=\_filter, base\_retriever=retriever  
)  
  
# Set the question  
question = "What did the tortoise do in the race"  
compressed\_docs = compression\_retriever.invoke(question)  
pretty\_print\_docs(compressed\_docs)  
  
# Example of output of result  
"""  
Document 1:  
Tortoise and the Hare  
Deep within a sun-dappled clearing of the DC Forest, lived a hare named Piti, famed for his lightning speed. He would streak past the other animals, a blur of brown fur that left them breathless in his wake. One crisp morning, Piti was bragging about his agility, puffing out his chest and flicking his tail with unconcealed pride.  
"There's no creature in this entire forest faster than me!" he declared, his voice echoing through the trees.  
A slow, rumbling voice came from behind a nearby thicket. It was Donato, a tortoise known for his steady pace and unwavering determination.  
"Speed isn't everything, Piti," Donato rumbled. "Even the slowest can achieve victory, if they set their mind to it."  
Piti burst into laughter. "You? Win a race against me? Donato, that's the most ludicrous notion I've ever heard!"  
----------------------------------------------------------------------------------------------------  
Document 2:  
He drifted off to sleep, picturing himself crossing the finish line first to a chorus of cheers. But time crawled by for the sleeping hare, while for Donato, it marched on relentlessly.  
Donato, inch by inch, made his way towards the finish line. The animals, who had initially mocked him, now cheered him on, their voices echoing through the forest. They were impressed by his unwavering perseverance.  
Finally, Donato, with a triumphant plod, crossed the finish line. Piti woke up with a start, his ears twitching in disbelief. He saw, to his utter humiliation, the crowd celebrating Donato's victory.  
The hare had lost the race, not to speed, but to slow and steady determination. The cheers of the animals resonated through the DC Forest, a testament to the fact that slow and steady truly does win the race.  
----------------------------------------------------------------------------------------------------  
Document 3:  
To everyone's surprise, Donato challenged Piti to a race. The other animals gathered around, buzzing with excitement at the prospect of such an unequal competition. Even the wise old owl hooted in amusement, his amber eyes twinkling with anticipation.  
The race began. Piti shot off like a furry bullet, leaving Donato in a cloud of dust. The animals cheered for the hare, certain of his victory. But Piti, brimming with overconfidence, spotted a patch of wildflowers bursting with color. He darted off the track, unable to resist the temptation of a tasty treat.  
Meanwhile, Donato plodded on steadily, never stopping, never wavering. He may have been slow, but his determination burned bright.  
Back on the track, Piti, feeling sluggish from his snack, decided to take a nap under the shade of a towering oak. "Old Donato won't catch up to me anyway," he thought arrogantly.  
"""

5.4) **Contextual Compression with an EmbeddingsFilter**[**​**](https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/contextual_compression/#embeddingsfilter)

This method provides a cost-effective alternative by using embeddings to filter out documents that are not semantically similar to the query.

# Example - Parent Document Retriever - EmbeddingsFilter  
  
# Load and split documents  
loaders = [  
 TextLoader("../../Tortoise and the Hare.txt"),  
 TextLoader("../../ลูกหมูสามตัว.txt"),  
]  
documents = []  
for loader in loaders:  
 documents.extend(loader.load())  
  
# Split documents  
text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)  
texts = text\_splitter.split\_documents(documents)  
  
# Create retriever  
embeddings = OpenAIEmbeddings(api\_key=OPENAI\_API\_KEY)  
retriever = FAISS.from\_documents(texts, embeddings).as\_retriever()  
  
# Create EmbeddingsFilter by using OpenAIEmbeddings  
embeddings\_filter = EmbeddingsFilter(embeddings=embeddings, similarity\_threshold=0.76)  
  
# Initialize ContextualCompressionRetriever  
compression\_retriever = ContextualCompressionRetriever(  
 base\_compressor=embeddings\_filter, base\_retriever=retriever  
)  
  
# Set the question  
question = "What did the tortoise do in the race?"  
compressed\_docs = compression\_retriever.invoke(question)  
pretty\_print\_docs(compressed\_docs)  
  
# Example of output of result  
"""  
Document 1:  
Tortoise and the Hare  
Deep within a sun-dappled clearing of the DC Forest, lived a hare named Piti, famed for his lightning speed. He would streak past the other animals, a blur of brown fur that left them breathless in his wake. One crisp morning, Piti was bragging about his agility, puffing out his chest and flicking his tail with unconcealed pride.  
"There's no creature in this entire forest faster than me!" he declared, his voice echoing through the trees.  
A slow, rumbling voice came from behind a nearby thicket. It was Donato, a tortoise known for his steady pace and unwavering determination.  
"Speed isn't everything, Piti," Donato rumbled. "Even the slowest can achieve victory, if they set their mind to it."  
Piti burst into laughter. "You? Win a race against me? Donato, that's the most ludicrous notion I've ever heard!"  
----------------------------------------------------------------------------------------------------  
Document 2:  
He drifted off to sleep, picturing himself crossing the finish line first to a chorus of cheers. But time crawled by for the sleeping hare, while for Donato, it marched on relentlessly.  
Donato, inch by inch, made his way towards the finish line. The animals, who had initially mocked him, now cheered him on, their voices echoing through the forest. They were impressed by his unwavering perseverance.  
Finally, Donato, with a triumphant plod, crossed the finish line. Piti woke up with a start, his ears twitching in disbelief. He saw, to his utter humiliation, the crowd celebrating Donato's victory.  
The hare had lost the race, not to speed, but to slow and steady determination. The cheers of the animals resonated through the DC Forest, a testament to the fact that slow and steady truly does win the race.  
----------------------------------------------------------------------------------------------------  
Document 3:  
To everyone's surprise, Donato challenged Piti to a race. The other animals gathered around, buzzing with excitement at the prospect of such an unequal competition. Even the wise old owl hooted in amusement, his amber eyes twinkling with anticipation.  
The race began. Piti shot off like a furry bullet, leaving Donato in a cloud of dust. The animals cheered for the hare, certain of his victory. But Piti, brimming with overconfidence, spotted a patch of wildflowers bursting with color. He darted off the track, unable to resist the temptation of a tasty treat.  
Meanwhile, Donato plodded on steadily, never stopping, never wavering. He may have been slow, but his determination burned bright.  
Back on the track, Piti, feeling sluggish from his snack, decided to take a nap under the shade of a towering oak. "Old Donato won't catch up to me anyway," he thought arrogantly.  
"""

**5.5) Stringing compressors and document transformers together**[**​**](https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/contextual_compression/#stringing-compressors-and-document-transformers-together)

Using the DocumentCompressorPipeline, we can combine multiple compressors and document transformers in sequence.

# Example - Parent Document Retriever - EmbeddingsFilter  
  
# Load and split documents  
loaders = [  
 TextLoader("../../Tortoise and the Hare.txt"),  
 TextLoader("../../ลูกหมูสามตัว.txt"),  
]  
documents = []  
for loader in loaders:  
 documents.extend(loader.load())  
  
# Split documents  
text\_splitter = CharacterTextSplitter(chunk\_size=300, chunk\_overlap=0, separator=". ")  
texts = text\_splitter.split\_documents(documents)  
  
# Create retriever  
embeddings = OpenAIEmbeddings(api\_key=OPENAI\_API\_KEY)  
retriever = FAISS.from\_documents(texts, embeddings).as\_retriever()  
  
# Create EmbeddingsRedundantFilter (delete duplicate document)  
redundant\_filter = EmbeddingsRedundantFilter(embeddings=embeddings)  
relevant\_filter = EmbeddingsFilter(embeddings=embeddings, similarity\_threshold=0.76)  
  
# Create DocumentCompressorPipeline  
pipeline\_compressor = DocumentCompressorPipeline(  
 transformers=[text\_splitter, redundant\_filter, relevant\_filter]  
)  
  
# Initialize ContextualCompressionRetriever  
compression\_retriever = ContextualCompressionRetriever(  
 base\_compressor=pipeline\_compressor, base\_retriever=retriever  
)  
  
# Set the question  
question = "What did the tortoise do in the race?"  
compressed\_docs = compression\_retriever.invoke(question)  
pretty\_print\_docs(compressed\_docs)  
  
# Example of output of result  
"""  
Document 1:  
Tortoise and the Hare  
Deep within a sun-dappled clearing of the DC Forest, lived a hare named Piti, famed for his lightning speed. He would streak past the other animals, a blur of brown fur that left them breathless in his wake  
----------------------------------------------------------------------------------------------------  
Document 2:  
It was Donato, a tortoise known for his steady pace and unwavering determination.  
"Speed isn't everything, Piti," Donato rumbled. "Even the slowest can achieve victory, if they set their mind to it."  
Piti burst into laughter  
----------------------------------------------------------------------------------------------------  
Document 3:  
Even the wise old owl hooted in amusement, his amber eyes twinkling with anticipation.  
The race began. Piti shot off like a furry bullet, leaving Donato in a cloud of dust. The animals cheered for the hare, certain of his victory  
----------------------------------------------------------------------------------------------------  
Document 4:  
He saw, to his utter humiliation, the crowd celebrating Donato's victory.  
The hare had lost the race, not to speed, but to slow and steady determination. The cheers of the animals resonated through the DC Forest, a testament to the fact that slow and steady truly does win the race.  
"""

**6) Time-Weighted Vector Store Retriever: Prioritizing Recency**

Sometimes, recent information is more valuable than older data. The Time-Weighted Vector Store Retriever factors in the recency of the information when ranking results, ensuring that the most relevant and up-to-date documents are prioritized.

* **Index Type:** Vectorstore
* **Uses an LLM:** No
* **Use Cases**: When you need to retrieve the most recent and relevant information from a time-sensitive dataset.

The core formula is:  
semantic\_similarity + (1.0 — decay\_rate) ^ hours\_passed

Here, hours\_passed refers to the time since the document was last accessed (not the creation time). Let’s examine with low and high decay rates.

**6.1) Low Decay Rate**

Setting a low decay rate (close to 0) means that memories are “remembered” for longer. A decay rate of 0 makes this retriever equivalent to a vector lookup.

# Example - Time-weighted vector store retriever - Low decay rate  
  
# Define embedding model  
embeddings\_model = OpenAIEmbeddings(api\_key=OPENAI\_API\_KEY)  
  
# Initialize vectorstore  
embedding\_size = 1536  
index = faiss.IndexFlatL2(embedding\_size)  
vectorstore = FAISS(embeddings\_model, index, InMemoryDocstore({}), {})  
  
# Initialize TimeWeightedVectorStoreRetriever with very low decay\_rate  
retriever = TimeWeightedVectorStoreRetriever(  
 vectorstore=vectorstore, decay\_rate=0.0000000000000000000000001, k=1  
)  
  
# Add documents with access times  
yesterday = datetime.now() - timedelta(days=1)  
retriever.add\_documents(  
 [Document(page\_content="hello world", metadata={"last\_accessed\_at": yesterday})]  
)  
retriever.add\_documents([Document(page\_content="hello foo")])  
  
# Retrieve and print result  
result = retriever.invoke("hello world")  
print(result)  
  
# Example of output  
"""  
[Document(metadata={'last\_accessed\_at': datetime.datetime(2024, 12, 29, 4, 55, 19, 937951), 'created\_at': datetime.datetime(2024, 12, 29, 4, 55, 19, 67516), 'buffer\_idx': 0}, page\_content='hello world')]  
"""

**6.2) High Decay Rate**

Conversely, a high decay rate (close to 1) means that the recency score quickly diminishes, making the retriever prioritize semantic similarity more.

# Example - Time-weighted vector store retriever - High decay rate  
  
# Define embedding model  
embeddings\_model = OpenAIEmbeddings(api\_key=OPENAI\_API\_KEY)  
  
# Initialize vectorstore  
embedding\_size = 1536  
index = faiss.IndexFlatL2(embedding\_size)  
vectorstore = FAISS(embeddings\_model, index, InMemoryDocstore({}), {})  
  
# Initialize TimeWeightedVectorStoreRetriever with high decay\_rate  
retriever = TimeWeightedVectorStoreRetriever(  
 vectorstore=vectorstore, decay\_rate=0.999, k=1  
)  
  
# Add documents with access times  
yesterday = datetime.now() - timedelta(days=1)  
retriever.add\_documents(  
 [Document(page\_content="hello world", metadata={"last\_accessed\_at": yesterday})]  
)  
retriever.add\_documents([Document(page\_content="hello foo")])  
  
# Retrieve and print result  
result = retriever.invoke("hello world")  
print(result)  
  
# Example of output  
"""  
[Document(metadata={'last\_accessed\_at': datetime.datetime(2024, 12, 29, 5, 53, 1, 467568), 'created\_at': datetime.datetime(2024, 12, 29, 5, 53, 1, 134997), 'buffer\_idx': 1}, page\_content='hello foo')]  
"""