Advanced RAG Techniques: an Illustrated Overview

towardsai.net/p/machine-learning/advanced-rag-techniques-an-illustrated-overview

17 de dezembro de 2023

Last Updated on December 21, 2023 by Editorial Team

Author(s): IVAN ILIN

Groningen, Martinitoren, where the article was composed in the peace of the Noorderplatsoen

A comprehensive study of the advanced retrieval augmented generation techniques and algorithms, systemising various approaches. The article comes with a collection of links in my knowledge base referencing various implementations and studies mentioned.

Since the goal of the post is to make an overview & explanation of avaliable RAG algorithms and techniques, I won't dive into implementations details in code, just referencing them and leaving it to the vast documentation & tutorials available.

Intro

If you are familiar with the RAG concept, please skip to the Advanced RAG part.

Retrieval Augmented Generation, aka RAG, provides LLMs with the information retrieved from some data source to ground its generated answer on. **Basically RAG is Search + LLM prompting**, where your ask the model to answer the query provided the information found with the search algorithm as a context. Both the query and the retrieved context are injected into the prompt that is sent to the LLM.

RAG is the most popular architecture of the LLM based systems in 2023. There are many products build almost solely on RAG — from Question Answering services combining web search engines with LLMs to hundreds of chat-with-your-data apps.

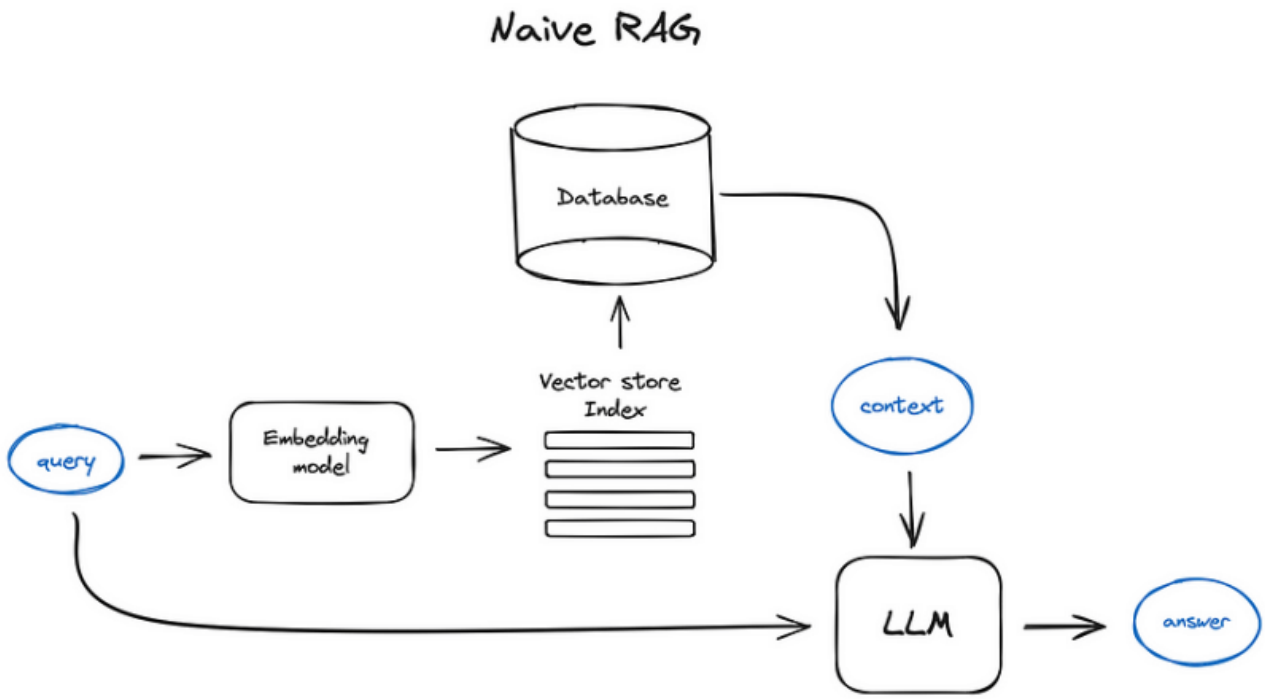
Even the **vector search** area got pumped by that hype although embedding based search engines were made with faiss back in 2019. Vector database startups like chroma, weavaite.io and pinecone have been built upon existing open source search indices — mainly faiss and nmslib — and added an extra storage for the input texts plus some other tooling lately.

There are two most prominent open source libraries for LLM-based pipelines & applications — LangChain and LlamaIndex, founded with a month difference in October and November 2022, respectfully, inspired by the ChatGPT launch and having gained massive adoption in 2023.

> *The purpose of this article is to systemise the key advanced RAG techniques with references to their implementations — mostly in the LlamaIndex — in order to facilitate other developers' dive into the technology.*
>
> *The problem is that most of the tutorials cherry-pick one or several techniques and explain in details how to implement them rather than decribing the full variety of the avaliable tools .*
>
> *Another thing is that both LlamaIndex and LangChian are amazing open source projects, developing at such a pace that their documentation is already thicker than a machine learning textbook in 2016.*

Naive RAG

The starting point of the RAG pipeline in this article would be a corpus of text documents — we skip everything before that point, leaving it to the amazing open source data loaders connecting to any imaginable source from Youtube to Notion.



A scheme by author, as well all the schemes further in the text

Vanilla RAG case in brief looks the following way: you split your texts into chunks, then you embed these chunks into vectors with some Transformer Encoder model, you put all those vectors into an index and finally you create a prompt for an LLM that tells the model to answers user's query given the context we found on the search step.
In the runtime we vectorise user's query with the same Encoder model and then execute search of this query vector against the index, find the top-k results, retrieve the corresponding text chunks from our database and feed them into the LLM prompt as context.

The prompt can look like that:

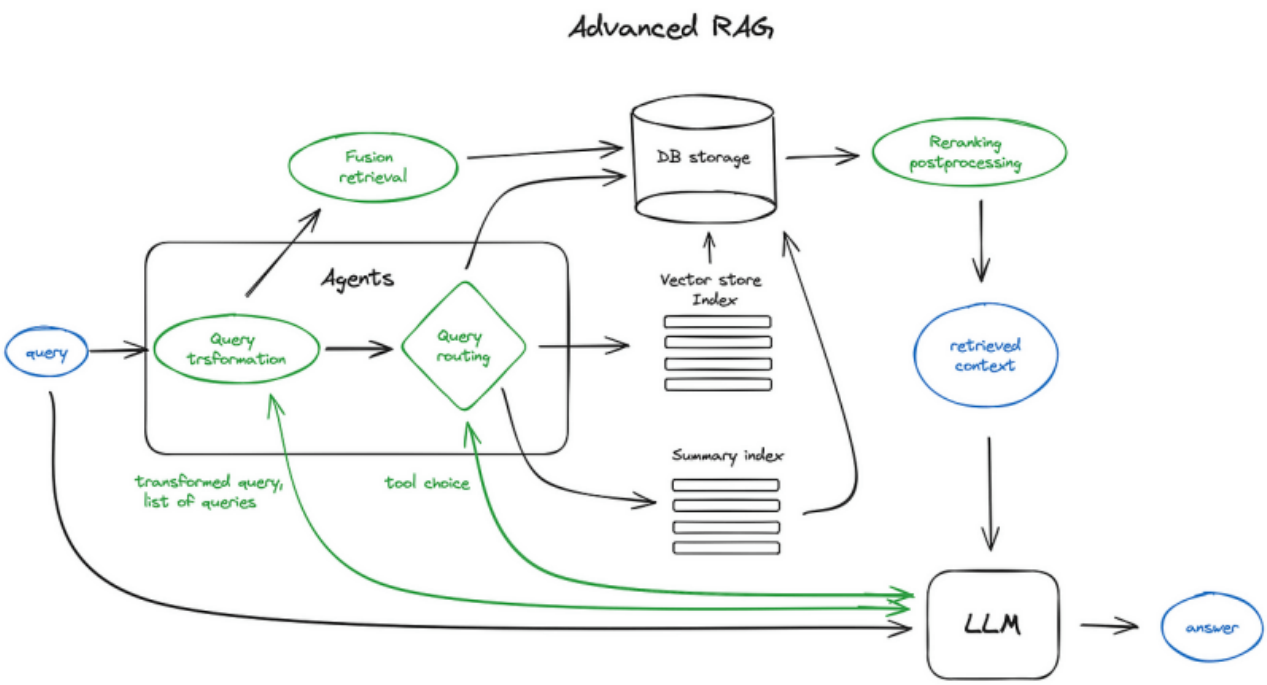
an example of a RAG prompt

Prompt engineering is the cheapest thing you can try to improve your RAG pipeline. Make sure you've checked a quite comprehensive OpenAI prompt engineering guide.

Obviously despite OpenAI being the market leader as an LLM provider there is a number of alternatives such as Claude from Anthropic, recent trendy smaller but very capable models like Mixtral form Mistral, Phi-2 from Microsoft and many open source options like Llama2, OpenLLaMA, Falcon, so you have a choice of the brain for your RAG pipeline.

Advanced RAG

Now we'll dive into the overview of the advanced RAG techniques.
Here is a scheme depicting core steps and algorithms involved.
Some logic loops and complex multistep agentic behaviours are omitted to keep the scheme readable.



Some key components of an advanced RAG architecture. It's more a choice of available instruments than a blueprint.

The green elements on the scheme are the core RAG techniques discussed further, the blue ones are texts. Not all the advanced RAG ideas are easily visualised on a single scheme, for example, various context enlarging approaches are omitted — we'll dive into that on the way.

1. Chunking & vectorisation

First of all we want to create an index of vectors, representing our document contents and then in the runtime to search for the least cosine distance between all these vectors and the query vector which corresponds to the closest semantic meaning.

1.1 Chunking

Transformer models have fixed input sequence length and even if the input context window is large, the vector of a sentence or a few better represents their semantic meaning than a vector averaged over a few pages of text (depends on the model too, but true in general), so **chunk your data** — split the initial documents in chunks of some size without loosing their meaning (splitting your text in sentences or in paragraphs, not cutting a single sentence in two parts). There are various text splitter implementations capable of this task.

The size of the chunk is a parameter to think of — it depends on the embedding model you use and its capacity in tokens, standard transformer Encoder models like BERT-based Sentence Transformers take 512 tokens at most, OpenAI ada-002 is capable of handling longer sequences like 8191 tokens, but **the compromise here is enough context for the LLM to reason upon vs specific enough text embedding in order to efficiently execute search upon**. Here you can find a research illustrating

chunk size selection concerns. In LlamaIndex this is covered by the NodeParser class with some advanced options as defining your own text splitter, metadata, nodes / chunks relations, etc.

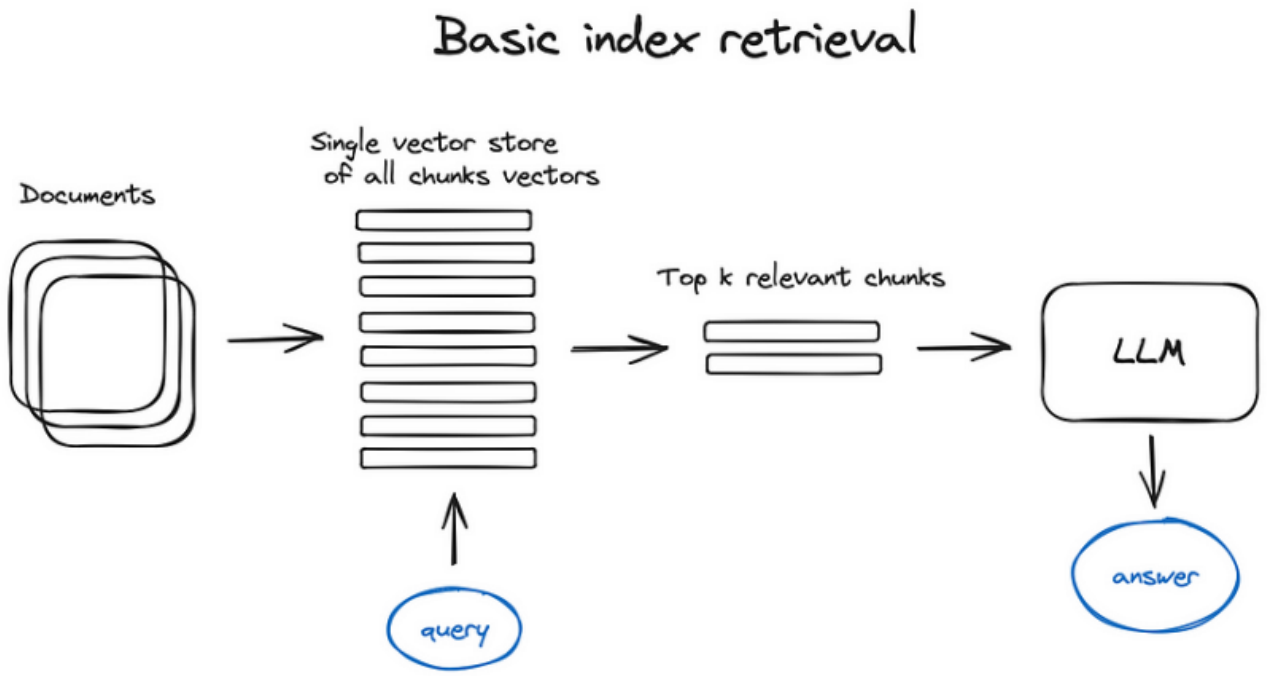
1.2 Vectorisation

The next step is to choose a **model to embed our chunks** — there are quite some options, I go with the **search optimised models** like bge-large or E5 embeddings family — just check the MTEB leaderboard for the latest updates.

For an end2end implementation of the chunking & vectorisation step check an example of a full data ingestion pipeline in LlamaIndex.

2. Search index

2.1 Vector store index



In this scheme and everywhere further in the text I omit the Encoder block and send our query straight to the index for the scheme simplicity. The query always gets vectorised first of course. Same with the top k cunks — index retrieves top k vectors, not chunks, but I replace them with chunks as fetching them is a trivial step.

The crucial part of the RAG pipeline is the search index, storing your vectorised content we got in the previous step. The most naive implementation uses a flat index — a brute force distance calculation between the query vector and all the chunks' vectors.

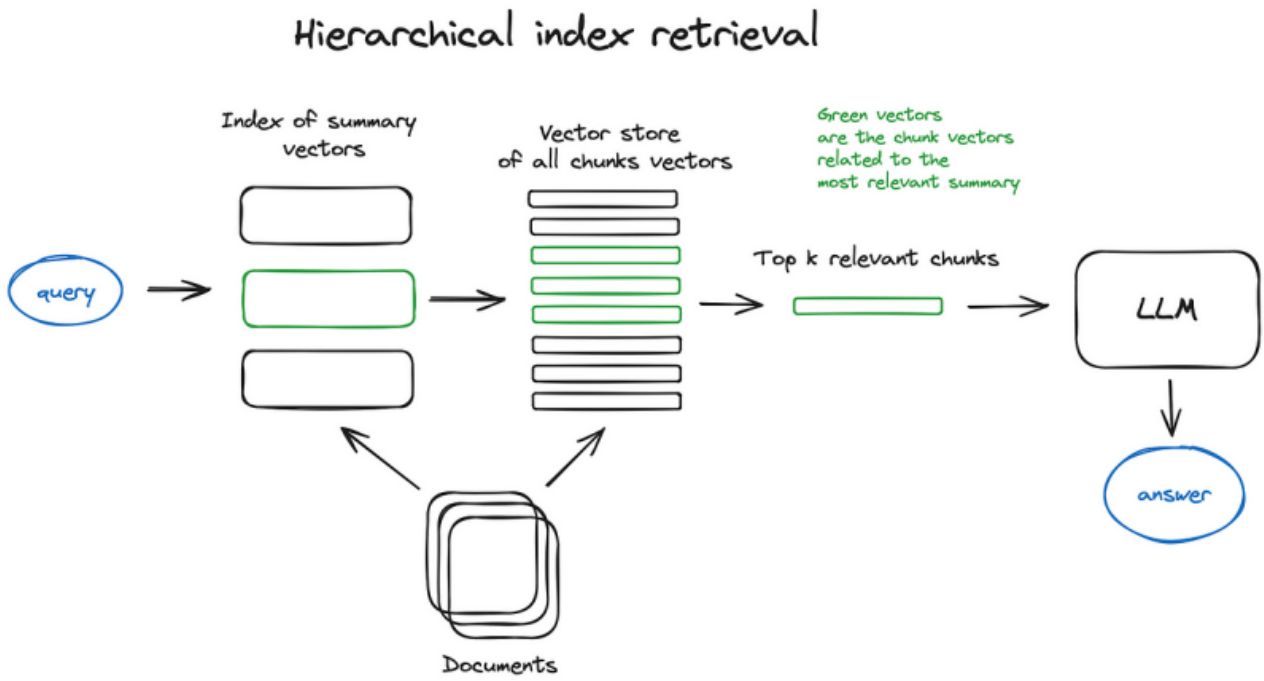
A proper search index, optimised for efficient retrieval on 10000+ elements scales **is a vector index** like faiss, nmslib or annoy, using some Approximate Nearest Neighbours implementation like clustring, trees or HNSW algorithm.

There are also managed solutions like OpenSearch or ElasticSearch and vector databases, taking care of the data ingestion pipeline described in step 1 under the hood, like Pinecone, Weaviate or Chroma.

Depending on your index choice, data and search needs **you can also store metadata along with vectors** and then use **metadata filters** to search for information within some dates or sources for example.

LlamaIndex supports lots of vector store indices but there are also other simpler index implementations supported like list index, tree index, and keyword table index — we'll talk about the latter in the Fusion retrieval part.

2. 2 Hierarchical indices



In case you have many documents to retrieve from, you need to be able to efficiently search inside them, find relevant information and synthesise it in a single answer with references to the sources. An efficient way to do that in case of a large database is to **create two indices — one composed of summaries and the other one composed of document chunks** , and to search in two steps, first filtering out the relevant docs by summaries and then searching just inside this relevant group.

2.3 Hypothetical Questions and HyDE

Another approach is to ask an LLM to **generate a question for each chunk and embed these questions in vectors**, at runtime performing query search against this index of question vectors (replacing chunks vectors with questions vectors in our index) and then after retrieval route to original text chunks and send them as the context for the LLM to get an answer.
This approach improves search quality due to a **higher semantic similarity between query and hypothetical question** compared to what we'd have for an actual chunk.

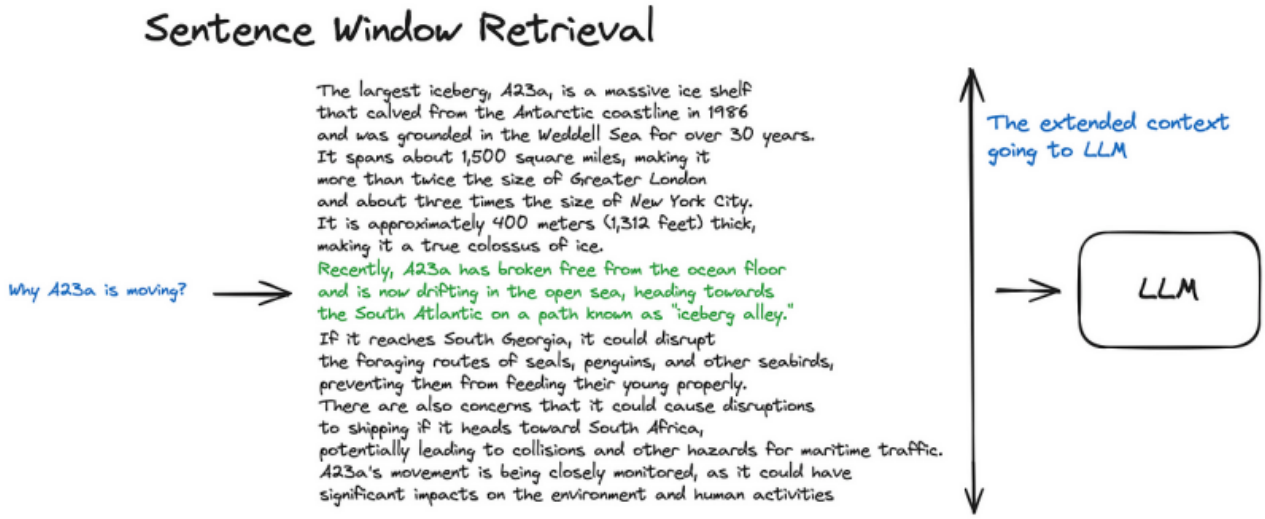
There is also the reversed logic apporach called **HyDE**— you ask an LLM to generate a hypothetical response given the query and then use its vector along with the query vector to enhance search quality.

2.4 Context enrichment

The concept here is to retrieve smaller chunks for better search quality, **but add up surrounding context for LLM to reason upon**.
There are two options — to expand context by sentences around the smaller retrieved chunk or to split documents recursively into a number of larger parent chunks, containing smaller child chunks.

2.4.1 Sentence Window Retrieval

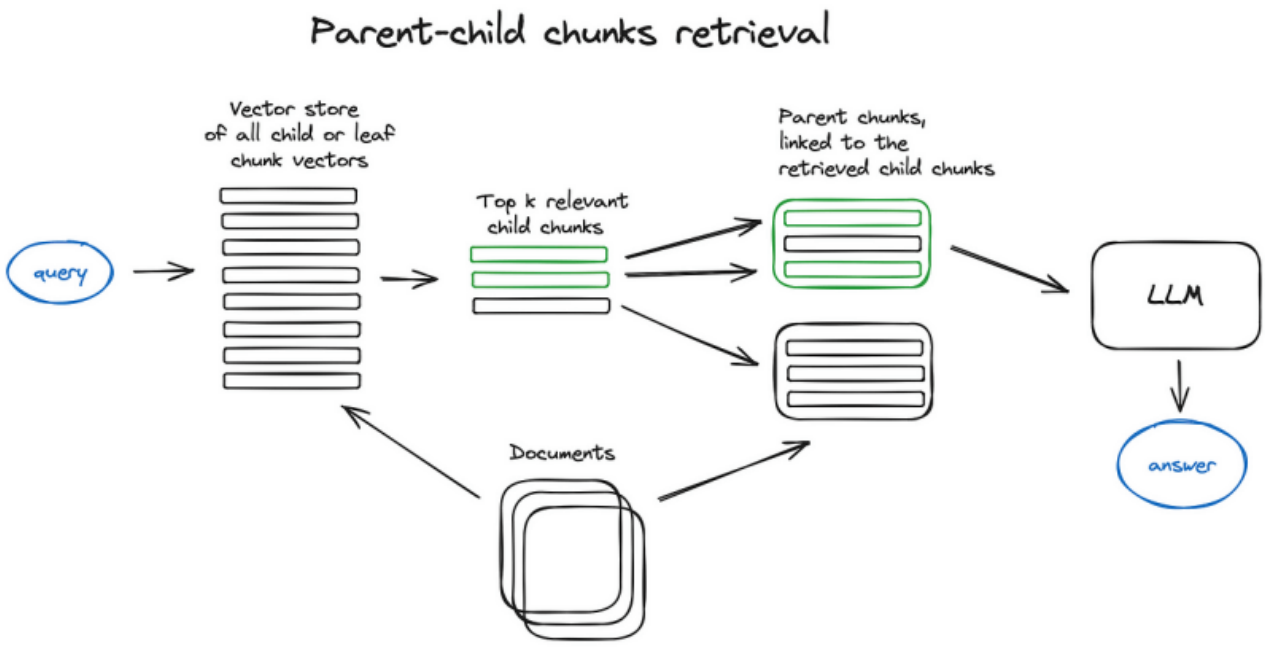
In this scheme each sentence in a document is embedded separately which provides great accuracy of the query to context cosine distance search.
In order to better reason upon the found context after fetching the most relevant single sentence we extend the context window by *k* sentences before and after the retrieved sentence and then send this extended context to LLM.



The green part is the sentence embedding found while search in index, and the whole black + green paragraph is fed to the LLM to enlarge its context while reasoning upon the provided query

2.4.2 Auto-merging Retriever (aka Parent Document Retriever)

The idea here is pretty much similar to Sentence Window Retriever — to search for more granular pieces of information and then to extend the context window before feeding said context to an LLM for reasoning. Documents are split into smaller child chunks referring to larger parent chunks.



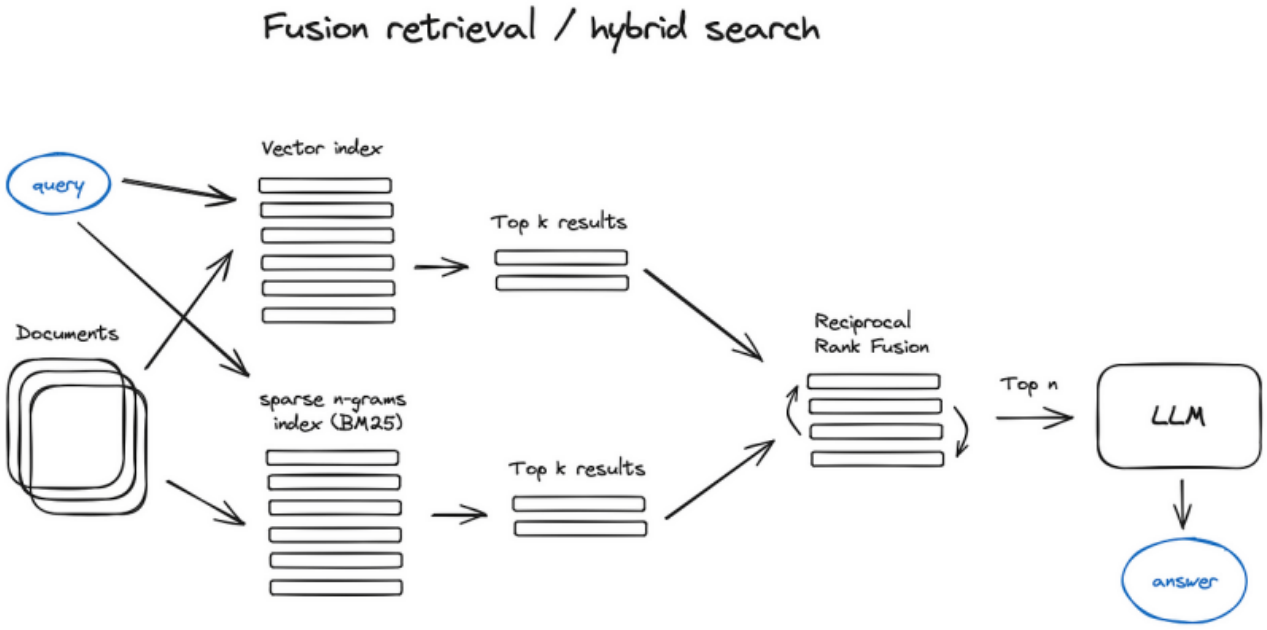
Documents are splitted into an hierarchy of chunks and then the smallest leaf chunks are sent to index. At the retrieval time we retrieve k leaf chunks, and if there is n chunks referring to the same parent chunk, we replace them with this parent chunk and send it to LLM for answer generation.

Fetch smaller chunks during retrieval first, then if more than *n* chunks in top *k* retrieved chunks are linked to the same parent node (larger chunk), we replace the context fed to the LLM by this parent node — works like auto merging a few retrieved chunks into a larger parent chunk, hence the method name. Just to note — search is performed just within the child nodes index. Check the LlamaIndex tutorial on Recursive Retriever + Node References for a deeper dive.

2.5 Fusion retrieval or hybrid search

A relatively old idea that you could **take the best from both worlds — keyword-based old school search** — sparse retrieval algorithms like tf-idf or search industry standard BM25 — **and modern** semantic or **vector search and combine it in one retrieval result.**

The only trick here is to properly combine the retrieved results with different similarity scores — this problem is usually solved with the help of the Reciprocal Rank Fusion algorithm, reranking the retrieved results for the final output.



In LangChain this is implemented in the Ensemble Retriever class, combining a list of retrievers you define, for example a faiss vector index and a BM25 based retriever and using RRF for reranking.

In LlamaIndex this is done in a pretty similar fashion.

Hybrid or fusion search usually provides better retrieval results as two complementary search algorithms are combined, taking into account both semantic similarity and keyword matching between the query and the stored documents.

3. Reranking & filtering

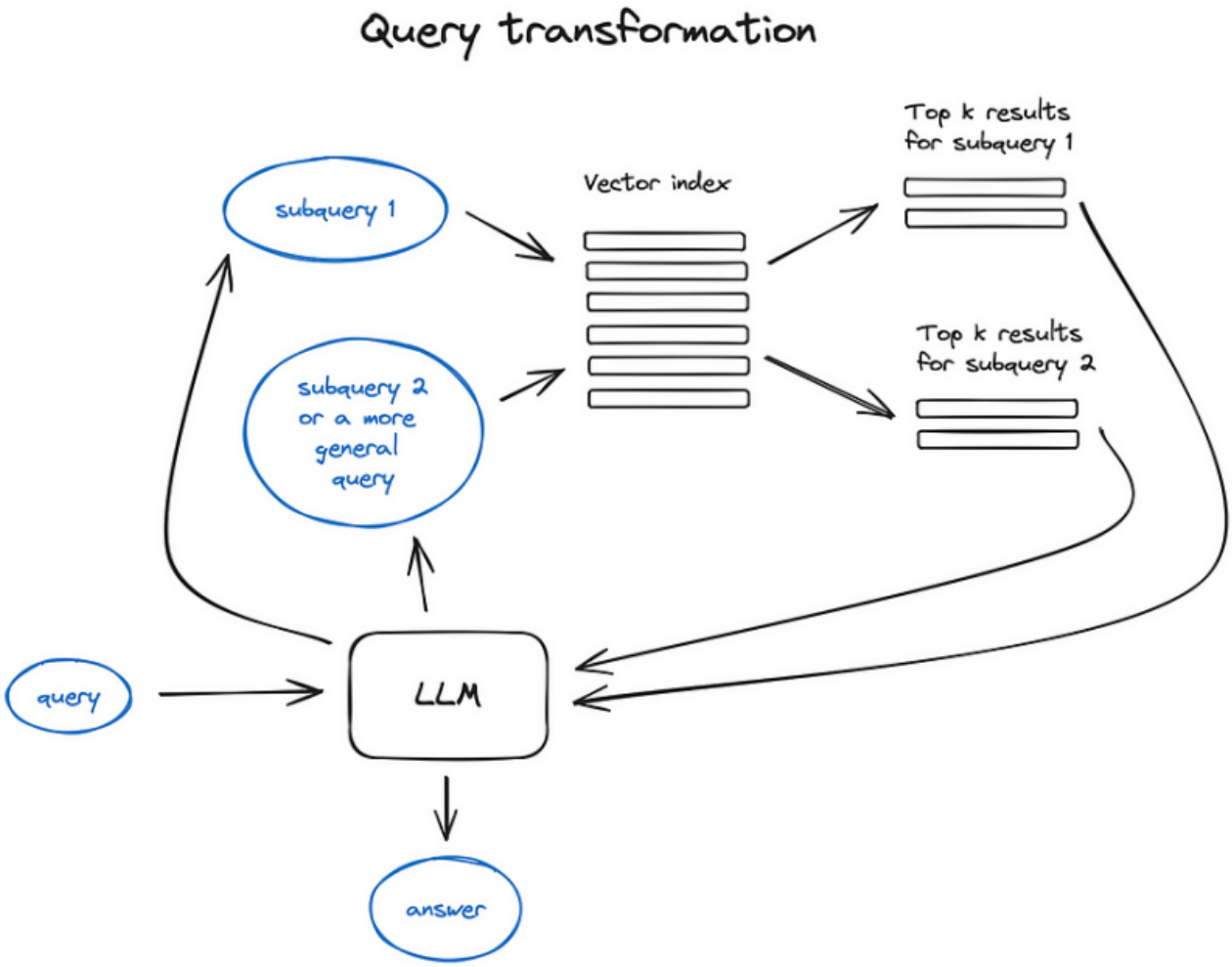
So we got our retrieval results with any of the algorithms described above, now it is time to refine them through filtering, re-ranking or some transformation. In LlamaIndex there is a variety of available **Postprocessors**, **filtering out results based on similarity score, keywords, metadata or reranking them with other models** like an LLM, sentence-transformer cross-encoder, Cohere reranking endpoint or based on metadata like date recency — basically, all you could imagine.

This is the final step before feeding our retrieved context to LLM in order to get the resulting answer.

Now it is time to get to the more sophisticated RAG techniques like Query transformation and Routing, both involving LLMs and thus representing **agentic behaviour — some complex logic involving LLM reasoning within our RAG pipeline.**

4. Query transformations

Query transformations are a family of techniques using an LLM as a reasoning engine to modify user input in order to improve retrieval quality. There are different options to do that.



Query transformation principles illustrated

If the query is complex, LLM can decompose it into several sub queries. For examle, if you ask:
— *"What framework has more stars on Github, Langchain or LlamaIndex?",*
and it is unlikely that we'll find a direct comparison in some text in our corpus so it makes sense to decompose this question in two sub-queries presupposing simpler and more concrete information retrieval:
— *"How many stars does Langchain have on Github?"*
— *"How many stars does Llamaindex have on Github?"*
They would be executed in parallel and then the retrieved context would be combined in a single prompt for LLM to synthesize a final answer to the initial query. Both libraries have this functional implemented — as a Multi Query Retriever in Langchain and as a Sub Question Query Engine in Llamaindex.

1. **Step-back prompting uses LLM to generate a more general query**, retrieving for which we obtain a more general or high-level context useful to ground the answer to our original query on.
   Retrieval for the original query is also performed and both contexts are fed to the LLM on the final answer generation step.
   Here is a LangChain implementation.

2. **Query re-writing uses LLM to reformulate initial query** in order to improve retrieval. Both LangChain and LlamaIndex have implementations, tough a bit different, I find LlamaIndex solution being more powerful here.

Reference citations

This one goes without a number as this is more an instrument than a retrieval improvement technique, although a very important one.
If we've used multiple sources to generate an answer either due to the initial query complexity (we had to execute multiple subqueries and then to combine retrieved context in one answer), or because we found relevant context for a single query in various documents, the question rises if we could **accurately back reference our sources**.

There are a couple of ways to do that:

1. **Insert this referencing task into our prompt** and ask LLM to mention ids of the used sources.
2. **Match the parts of generated response to the original text chunks** in our index — llamaindex offers an efficient fuzzy matching based solution for this case. In case you have not heard of fuzzy matching, this is an incredibly powerful string matching technique.

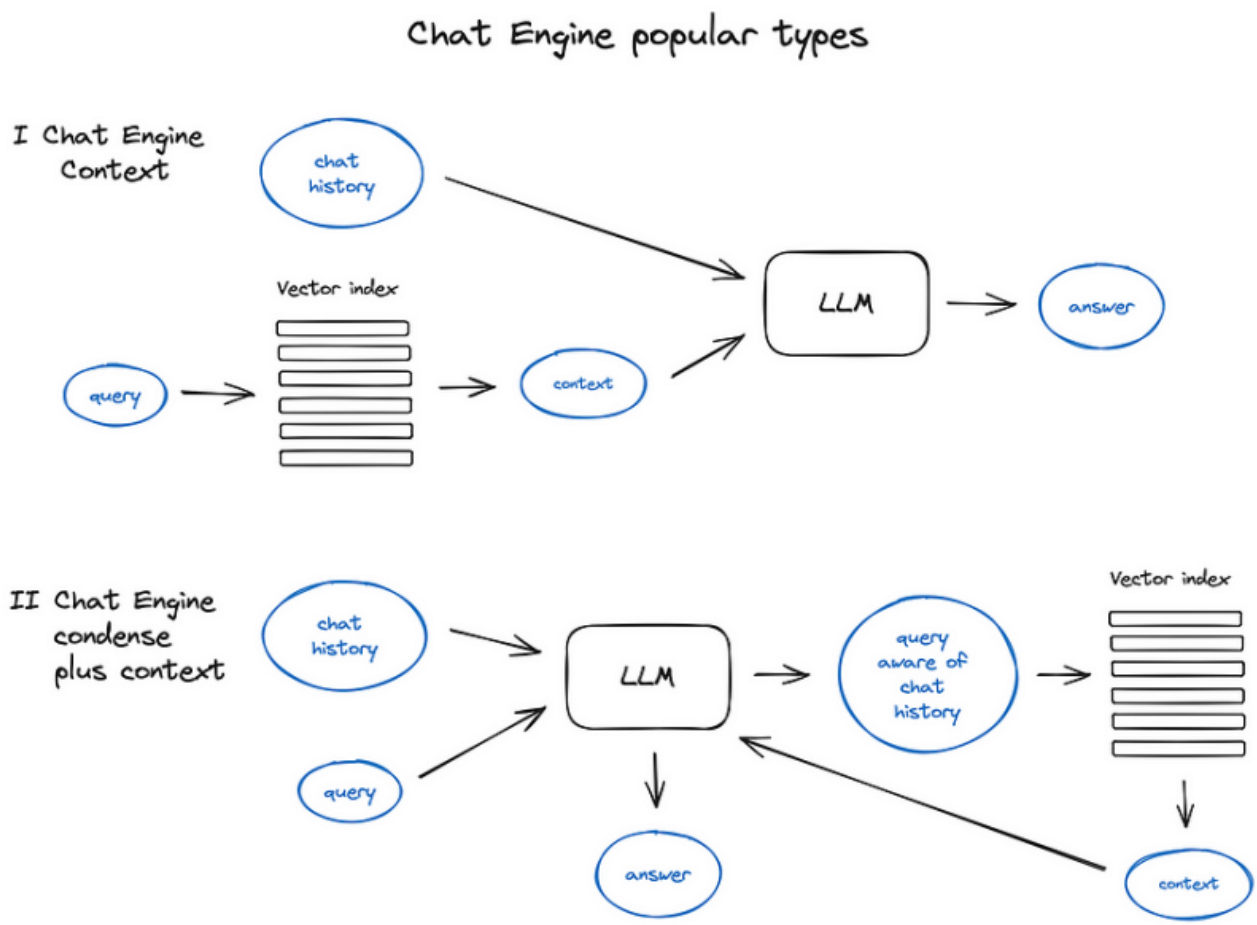
5. Chat Engine

The next big thing about building a nice RAG system that can work more than once for a single query is the **chat logic, taking into account the dialogue context**, same as in the classic chat bots in the pre-LLM era.
This is needed to support follow up questions, anaphora, or arbitrary user commands relating to the previous dialogue context. It is solved by **query compression technique, taking chat context into account** along with the user query.

As always, there are several approaches to said context compression —
a popular and relatively simple ContextChatEngine, first retrieving context relevant to user's query and then sending it to LLM along with chat history from the *memory* buffer for LLM to be aware of the previous context while generating the next answer.

A bit more sophisticated case is CondensePlusContextMode — there in each interaction the chat history and last message are condensed into a new query, then this query goes to the index and the retrieved context is passed to the LLM along with the original user message to generate an answer.

It's important to note that there is also support for OpenAI agents based Chat Engine in LlamaIndex providing a more flexible chat mode and Langchain also supports OpenAI functional API.



An illustration of different Chat Engine types and principles

There are other Chat engine types like ReAct Agent, but let's skip to Agents themselves in section 7.

6. Query Routing

Query routing is the step of LLM-powered decision making upon what to do next given the user query — the options usually are to summarise, to perform search against some data index or to try a number of different routes and then to synthesise their output in a single answer.

Query routers are also used to select an index, or, broader, data store, where to send user query — either you have multiple sources of data, for example, a classic vector store and a graph database or a relational DB, or you have an hierarchy of indices — for a multi-document storage a pretty classic case would be an index of summaries and another index of document chunks vectors for example.

Defining the query router includes setting up the choices it can make.
The selection of a routing option is performed with an LLM call, returning its result in a predefined format, used to route the query to the given index, or, if we are taking of the agnatic behaviour, to sub-chains or even other agents as shown in the **Multi documents agent scheme** below.

Both LlamaIndex and LangChain have support for query routers.

7. Agents in RAG

Agents (supported both by Langchain and LlamaIndex) have been around almost since the first LLM API has been released — **the idea was to provide an LLM, capable of reasoning, with a set of tools and a task to be completed**. The tools might include some deterministic functions like any code function or an external API or even other agents — this LLM chaining idea is where LangChain got its name from.

Agents are a huge thing itself and it's impossible to make a deep enough dive into the topic inside a RAG overview, so I'll just continue with the agent-based multi document retrieval case, making a short stop at the OpenAI Assistants stationas it's a relatively new thing, presented at the recent OpenAI dev conference as GPTs, and working under the hood of the RAG system described below.

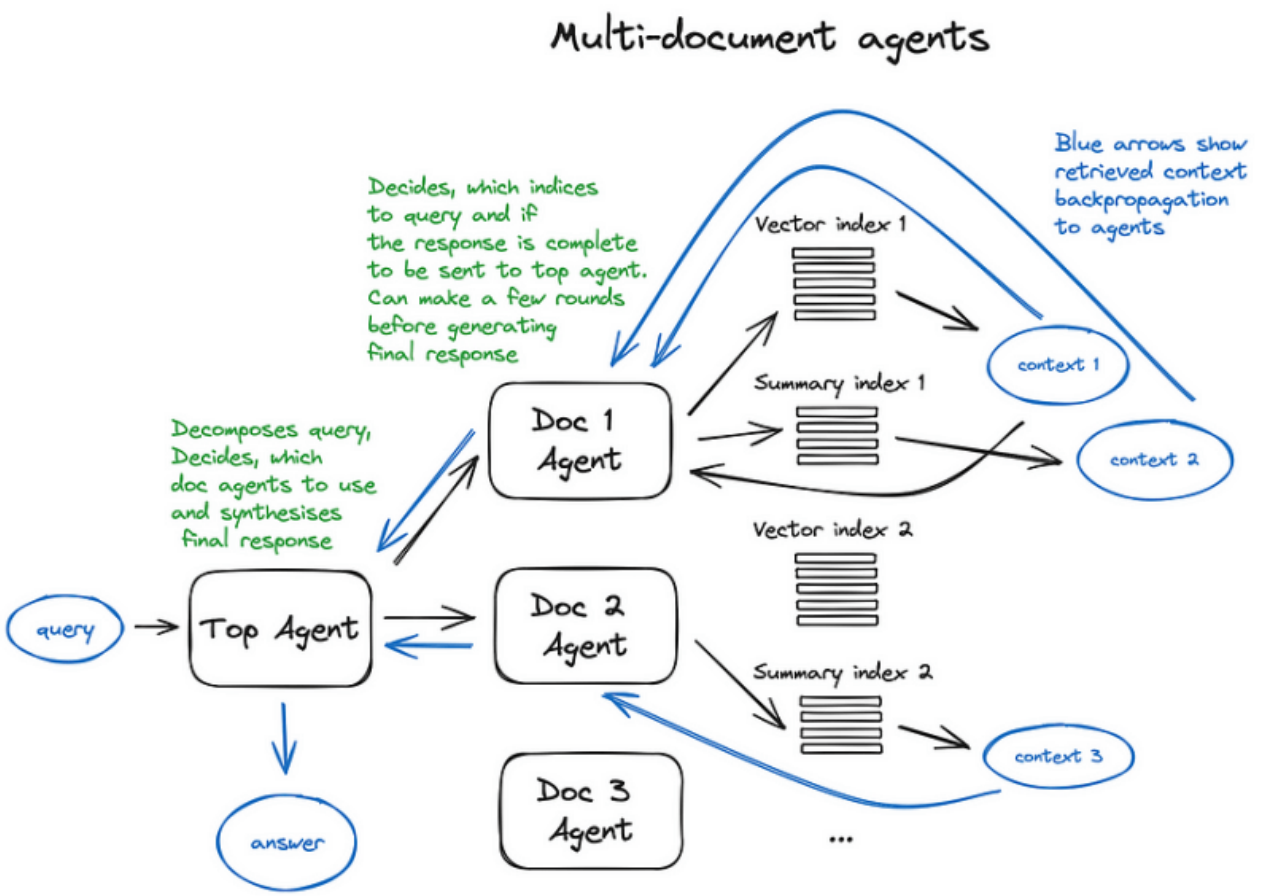
OpenAI Assistants basically have implemented a lot of tools needed around an LLM that we previously had in open source — a chat history, a knowledge storage, a document uploading interface and, maybe most important, **function calling API**. This latter provides capabilities to **convert natural language into API calls to external tools or database queries.**

In LlamaIndex there is an OpenAIAgent class marrying this advanced logic with the ChatEngine and QueryEngine classes, providing knowledge-based and context aware chatting along with the ability of multiple OpenAI functions calls in one conversation turn, which really brings the smart agentic behaviour.

Let's take a look at the **Multi-Document Agents scheme** — a pretty sophisticated setting, involving initialisation of an **agent** (OpenAIAgent) **upon each document**, capable of doc summarisation and the classic QA mechanics, **and a top agent**, responsible for queries routing to doc agents and for the final answer synthesis.

Each document agent has two tools — a vector store index and a summary index, and based on the routed query it decides which one to use.
And for the top agent, all document agents are tools respectfully.

This scheme illustrates an advanced RAG architecture with a lot of routing decisions made by each involved agent. **The benefit of such approach is the ability to compare different solutions or entities, described in different documents and their summaries** along with the classic single doc summarisation and QA mechanics — this basically covers the most frequent chat-with-collection-of-docs usecases.



A scheme illustrating multi document agents, involving both query routing and agentic behavior patterns.

The drawback of such a complex scheme can be guessed from the picture — it's a bit slow due to multiple back and forth iterations with the LLMs inside our agents. Just in case, an LLM call is always the longest operation in a RAG pipeline — search is optimised for speed by design. So for a large multi document storage I'd recommed to think of some simplifications to this scheme making it scalable.

8. Response synthesiser

This is the final step of any RAG pipeline — generate an answer based on all the context we carefully retrieved and on the initial user query.
The simplest approach would be just to concatenate and feed all the fetched context (above some relevance threshold) along with the query to an LLM at once.
But, as always, there are other more sophisticated options involving multiple LLM calls to refine retrieved context and generate a better answer.

The main approaches to response synthesis are:
1. **iteratively refine the answer** by sending retrieved context to LLM chunk by chunk
2. **summarise the retrieved context** to fit into the prompt
3. **generate multiple answers** based on different context chunks and then to concatenate or **summarise them**.

For more details please check the Response synthesizer module docs.

Encoder and LLM fine-tuning

This approach involves fine-tuning of some of the two DL models involved in our RAG pipeline — either the Transformer **Encoder, resposible for embeddings quality and thus context retrieval quality** or an **LLM, responsible for the best usage of the provided context to answer user query** — luckily, the latter is a good few shot learner.

One big advantage nowadays is the availability of high-end LLMs like GPT-4 to generate high quality synthetic datasets.
But you should always be aware that taking an open-source model trained by professional research teams on carefully collected, cleaned and validated large datasets and making a quick tuning using small synthetic dataset might narrow down the model's capabilities in general.

Encoder fine-tuning

I've also been a bit skeptical about the Encoder funetuning approach as the latest Transformer Encoders optimised for search are pretty efficient.
So I have tested the performance increase provided by finetuning of bge-large-en-v1.5 (top 4 of the MTEB leaderboard at the time of writing) in the LlamaIndex notebook setting, and it demonstrated a 2% retrieval quality increase. Nothing dramatic but it is nice to be aware of that option, especially if you have a narrow domain dataset you're building RAG for.

Ranker fine-tuning

The other good old option is to have a cross-encoder for reranking your retrieved results if you dont trust your base Encoder completely.
It works the following way — you pass the query and each of the top k retrieved text chunks to the cross-encoder, separated by a SEP token, and fine-tune it to output 1 for relevant chunks and 0 for non-relevant.
A good example of such tuning process could be found here, the results say the pairwise score was improved by 4% by cross-encoder finetuning.

LLM fine-tuning

Recently OpenAI started providing LLM finetuning API and LlamaIndex has a tutorial on **finetuning GPT-3.5-turbo in RAG setting** to "distill" some of the GPT-4 knowledge. The idea here is to take a document, generate a number of questions with GPT-3.5-turbo, then use GPT-4 to generate answers to these questions based on the document contents (build a GPT4-powered RAG pipeline) and then to fine-tune GPT-3.5-turbo on that dataset of question-answer pairs. The ragas framework used for the RAG pipeline evaluation shows a **5% increase in the faithfulness metrics, meaning the fine-tuned GPT 3.5-turbo model made a better use of the provided context** to generate its answer, than the original one.

A bit more sophisticated approach is demonstrated in the recent paper **RA-DIT: Retrieval Augmented Dual Instruction Tuning** by Meta AI Research, **suggesting a technique to tune both the LLM and the Retriever**
(a Dual Encoder in the original paper) **on triplets of query, context and answer**. For the implementations details please refer to this guide.
This technique was used both to fine-tune OpenAI LLMs through the fine-tuning API and Llama2 open-source model (in the original paper), resulting in ~5% increase in knowledge-intense tasks metrics (compared to Llama2 65B with RAG) and a couple percent increase in common sense reasoning tasks.

In case you know better approaches to LLM finetuning for RAG, please share your expertise in the comments section, especially if they are applied to the smaller open source LLMs.

Evaluation

There are **several frameworks for RAG systems performance evaluation** sharing the idea of having a few separate metrics like overall **answer relevance, answer groundedness, faithfulness and retrieved context relevance**.

Ragas, mentioned in the previous section, uses faithfulness and answer relevance as the generated answer quality metrics and classic context precision and recall for the retrieval part of the RAG scheme.

In a recently released great short course Building and Evaluating Advanced RAG by Andrew NG, LlamaIndex and the evaluation framework Truelens, they suggest the **RAG triad** — **retrieved context relevance** to the query, **groundedness** (how much the LLM answer is supported by the provided context) and **answer relevance** to the query.

The key and the most controllable metric is the **retrieved context relevance** — basically parts 1–7 of the advanced RAG pipeline described above plus the Encoder and Ranker fine-tuning sections are meant to improve this metric, while part 8 and LLM fine-tuning are focusing on answer relevance and groundedness.

A good example of a pretty simple retriever evaluation pipeline could be found here and it was applied in the Encoder fine-tuning section.
A bit more advanced approach taking into account not only the **hit rate**, but the **Mean Reciprocal Rank**, a common search engine metric, and also generated answer metrics such as faithfulness abd relevance, is demonstrated in the OpenAI cookbook.

LangChain has a pretty advanced evaluation framework LangSmith where custom evaluators may be implemented plus it monitors the traces running inside your RAG pipeline in order to make your system more transparent.

In case you are building with LlamaIndex, there is a rag_evaluator llama pack, providing a quick tool to evaluate your pipeline with a public dataset.

Conclusion

I tried to outline the core algorithmic approaches to RAG and to illustrate some of them in hopes this might spark some novel ideas to try in your RAG pipeline, or bring some system to the vast variety of tecniques that have been invented this year — for me 2023 was the most exciting year in ML so far.

There are many more other things to consider like **web search based RAG** (RAGs by LlamaIndex, webLangChain, etc), taking a deeper dive into **agentic architectures** (and the recent OpenAI stake in this game) and some ideas on **LLMs Long-term memory**.

The main production challenge for RAG systems besides answer relevance and faithfulness is speed, especially if you are into the more flexible agent-based schemes, but that's a thing for another post. This streaming feature ChatGPT and most other assistants use is not a random cyberpunk style, but merely a way to shorten the perceived answer generation time.
That is why I see a very bright future for the smaller LLMs and recent releases of Mixtral and Phi-2 are leading us in this direction.

Thank you very much for reading this long post!

The main references are collected in my knowledge base, there is a co-pilot to chat with this set of documents: https://app.iki.ai/playlist/236.

Published via Towards AI