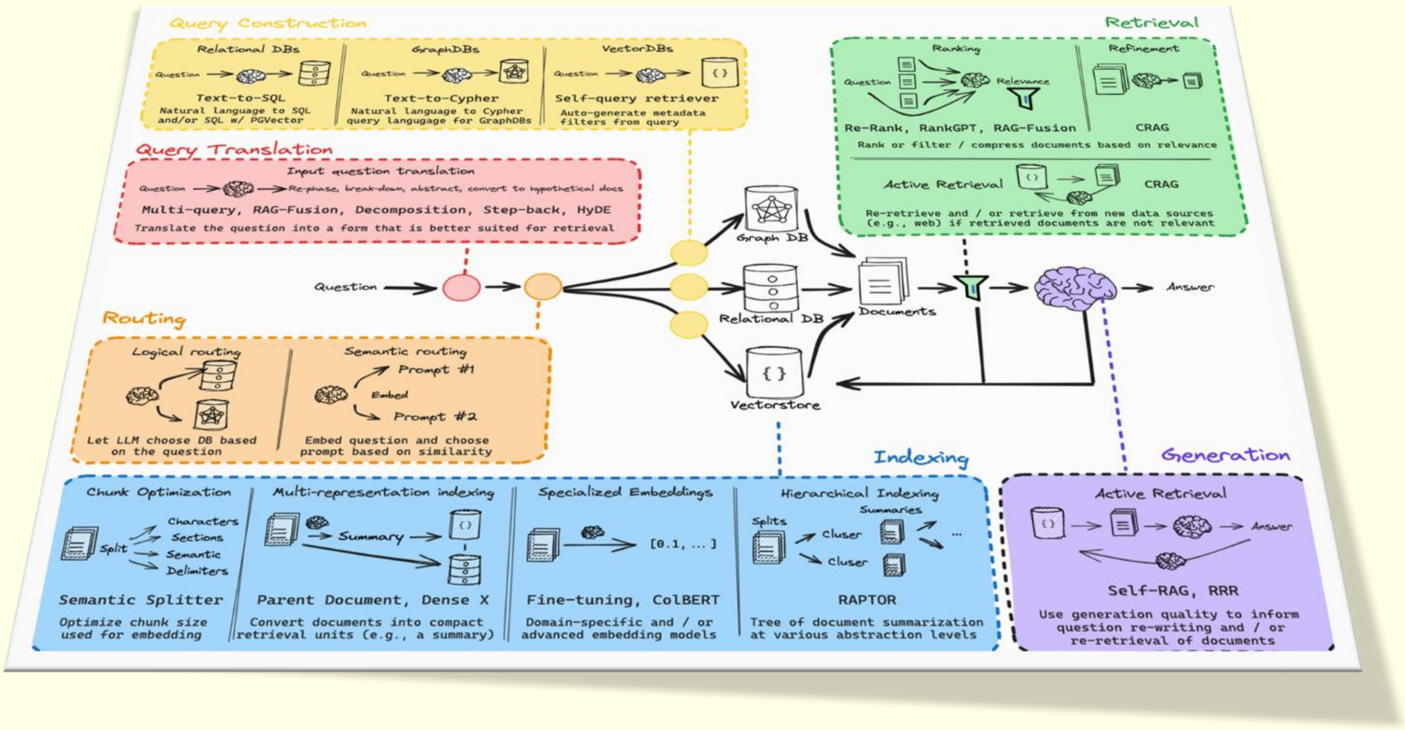
RAG Tutorial

(Retrieval Augmented Generation)

Indexing and Loading Blog Post Content Using DocumentLoaders



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
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
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
Retrieval
Ranking
Relevance
Refinement
Re-Rank, RankGPT, RAG-Fusion
CRAG
Rank or filter / compress documents based on relevance
Active Retrieval
Re-retrieve and / or retrieve from new data sources (e.g., web) if retrieved documents are not relevant
Graph DB
Relational DB
Vectorstore
Documents
Answer
Indexing
Chunk Optimization
Characters
Sections
Semantic
Delimiters
Split
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
RAPTOR
Tree of document summarization at various abstraction levels
Generation
Self-RAG, RRR
Use generation quality to inform question re-writing and / or re-retrieval of documents

Day 3 of 7

Table of Contents

1. Introduction

In the previous sessions, we set up our environment and previewed the construction of a question-answering app.

Now, we need to load the content we want to index and retrieve from. For this, we'll use DocumentLoaders, which are designed to fetch data from various sources and return it as a list of Documents.

2. Loading Blog Post Contents

We'll DocumentLoaders simplify the task of fetching and parsing data from different sources.

Each Document consists of page_content (a string) and metadata (a dictionary).

In this tutorial, we'll use the WebBaseLoader to load and parse HTML content from a specified URL.

3. Customizing HTML Parsing

The WebBaseLoader utilizes urllib to fetch HTML content and BeautifulSoup to parse it.

We can customize the parsing process by passing parameters to the BeautifulSoup parser through bs_kwargs.

For our purpose, we will filter and keep only HTML tags with the classes "post-content", "post-title", and "post-header"

4. Sample Code and Explanation

In this Google Collab Notebook, we will demonstrates how to extract specific content from a website using the `langchain_community` and `bs4` (Beautiful Soup) libraries.

First, it defines a `bs4_strainer` using `SoupStrainer` to target only HTML elements with classes "post-title", "post-header", and "post-content".

Then, it initializes a `WebBaseLoader` with the target website URL and the strainer. When `loader.load()` is called, it fetches the website content and parses it, keeping only the specified elements.

Finally, the code prints the length of the extracted content and displays the first 500 characters as a preview.

This technique is useful for extracting relevant information from websites while discarding unnecessary elements.

Link of collab Notebook

5. Conclusion

In this tutorial, we explored how to use the *WebBaseLoader* to load and parse HTML content from a web URL.

We customized the parsing process to filter and retain only relevant sections of the HTML using *BeautifulSoup*.

This forms the basis for our content indexing, which is crucial for building our question-answering app.

Stay tuned for Day 4, where we will delve deeper into the indexing process and prepare our data for efficient retrieval.

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