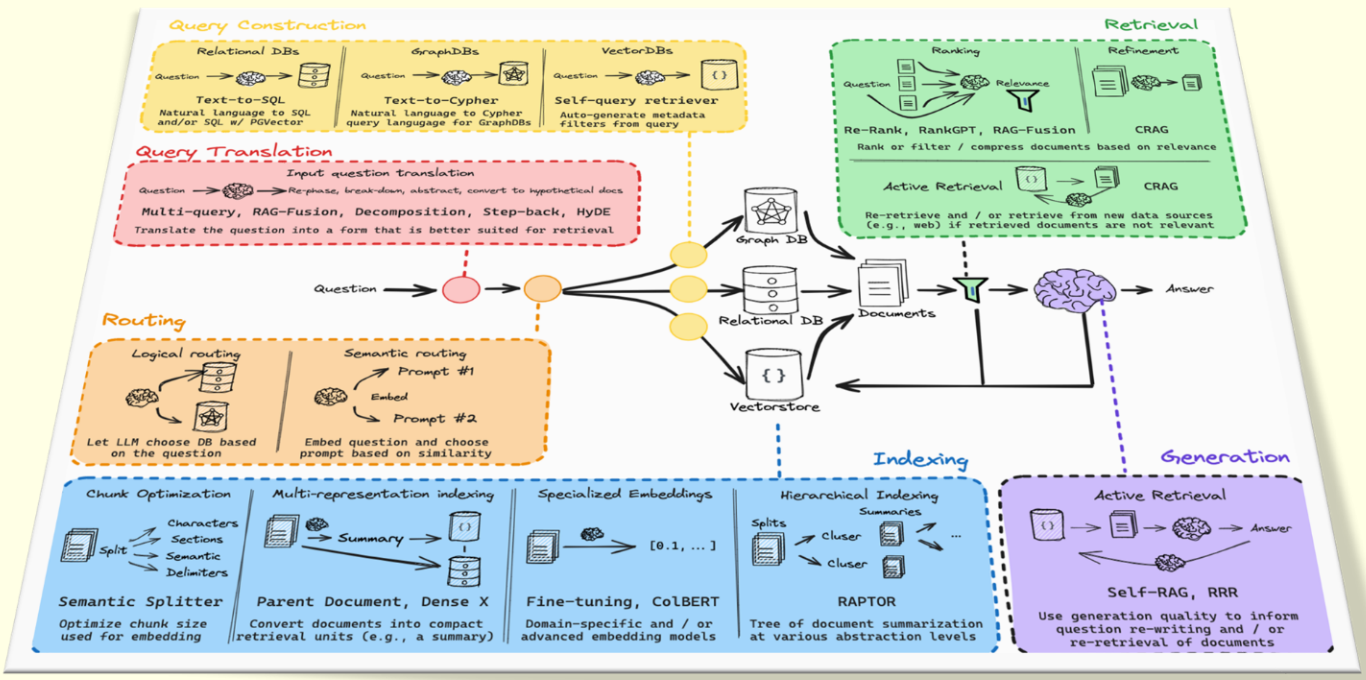
RAG Tutorial

(Retrieval Augmented Generation)

Indexing - Store



Query Construction
Relational DBs
Question
Text-to-SQL
Natural language to SQL and/or SQL w/ PGVector
GraphDBs
Text-to-Cypher
Natural language to Cypher query language for GraphDBs
VectorDBs
Self-query retriever
Auto-generate metadata filters from query
Retrieval
Ranking
Relevance
Re-Rank, RankGPT, RAG-Fusion
Refinement
CRAG
Rank or filter / compress documents based on relevance
Active Retrieval
CRAG
Re-retrieve and / or retrieve from new data sources (e.g., web) if retrieved documents are not relevant
Query Translation
Input question translation
Re-phase, break-down, abstract, convert to hypothetical docs
Multi-query, RAG-Fusion, Decomposition, Step-back, HyDE
Translate the question into a form that is better suited for retrieval
Graph DB
Relational DB
Documents
Vectorstore
Answer
Routing
Logical routing
Let LLM choose DB based on the question
Semantic routing
Prompt #1
Embed
Prompt #2
Embed question and choose prompt based on similarity
Indexing
Generation
Chunk Optimization
Characters
Sections
Split
Semantic
Delimiters
Semantic Splitter
Optimize chunk size used for embedding
Multi-representation indexing
Summary
Parent Document, Dense X
Convert documents into compact retrieval units (e.g., a summary)
Specialized Embeddings
[0.1, …]
Fine-tuning, ColBERT
Domain-specific and / or advanced embedding models
Hierarchical Indexing
Summaries
Splits
Cluser
Cluser
RAPTOR
Tree of document summarization at various abstraction levels
Active Retrieval
Answer
Self-RAG, RRR
Use generation quality to inform question re-writing and / or re-retrieval of documents

Day 5 of 7

Table of Contents

1. Introduction

Welcome to Day 5 of our LangChain Retrieval-Augmented Generation (RAG) tutorial series.

In the previous article, we delved into the workings of splitting documents into manageable chunks, setting the foundation for efficient indexing and retrieval.

Today, we'll take the next step: storing these chunks in a vector store, enabling us to efficiently search and retrieve relevant content based on user queries.

2. What is Embeddings?

1. **Embeddings are numerical representations of text, essential for tasks like similarity search.**

2. **Imagine you have a bunch of words, like "cat," "dog," "car," and "truck."**

3. **Embeddings are like secret codes for these words, turning them into lists of numbers.**

4. **Words with similar meanings, like "cat" and "dog," will have codes that are very similar.**

5. **Words with different meanings, like "cat" and "car," will have codes that are very different.**

6. **These secret codes help computers understand the meaning of words and how they relate to each other.**

7. **It's like a magic trick that helps computers understand language better**

8. **The *OpenAIEmbeddings* class is a wrapper around an embedding model, converting text into high-dimensional vectors.**

3. What is VectorStore?

1. **VectorStore is where the embeddings are stored and queried.**

2. **Imagine you have a big box full of toys. You want to be able to find any toy quickly and easily.**

3. **So, you organize them neatly, putting all the cars together, all the dolls together, and all the building blocks together.**

4. **A VectorStore is like that big box, but instead of toys, it holds information, like sentences from a book or facts about animals.**

5. **Each piece of information is given a special code, like a secret number, that describes what it's about.**

6. This special code helps the VectorStore find information quickly, just like organizing your toys helps you find the one you want.

7. When you ask the VectorStore a question, it uses the codes to find the information that's most related to your question.

8. So, think of a VectorStore as a super-organized box of information that uses secret codes to help you find what you're looking for quickly and easily

9. The Chroma vector store allows for efficient retrieval of embeddings based on similarity measures like cosine similarity.

4. Understanding the Importance of Indexing

Indexing is a crucial part of the RAG pipeline. It allows us to store and query large amounts of text data in an efficient manner.

By converting text into embeddings numerical representations in high-dimensional space.

we can perform similarity searches to find the most relevant chunks of information based on a user's query.

This process underpins the ability to retrieve accurate and contextually relevant answers from a large corpus of text.

5. Embedding and Storing Document Splits

To achieve this, we need to embed our 66 text chunks and store these embeddings in a vector database (or vector store).

The vector store allows us to perform similarity searches, where we measure how close the stored embeddings are to a query embedding.

A common similarity measure is cosine similarity, which calculates the cosine of the angle between two high-dimensional vectors.

In this article, we'll use the Chroma vector store in combination with the OpenAIEmbeddings model to embed and store our document splits.

6. Sample Code and Explanation

Define the Text Chunks

Instead of defining a pre-set list, we'll demonstrate how to split a larger document into smaller, more manageable text chunks. This process is crucial for efficient retrieval and will be explored in detail

Embed and Store Document Splits

Each text chunk will be transformed into a numerical representation (embedding) using *OpenAIEmbeddings*. These embeddings will then be stored within the Chroma vector store.

We'll leverage the *Chroma.from_documents()* function to streamline both embedding and storage in a single step.

Querying the Vector Store

We'll formulate a user query and convert it into an embedding using *OpenAIEmbeddings*.

This query embedding will then be compared against the embeddings stored in the Chroma vector store.

Utilizing cosine similarity, we'll retrieve the most relevant document chunk(s) matching the user's query.

Inspect the Vector Store

Finally, we'll examine the contents of our populated vector store. This includes checking the number of stored document chunks and previewing a sample chunk to understand its structure and content.

Link of collab Notebook

7. Conclusion

In this tutorial, With our document chunks now embedded and stored in a vector store, we've completed the indexing portion of our pipeline.

This is a crucial step in building a powerful question-answering system, as it allows us to efficiently search and retrieve the most relevant chunks of information based on user queries.

In the next part of our series, we'll explore how to perform similarity searches over our indexed embeddings, enabling us to retrieve the most relevant answers to user questions.

Stay tuned for Day 6, where we will delve deeper into the Retrieval and Generation: Retrieve