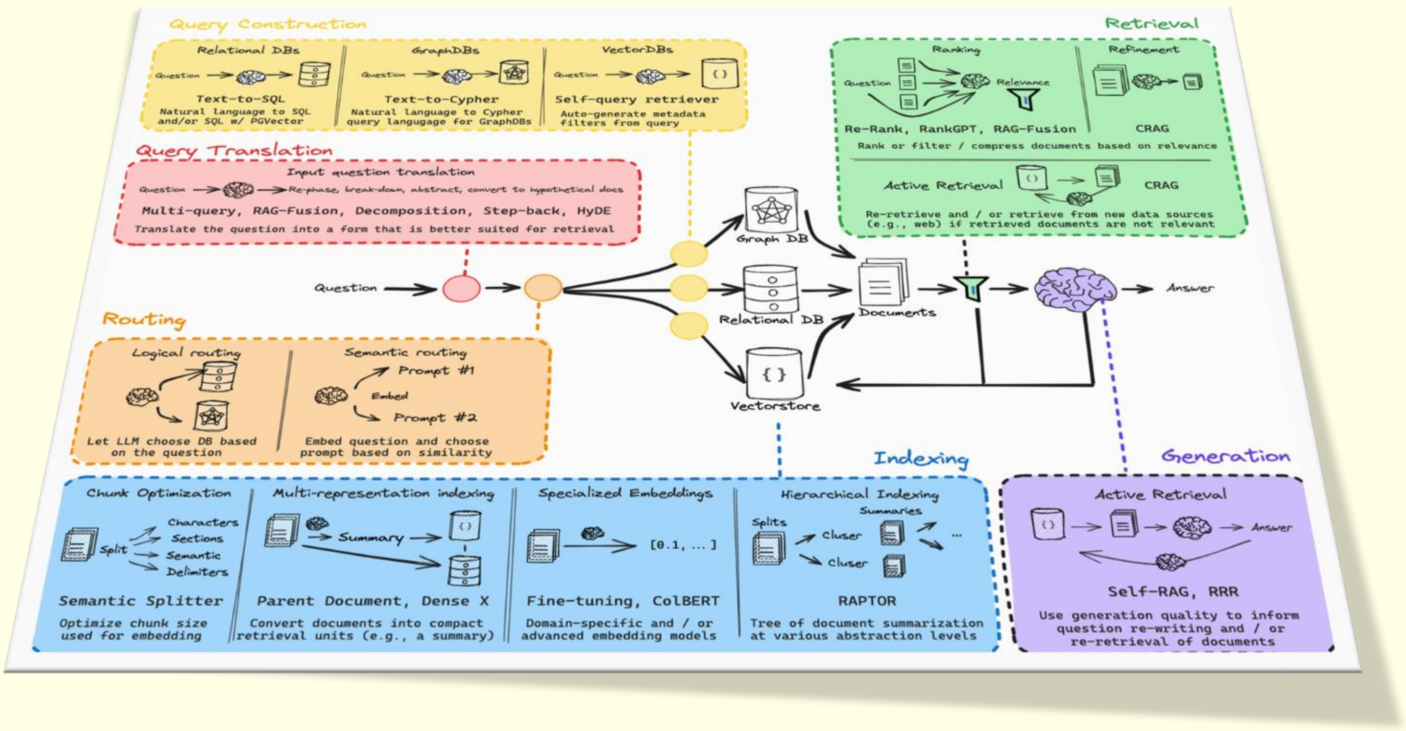
RAG Tutorial

(Retrieval Augmented Generation)

Indexing - Splitting Documents for Effective Retrieval



Query Construction
Relational DBs
Question
Text-to-SQL
Natural language to SQL and/or SQL w/ PGVector
GraphDBs
Question
Text-to-Cypher
Natural language to Cypher query langugage for GraphDBs
VectorDBs
Question
Self-query retriever
Auto-generate metadata filters from query
Query Translation
Input question translation
Question
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
Question
Graph DB
Relational DB
Vectorstore
Documents
Answer
Retrieval
Ranking
Question
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
Indexing
Chunk Optimization
Split
Characters
Sections
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
Generation
Active Retrieval
Answer
Self-RAG, RRR
Use generation quality to inform question re-writing and / or re-retrieval of documents

Day 4 of 7

Table of Contents

1. Introduction
2. Why Split the Document?
3. Splitting Strategy:
    - Recursive Character Text Splitter
4. Sample Code & Explanation
    - Defining the Text Splitter
    - Splitting the Document
    - Output the Number of Chunks
    - Check Chunk Content Length
    - Inspect Metadata
5. Conclusion

1. Introduction

In the previous tutorial, we successfully loaded and parsed a lengthy blog post, setting the stage for the next crucial step in building our question-answering app: indexing.

Our loaded document is over 42,000 characters long, which is far too lengthy to fit within the context window of most language models.

Even for models that can accommodate such a large input, processing very long text can be inefficient and lead to suboptimal retrieval of relevant information.

To overcome these challenges, we'll break down our document into manageable chunks.

2. Why Split the Document?

Splitting the document into smaller chunks enables us to efficiently embed and store these chunks as vectors, which will be crucial for our retrieval process.

By doing so, our model will be able to focus on retrieving the most relevant sections of the blog post when answering queries.

This method enhances both performance and accuracy, ensuring that the context remains intact and that important information isn't lost.

3. Splitting Strategy:

Recursive Character Text Splitter

1. For this task, we'll use the *RecursiveCharacterTextSplitter* from LangChain.

2. This tool is designed to split a document into chunks based on common separators like new lines, sentences, or paragraphs.

3. It recursively divides the document until each chunk reaches the desired size.

4. This approach is particularly effective for generic text, where the logical structure (e.g., paragraphs) is essential.

5. We'll configure our splitter to create chunks of 1,000 characters, with a 200-character overlap between consecutive chunks.

6. The overlap helps preserve context, ensuring that a statement at the end of one chunk retains its connection to the beginning of the next.

7. Additionally, we'll set add_start_index=True to preserve the starting character index of each chunk as a metadata attribute called start_index.

8. This metadata can be crucial when tracking the original location of a chunk within the full document.

4. Sample Code and Explanation

Defining the Text Splitter

We start by importing the *`RecursiveCharacterTextSplitter`* and defining it with a `chunk_size` of 1,000 characters and a `chunk_overlap` of 200 characters. The `add_start_index=True` parameter ensures that each chunk retains information about where it begins within the original document.

Splitting the Document:

The `split_documents()` method is then called on our document (`docs`). This method returns a list of smaller document chunks (`all_splits`), each containing part of the original document's content.

Output the Number of Chunks

Using `len(all_splits)`, we can check how many chunks were created. For a document of 42,000 characters, this should give us a manageable number of chunks for indexing.

Check Chunk Content Length

We inspect the length of the content in the first chunk using `len(all_splits[0].page_content)`. This should be close to the 1,000 characters we defined, minus any adjustments made by the splitter for logical breaks.

Inspect Metadata

Finally, we look at the metadata for the 11th chunk (`all_splits[10].metadata`). This metadata includes the `start_index`, showing where this chunk begins within the original document, which can be useful for tracing back to the full text.

Link of collab Notebook

5. Conclusion

In this tutorial, Splitting documents into smaller, manageable chunks is a critical step in creating an effective retrieval-augmented generation (RAG) system.

By leveraging the *RecursiveCharacterTextSplitter*, we can ensure that our documents are split in a way that maintains context and logical coherence.

This preparation sets the stage for embedding these chunks and using them effectively in our question-answering app.

Stay tuned for Day 5, where we will delve deeper into the indexing process and storing our data for efficient retrieval.