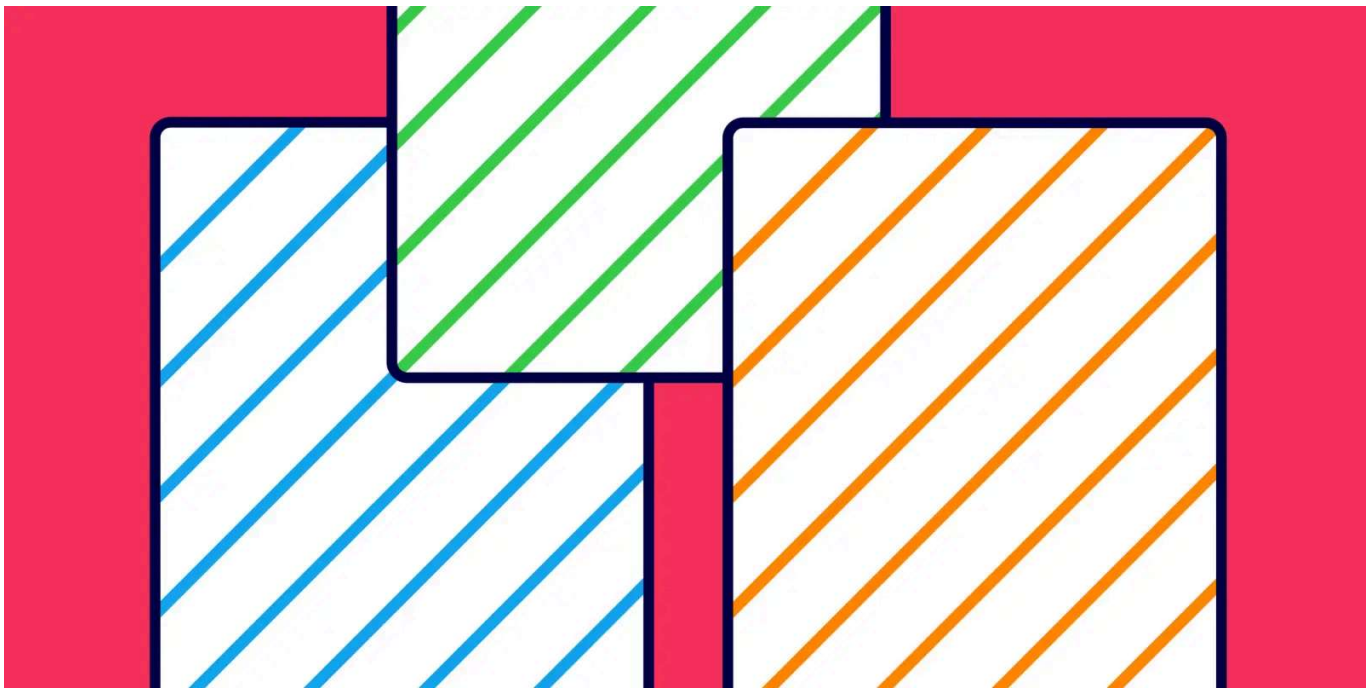
BLOG

RAG Strategies - Hierarchical Index Retrieval

Looking into the scalability problem of the RAG, the article explores the idea of Hierarchical Index Retrieval. A retrieval strategy that aims to deal with the problem through progressive narrowing of the search space.

Franjo Mindek

2024-04-03



The problem of scalability

In the current modern world, data comes in massive volumes. Organizing and retrieving this data is challenging, especially when accuracy and scalability are at stake. Simple index retrieval strategies serve the purpose initially, but as data volume

better accuracy. However, context enrichment is of no help when it comes to scalability.

Of course, everything varies depending on the data you work with, and **knowing your domain is your biggest asset for increasing retrieval accuracy**. So, how do we traverse larger chunks without introducing noise to data? One attempt at that is using a hierarchical retrieval strategy. A strategy that **hierarchically narrows down data** to relevant chunks.

How it works

Hierarchical retrieval, as its name suggests, **executes the retrieval process in a hierarchical order**.

The **number of levels inside a hierarchy can vary** depending on the usage context. If the data structure naturally exists in a hierarchy, it could be worthwhile to mirror it during the retrieval. Yet, to know the optimal number of levels in a hierarchy, there is no workaround to **testing the implementation variations on actual data**.

To illustrate how the hierarchical retrieval strategy operates, let's consider documents composed of titles, chapters, and sections. If we strictly followed the document structure, we would then retrieve only the relevant documents. From those documents, we would then retrieve relevant chapters. From those chapters, we would retrieve relevant sections, and depending on how large they are, from those sections, we would either retrieve a whole section or its chunks. This process, though thorough, results in a lengthy path to retrieve the relevant chunks.

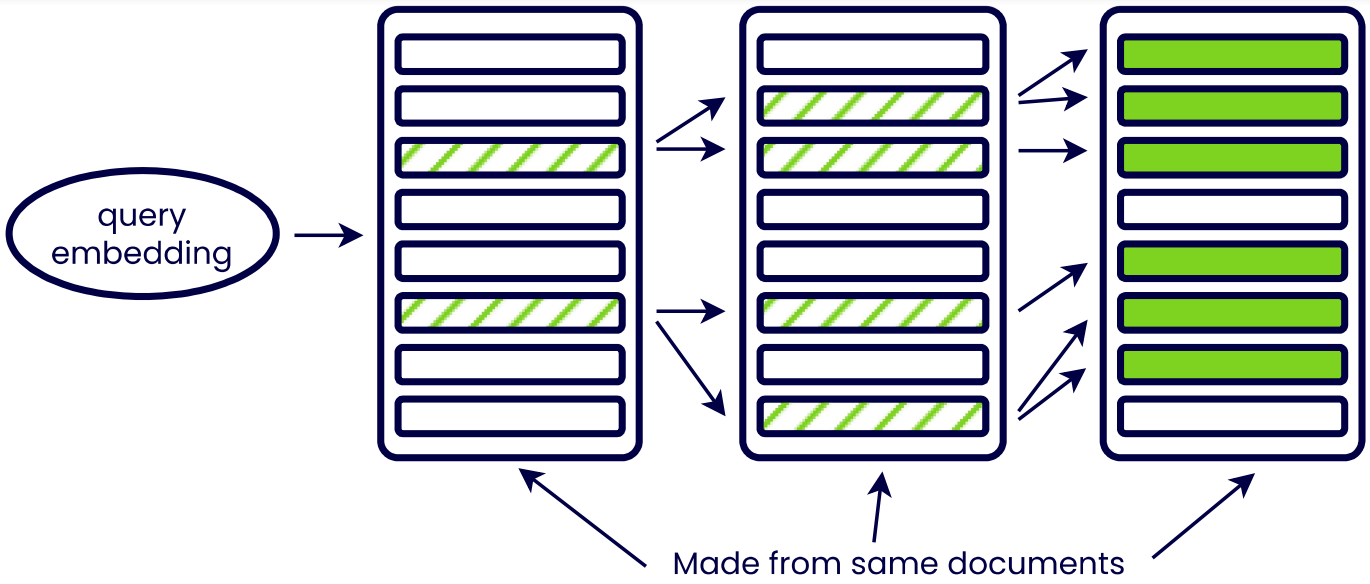


Image 1 - Hierarchical index retrieval overview. The retrieval executes in steps. We start with larger document segments and work towards the smaller chunks. On each level of the hierarchy, we only work with segments that we retrieved as relevant. Each level represents an arbitrary structure of a document.

However, depending on the size of chapters and sections, we might opt for a less complicated plan. We can start with documents again but then retrieve the relevant chapters. From those chapters, we would retrieve relevant chunks. This approach provides a much shorter path while capitalizing on the strengths of the hierarchical approach.

In practice, the number of levels and their configuration can vary. Ultimately, the correct choice boils down to **striking a balance** among factors such as accuracy, vector similarity search time, database round trip time, and document composition.

It's not easy to predict the perfect solution. The easiest way to reduce the effectiveness of a strategy is to **over-complicate** the strategy by introducing too many levels. Most of the time, having just **one level of hierarchy** before retrieving chunks of the document **is sufficient**.

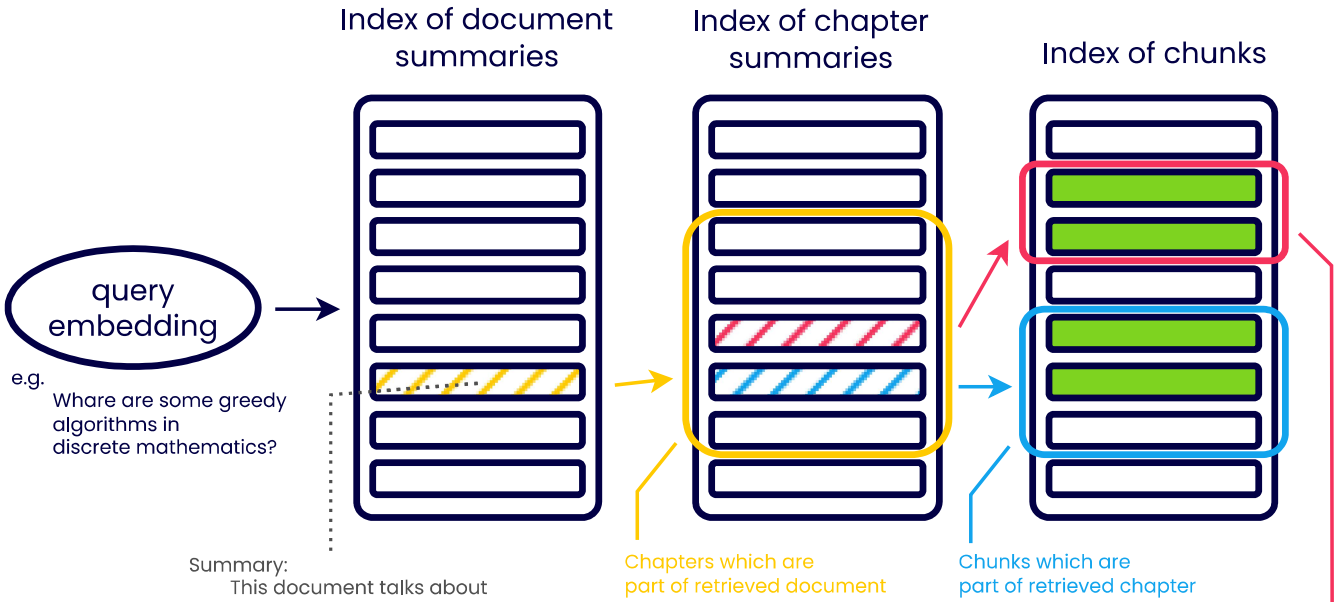
The details

To enable the strategy, we aim to create content summaries at each hierarchy level, except the last one, which contains the actual chunks of the document.

Summaries at the top of the hierarchy **start quite abstract**, highlighting only the key points, as these levels represent large document segments. As we continue down the hierarchy, each summary becomes **more discrete and detailed**. This structure allows the retrieval process to **progressively narrow down** to the relevant data.

This is necessary because when we retrieve relevant entries from the current level, we assess the relevance of the **query embedding against the summary embedding** rather than the content embedding itself. This, in turn, allows us to search based on the semantics of an element without needing to use the entire content as a chunk, which could be impossible due to size limitations.

For example, as of the time of writing this article, OpenAI's embedding models have a maximum input of 8191 tokens, approximately 6100 words.



scaling to millions of documents. The solution lies in using an **AI model trained in summarization**. Choosing the model is a crucial part of the strategy because it determines the quality of summaries, which heavily influences the retrieval accuracy.

It is also important to consider **potential bottlenecks** when implementing this strategy. During the data loading, the **API calls to a summarization model** can slow down the whole process. To optimize the throughput, the loading pipeline should be designed in a manner that allows simultaneous execution of multiple summarizations.

The data we work with has the most impact on the strategy implementation. While the core idea of the strategy remains the same, the implementation will differ, for example, based on whether we work with structured or unstructured documents.

Approaching the document as structured

Working with structured data generally allows us **more precise control** over the strategy implementation. Whether it's HTML, Markdown, or a structured Txt file, as long as there are structural units such as chapters or sections, the file can be structurally parsed. Nevertheless, it's crucial to remember that structured data **doesn't have a one-size-fits-all solution**. Different types of documents will each require their own parsing strategy to fit our retrieval strategy.

How do we transfer the structure from our documents to our strategy and vector database? The answer lies in the previously mentioned metadata, both document and structural.

Document metadata contains information like the title, abstract, and keywords, which can assist the AI model with contextualization. **Structural metadata** provides the data needed to **pinpoint the exact location** of each chunk inside the document. So, when we look at a chunk in our database, through our metadata, we can precisely determine from which document, title, chapter, and section the chunk originates (or any other structure the document adheres to).

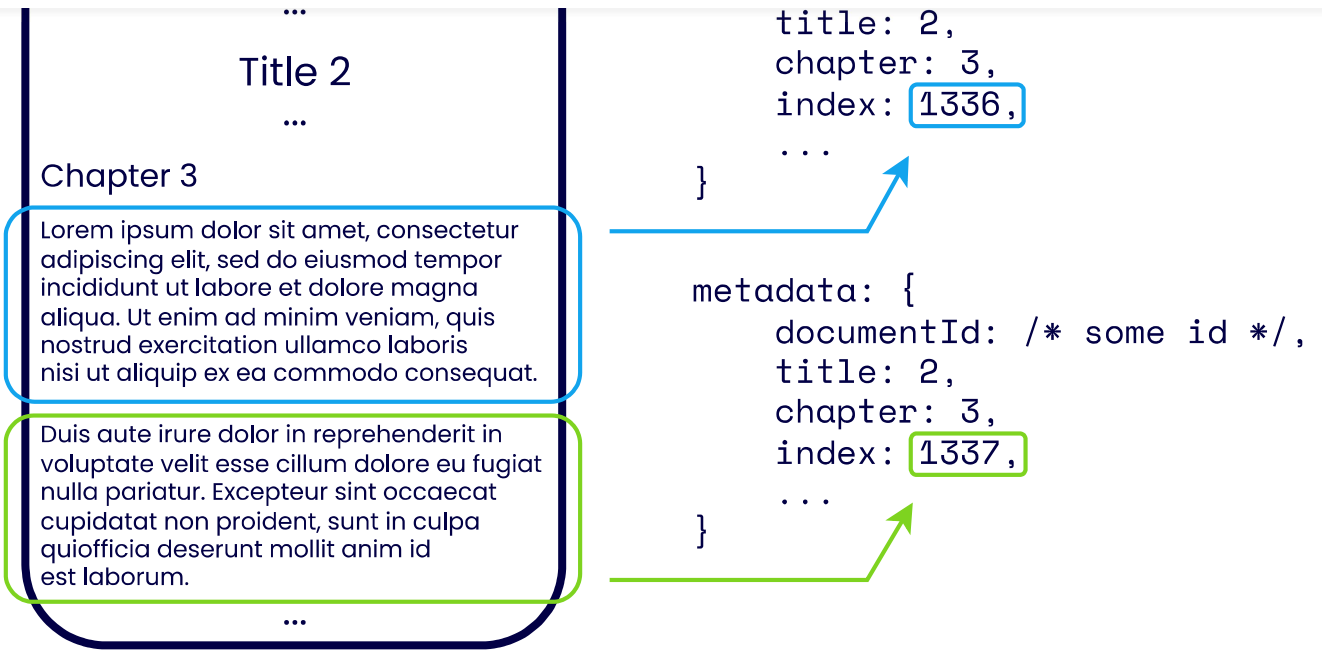


Image 3 - Structural metadata. Its structure should allow us to pinpoint the location of every chunk in our database.

Approaching the document as unstructured

This approach assumes we don't know the structure of the data we are working with beforehand, don't have a structure, or don't want to invest in parsing said structure.

Not knowing or using the structure will make the optimization harder, as we are implementinga a more **generic solution**. However, the upside is that once we implement the solution, we can use it on **hypothetically any textual data.**

One summary per document

One common implementation variant begins with the **summary of an entire document** as the first level of the hierarchy. That allows us to proceed with the

BLOG

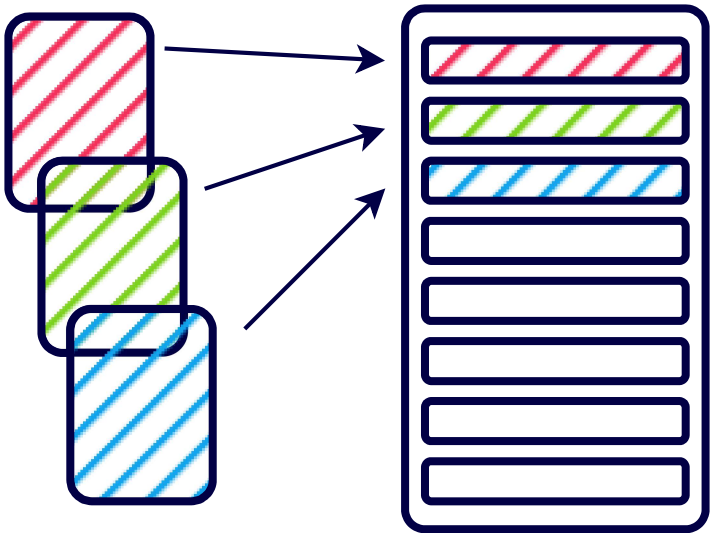


Image 4 - Summary per document approach. While the idea of narrowing down a whole document is appealing, we can end up with summaries of varying quality. Quality can depend on factors such as document size or the performance of the summarization model.

While this solution can help, it can also potentially hinder the accuracy of the retrieval process. Working with hierarchies is all about balancing the size of elements, or more specifically, **balancing the semantics** of the elements. This balance can be challenging to strike without a clear structure.

An edge case where this approach could be problematic is when we can't fit the entire document into the summarization model's context. In the worst-case scenario, we might not even be aware of it, which could result in us summarizing only a part of the document. However, if we are aware, we can attempt to solve it.

One approach to address the size issue is introducing a **"map and reduce"** style of summarization, which splits a document into chunks that can comfortably fit into the summarization model. After summarization, we collect the chunk summaries and combine them into a single chunk. If necessary, this **whole process is repeated** until we have a chunk of a satisfactory size.

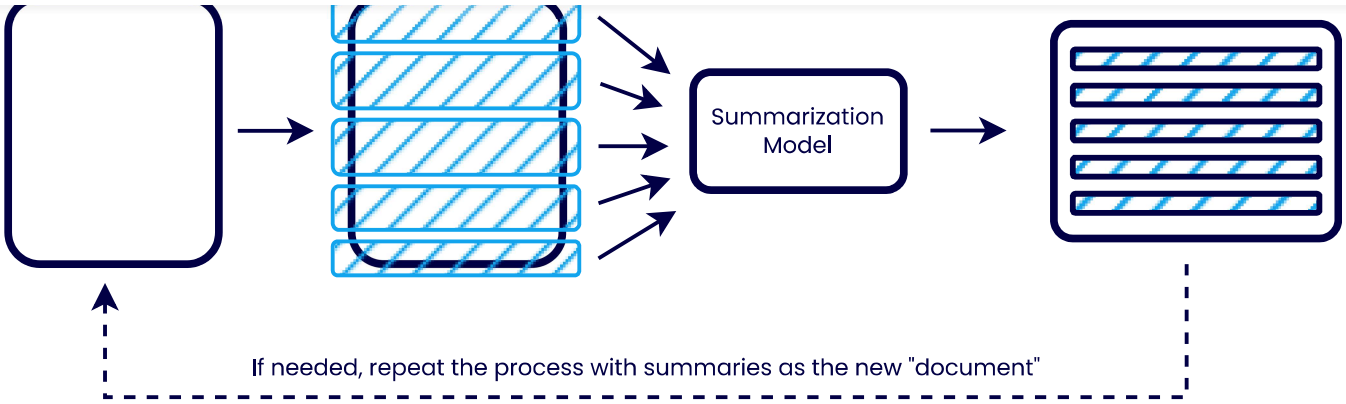


Image 5 - "Map and reduce" summarization.

However, this approach raises a question about the **quality of the final summarization**. Is it possible that the summarization process has omitted a significant part of the document, never to be retrieved due to an unreliable process?

As the algorithm progresses, we gradually lose more of the initial context upon which the summaries were originally based, leaving the summarization model with the task of correctly interpreting them. Therefore, it's critical to ensure that the summarization process is thorough and accurate, to maintain the integrity of the original document.

Multiple summaries per document

In theory, even if there were a model capable of summarizing any document regardless of its size, another issue that might arise is the document being too encyclopedic. When a document covers **too many topics**, it introduces noise to the semantic values, and, as we know, **noise leads to reduced accuracy** during the retrieval. For this reason, there is an alternative way of creating multiple summaries per document.

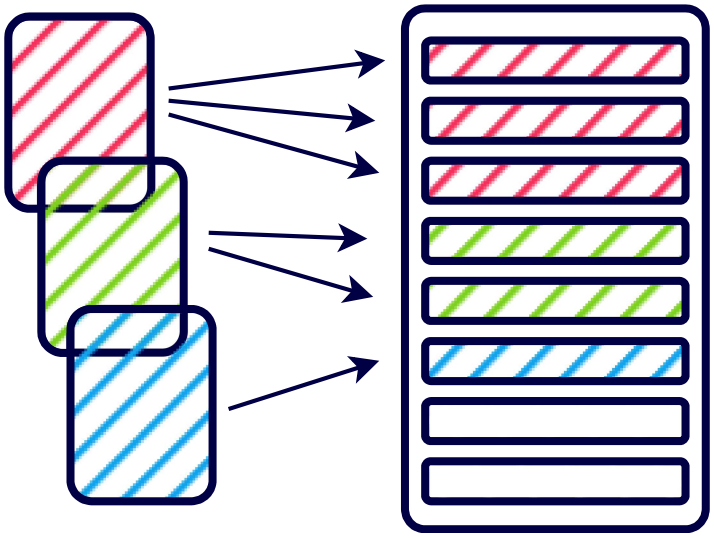


Image 6 - Multiple summaries per document approach. The number of summaries per document will scale with document size.

In this approach, the first level of hierarchy is decided by either the word limit or by reaching the end of the document. This means we can have **any number of starting summaries** for any document. The only parameter to adjust here is how large a chunk we want to summarize.

For example, if we wanted to use 20 pages per starting summary, and if we approximately use 3/4 of a token per word, we can use the following calculation:

```
pages = 20
words_per_page = 500
tokens_per_word = 4/3 // approximation for OpenAI models
token_in_20_pages = 20 * 500 * 4/3 ~= 13333 tokens
```

One thing to keep note of here is to check the size of the last chunk and **redistribute data** to it if it's below a specific limit. This is because, in edge cases, we could end up with something like 10,000 words per summary for the first few chunks and only 300

When creating a hierarchy, we should be aware of the combinatorial explosion. Making our data **too granular could do more harm than good**. It's good to keep the average size of hierarchy elements in mind when deciding how many levels we want.

As mentioned earlier, most of the time, having just one level is sufficient. Even if the document structure supports another level, we should be cautious about whether to include it.

If we are in a situation where we must support a larger number of levels, there are still ways to control the explosion. One solution is that instead of retrieving the *k* most relevant entries, we retrieve only what is relevant. We can do this by introducing a **relevancy cutoff** where entries below a certain similarity threshold won't be retrieved even if they are in the "top 5 most relevant".

If even that solution isn't enough and you are still left with too many chunks, you can **use a reranker**. A reranker is a model that can calculate the relevancy of chunks more accurately than a usual similarity search can, with the tradeoff being that it's much slower. However, when working with a smaller set of retrieved chunks instead of millions of chunks in the database there is no significant slowdown.

Conclusion

Hierarchical index retrieval presents an effective solution for the scalability challenge of ever-increasing data volumes. Its biggest strength lies in the **progressive narrowing** of the relevant data.

We should take our **data** and **domain** into account before building a hierarchy. Knowing the answer to questions such as "Do we work with structured or unstructured data?" gives us a huge advantage. It's important to balance the number of hierarchy levels to prevent unnecessary over-complication and inefficiency.

The use of **metadata** and AI models for **summarization** is essential for successful

In the face of growing data volumes, hierarchical index retrieval stands as a robust strategy to **improve data accuracy and scalability**. This strategy, which organizes data into a **hierarchy** and **progressively narrows down** to the most relevant data, is versatile and adaptable to various use cases, from **structured to unstructured data**.

Key to implementing this strategy is the use of **summaries** and **metadata** to navigate large chunks of data, and the use of AI models for automated summarization. However, we should be wary of keeping the **balance of hierarchy levels** to avoid complexity and inefficiency.

Potential bottlenecks, such as the speed of **API calls** for summarization, need to be considered and mitigated. The combinatorial explosion problem further highlights the need for careful hierarchy design and the possible use of **relevancy cutoffs** or **rerankers**.

While the hierarchical index is a good introduction to leveraging LLMs to improve RAG performance, it's just the **tip of the LLM iceberg**. More on how to use LLMs will be discussed in the following article about augmenting user queries and chunks themselves.

BLOG

Retrieval Augmented Generation AI Blog Series Chunking

Get blog post updates

Sign up to our newsletter and never miss an update about relevant topics in the industry.

E-mail

Sign up

Hi, our name is — **PIXION**

Our best — **WORK**

BLOG