Medium Search Write Sign up Sign in

Member-only story

Improving RAG performance — A Structured Approach (Part 6(B) of RAG Series)

A comprehensive and structured approach

Chandan Durgia · Follow

17 min read · Mar 5, 2024

--

Photo by Lukas Blazek on Unsplash

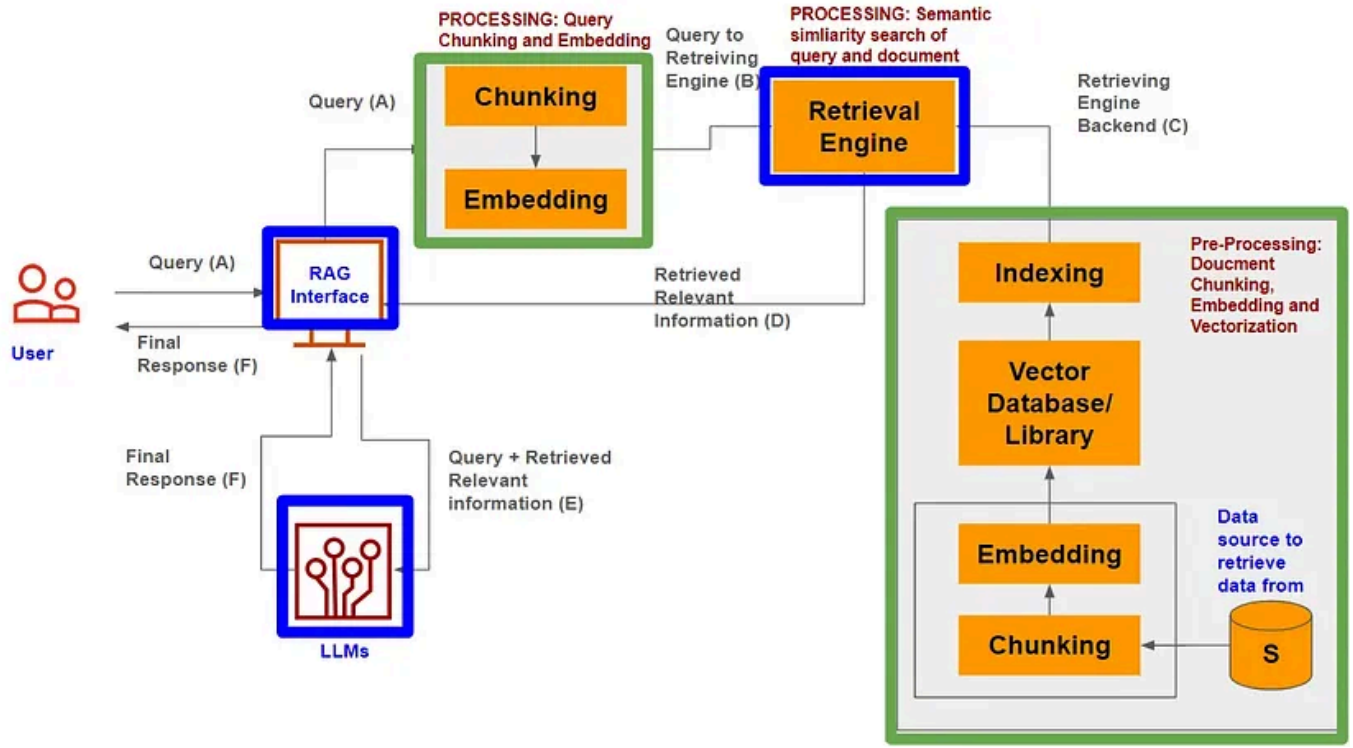
This is part 6 of the “Retrieval-Augmented Generation (RAG) — Basics to Advanced Series”. Links to other blogs in the series are at the bottom of this blog. Taking forward from part 1 (RAG Basics), part 2 (Chunking), part 3 (Embedding) and part 4 (Vector Databases and Vector Libraries) and part 5 (Evaluation of RAG). In this blog, we will focus on one of the most

challenging but also the most important step i.e. “Improving RAG performance”

This part 6 is further divided into Part A and Part B. Part A of the blog **covered the Ingestion Stage and covers key topics like** Data clean up, Data Enrichment, Chunking, Embedding, Vector database and Indexing and Query enhancement and Prompt Engineering

This blog would cover **the retrieval and generation stage** which would include experimentation around areas like Retrieval, Re-ranking, Fine Tuning and bringing everything together.

Diagrammatically, going back to our RAG Architecture, Part A covers the components in the Green boxes and Part B covers the components in the Blue boxes.



RAG Architecture (Image from Author)

In this blog, we will first discuss the Retrieval optimization, Re-ranking optimization followed by Fine-tuning approaches.

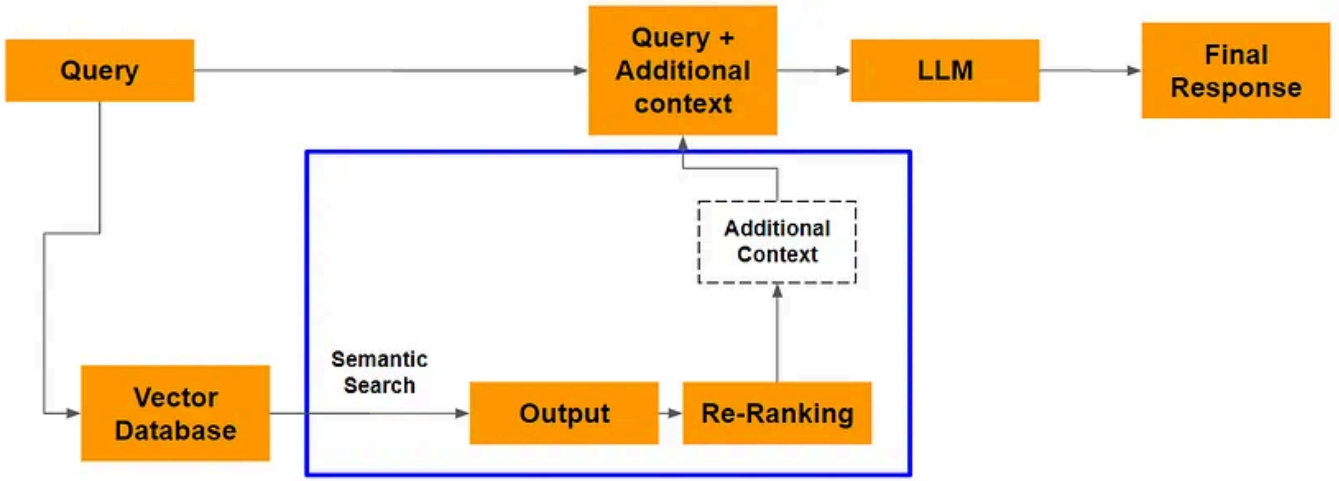
Retrieval optimization

Retrieval optimization is one of the other key components which can yield significant improvement in performance if calibrated appropriately. These

are some of the key techniques from the retrieval perspective which can improve RAG performance.

Important Note: It is important to acknowledge that retrieval methodologies have some overlap and commonalities with some techniques covered in Part A of this blog under the sections “Data Enrichment” and “Vector databases and Indexing”. This is because for the cases where data enrichment and/or indexing is not done efficiently, some sophisticated retrieval techniques can be used which compensates for it during the retrieval process. In the techniques below, the relevant techniques from “Data Enrichment” and “Vector databases and Indexing” are mentioned for ease of understanding.

Before proceeding on various ways of retrieval optimization, let’s visualize a simple retrieval engine, highlighted in blue below:

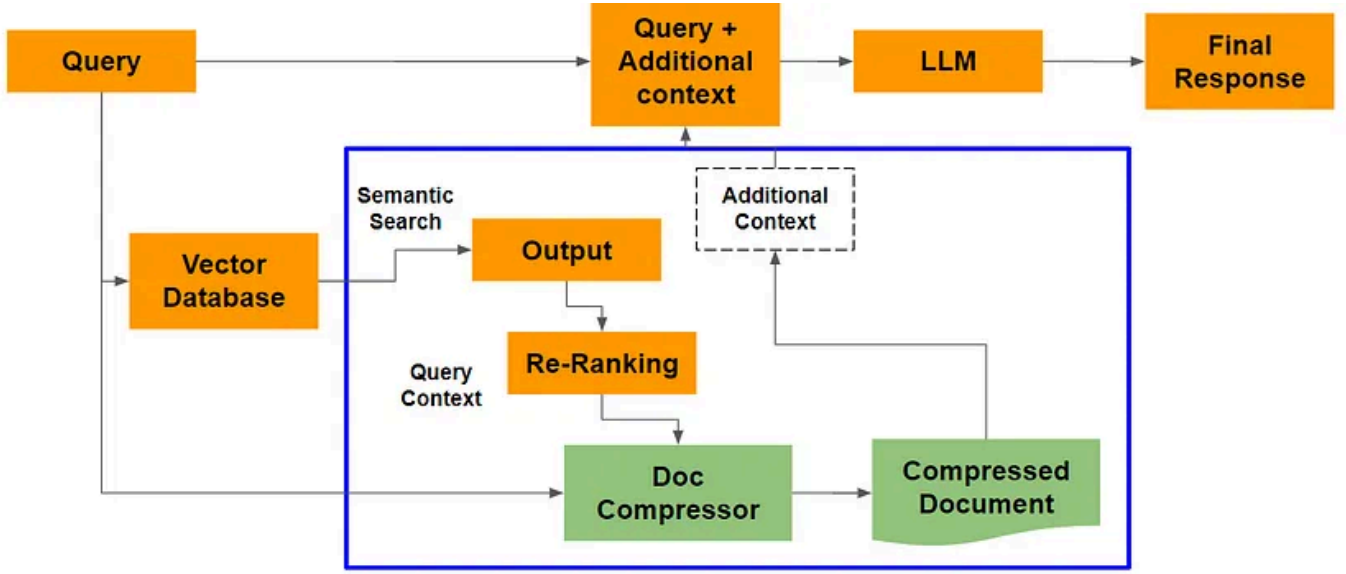


Simple Retriever (Image from Author)

1. Contextual Compression — Retrieval mechanism

Imagine a case where the output of your query is too long and not to the point. It could be because the base retriever worked well, but the underlying information related to the query is intermingled with a lot of noise in the document. In this case, a lot of information is passed to LLM as “Additional Context” which could be expensive and inefficient. This is where Contextual Compression comes into play.

Let’s first compare this with how it is different from simple retrieval engine



Contextual Compression (Image from Author)

The output of the long retrieved document is first sent to “Doc Compressor” which compresses the document — given “query” context. Contextually compressing the document covers two aspects:

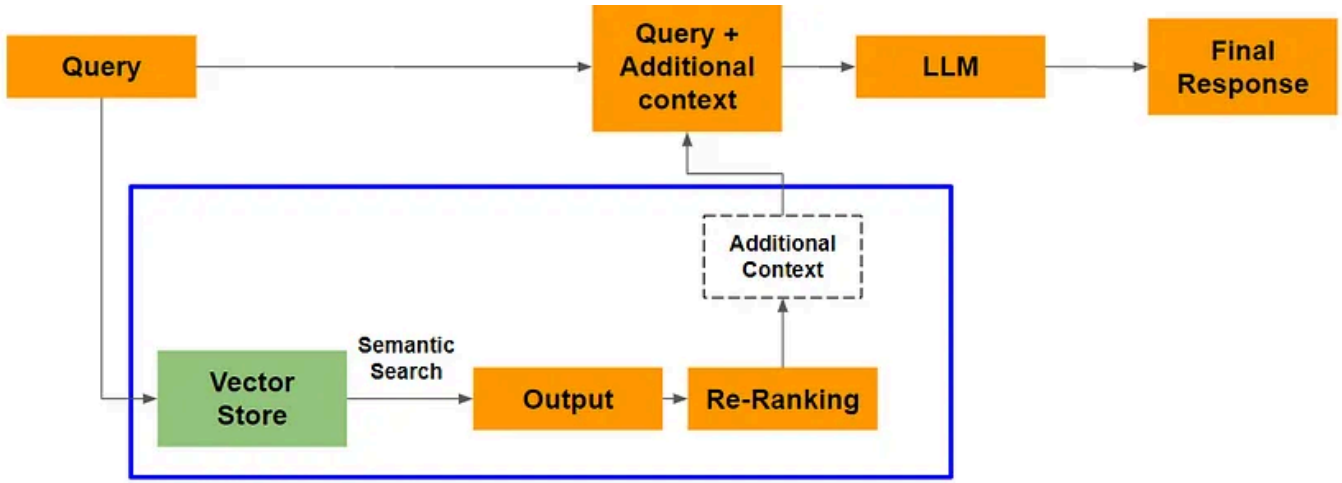
1. Filter the relevant text from the range of documents
2. Compress (summarize) the content of a specific document.

The compressed document additional context is to the point which is sent to LLM — it can be acknowledged that this would be less expensive and quite efficient.

Both Langchain and Llama index provide functionality for contextual compression.

2. Vector store — retriever mechanism

As discussed in the previous blog, a vector store is just like a memory based Vector database — without a need for a database. These vector stores provide a lightweight wrapper class around them so that the retrieving with similarity search can easily be done in an efficient manner.



Vector Store (Image from Author)

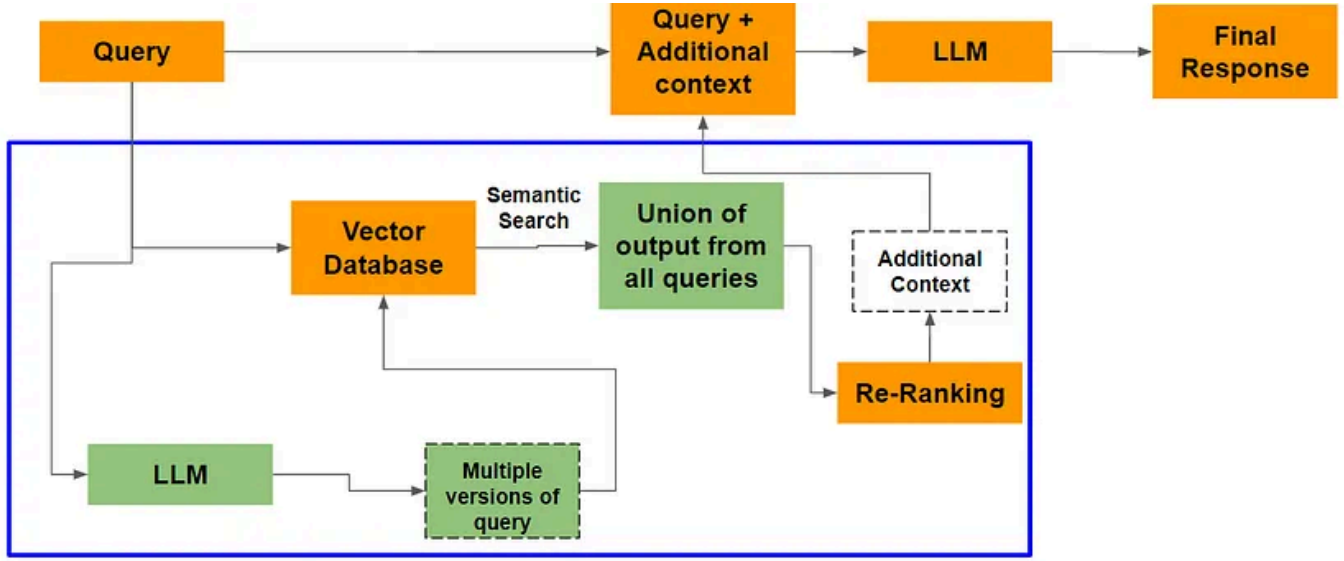
3. MultiQueryRetriever mechanism

The issue with the simple retrieval is that it gives the output which is closest to the query and sometimes the output from a query might not be relevant.

In this case, practitioners try a couple potential solutions:

1. Perform prompt engineering till they get the desired output (which is time consuming and takes lot of efforts)
2. Create a mechanism through which they get all possible outputs which could be relevant to that query. This is what MultiQueryRetriever does.

A MultiQueryRetriever uses an LLM to generate multiple versions of the input query and these all queries are used in the retriever to get multiple outputs and then a union is performed on all the outputs to get a comprehensive output which would have a high probability of matching the expected response.



MultiQueryRetriever (Image from Author)

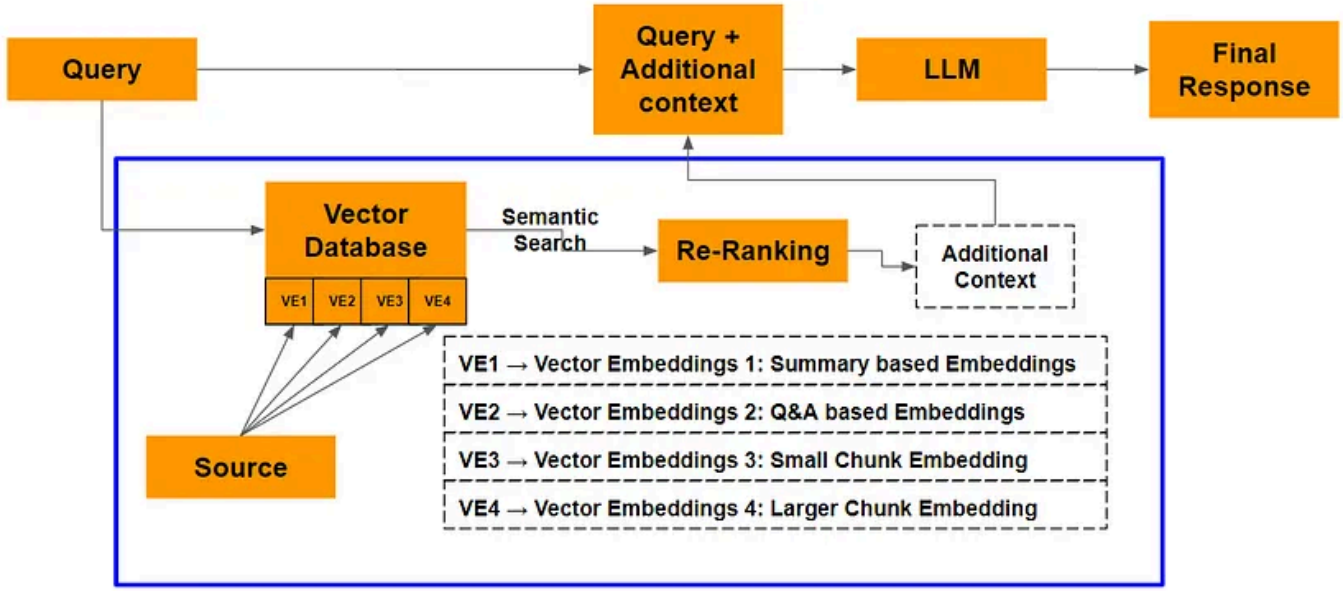
4. MultiVector Retriever Mechanism

Leveraging the same vector (Embeddings) agnostic of the use cases (summarization, Q&A etc.) could be quite inefficient sometimes. So MultiVector Retriever as the name suggests, stores multiple Vector Embeddings for the same document in multiple ways. Depending on the query and input, the mechanism uses the appropriate Vector and provides the relevant output accordingly. Various kinds of vectors that can be created from the document:

- **Smaller chunks Vector (ParentDocumentRetriever).** This uses small chunks in isolation to provide the response or clubs these small chunks

together with the linked larger chunks (Parent) to spit out the response.

- **Summary Vector:** Create a summary of the document and which can either be appended to the document or just be saved alone.
- **Q&A vector:** Create anticipated question basis the context of the document and these Q&A can either be appended to the document or just be saved alone.



MultiVector Retriever (Image from Author)

5. Time-weighted vector store retriever

There are cases wherein some part of the documents have almost all answers and other parts are just noises or additional information which would never be used. For these cases, Time-weighted vector store retrievers are used. These retrievers in addition to the semantic search also give weightage to time a particular part of the document was accessed. This implies that the part of the documents which are commonly accessed have higher probability of getting re-accessed again — which is aligned with the objective. It is important to note that the Time-weighted component is not related to the recency of the part of the document, as mentioned it is related to the time a particular part of the document was accessed.

6. Recursive or Iterative Retrieval

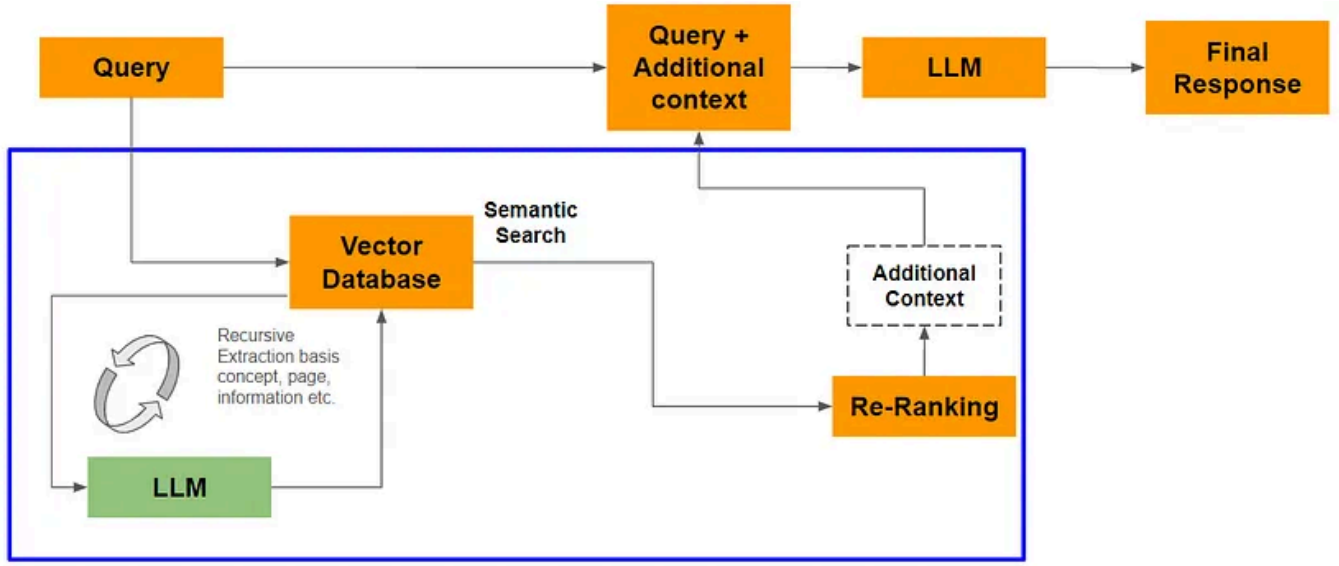
Recursive retrieval works in majorly 3 steps:

1. In the first step, like a base retriever smaller chunks and data are retrieved.
2. In the next step, these intermediate results are “iteratively and continuously” analyzed by the LLM RAG to find more relevant information. This step continues till there are no more references to other parts that hints at a final response. (second, third, fourth order degree).

3. Finally, the aggregation of all the key retrieved information across the intermediate steps is done.

In other words, first the small chunks are retrieved and then parent documents corresponding to these smaller chunks are further retrieved and amended.

This mechanism ensures that the retrieval is exhaustive. Note that this recursiveness can be performed on the basis of page, information, concept etc.



Recursive Retrieval (Image by Author)

7. Auto-merging retrieval

This is a different form of recursive retrieval with the key difference being that for Auto-merging the documents are organized in a tree structure. During retrieval, multiple relevant leaf nodes (a subset of the parent nodes) are merged in a recursive way. In this manner, potentially different and disparate texts are merged together and retrieved which helps in synthesizing a holistic output from LLM.

8. Small to Big Retrieval (Llama)/ Parent Retrieval(Langchain)

Many times the response from RAG could be irrelevant or incorrect because the retrieval engine looks for the specific text and passess it to the LLM for the final response. However, this text without additional context might lead to wrong inferences.

In these cases, Small to Big Retrieval is of great importance. The idea behind this mechanism is that the text could be Embedded at the sentence level however during the LLM synthesis this could be expanded in some fashion. A couple of common ways to achieve this are:

1. **Sentence-window retrieval:** As the name suggests, the Sentence-window retrieval not only retrieves the specific text, but also retrieves some additional text around the sentence (before and after) for the defined text

window. This additional information, sometimes considerably improves overall RAG accuracy as LLM has additional information and context to provide appropriate response.

2. **Parent-Child retrieval:** In this technique, the smaller chunks are fetched/retrieved first and then the parent IDs of these smaller chunks are referenced to provide additional context.

Note that this technique together with additional metadata like summary etc. could lead to considerable improvement in performance and accurate results.

Small to Big Retrieval (Image by Author)

9. Summary based retrieval

There are always texts and documents which have a lot of repetition and unnecessary words. While looking for the inference based responses from these documents, the retrieval engine might give erroneous results. One effective method to improve the quality of information and reduce the amount of text is to summarize the data chunks and save these summaries in a vector store. By summarization, these extra chunks can be removed and the relevant response can be retrieved.

This type of technique is quite useful in addressing the ‘Lost in the Middle’ problem which is quite prominent for these types of documents. “Lost in the middle” problem occurs as the LLM by design are trained to give more heed/attention to the initial and later part of the text and not the middle part. By summarizing the text, the text length decreases considerably which minimizes the “lost in the middle” problem to a good extent.

10. Hierarchical retrieval

Hierarchical retrieval in a way is an extension of summary base retrieval. In contrast to summary base retrieval, wherein the summary of the text is retrieved, in this retrieval mechanism, once the summary chunks are retrieved, the actual text from which the summary is derived is further searched for specific chunks.

As it can be acknowledged, this considerably improves the overall performance and accuracy of the RAG model.

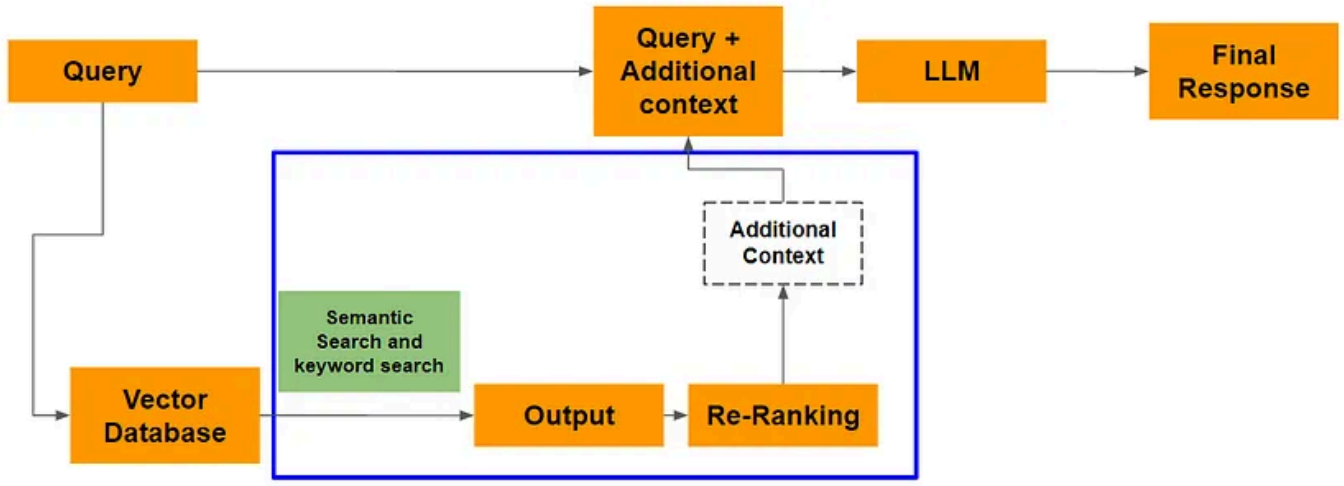
11. Ensemble Retrievers

As the name indicates, Ensemble Retrievers use various different retrievers, ensemble the results and rank the output basis ranking algorithm. Though there are many ways to ensemble the retrievers, one of the most famous and common retriever is Hybrid search Retriever

Hybrid search Retriever

Hybrid search Retriever, as the name indicates, is an ensemble of semantic search (embedding similarity) and keyword search (sparse retriever) i.e. it ensures that the “exact keywords” are searched with the right balance of searching for the “semantics”. This mechanism leads to better results across a range of use cases.

By searching for both keywords (which improves precision of the model together with semantic (which improves recall), overall this mechanism provides a very comprehensive search with a high F1 score. Furthermore, this is extremely useful for higher dimensional vector search. A good example for this is retrieval search for a “product” from a retail perspective, this would yield better results due to “keywords” used the products together with the “semantic” search from products features perspective. Though this adds complexity and cost, there has been multiple researches performed which proves that this is considerably better than the basic semantic search.



Hybrid Search Retriever (Image by Author)

12. Rules based Retriever

To improve the accuracy and relevance of the retrievers, this is a manual but critical step. Even the most high performing RAG would be based on its understanding of the Embeddings and keywords. However, it might not understand the domain and the style of the documentation. This is where “rules” come into play. This mechanism provides an additional layer of hardcoded rules. Some examples of rules could be: for this category of query refer to this section or page num, for query outside the following domain pls return — “results not found” etc.

Re-ranking Optimisation

As discussed in the past blogs, there are two key elements to the RAG performance:

a) Accuracy (defined by similarity of outcome to the query)

b) Relevance (how relevant is the output)

The process till the Retrieval (Output box in the picture) uses only “semantic search” which identifies chunks of information closely related to the user’s query and presents the top matches. This process doesn’t cater to the relevance aspect at all. This is where Re-ranking plays a key role.

Re-ranking rank orders the output from the retriever (which is based on similarity search) by “relevance”. This is done by computing a score for the relevance of the query for each retrieved context and eliminating irrelevant search results by ensuring that relevant chunks with high scores are provided as output.

Re-ranking is a good mechanism to enable RAG systems:

- understand domain-specific data nuances (jargons)

- diversify search results and consider other factors like metadata and keyword matches for providing a more comprehensive context for the language model to generate responses.

Expectedly, if Re-ranking is optimized/tuned appropriately for a particular use case, it improves the RAG performance considerably. Optimization can be done using a task-specific dataset of question-and-answer pairs, similar to how the embedding models are trained.

Though the market is evolving fast on a daily basis, as of now the Cohere Reranker is the most commonly used mechanism for Re-ranking.

Fine Tuning

Generally, fine-tuning is a process of taking a pre-trained model (which is usually trained on a larger dataset) and adapting it to perform a specific task or improve its performance on a specific task. Instead of training a model from scratch, fine-tuning allows leveraging the knowledge and features learned by a pre-trained model on a large dataset and applying it to a new, smaller dataset. This saves time and computational resources.

Fine-tuning can be beneficial for improving the efficiency and effectiveness of a model. It can reduce training time and cost, as it does not require starting from scratch. Additionally, fine-tuning can increase performance and accuracy by leveraging the features and knowledge of the pre-trained model.

From RAG perspective, fine tuning can be done at two levels:

1. **Embedding model fine-tuning:** To ensure that Embedding model understands specialized domain specific words, specific terminologies and Abbreviations.
2. **LLM Model fine-tuning:** To enhance existing knowledge base; changing the structure or tone of responses and teaching how to think in a complex scenario (more details below)

Similar to improving any ML models, there is no easy way or defined way for fine tuning, one needs to try a lot of experimentation to find the best fit for the given use cases.

Embedding Model Fine Tuning

For RAG purposes, usually, pre-trained embedding models (e.g. BERT and Ada) are used which usually provides a good retrieval outcome for many use cases. However, for certain specific domains, there are considerable issues in the accuracy and relevance of responses using these pre-trained models.

Say you want to execute RAG on documents on trading advice from brokers. In these documents there could be some terminologies like call, put, options, derivatives etc which the Embedding model is not conversant with. For these cases, though the model might perform well for general queries, since it has not been trained on these specific terminologies the model would not be able to understand the context appropriately.

This is where fine-tuning plays an important role. The model fine tuning helps in educating the model about specific new words, terminologies and context for a new specific domain. Depending on the use case the Embedding fine tuning can improve the accuracy of RAG up to 10%.

Though it sounds complex, In practice it is not difficult to perform fine tuning. (though not that straightforward as well). Libraries like “sentence transformer” in Llama help train embedding models with ease. Also this library also provides different training methods for this purpose.

For fine tuning, the dataset required includes a set of questions and document pairs. This can include both cases where you can find answers in the document and also the cases where it can't. Fundamentally, based on the query/data source, the library generates a synthetic dataset (leveraging LLM model) which can be used to finetune the pre-trained Embedding model.

The fine-tuning leads to better and efficient embedding representations for the given data, which in-turn leads to better retrieval accuracy and relevance.

To ensure a fast and accurate RAG implementation, it is important to periodically update the embeddings in order to capture the changing semantics in your corpus. This will make them more efficient for both retrieval and matching purposes.

LLM Fine tuning

Before we delve into the details of LLM fine tuning, it is important to highlight that unlike the popular myth, fine-tuning is certainly not the panacea for all the LLM related problems. Though it certainly helps improve accuracy to a certain extent.

The key areas where LLM fine tuning works well are:

1. **Knowledge base:** “Emphasizing” existing knowledge of the LLM model. Fine-tuning is not for adding new knowledge.
2. **Outcome:** Change the structure or tone of responses e.g. you can train LLM to be causal in the response rather than being formal.
3. **Teaching how to think in a complex scenario:** When complex instructions are provided to LLM, it could get confused and might provide responses which are not accurate or relevant.

Fine-tuning a pre-trained strong LLM model could do wonders in terms of accuracy and performance of the LLM.

Fine-tuning LLM is very similar, operationally to fine-tuning Embedding models — it’s about creating a synthetic dataset and retraining the model to perform better on specific task and domain. However, there is a catch, LLM are “focussed models” which implies that if you fine tune LLM for a specific data and use case — it would not be as efficient for the other use cases. For example: a model fine tuned on say legal data will not be good in conversations and coding.

PS: For the purpose of this blog, we will not delve into the details of how to do fine-tuning. I will try to publish another blog on this soon.

There are multiple ways to do perform fine-tuning, each having its own pros and cons:

1. **Full fine-tuning approaches**: In this approach, one amends every single model parameter to optimize performance. This involves retraining or adjusting on a specific dataset for specialized understanding.
2. **Parameter-efficient fine-tuning methods (PEFT):** In this approach, like fine-tuning of many other transfer learning models, one amends the final (adapter) layer of the model (limited parameters). This is very lightweight fine-tuning and doesn't require much data.
3. **Use cases specific fine-tuning**: Fine-tuning to tailor specific use cases and tasks rather than extensive set of use case support.
4. **Multi-task fine-tuning**: Fine-tuning to tailor multiple tasks for a specific use case for example — summarization, Q&A etc.
5. **Few-shot fine-tuning:** Rapidly fine-tuning the model with a few examples.

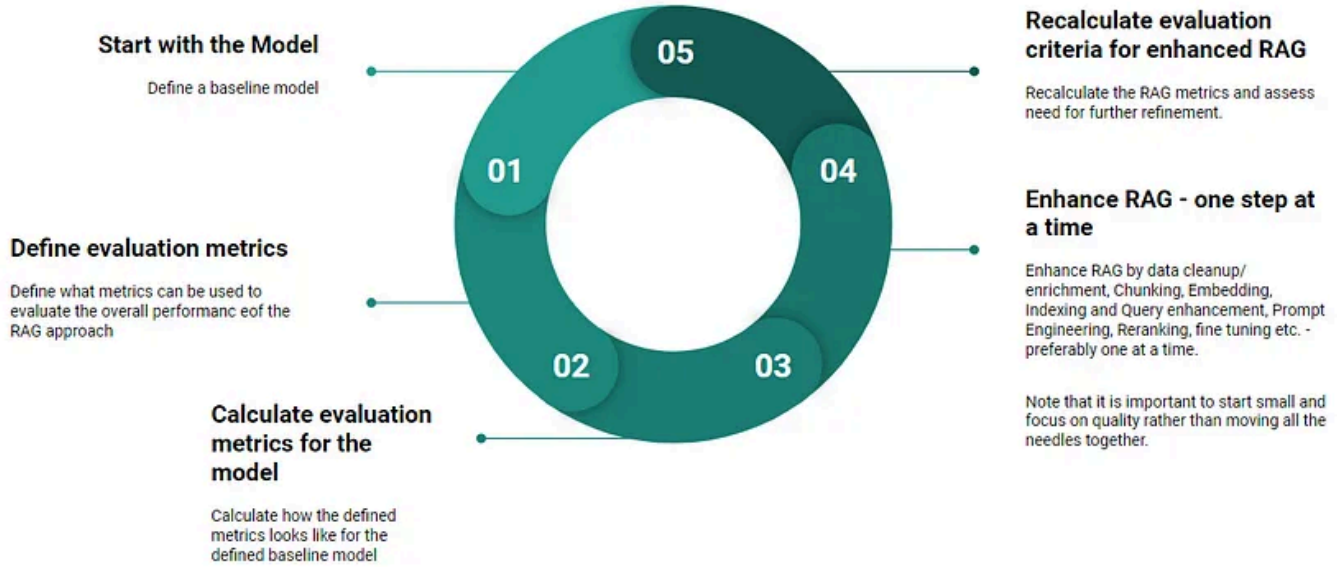
These are some of the key LLM fine tuning techniques, by no means this is an exhaustive list. It is important to note that usually LLM fine tuning is

performed after one has tried extensive experimentation with the Embedding fine tuning.

Overall, LLM fine-tuning is a powerful technique that allows for the customization and optimization of language models to specific tasks and domains. It offers numerous benefits, including improved performance, reduced training time, and increased efficiency and enables us to leverage large-scale pre-training while still tailoring the model to specific requirements.

However, it is important to note that LLM fine-tuning requires careful consideration and expertise. Proper data selection, preprocessing, and hyperparameter tuning are crucial for achieving optimal results. Additionally, ethical considerations, such as bias mitigation and fairness, should be taken into account to ensure responsible and unbiased use of language models.

A structured approach to improving RAG performance — an iterative approach



Improving RAG performance (Image by Author)

The RAG improvement approach starts with a baseline model which is an initial version of the RAG model that serves as a starting point for comparison and evaluation. It represents the performance level before any enhancements or modifications are made.

Following the evaluation criterias are defined to assess the performance of the model. Evaluation criteria metrics are the measures used to assess the performance and effectiveness of a model. These metrics can include accuracy, precision, recall, F1 score, perplexity, or any other relevant measures depending on the specific task. These metrics provide quantitative

insights into how well the model is performing and help in comparing different models or versions.

Then the evaluation metrics for the baseline model are determined by testing the model on a representative dataset or set of examples. The predictions made by the model are compared to the ground truth or expected outputs, and the evaluation metrics are computed based on these comparisons. This allows for an objective assessment of the baseline model's performance.

To enhance the RAG (Retrieval-Augmented Generation) model, several techniques can be applied one at a time. These techniques include data cleanup/enrichment, chunking, embedding, indexing and query enhancement, prompt engineering, re-ranking, and fine-tuning. It is important to start with small enhancements and focus on quality rather than making all the changes simultaneously. This iterative approach allows for better understanding of the impact of each enhancement and ensures that the model is improved step by step.

After applying the enhancements to the RAG model, the evaluation criteria metrics need to be recalculated. The enhanced model is tested on the same dataset or examples used for the baseline model, and the evaluation metrics

are computed based on the new predictions. The evaluation metrics for the enhanced model can then be compared and contrasted with those of the baseline model. This comparison helps in understanding the effectiveness of the enhancements and provides insights into the improvements achieved by the enhanced model compared to the baseline.

Concluding Remarks

This concludes the two part series on improving RAG performances.

In conclusion, improving RAG performances is a complex process that involves multiple stages and key topics. The first part of this blog explored the Ingestion Stage, which included crucial steps such as data clean up, data enrichment, chunking, embedding, vector database and indexing, and query enhancement and prompt engineering. These steps lay the foundation for effective retrieval and ranking in the second part of the process.

It is important to note that while there is a structured approach to improving RAG performances, there is also a significant amount of experimentation involved. Each dataset and scenario may require unique strategies and adjustments to achieve optimal results. This experimentation allows for continuous learning and refinement of the techniques used.

By understanding and implementing the key topics discussed in this blog, individuals and organizations can enhance their RAG performances and achieve more accurate and relevant results. As this area continues to evolve, it is an ongoing journey of exploration and improvement — however, it is essential to grasp the intricacies of these techniques in order to fully exploit their capabilities and successfully implement them in various real-life situations.

Here is a view of what's in the series, feel free to let me know if you would like me to cover any specific aspect.

1. Retrieval-Augmented Generation (RAG) — Basics to Advanced Series (Part 1)

2. Pre-processing block — Chunking (part 2): strategies, considerations and optimization

3. Pre-processing block — Embedding (part 3): Types, Use cases and Evaluation.