LlamaIndex Introduction: Precision & Simplicity in Information Retrieval

In this guide, we will comprehensively explore the LlamaIndex framework, which helps in building Retrieval-Augmented Generation (RAG) systems for LLM-based applications.

LlamaIndex, like other RAG frameworks, combines the fetching of relevant information from a vast database with the generative capabilities of Large Language Models. It involves providing supplementary information to the LLM for a posed question to ensure that the LLM does not generate inaccurate responses.
The aim is to provide a clear understanding of the best practices in developing LLM-based applications. We will explain LlamaIndex's fundamental components in the following sections.

Vector Stores

Vector store databases enable the storage of large, high-dimensional data and provide the essential tools for semantically retrieving relevant documents. This implies that rather than naively checking for the presence of specific words in a document, these systems **analyze the embedding vectors** that encapsulate the entire document's meaning. This approach simplifies the search process and enhances its accuracy.
Searching in vector stores focuses on fetching data according to **its vector-based representations.** These databases are integral across various domains, including Natural Language Processing **(NLP) and multimodal** applications, allowing for the efficient storing and analysis of high-dimensional datasets.

A primary function in vector stores is the **similarity search,** aiming to locate vectors closely resembling a specific query vector. This functionality is critical in numerous AI-driven systems, such as recommendation engines and image retrieval platforms, where pinpointing contextually relevant data is critical.

Semantic search transcends traditional keyword matching by seeking information that aligns conceptually with the user's query. It captures the **meaning in vectorized representations** by employing advanced ML techniques. Semantic search can be applied to all data formats, as we vectorize the data before storing it in a database. Once we have an embedded format, we can calculate indexed similarities or capture the context embedded in the query. This ensures that the results are relevant and in line with the contextual and conceptual nuances of the user's intent.

Data Connectors

The effectiveness of a RAG-based application is significantly enhanced by accessing a vector store that compiles information from various sources. However, managing data in diverse formats can be challenging.

Data connectors, also called `Readers`, are **essential in LlamaIndex**. Readers are responsible for **parsing and converting the data into a simplified** `Document` representation consisting of text and basic metadata. Data connectors are designed to streamline the data ingestion process, automate the task of fetching data from various sources (**like APIs, PDFs, and SQL databases**), and format it.

LlamaHub is an open-source project that hosts data connectors. LlamaHub repository offers data connectors for ingesting all possible data formats into the LLM.

You can check out the LlamaHub repository and test some of the loaders here. You can explore various integrations and data sources with the embedded link below. These implementations make the preprocessing step as simple as executing a function. Take the Wikipedia integration, for instance.
Before testing loaders, we must install the required packages and set the OpenAI API key for LlamaIndex. You can get the API key on OpenAI's website and set the environment variable with `OPENAI_API_KEY`.
Please note that **LlamaIndex defaults to using OpenAI's `get-3.5-turbo` for text generation and `text-embedding-ada-002` model for embedding generation**.

```
pip install -q llama-index==0.9.14.post3 openai==1.3.8 cohere==4.37
```

The commands to install libraries.

```
# Add API Keys

import os

os.environ['OPENAI_API_KEY'] = '<YOUR_OPENAI_API_KEY>'

# Enable Logging

import logging

import sys

#You can set the logging level to DEBUG for more verbose output,

# or use level=logging.INFO for less detailed information.

logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)

logging.getLogger().addHandler(logging.StreamHandler(stream=sys.stdout))
```

The sample code.

We have also added a **logging mechanism** to the code. Logging in LlamaIndex is a way to monitor the operations and events that occur during the execution of your application. Logging helps develop and debug the process and understand the details of what the application is doing. In a **production environment, you can configure the logging module** to output log messages to a file or a logging service.

⚠

The configuration of the logging module, which directs log messages to the standard output (`sys.stdout`) and sets the logging level as INFO, leads to the logging of all messages with a severity level of INFO or higher. Also, you can use **`logging.debug`** to get more detailed information.

We can now use the **`download_loader`** method to **access integrations** from LlamaHub and activate them by passing the integration name to the class. In our sample code, the `WikipediaReader` class takes in several page titles and returns the text contained within them as `Document` objects.

```
from llama_index import download_loader

WikipediaReader = download_loader("WikipediaReader")

loader = WikipediaReader()

documents = loader.load_data(pages=['Natural Language Processing', 'Artificial Intelligence'])

print(len(documents))
```

The sample code.

This retrieved information can be stored and utilized to enhance the knowledge base of our chatbot.

Nodes

In LlamaIndex, once data is ingested as documents, it passes through a processing structure that transforms these documents into `Node` objects. **Nodes are smaller, more granular data units created from the original documents**. Besides their primary content, these nodes **also contain metadata and contextual information.**

LlamaIndex features a `NodeParser` class designed to convert the content of documents into structured nodes automatically. The `SimpleNodeParser` converts a list of document objects into nodes.

```
from llama_index.node_parser import SimpleNodeParser

# Assuming documents have already been loaded

# Initialize the parser

parser = SimpleNodeParser.from_defaults(chunk_size=512, chunk_overlap=20)

# Parse documents into nodes

nodes = parser.get_nodes_from_documents(documents)

print(len(nodes))
```

The sample code.

The output.

The code above splits the two retrieved documents from the Wikipedia page into 48 smaller chunks with slight overlap.

Indices

At the heart of LlamaIndex is the **capability to index and search various data formats** like documents, PDFs, and database queries. Indexing is an i**nitial step for storing information in a database**; it essentially **transforms the unstructured data into embeddings** that capture semantic meaning and optimize the data format so it can be easily accessed and queried.

LlamaIndex has a variety of index types, each fulfills a specific role. We have highlighted some of the popular index types in the following subsections.

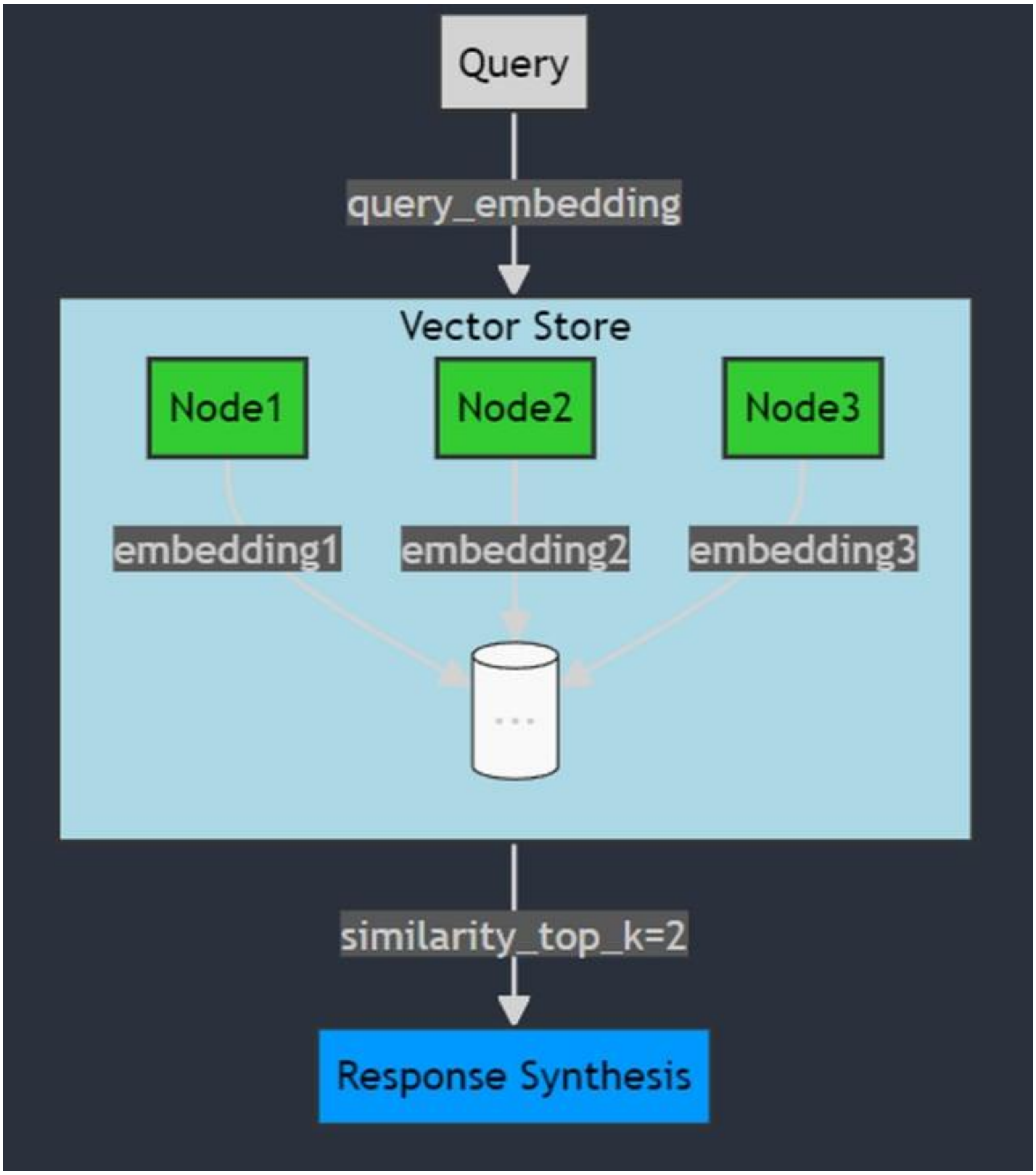
Summary Index

The Summary Index extracts a summary from each document and stores it with all the nodes in that document. Since it's not always easy to match small node embeddings with a query, sometimes having a document summary helps.

Vector Store Index

The Vector Store Index generates embeddings during index construction to identify the top-k most similar nodes in response to a query.

It's suitable for small-scale applications and easily scalable to accommodate larger datasets using high-performance vector databases.



Fetching the top-k nodes and passing them for generating the final response

The crawled Wikipedia documents can be stored in a Deep Lake vector store, and an index object can be created based on its data. We can create the dataset in Activeloop and append documents to it by employing the `DeepLakeVectorStore` class. First, we need to set the Activeloop and OpenAI API keys in the environment using the following code.

```
import os
os.environ['OPENAI_API_KEY'] = '<YOUR_OPENAI_API_KEY>'
os.environ['ACTIVELOOP_TOKEN'] = '<YOUR_ACTIVELOOP_KEY>'
```

To connect to the platform, use the `DeepLakeVectorStore` class and provide the `dataset path` as an argument. To save the dataset on your workspace, you can replace the `genai360` name with your organization ID (which defaults to your Activeloop username). Running the following code will create an empty dataset.

```
from llama_index.vector_stores import DeepLakeVectorStore
my_activeloop_org_id = "genai360"
my_activeloop_dataset_name = "LlamaIndex_intro"
dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"
# Create an index over the documnts
vector_store = DeepLakeVectorStore(dataset_path=dataset_path, overwrite=False)
```

```
Your Deep Lake dataset has been successfully created!
```

The output.

Now, we need to create a storage context using the **`StorageContext`** class and the Deep Lake dataset as the source. Pass this storage to a **`VectorStoreIndex`** class to create the index (generate embeddings) and store the results on the defined dataset.

```
from llama_index.storage.storage_context import StorageContext
from llama_index import VectorStoreIndex
storage_context = StorageContext.from_defaults(vector_store=vector_store)
index = VectorStoreIndex.from_documents(documents, storage_context=storage_context)
```

```
Uploading data to deeplake dataset.
100%|██████████| 23/23 [00:00<00:00, 69.43it/s]
Dataset(path='hub://genai360/LlamaIndex_intro', tensors=['text', 'metadata', 'embedding', 'id'])

  tensor      htype      shape      dtype  compression
 -------     -------    -------    -------  -------
   text       text      (23, 1)      str     None
 metadata     json      (23, 1)      str     None
 embedding  embedding  (23, 1536)  float32   None
    id         text      (23, 1)      str     None
```

The output.

The created database will be accessible in the future. The Deep Lake database efficiently stores and retrieves high-dimensional vectors.

Learn more about other Index types from LlamaIndex documentation.

Query Engines

The next step is to leverage the generated indexes to query through the information. The Query Engine is a **wrapper that combines a Retriever and a Response Synthesizer into a pipeline**. The pipeline uses the query string to fetch nodes and then sends them to the LLM to generate a response. A query engine can be created by calling the `as_query_engine()` method on an already-created index.

The code below uses the documents fetched from the Wikipedia page to construct a Vector Store Index using the `GPTVectorStoreIndex` class. The `.from_documents()` method simplifies building indexes on these processed documents. The created index can then be utilized to generate a `query_engine` object, allowing us to ask questions based on the documents using the `.query()` method.

```
from llama_index import GPTVectorStoreIndex

index = GPTVectorStoreIndex.from_documents(documents)

query_engine = index.as_query_engine()

response = query_engine.query("What does NLP stands for?")

print( response.response )
```

```
NLP stands for Natural Language Processing.
```

The output.

The indexes can also function solely as retrievers for fetching documents relevant to a query. This capability enables the creation of a **Custom Query Engine**, offering more control over various aspects, such as the **prompt or the output format**. You can learn more here.

Routers

Routers play a role in determining the most appropriate retriever for extracting context from the knowledge base. The routing function **selects the optimal query engine for each task**, improving performance and accuracy.
These functions are beneficial **when dealing with multiple data sources**, each holding unique information.
Consider an application that employs a SQL database and a Vector Store as its knowledge base. In this setup, the router can determine which data source is most applicable to the given query.

You can see a working example of incorporating the routers in this tutorial.

Saving and Loading Indexes Locally

All the examples we explored involved storing indexes on cloud-based vector stores like Deep Lake. However, there are scenarios where saving the data on a disk might be necessary for rapid testing. The concept of storing refers to saving the index data, which includes the nodes and their associated embeddings, to disk. This is done using the `persist()` method from the **`storage_context`** object related to the index.

```
# store index as vector embeddings on the disk
index.storage_context.persist()
# This saves the data in the 'storage' by default
# to minimize repetitive processing
```

If the index already exists in storage, you can load it directly instead of recreating it. We simply need to determine whether the index already exists on disk and proceed accordingly; here is how to do it:

```
# Index Storage Checks
import os.path
from llama_index import (
    VectorStoreIndex,
    StorageContext,
    load_index_from_storage )
from llama_index import download_loader
# Let's see if our index already exists in storage.
if not os.path.exists("./storage"):
   # If not, we'll load the Wikipedia data and create a new index
   WikipediaReader = download_loader("WikipediaReader")
   loader = WikipediaReader()
   documents = loader.load_data(pages=['Natural Language Processing', 'Artificial Intelligence'])
   index = VectorStoreIndex.from_documents(documents)
   # Index storing
   index.storage_context.persist()
else:
   # If the index already exists, we'll just load it:
   storage_context = StorageContext.from_defaults(persist_dir="./storage")
   index = load_index_from_storage(storage_context)
```

In this example, the `os.path.exists("./storage")` function is used to check if the 'storage' directory exists. If it does not exist, the Wikipedia data is loaded, and a new index is created.

LangChain vs. LlamaIndex

LangChain and LlamaIndex are designed to improve LLMs' capabilities, each with their unique strengths.

LlamaIndex: LlamaIndex specializes in **processing, structuring, and accessing private or domain-specific data, with a focus on specific LLM interactions**. It works for tasks that demand high precision and quality when dealing with specialized, domain-specific data. Its main strength lies in linking Large Language Models (LLMs) to any data source.

LangChain is dynamic, suited for **context-rich interactions,** and effective for applications like chatbots and virtual assistants. These features render it highly appropriate for quick prototyping and application development.
While generally used independently, it is worth noting that it can be possible to combine functions from both LangChain and LlamaIndex where they have different strengths. Both can be complementary tools. We also designed a little table below to help you understand the differences better. The attached video in the course also aims to help you decide which tool you should use for your application: LlamaIndex, LangChain, OpenAI Assistants, or doing it all from scratch (yourself).

Here's a clear comparison of each to help you quickly grasp the essentials on a few relevant topics you may consider when choosing:

In an upcoming lesson, we will explore the functionalities of both frameworks and aim to determine the ideal scenarios for using each.

	LangChain	**LlamaIndex**	**OpenAI Assistants**
	Interact with LLMs - Modular and more flexible	Data framework for LLMs - Empower RAG	Assistant API - SaaS
Data	• Standard formats like CSV, PDF, TXT, … • Mostly focus on Vector Stores.	• LlamaHub with dedicated data loaders from different sources. (Discord, Slack, Notion, …) • Efficient indexing and retrieving + easily add new data points without calculating embeddings for all. • Improved chunking strategy by linking them and using metadata. • Support multimodality.	• 20 files where each can be up to 512mb. • Accept a wide range of file types.

LLM Interaction	• Prompt templates to facilitate interactions. • Very flexible, easily defining chains and using different modules. Choose the prompting strategy, model, and output parser from many options. • Can directly interact with LLMs and create chains without the need to have additional data.	• Mostly use LLMs in the context of manipulating data. Either for indexing or querying.	• Either GPT-3.5 Turbo or GPT-4 + any fine-tuned model.
Optimizations	-	• LLM fine-tuning • Embedding fine-tuning	-
Querying	• Use retriever functions.	•Advanced indexing/querying techniques like subquestions, HyDe,... • Routing: enable to use multiple data sources	• thread and messages to keep track of users conversations.
Agents	• LangSmith	• LlamaHub	• Code interpreter, knowledge retriever, and custom function calls.
Documentation	• Easy to debug. • Easy to find concepts and understand the function usage.	• As of November 2023, the methods are mostly explained in the form of tutorials or blog posts. A bit harder to debug.	• Great.
Pricing	FREE	FREE	• $0.03 / code interpreter session • $0.20 / GB / assistant / day • + The usual usage of the LLM

A smart approach involves closely examining your unique use case and its specific demands. The key decision boils down to whether you need interactive agents or powerful search capabilities for retrieving information.
If your aim leans towards the latter, LlamaIndex stands out as a likely choice for enhanced performance.

Conclusion

We explored the features of the LlamaIndex library and its building blocks. During our exploration of the LlamaIndex library and its building blocks, we learned about various features. Firstly, we discussed data connectors and LlamaHub, which help to retrieve relevant data through vector stores and retrievers. We also talked about routers that assist in selecting the most suitable retrievers. Additionally, we looked at data connectors that are capable of ingesting data from various sources, such as APIs, PDFs, and SQL databases, and how the indexing step structures the data into embedding representations. Lastly, we covered the query engines that provide knowledge retrieval.

LangChain and LlamaIndex are valuable and popular frameworks for developing apps powered by language models. LangChain offers a broader range of capabilities and tool integration, while LlamaIndex specializes in indexing and retrieval of information.

Read the LlamaIndex documentation to explore its full potential.
>> Notebook.