[Week 4] Retrieval Augmented Generation

ETMI5: Explain to Me in 5

In this week's content, we will do an in-depth exploration of Retrieval Augmented Generation (RAG), an AI framework that enhances the capabilities of Large Language Models by integrating real-time, contextually relevant information from external sources during the response generation process. It addresses the limitations of LLMs, such as inconsistency and lack of domain-specific knowledge, hence reducing the risk of generating incorrect or hallucinated responses.

RAG operates in three key phases: ingestion, retrieval, and synthesis. In the ingestion phase, documents are segmented into smaller, manageable chunks, which are then transformed into embeddings and stored in an index for efficient retrieval. The retrieval phase involves leveraging the index to retrieve the top-k relevant documents based on similarity metrics when a user query is received. Finally, in the synthesis phase, the LLM utilizes the retrieved information along with its internal training data to formulate accurate responses to user queries.

We will discuss the history of RAG and then delve into the key components of RAG, including ingestion, retrieval, and synthesis, providing detailed insights into each phase's processes and strategies for improvement. We will also go over various challenges associated with RAG, such as data ingestion complexity, efficient embedding, and fine-tuning for generalization and propose solutions to each of them.

What is RAG? (Recap)

Retrieval Augmented Generation (RAG) is an AI framework that enhances the quality of responses generated by LLMs by incorporating up-to-date and contextually relevant information from external sources during the generation process. It addresses the inconsistency and lack of domain-specific knowledge in LLMs, reducing the chances of hallucinations or incorrect responses. RAG involves two phases: retrieval, where relevant information is searched and retrieved, and content generation, where the LLM synthesizes an answer based on the retrieved information and its internal training data. This approach improves accuracy, allows source verification, and reduces the need for continuous model retraining.

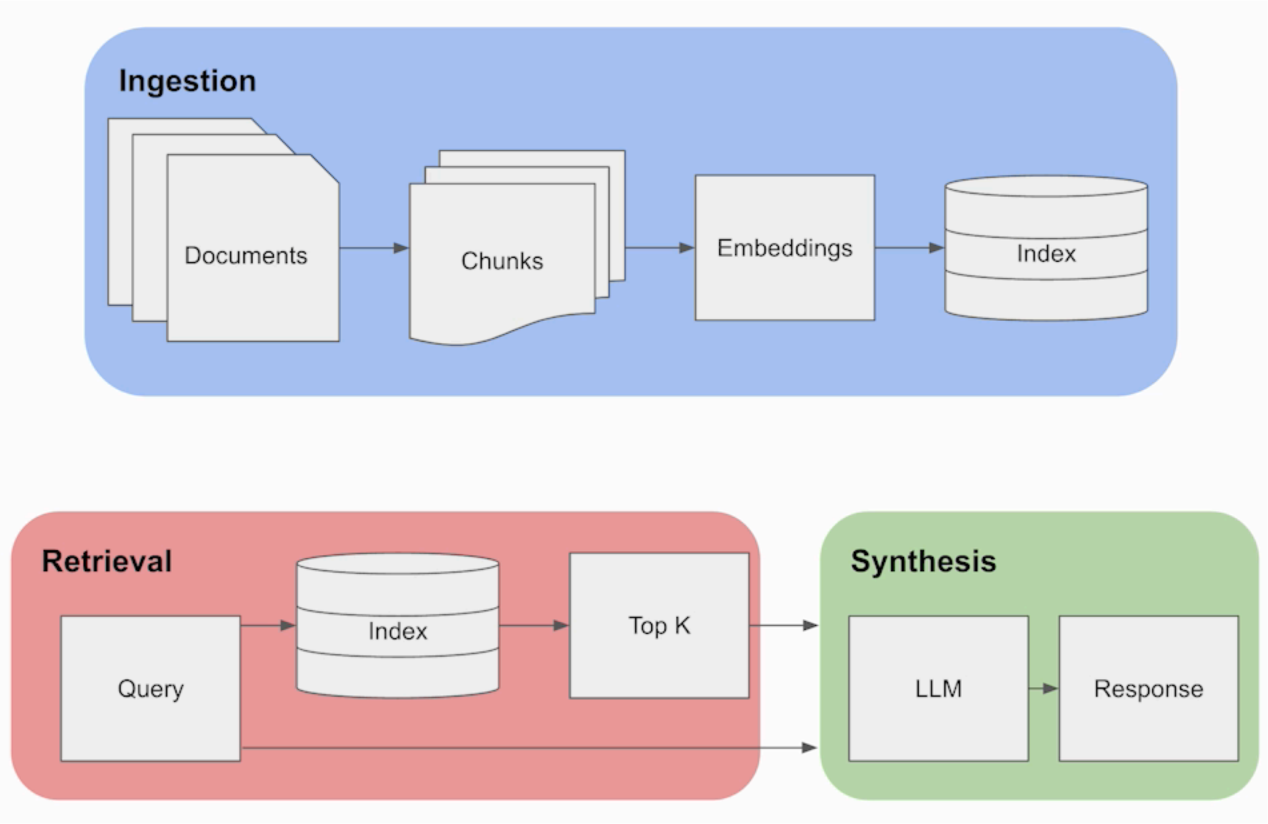


Image Source: https://www.deeplearning.ai/short-courses/langchain-for-llm-application-development/

The diagram above outlines the fundamental RAG pipeline, consisting of three key components:

- **Ingestion:**
  Documents undergo segmentation into chunks, and embeddings are generated from these chunks, subsequently stored in an index.
  Chunks are essential for pinpointing the relevant information in response to a given query, resembling a standard retrieval approach.
- **Retrieval:**
  Leveraging the index of embeddings, the system retrieves the top-k documents when a query is received, based on the similarity of embeddings.
- **Synthesis:**
  Examining the chunks as contextual information, the LLM utilizes this knowledge to formulate accurate responses.

💡Unlike previous methods for domain adaptation, it's important to highlight that RAG doesn't necessitate any model training whatsoever. It can be readily applied without the need for training when specific domain data is provided.

History

RAG, or Retrieval-Augmented Generation, made its debut in this paper by Meta. The idea came about in response to the limitations observed in large pre-trained language models regarding their ability to access and manipulate knowledge effectively.

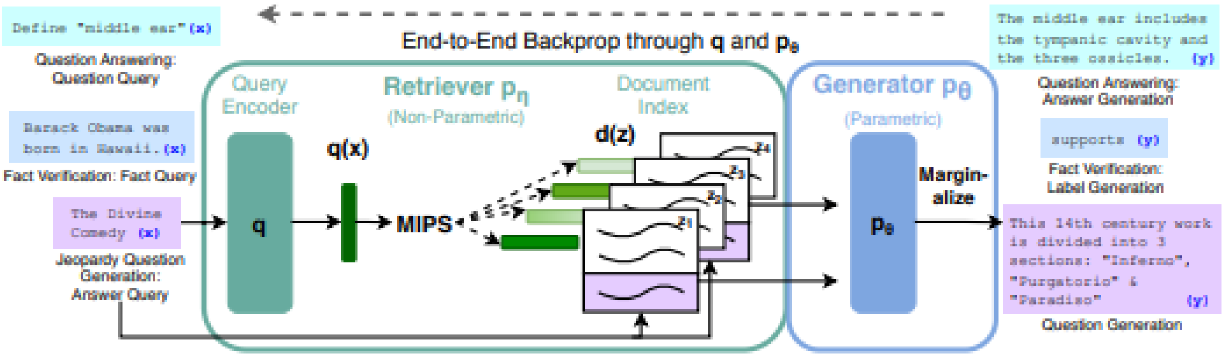

Define "middle ear" (x)
Question Answering: Question Query
Barack Obama was born in Hawaii. (x)
Fact Verification: Fact Query
The Divine Comedy (x)
Jeopardy Question Generation: Answer Query
End-to-End Backprop through q and p_θ
Query Encoder
Retriever p_η
(Non-Parametric)
Document Index
q
q(x)
MIPS
d(z)
z_4
z_3
z_2
z_1
Generator p_θ
(Parametric)
p_θ
Margin-alize
The middle ear includes the tympanic cavity and the three ossicles. (y)
Question Answering: Answer Generation
supports (y)
Fact Verification: Label Generation
This 14th century work is divided into 3 sections: "Inferno", "Purgatorio" & "Paradiso" (y)
Question Generation

Image Source: https://arxiv.org/pdf/2005.11401.pdf

Below is a short summary of how the authors introduce the problem and provide a solution:

RAG came about because, even though big language models were good at remembering facts and performing specific tasks, they struggled when it came to precisely using and manipulating that knowledge. This became evident in tasks heavy on knowledge, where other specialized models outperformed them. The authors identified challenges in existing models, such as difficulty explaining decisions and keeping up with real-world changes. Before RAG, there were promising results with hybrid models that mixed both parametric and non-parametric memories. Examples like REALM and ORQA combined masked language models with a retriever, showing positive outcomes in this direction.

Then, along came RAG, a game-changer in the form of a flexible fine-tuning method for retrieval-augmented generation. RAG combined pre-trained parametric memory (like a seq2seq model) with non-parametric memory from a dense vector index of Wikipedia, accessed through a pre-trained neural retriever like Dense Passage Retriever (DPR). RAG models aimed to enhance pre-trained, parametric-memory generation models by combining them with non-parametric memory through fine-tuning. The seq2seq model in RAG used latent documents retrieved by the neural retriever, creating a model trained end-to-end. Training involved fine-tuning on any seq2seq task, learning both the generator and retriever. Latent documents were then handled using a top-K approximation, either per output or per token.

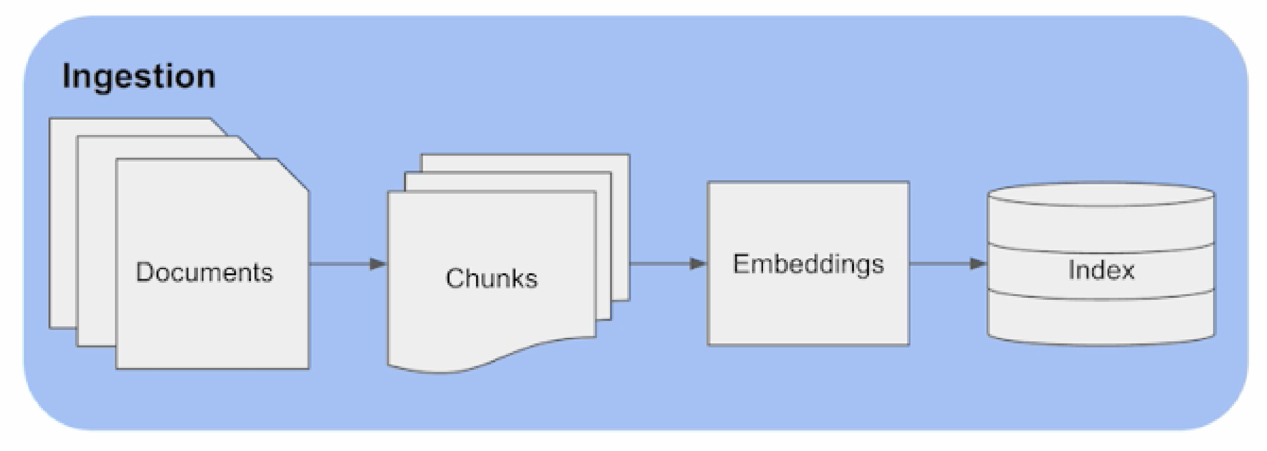
RAG's main significance was moving away from past approaches that proposed adding non-parametric memory to systems. Instead, RAG explored a new approach where both parametric and non-parametric memory components were pre-trained and filled with lots of knowledge. In experiments, RAG proved its worth by achieving top-notch results in open-domain question answering and surpassing previous models in fact verification and knowledge-intensive generation. Another win for RAG was showing it could adapt, allowing the non-parametric memory to be swapped out and updated to keep the model's knowledge fresh in a changing world.

Key Components

As mentioned earlier, the key elements of RAG involve the processes of ingestion, retrieval, and synthesis. Now, let's delve deeper into each of these components.

Ingestion

In RAG, the ingestion process refers to the handling and preparation of data before it is utilized by the model for generating responses.

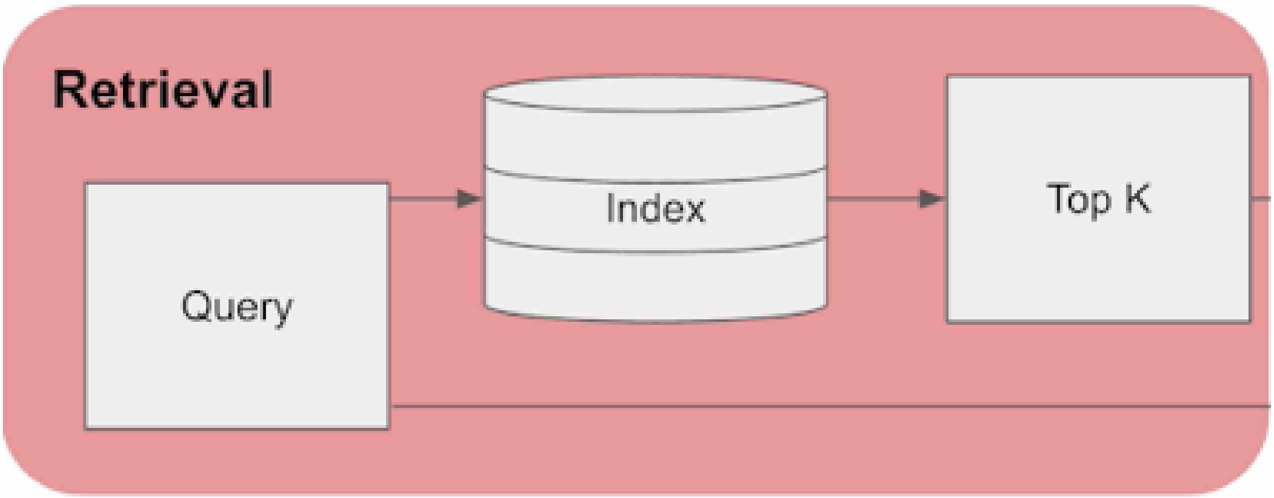


This process involves 3 key steps:

- **Chunking: B**reaking down input text into smaller, more manageable segments or chunks. This can be based on size, sentences, or other natural divisions within the text. We will dig deeper into chunking strategies in the next sections. As an example, consider a comprehensive article on the Renaissance. The chunking process involves breaking down the article into manageable segments based on natural breaks, such as paragraphs or distinct historical periods (e.g., Early Renaissance, High Renaissance). Each of these segments becomes a chunk, enabling focused analysis by the language model.
- **Embedding**: Transforming the text or chunks into a vector format that captures essential qualities in a computationally friendly way. This step is crucial for efficient processing by the language model. Following from the previous example- once the article segments are identified, the embedding process transforms the content of each chunk into a vector format. For instance, the section on the High Renaissance could be embedded into a vector that captures key artistic, cultural, and historical aspects. This vector representation enhances the model's ability to understand and process the nuanced information within the chunk.
- **Indexing:** Organizing the embedded data in a structured format optimized for quick and efficient retrieval. This often involves creating a vector representation for each document and storing these vectors in a searchable format, such as a vector database or search engine. In the example we discussed- The indexed database is created by organizing these vector representations of historical events. Each chunk, now represented as a vector, is indexed for efficient retrieval. When a user queries about a specific aspect of the Renaissance, the indexing enables the quick identification and retrieval of the most relevant chunks, providing contextually rich responses.

Retrieval

The retrieval component involves the following steps:



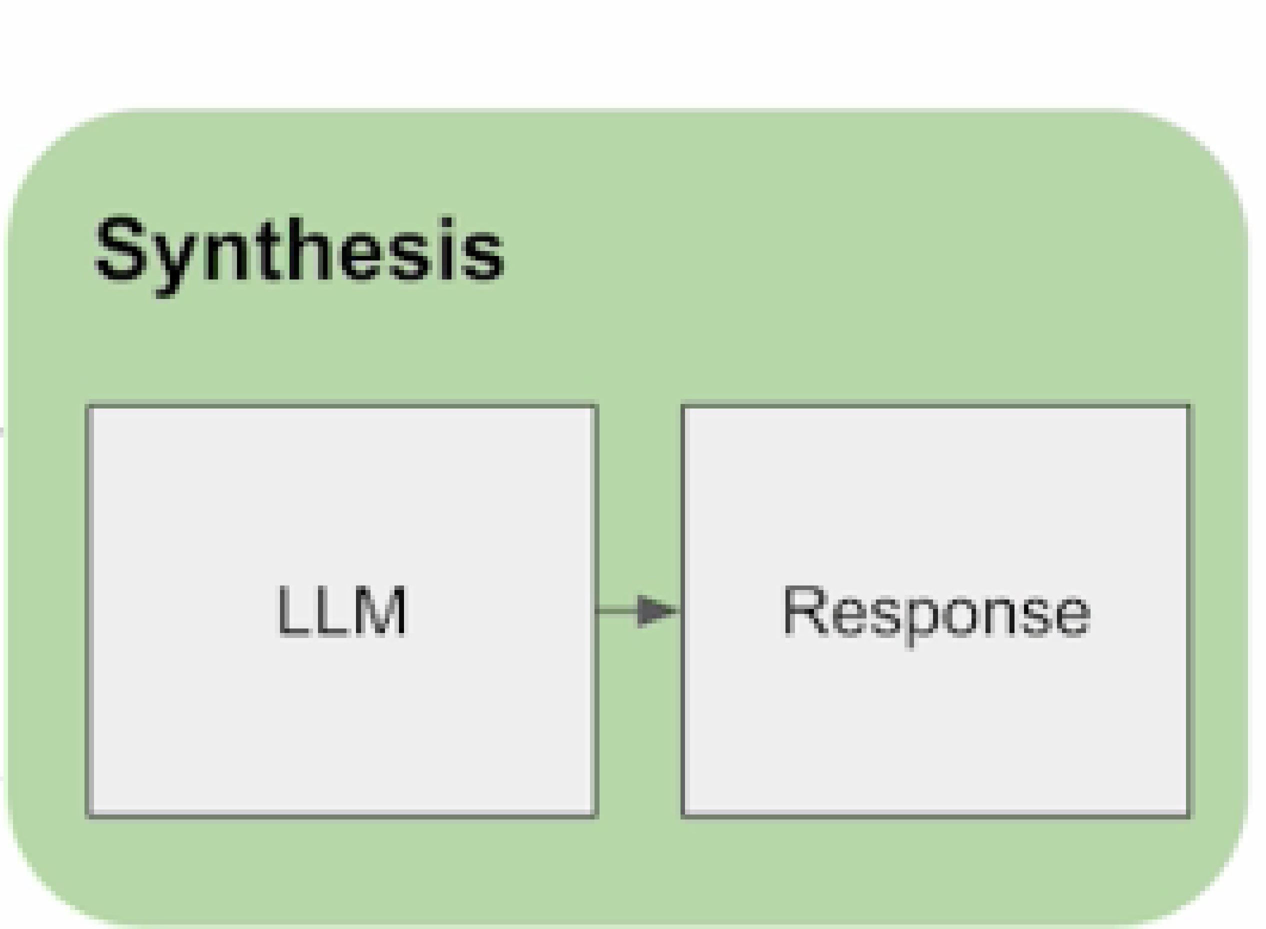
- **User Query:** A user poses a natural language query to the LLM. For instance, let’s say we’ve completed the ingestion process for renaissance articles as explained in the above method and a user poses a query, "Tell me about the Renaissance period.”

- **Query Conversion:** The query is sent to an embedding model, which converts the natural language query into a numeric format, creating an embedding or vector representation. The embedding model is the same as the model used to embed articles in the ingestion phase.
- **Vector Comparison:** The numeric vectors of the query are compared to vectors in a index of a knowledge base created in the previous phase. This involves measuring similarity or distance metrics between the query vector and vectors stored in the index (often cosine similarity).
- **Top-K Retrieval:** The system then retrieves the top-K documents or passages from the knowledge base that have the highest similarity to the query vector. This step involves selecting a predefined number (K) of the most relevant documents based on the vector similarities. These embeddings may include information about different aspects of the Renaissance.
- **Data Retrieval:** The system retrieves the actual content or data from the selected top-K documents in the knowledge base. This content is typically in human-readable form, representing relevant information related to the user's query.

Therefore, at the end of the retrieval phase, the LLM has access to relevant context regarding the segments of the knowledge base that hold utmost relevance to the user's query. In this example, the retrieval process ensures that the user receives a well-informed response about the Renaissance, drawing on historical documents stored in the knowledge base to provide contextually rich information.

Synthesis

The Synthesis phase is very similar to regular LLM generation, except that now the LLM has access to additional context from the knowledge base. The LLM presents the final answer to the user, combining its own language generation with information retrieved from the knowledge base. The response may include references to specific documents or historical sources.



RAG Challenges

Although RAG seems to be a very straightforward way to integrate LLMs with knowledge, there are still the below mentioned open research and application challenges with RAG.

- **Data Ingestion Complexity:** Dealing with the complexity of ingesting extensive knowledge bases involves overcoming engineering challenges. For instance, parallelizing requests effectively, managing retry mechanisms, and scaling infrastructure are critical considerations. Imagine ingesting large volumes of diverse data sources, such as scientific articles, and ensuring efficient processing for subsequent retrieval and generation tasks.
- **Efficient Embedding:** Ensuring the efficient embedding of large datasets poses challenges like addressing rate limits, implementing robust retry logic, and managing self-hosted models. Consider the scenario where an AI system needs to embed a vast collection of news articles, requiring strategies to handle changing data, syncing mechanisms, and optimizing embedding costs.
- **Vector Database Considerations:** Storing data in a vector database introduces considerations such as understanding compute resources, monitoring, sharding, and addressing potential bottlenecks. Think about the challenges involved in maintaining a vector database for a diverse range of documents, each with varying levels of complexity and importance.
- **Fine-Tuning and Generalization:** Fine-tuning RAG models for specific tasks while ensuring generalization across diverse knowledge-intensive NLP tasks is challenging. For instance, achieving optimal performance in question-answering tasks might require different fine-tuning approaches compared to tasks involving creative language generation, requiring careful balance.
- **Hybrid Parametric and Non-Parametric Memory: I**ntegrating parametric and non-parametric memory components in models like RAG presents challenges related to knowledge revision, interpretability, and avoiding hallucinations. Consider the difficulty in ensuring that a language model combines its pre-trained knowledge with dynamically retrieved information, avoiding inaccuracies and maintaining coherence.
- **Knowledge Update Mechanisms:** Developing mechanisms to update non-parametric memory as real-world knowledge evolves is crucial. Imagine a scenario where RAG models need to adapt to changing information in domains like medicine, where new research findings and treatments continually emerge, requiring timely updates for accurate responses.

Improving RAG components (Ingestion)

1. Better Chunking Strategies

In the context of enhancing the Ingestion process for the RAG components, adopting advanced chunking strategies is necessary for efficient handling of textual data. In a simple RAG pipeline, a fixed strategy is adopted, i.e., a fixed number of words or characters form a single chunk.

Considering the complexities involved in large datasets, the following strategies are being used recently:

- **Content-Based Chunking:** Breaks down text based on meaning and sentence structure using techniques like part-of-speech tagging or syntactic parsing. This preserves the sense and coherence of the text. However, one consideration of this chunking is it requires additional computational resources and algorithmic complexity.
- **Sentence Chunking:** Involves breaking text into complete and grammatically correct sentences using sentence boundary recognition or speech segment. Maintains the unity and completeness of the text but can generate chunks of varying sizes, lacking homogeneity.
- **Recursive Chunking:** Splits text into chunks of different levels, creating a hierarchical and flexible structure. Offers greater granularity and variety in text, but managing and indexing these chunks involves increased complexity.

2. Better Indexing Strategies

Improved indexing allows for more efficient search and retrieval of information. When chunks of data are properly indexed, it becomes easier to locate and retrieve specific pieces of information quickly. Some improved strategies include:

- **Detailed Indexing:** Chunks through sub-parts (e.g., sentences) and assigns each chunk an identifier based on its position and a feature vector based on content. Provides specific context and accuracy but requires more memory and processing time.
- **Question-Based Indexing:** Chunks through knowledge domains (e.g., topics) and assigns each chunk an identifier based on its category and a vector of characteristics based on relevance. Aligns directly with user requests, enhancing efficiency, but may result in information loss and lower accuracy.
- **Optimized Indexing with Chunk Summaries:** Generates a summary for each chunk using extraction or compression techniques. Assigns an identifier based on the summary and a feature vector based on similarity.

Provides greater synthesis and variety but demands complexity in generating and comparing summaries.

Improving RAG components (Retrieval)

1. Hypothetical Questions and HyDE:

The introduction of hypothetical questions involves generating a question for each chunk, embedding these questions in vectors, and performing a query search against this index of question vectors. This enhances search quality due to higher semantic similarity between queries and hypothetical questions compared to actual chunks. Conversely, HyDE (Hypothetical Response Extraction) involves generating a hypothetical response given the query, enhancing search quality by leveraging the vector representation of the query and its hypothetical response.

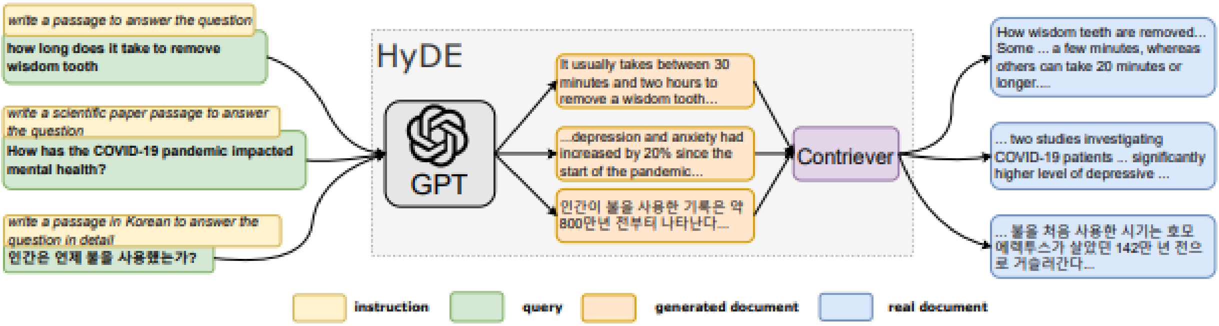

write a passage to answer the question
how long does it take to remove wisdom tooth
write a scientific paper passage to answer the question
How has the COVID-19 pandemic impacted mental health?
write a passage in Korean to answer the question in detail
인간은 언제 불을 사용했는가?
HyDE
GPT
It usually takes between 30 minutes and two hours to remove a wisdom tooth...
...depression and anxiety had increased by 20% since the start of the pandemic...
인간이 불을 사용한 기록은 약 800만년 전부터 나타난다...
Contriever
How wisdom teeth are removed... Some ... a few minutes, whereas others can take 20 minutes or longer....
... two studies investigating COVID-19 patients ... significantly higher level of depressive ...
... 불을 처음 사용한 시기는 호모 에렉투스가 살았던 142만 년 전으로 거슬러간다...
instruction
query
generated document
real document

Image Source: https://arxiv.org/pdf/2212.10496.pdf

2. Context Enrichment:

The strategy here aims for smaller chunk retrieval for improved search quality while incorporating surrounding context for reasoning by the Language Model. Two options can explored:

- Sentence Window Retrieval: Embedding each sentence in a document separately to achieve high accuracy in the cosine distance search between the query and the context. After retrieving the most relevant single sentence, a context window is extended by including a specified number of sentences before and after the retrieved sentence. This extended context is then sent to the LLM for reasoning upon the provided query. The goal is to enhance the LLM's understanding of the context surrounding the retrieved sentence, enabling more informed responses.

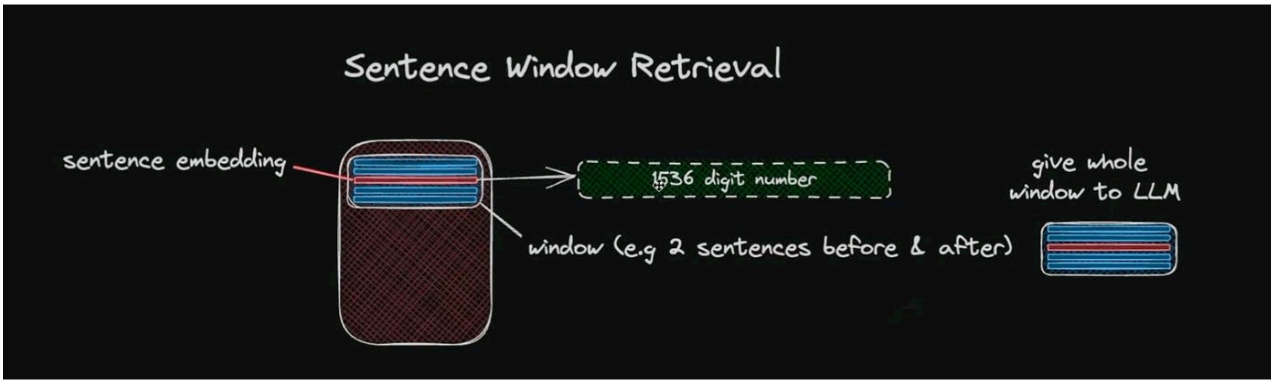

Sentence Window Retrieval
sentence embedding
1536 digit number
give whole window to LLM
window (e.g 2 sentences before & after)

Image Source: https://medium.com/@shivansh.kaushik/advanced-text-retrieval-with-elasticsearch-llamaindex-sentence-window-retrieval-cb5ea720aa44

- Auto-Merging Retriever: In this approach, documents are initially split into smaller child chunks, each referring to a larger parent chunk. During retrieval, smaller chunks are fetched first. If, among the top retrieved chunks, more than a specified number are linked to the same parent node (larger chunk), the context fed to the LLM is replaced by this parent node. This process can be likened to automatically merging several retrieved chunks into a larger parent chunk, hence the name "auto-merging retriever." The method aims to capture both granularity and context, contributing to more comprehensive and coherent responses from the LLM.

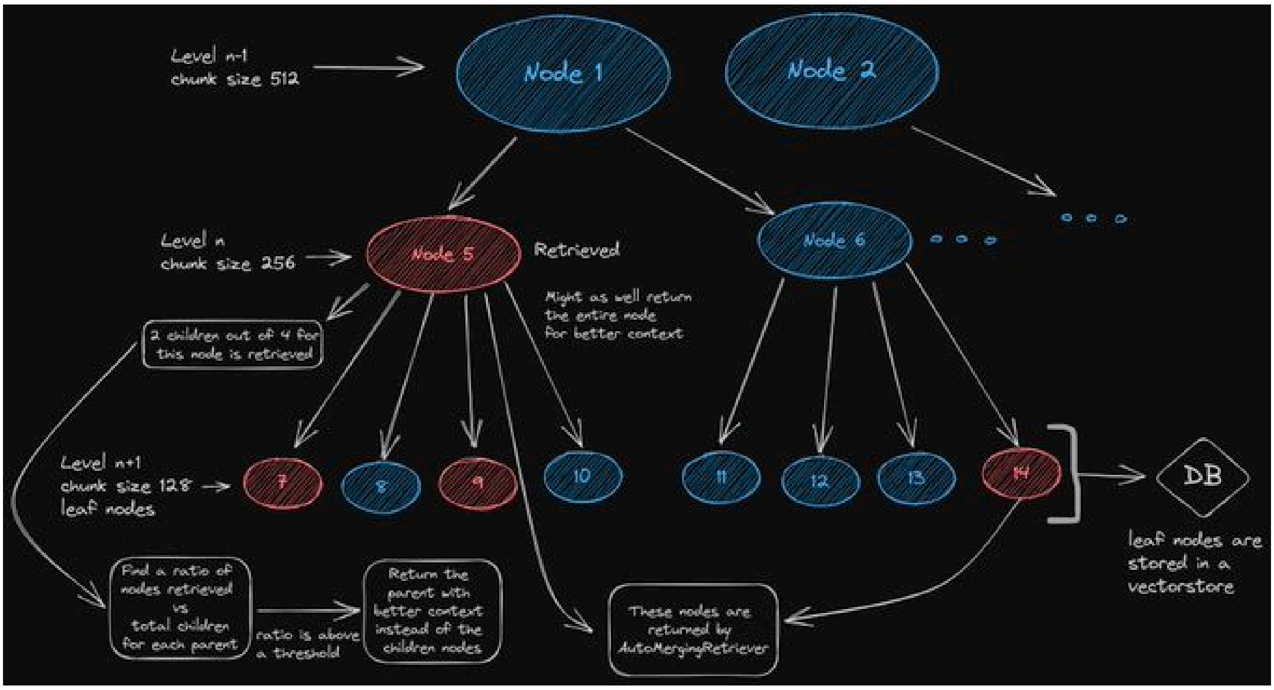

Level n-1
chunk size 512
Node 1
Node 2
Level n
chunk size 256
Node 5
Retrieved
Node 6
Might as well return the entire node for better context
2 children out of 4 for this node is retrieved
Level n+1
chunk size 128
leaf nodes
7
8
9
10
11
12
13
14
DB
leaf nodes are stored in a vectorstore
Find a ratio of nodes retrieved vs total children for each parent
ratio is above a threshold
Return the parent with better context instead of the children nodes
These nodes are returned by AutoMergingRetriever

Image Source: https://twitter.com/clusteredbytes

3. Fusion Retrieval or Hybrid Search:

This strategy integrates conventional keyword-based search approaches with contemporary semantic search techniques. By incorporating diverse algorithms like tf-idf (term frequency–inverse document frequency) or BM25 alongside vector-based search, RAG systems can harness the benefits of both semantic relevance and keyword matching, resulting in more thorough and inclusive search outcomes.

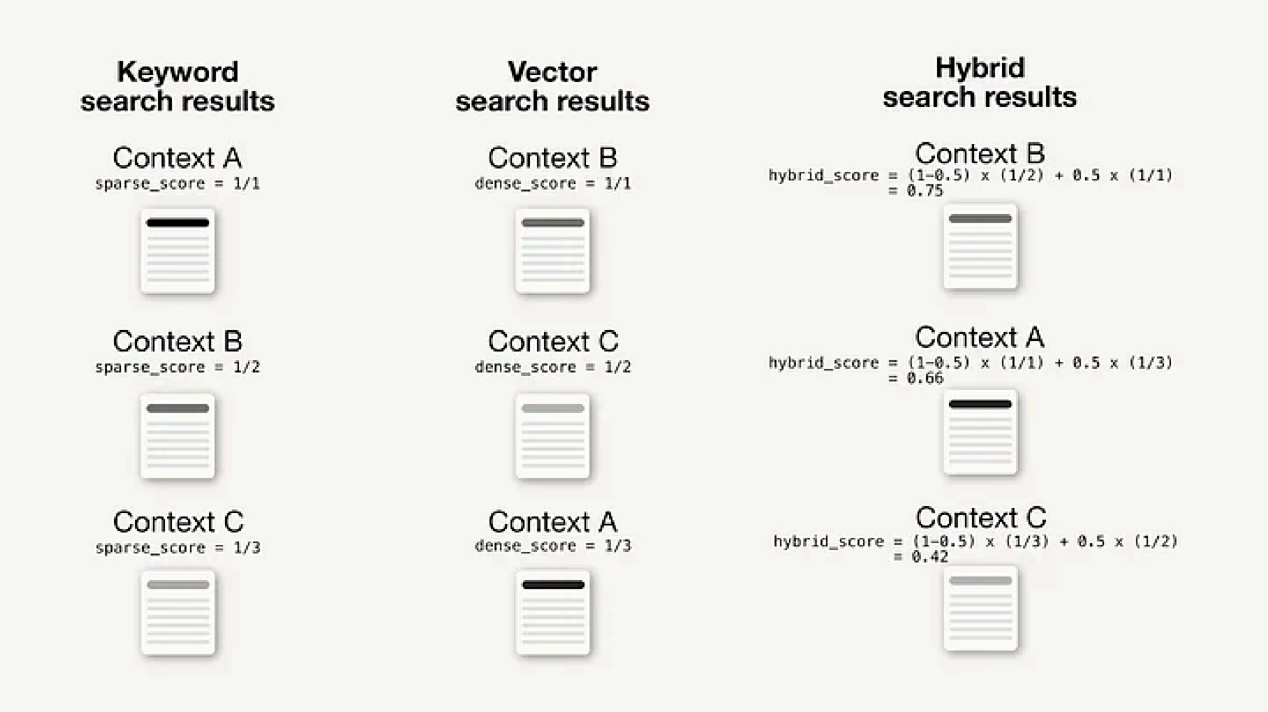


Image Source: https://towardsdatascience.com/improving-retrieval-performance-in-rag-pipelines-with-hybrid-search-c75203c2f2f5

4. Reranking & Filtering:

Post-retrieval refinement is performed through filtering, reranking, or transformations. LlamaIndex provides various Postprocessors, allowing the filtering of results based on similarity score, keywords, metadata, or reranking with models like LLMs or sentence-transformer cross-encoders. This step precedes the final presentation of retrieved context to the LLM for answer generation.

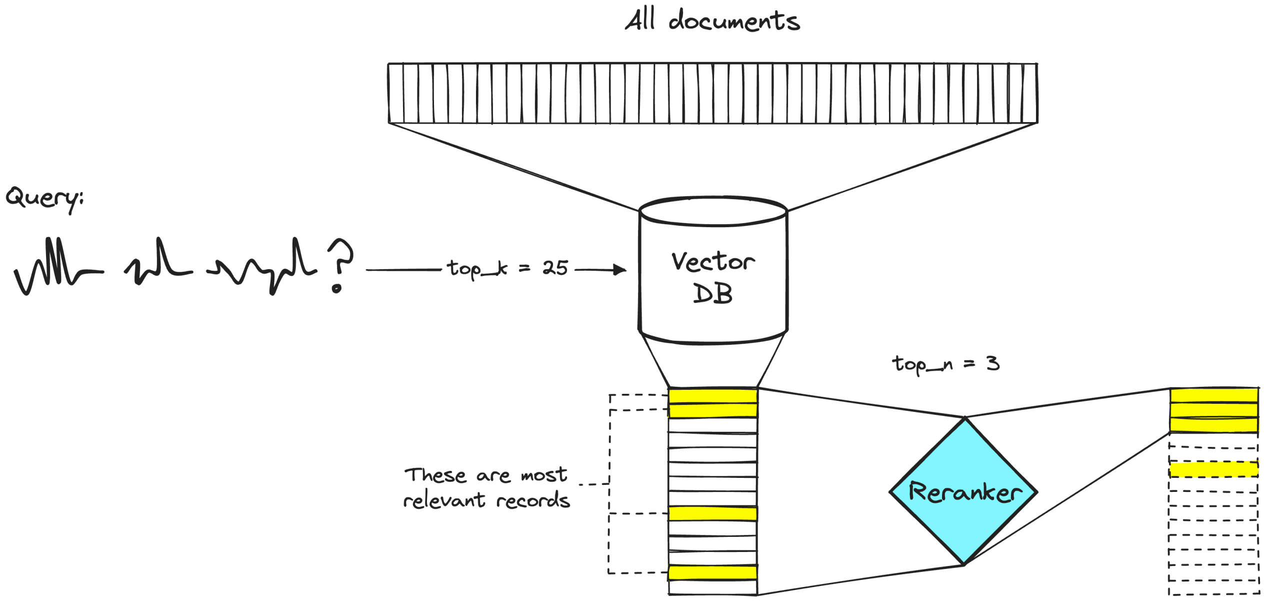


Image Source: https://www.pinecone.io/learn/series/rag/rerankers/

4. Query Transformations and Routing [Source]

Query transformation methods enhance retrieval by breaking down complex queries into sub-questions (Expansion) and improving poorly worded queries through re-writing. While dynamic Query Routing optimizes data retrieval in diverse sources. The below are popular approaches

Query Transformations

- **Query Expansion***:* Query expansion decomposes the input into sub-questions, each of which is a more narrow retrieval challenge. For example, a question about physics can be stepped-back into a question (and LLM-generated answer) about the physical principles behind the user query.
- **Query Re-writing**: Addressing poorly framed or worded user queries, the Rewrite-Retrieve-Read approach involves rephrasing questions to enhance retrieval effectiveness. The method is explained in detail in the paper.
- Query Compression: In scenarios where a user question follows a broader chat conversation, the full conversational context may be necessary to answer the question. Query compression is utilized to condense chat history into a final question for retrieval.

Query Routing

- **Dynamic Query Routing**: The question of where the data resides is crucial in RAG, especially in production settings with diverse data-stores. Dynamic query routing, supported by LLMs, efficiently directs incoming queries to the appropriate datastores. This dynamic routing adapts to different sources and optimizes the retrieval process.

Improving RAG components (Generation)

The most straightforward method for LLM generation involves concatenating all the relevant context pieces, surpassing a predefined relevance threshold, and presenting them along with the query to the LLM in a single instance. However, more advanced alternatives exist, necessitating multiple calls to the LLM to iteratively enhance the retrieved context, ultimately leading to the generation of a more refined and improved answer. Some methods are illustrated below.

1. Response Synthesis Approaches:

Involves 3 steps

- **Iterative Refinement:** Refine the answer by sending retrieved context to the Language Model chunk by chunk.
- **Summarization:** Summarize the retrieved context to fit into the prompt and generate a concise answer.
- **Multiple Answers and Concatenation:** Generate multiple answers based on different context chunks and then concatenate or summarize them.

2. Encoder and LLM Fine-Tuning:

This approach involves the fine-tuning the LLM models within our RAG pipeline.

- **Encoder Fine-Tuning:** Fine-tune the Transformer Encoder for better embeddings quality and context retrieval.
- **Ranker Fine-Tuning:** Use a cross-encoder for reranking retrieved results, especially if there's a lack of trust in the base Encoder.
- **RA-DIT Technique:** Use a technique like RA-DIT to tune both the LLM and the Retriever on triplets of query, context, and answer.

Read/Watch These Resources (Optional)

- Building Production Ready RAG Applications: https://www.youtube.com/watch?v=TRjq7t2Ms5I
- Amazon article on RAG- https://docs.aws.amazon.com/sagemaker/latest/dg/jumpstart-foundation-models-customize-rag.html
- Huggingface tools for RAG- https://huggingface.co/docs/transformers/model_doc/rag
- 12 RAG Pain Points and Proposed Solutions- https://towardsdatascience.com/12-rag-pain-points-and-proposed-solutions-43709939a28c

Read These Papers (Optional)

Retrieval-Augmented Generation for Large Language Models: A Survey

Seven Failure Points When Engineering a Retrieval Augmented Generation System