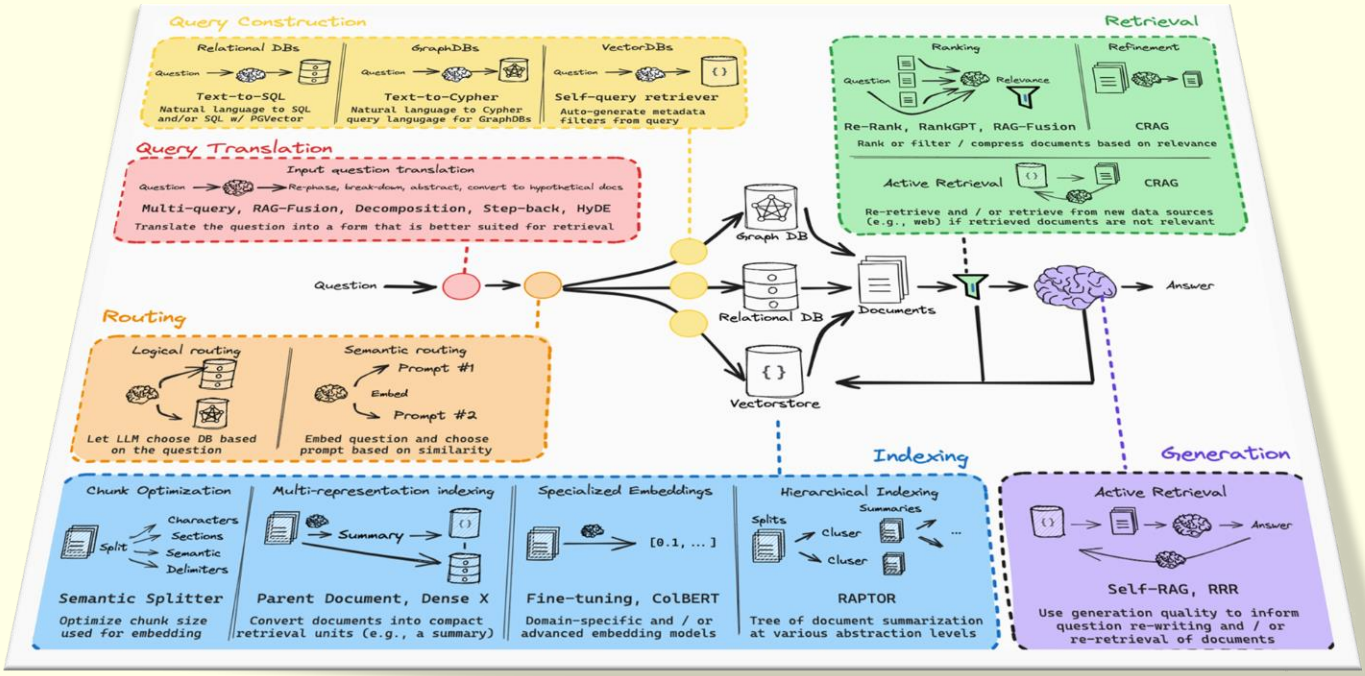
RAG Tutorial

(Retrieval Augmented Generation)



Day 1 of 7

Table of Contents

1. Introduction

Large language models (LLMs) have enabled the creation of powerful question-answering (Q&A) chatbots. These applications can provide answers to queries about specific information sources using a method called Retrieval Augmented Generation (RAG).

In this tutorial, we'll guide you through the process of building a basic Q&A application that works with text data sources. We'll cover a standard Q&A architecture, offer resources for advanced Q&A techniques, and demonstrate how LangSmith can assist in tracing and understanding your application, becoming increasingly useful as your application grows more complex.

2. What is RAG?

Retrieval Augmented Generation (RAG) is a technique that enhances the knowledge of LLMs with additional data. While LLMs can handle a wide range of topics, their knowledge is limited to the public data they were trained on up to a specific point in time. To build AI applications that can reason about private data or data introduced after a model's training cutoff date, we need to supplement the model's knowledge with the necessary information. This process of retrieving the relevant information and integrating it into the model's prompt is known as RAG.

LangChain provides various components designed to facilitate the development of Q&A applications and RAG applications more generally.

3. Concepts

A typical RAG application consists of two main components:

1. **Indexing**: This involves creating a pipeline for ingesting and indexing data from a source. This process usually occurs offline.
2. **Retrieval and Generation**: This component handles user queries at runtime, retrieving relevant data from the index and passing it to the model.

The complete sequence from raw data to the final answer generally includes:

4. Indexing

1. **Load**: The initial step is to load our data using Document Loaders.

2. **Split**: Large documents are divided into smaller chunks using text splitters. This is beneficial for both indexing and for fitting the data into a model's limited context window.

3. **Store**: We store and index these chunks so they can be searched later. This is typically done using a VectorStore and an Embeddings model.

5. Retrieval and Generation

1. **Retrieve**: For a given user query, relevant chunks are retrieved from storage using a Retriever.

2. **Generate**: A ChatModel/LLM generates an answer by using a prompt that includes both the question and the retrieved data.

6. Sample Code

Let's build our simple Q&A application step-by-step

Step 1: Load Data

First, make sure you have the necessary libraries installed. You can install them using pip:

```
pip install langchain
pip install some-vectorstore-library
pip install some-embeddings-library
```

Step 2: Load Data

We'll start by loading the data using a Document Loader. For simplicity, let's assume we have a text file containing our data.

```
from langchain.document_loaders import TextLoader

# Load data from a text file
loader = TextLoader('path/to/your/data.txt')
documents = loader.load()
```

Step 3: Split Data

Next, we split the loaded documents into smaller chunks.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Split documents into smaller chunks
splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
                                          chunk_overlap=200)
chunks = splitter.split(documents)
```

Step 4: Store the Chunks

We'll store the split data using a VectorStore and Embeddings model.

```
from some_vectorstore_library import VectorStore
from some_embeddings_library import EmbeddingsModel

# Initialize embeddings model and vector store
embeddings_model = EmbeddingsModel()
vector_store = VectorStore()

# Embed the chunks and store them
for chunk in chunks:
    embedding = embeddings_model.embed(chunk)
    vector_store.add(embedding, chunk)
```

Step 5: Retrieve Relevant Data

We now need to store these chunks in a VectorStore for easy retrieval later.

```
from some_retriever_library import Retriever

# Initialize retriever
retriever = Retriever(vector_store)

# User query
query = "Your question here"

# Retrieve relevant chunks
relevant_chunks = retriever.retrieve(query)
```

Step 6: Generate the Answer

For a given user query, we retrieve the relevant chunks from the VectorStore.

```
from some_chatmodel_library import ChatModel

# Initialize chat model
chat_model = ChatModel()

# Generate answer
answer = chat_model.generate(query, relevant_chunks)
print(answer)
```

7. Conclusion

In this tutorial, we built a simple Q&A application using the Retrieval Augmented Generation (RAG) technique. We covered the basic components of a RAG application, including indexing and retrieval/generation, and saw how LangChain and LangSmith can facilitate the development of such applications. As your application grows in complexity, LangSmith can provide valuable insights and traceability. For more advanced techniques and further reading, explore the additional resources provided by LangChain.

Stay tuned for the next tutorial in this series, where we will dive deeper into optimizing and scaling your RAG applications.