Notes on Building Production RAG over complex documents

compiled by D. Gueorguiev, 7/14/2025

# Introduction

Large Language Models (LLMs) are revolutionizing how users search for, interact with, and generate new content. Some recent stacks and toolkits around Retrieval-Augmented Generation (RAG) have emerged, enabling users to build applications such as chatbots using LLMs on their private data. However, while setting up naive RAG is straightforward, building production RAG is very challenging, especially as users scale to larger and more complex data sources. A classic example is a large number of PDFs with embedded tables. RAG is only as good as your data, and developers must carefully consider how to parse, ingest, and retrieve their data to successfully build RAG over complex documents. This Notes record the in-depth exploration of this entire process; you will get an overview of the process around building a RAG pipeline that can handle messy, complicated PDF documents. This includes implementing a parsing strategy for parsing a complex document with embedded objects. This consists of an indexing strategy to process these documents beyond simple chunking techniques. We will then explore various advanced retrieval algorithms to handle questions about the tabular and unstructured data and discuss their use cases and tradeoffs.

Knowledge Assistant Use Case:

Human wants to query a Knowledge Base and wants to get back an answer.

The types of question can be asked are various – it could be a question for specific details on specific issue / aspect / concept or it could be a question which can be answered only generally / in very broad terms, or it could be a question which would require detailed analysis and research before providing an answer.

The output could be structured output , short answer to question of the first type or research report for question of the third type.

A computer screen shot of a computer

Description automatically generated

A diagram of a data flow

Description automatically generated

Retrieval-Augmented Generation (RAG) contains of two main components –

* Data Parsing and Ingestion
* Data Querying

Data Parsing and Ingestion can be viewed as ETL applied to your unstructured data.

Being able to parse, process, and index that data is *the first step* toward building a RAG. *The second step* is querying relevant data using appropriate storage system like vector DB or SQL DB and you can treat access to this storage as API interface for the querying purposes. Querying will be done by the LLM. If we have more tools which we can use in order to obtain the final answer we can combine those with the storage facility.

Naïve RAG:

A diagram of a diagram

Description automatically generated

We have a tool which allows us to load some document (e.g. PyPDF) and then we do some sort of splitting from the loaded document – e.g. sentence or paragraph splitting with appropriate chunk size. We can choose also more elaborate splitting scheme such based on NLTK tokens or semantic similarity based splitting ([4], [6], [7]).

These text chunks are embedded and stored in the Vector DB for instance.

The Retrieval phase usually does top-k retrieval scheme. The retrieved data is injected into a prompt which already has context injected, we feed this prompt to an LLM and we get back an answer.

Naïve RAG tends to work well with simple questions over a simple small set of documents.

Naïve RAG is easy to explain and difficult to actually make it work.

Problems with Naïve RAG:

i) simple questions over complex data

ii) simple questions over multiple documents

iii) complex questions

The reasons for the failure modes i) - iii)

a) semantic dissonance and retrieval inefficiency

Naive RAG systems typically rely on basic similarity searches (vector and/or keyword-based) to retrieve relevant information. However, a direct match between the user's query and the data chunks doesn't guarantee the most relevant or complete information is retrieved, particularly with complex data.

This can lead to a semantic mismatch where the system misunderstands the user's intent or retrieves information that is only superficially related, resulting in inaccurate or incomplete answers.

b) limited retrieval scope (single shot)

Naive RAG typically performs a single retrieval attempt based on the user's initial query. If the answer is split across multiple documents or requires information that isn't directly addressed by the initial query terms, it might miss crucial pieces.

c) limited contextual understanding and memory

Naive RAG lacks the capability to deeply process and understand the nuances of a query or maintain context across turns in a conversation. This can lead to fragmented answers and an inability to connect related information scattered across different parts of the knowledge base.

For instance, if a query involves specialized terminology or acronyms, the system might fail to interpret them accurately, leading to incorrect retrieval. Furthermore, without contextual memory, the system cannot leverage previous interactions, making conversations feel disjointed and less effective for addressing evolving user needs.

d) suboptimal retrieval and ranking

Even if relevant documents are present, a simple RAG might not rank them highly enough for retrieval, especially when dealing with nuanced or domain-specific information.

e) incomplete answers and fragmentation

When retrieving multiple chunks, Naive RAG might struggle to synthesize all the relevant information into a coherent and comprehensive answer, potentially providing partial or fragmented responses.

f) conflicting information in the dataset

Naïve RAG do not handle gracefully conflicting information in the dataset

# References

[1] [Building Production RAG Over Complex Documents, Jerry Liu, LlamaIndex by Databricks, (youtube video) July 7, 2024](https://youtu.be/dI_TmTW9S4c?si=qP6wuZieU2SCZdc3)

[2] [The Big Book on MLOps and LLMOps, Databricks, 10/2023](https://github.com/dimitarpg13/rag_architectures_and_concepts/blob/main/docs/2023-10-EB-Big-Book-of-MLOps-2nd-Edition.pdf)

[3] PyPDF User Guide and API Reference: <https://pypdf.readthedocs.io/en/stable/>

[4] [Chunking(Text Splitting) Strategies- LLMs, Rahul, March 30, 2024, Medium](https://medium.com/@rahulpant.me/chunking-text-splitting-strategies-llms-579ab4ede2eb)

[5] the NLTK suite of libraries: <https://github.com/nltk/nltk>

[6] [LangChain NLTKTextSplitter API Reference](https://python.langchain.com/api_reference/text_splitters/nltk/langchain_text_splitters.nltk.NLTKTextSplitter.html)

[7] [LangChain NLTKTextSplitter Examples](https://lagnchain.readthedocs.io/en/latest/modules/indexes/text_splitters/examples/nltk.html)

[8] [Optimizing Retrieval Augmentation with Dynamic Top-K Tuning for Efficient Question Answering,](https://medium.com/@sauravjoshi23/optimizing-retrieval-augmentation-with-dynamic-top-k-tuning-for-efficient-question-answering-11961503d4ae)

[Saurav Joshi, Medium, Nov 3, 2023](https://medium.com/@sauravjoshi23/optimizing-retrieval-augmentation-with-dynamic-top-k-tuning-for-efficient-question-answering-11961503d4ae)

[9] [Adversarial Top-K Ranking, C. Suh et al, 2016](https://github.com/dimitarpg13/rag_architectures_and_concepts/blob/main/articles/Adversarial_Top-K_ranking_Suh_2016.pdf)

# Appendix

## Top-K Retrieval

Top-k retrieval refers to the process of selecting the k most relevant items from a larger set, based on a ranking or scoring system. This technique is widely used in various applications like web search, recommendation systems, and machine learning, where filtering a large dataset down to the most pertinent results is crucial.

Here's a more detailed breakdown:

***Key Concepts***:

*Ranking Function*: A function that assigns a score to each item in the dataset, indicating its relevance to a specific query or task.

*Top-k*: The top-k refers to the k items with the highest scores.

*Relevance*: The degree to which an item matches the user's query or need.

***Applications:***

Web Search:

Identifying the top k documents that are most relevant to a user's search query.

Recommendation Systems:

Suggesting the top k items (e.g., products, movies, music) that a user might be interested in.

Machine Learning:

Selecting the top k predicted labels for a classification task, or the top k nearest neighbors in a clustering algorithm.

Big Data Processing:

Filtering large datasets to extract the most significant data points.

Question Answering Systems:

Dynamically adjusting the number of documents retrieved based on the query complexity.

***Challenges***:

Scalability:

Handling massive datasets efficiently is a major challenge, requiring optimized algorithms and data structures.

Balance between Relevance and Diversity:

Sometimes, it's important to not only retrieve the highest-scoring items but also ensure a degree of diversity in the results.

Computational Cost:

Evaluating the ranking function for all items can be expensive, especially with large datasets.

Common Techniques:

i) Inverted Index:

A data structure used in information retrieval to efficiently locate documents containing specific terms.

ii) WAND Algorithm:

A two-level retrieval process that uses upper bound scores to skip irrelevant documents, improving efficiency.

iii) Adaptive RAG:

A technique that dynamically adjusts the number of documents retrieved based on feedback from the language model.

In essence, top-k retrieval is a fundamental building block for many information access and data analysis tasks, requiring careful consideration of efficiency, scalability, and the balance between relevance and other factors.