ISSN: 2634 - 8853

Journal of Engineering and Applied Sciences Technology

SCIENTIFIC Research and Community

Review Article

Open Access

Optimizing RAG with Hybrid Search and Contextual Chunking

Ashish Bansal

USA

ABSTRACT

Large pre-trained language models have been shown to store factual knowledge in their parameters, and achieve state- of-the-art results when fine-tuned on down- stream NLP tasks. However, their ability to access and precisely manip- ulate knowledge is still limited, and hence on knowledge- intensive tasks, their performance lags behind task-specific architectures. Additionally, providing provenance for their decisions and updating their world knowledge remain open research problems. Pre- trained models with a differentiable access mechanism to explicit non-parametric memory have so far been only investigated for extractive downstream tasks. Retrieval-Augmented Generation (RAG) is a prevalent ap- proach to infuse a private knowledge base of documents with Large Language Models (LLM) to build Generative Q&A(Question-Answering) systems. However, RAG accu- racy becomes increasingly challenging as the corpus of docu- ments scales up,with Retrievers playing an outsized role in the overall RAG accuracy by extracting the most relevant document from the corpus to provide context to the LLM. In this paper, we propose different ways to optimize the re- treivals, Reciprocal Rank Fusion, Reranking and dynamic chunking schemes.

***Corresponding author**

Ashish Bansal, USA.

Received: August 07, 2023; **Accepted:** August 14, 2023; **Published:** August 28, 2023

Keywords: RAG, Retrievers, Semantic Search, Dense In-Dex, Vector Search, Hybrid Search, Sparse Embeddings

Introduction

Generative AI is taking the world by storm. However, one of the first challenges organizations face in deploying large lan- guage models (LLMs) in their corporate environment is how to make LLMs understand their proprietary enterprise data. However, many of them ran into similar problems while try- ing to integrate generative AI into enterprise environments, like privacy breaches, lack of relevance, and a need for better personalization in the results they received.

To address this, most have concluded that the answer lies in retrieval augmented generation (RAG). Retrieval aug- mented generation (RAG) is the leading technique for en- hancing LLMs with enterprise data. For example, to ensure chatbots that are powered by LLMs are responding with accurate, relevant responses, companies use RAG to give LLMs domain-specific knowledge drawn from user manu- als or support documents.

RAG separates knowledge retrieval from the genera- tion process via external discovery systems like enterprise search. This enables LLMs and the responses they provide to be grounded in real, external enterprise knowledge that can be readily surfaced, traced, and referenced.

RAG represents an approach to text generation that is based not only on patterns learned during training but also on dynamically retrieved external knowledge. This method combines the creative flair of generative models with the encyclopedic recall of a search engine. The efficacy of the RAG system relies fundamentally on two components: the Retriever (R) and the Generator (G), the latter representing the size and type of LLM.

The language model can easily craft sentences, but it might not always have all the facts. This is where the Re- triever (R) steps in, quickly sifting through vast amounts of documents to find relevant information that can be used to inform and enrich the language model's output. Think of the retriever as a researcher part of the AI, which feeds the con- textually grounded text to generate knowledgeable answers to Generator (G). Without the retriever, RAG would be like a well-spoken individual who delivers irrelevant information. Retrieval-Augmented Generation (RAG) is revolutioniz- ing traditional search engines and AI methodologies for in- formation retrieval. However, standard RAG systems em- ploying simple semantic search often lack efficiency and precision when dealing with extensive data repositories. Hy- brid search, on the other hand, combines the strengths of different search methods, unlocking new levels of efficiency and accuracy. Hybrid search is flexible and can be adapted to tackle a wider range of information needs.

Hybrid search can also be paired with semantic rerank- ing (to reorder outcomes) to further enhance performance. Combining hybrid search with reranking holds immense po- tential for various applications, including natural language processing tasks like question answering and text summa- rization, even for implementation at a large-scale.

Hybrid Search

In current Retrieval-Augmented Generation (RAG) systems, word embeddings are used to represent data in the vector database, and vector similarity search is commonly used for searching through them. For LLMs and RAG systems, em- beddings - because they capture semantic relationships between words - are generally preferred over keyword-based representations like Bag-of-words

(BoW) approaches.

$$\sum_{i}^{n} IDF(q_i) \frac{f(q_i, D) * (k1 + 1)}{f(q_i, D) + k1 * (1 - b + b * \frac{fieldLen}{avgFieldLen})}$$

Figure 1: BM25 Algorithm

$$ln\left(1 + \frac{(docCount - f(q_i) + 0.5)}{f(q_i) + 0.5}\right)$$

Figure 2: idfequation

But each of vector similarity search and keyword search has its own strengths and weaknesses. Vector similarity search is good, for example, at dealing with queries that contain typos, which usually don't change the overall intent of the sentence. However, vector similarity search is not as good at precise matching on keywords, abbreviations, and names, which can get lost in vector embeddings along with the surrounding words. Here, keyword search performs bet- ter.

Therefore, combining semantic search with traditional keyword-based search (hybrid) proves beneficial in over- coming these limitations and yielding better results. By in- tegrating the strengths of both search algorithms, hybrid search enhances the relevance of returned search results, al- lowing for a comprehensive search over both document con- tent and underlying meaning in RAG applications. We com- bines vector and keyword search methods. In this context, we used the widely utilized BM25 algorithm as part of the keyword search component. BM25 relies on lexical match- ing, scoring documents based on query term frequency and document length normalization.

The BM25 Algorithm

We can see a few common components like qi, IDF(qi), f(qi,D), k1, b, and something about field lengths. Here's what each of these is all about:

- qi is the ith query term.
- For example, if I search for "top," there's only 1 query term, so q0 is "top". If I search for "top mate" in English, Elasticsearch will see the whitespace and tokenize this as 2 terms: q0 will be "top" and q1 will be "mate". These query terms are plugged into the other bits of the equation and all of it is summed up.
- IDF(gi) is the inverse document frequency of the ith query term. The IDF component of our formula mea- sures how often a term occurs in all of the documents and "penalizes" terms that are common. The actual formula Lucene/BM25 uses for this part is:Where docCount is the total number of doc- uments that have a value for the field in the shard (across shards, if you're using search_type=fs_query_then_fetch and f(qi) is the number of documents which contain the ith query term.

- We see that the length of the field is divided by the av- erage field length in the denominator as fieldLen/avg- FieldLen. We can think of this as how long a document is relative to the average document length. If a document is longer than average, the denominator gets bigger (decreasing the score) and if it's shorter than average, the denomi- nator gets smaller (increasing the score). Note that the implementation of field length in Elasticsearch is based on number of terms (vs something else like character length). This is exactly as described in the original BM25 paper, though we do have a special flag (discount over- laps) to handle synonyms specially if you so desire. The way to think about this is that the more terms in the docu- ment — at least ones not matching the query — the lower the score for the document. Again, this makes intuitive sense: if a document is 300 pages long and mentions my name once, it's less likely to have as much to do with me as a short tweet which mentions me once. Vector search represents documents as dense embed- dings, indexing them in a vector space. Queries are em- bedded into the same space and relevant documents are found by semantic similarity between the query and doc- ument vectors rather than exact term matching. The two techniques can be combined, with BM25 providing lexi- cal matching signals and vector search providing seman- tic matching.
- We see a variable b which shows up in the denominator and that it's multiplied by the ratio of the field length we just discussed. If b is bigger, the effects of the length of the document compared to the average length are more amplified. To see this, you can imagine if you set b to 0, the effect of the length ratio would be completely nulli- fied and the length of the document would have no bear- ing on the score.
- Finally, we see two components of the score which show up in both the numerator and the denominator: k1 and f(qi,D).
- f(qi,D) is "how many times does the ith query term oc- cur in document D
- k1 is a variable which helps determine term frequency saturation characteristics. That is, it limits how much a single query term can affect the score of a given docu- ment. It does this through approaching an asymptote.

A higher/lower k1 value means that the slope of "tf() of BM25" curve changes. This has the effect of changing how "terms occurring extra times add extra score." An interpre- tation of k1 is that for documents of the average length, it is the value of the term frequency that gives a score of half the maximum score for the considered term. The curve of the impact of tf on the score grows quickly when tf() k1 and slower and slower when tf() > k1.

Dense and Sparse Vectors

Benefits of using Hybrid search:

Precision: Keyword search enables exact matches to the query, leaving no room for ambiguity.

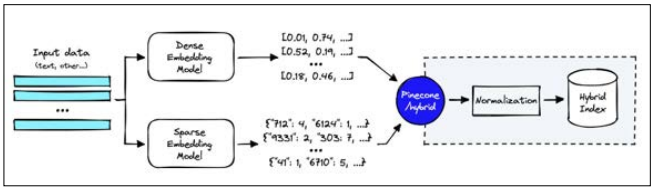

Figure 3: Hybrid Search Implementation

Context: Semantic search allows algorithms to under- stand the intent of the query. If no keywords are matched, the semantic search will step in to analyze the context and meaning behind the query, ensuring that relevant re- sults are still provided and covering any gaps in keyword- based matching.

Relevance: Both techniques complement each other and improve relevance for unseen queries.

That being said, keyword search is not as good as vec- tor similarity search at fetching relevant results based on se- mantic relationships or meaning, which are only available via word embeddings. For example, a keyword search will relate the words

"the river bank" and "the Bank of America" even though there is no actual semantic connection between the terms - a difference to which vector similarity search is sensitive. Keyword search would, therefore, benefit from vector search, but the prevailing approach is not to combine them but rather to implement them separately using distinct methodologies.

In hybrid search -a keyword-sensitive semantic search ap- proach, we combine vector search and keyword search algo- rithms to take advantage of their respective strengths while mitigating their respective limitations. Let's take a look at the components that make up the ar- chitecture of hybrid search. Hybrid search combines keyword-based and vector search techniques by fusing their search results and reranking them. Keyword-based search in the context of hybrid search often uses a representation called sparse embeddings, which is why it is also referred to as sparse vector search.Sparse embeddings can be generated with different algorithms. The most commonly used algorithm for sparse embeddings is BM25 (Best match 25), which builds upon the TF-IDF (Term Frequency-Inverse Document Frequency) approach and refines it. BM25 is expalined in the above section.

Sparse embeddings are vectors with mostly zero values with only a few non-zero values, as shown below.

[0, 0, 0, 12, 23, 0, 0, 0] Vector search Vector search is a modern search technique that has emerged with the advances in ML. Modern ML al- gorithms, such as Transformers, can generate a numerical representation of data objects in various modalities (text, im- ages, etc.) called vector embeddings. These vector embeddings are usually densely packed with information and mostly comprised of non-zero values (dense vectors), as shown below. This is why vector search is also known as dense vector search.

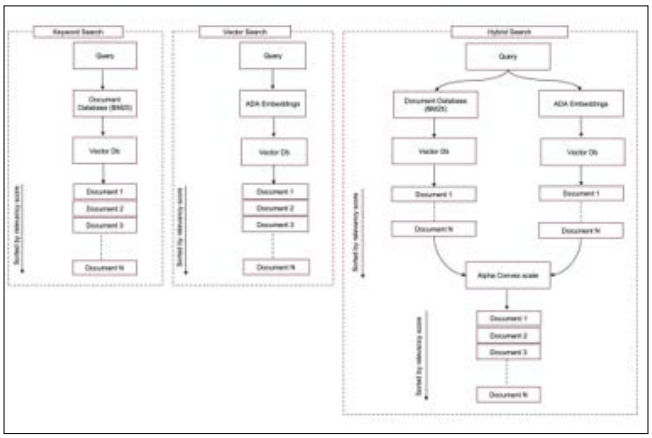

Figure 4: Keyword vs Vector vs Hybrid search

[0.634, 0.234, 0.867, 0.042, 0.249, 0.093, 0.029, 0.123, 0.234,] A search query is embedded into the same vector space as the data objects. Then, its vector embedding is used to cal- culate the closest data objects based on a specified similarity metric, such as cosine distance. The returned search results list the closest data objects ranked by their similarity to the search query.

Fusion of Keyword-Based and Vector Search Results

- **Weighting-based fusion**

Both the keyword-based search and the vector search re- turn a separate set of results, usually a list of search re- sults sorted by their calculated relevance. These separate sets of search results must be combined. There are many different strategies to combine the ranked results of two lists into one single ranking, as outlined in a paper by Benham and Culpepper [1].

Generally speaking, the search results are usually first scored. These scores can be calculated based on a spec- ified metric, such as cosine distance, or simply just the rank in the search results list.

Then, the calculated scores are weighted with a parame- ter alpha, which dictates each algorithm's weighting and impacts the results re-ranking.

hybrid_score = (1- alpha) * sparse_score + alpha * dense_score

Usually, alpha takes a value between 0 and 1, with

- alpha = 1: Pure vector search
- alpha = 0: Pure keyword search

Below, you can see a minimal example of the fusion be- tween keyword and vector search with scoring based on the rank and an alpha = 0.5

Below, you can see a minimal example of the fusion be- tween keyword and vector search with scoring based on the rank and an alpha = 0.5

A RAG pipeline has many knobs you can tune to improve its performance. One of these knobs is to improve the relevance of the retrieved context that is then fed into the LLM because if the retrieved context is not relevant for answering a given question, the LLM won't be able to generate a relevant answer either.

Depending on your context type and query, you have to determine which of the three search techniques is most beneficial for your RAG application. Thus, the parameter alpha, which controls the weighting between keyword- based and semantic search, can be viewed as a hyperpa- rameter that needs to be tuned

- **Rank fusion**

Rank fusion algorithms, particularly Reciprocal Rank Fusion (RRF), provided a promising alterna- tive.Reciprocal Rank Fusion (RRF), a simple method for combining the document rankings from multiple IR systems, consistently yields better results than any individual system, and better results than the standard method Condorcet Fuse. This result is demonstrated by using RRF to combine the results of several TREC experiments, and to build a meta-learner that ranks the LETOR 3 dataset better than any previously reported method. Here's how most rank fusion algorithms work:

- **Rank assignment:** Each document from the individual ranked lists is assigned a score based on its rank posi- tion. Typically, the score is the reciprocal of its rank (i.e., 1/rank). For example, a document ranked first gets a score of 1, the second gets 0.5, the third gets 0.33, and so on.
- **Score summation:** The scores from all ranked lists are summed for each document. Documents appearing in multiple lists accumulate higher combined scores.
- **Final ranking:** Documents are re-ranked based on their combined scores, producing a final ranked list that in- tegrates the rankings from all individual search en- gines.

Chunking

Chunking is an essential preprocessing step when preparing data for RAG for a number of reasons.

While the embedding model imposes a hard maximum limit on the number of tokens it can embed, that doesn't mean your chunks need to reach that length. It simply means they can't exceed it. In fact, utilizing the maximum length for each chunk, such as 6200

words (8K tokens), may be excessive in many scenarios. There are several compelling reasons to opt for smaller chunks.

Common Approaches to Chunking

- **Character splitting:**

The very basic way to split a large document into smaller chunks is to divide the text into N-character sized chunks. Often in this case, you would also specify a certain num- ber of characters that should overlap between consec- utive chunks. This somewhat reduces the likelihood of sentences or ideas being abruptly cut off at the boundary between two adjacent chunks. However, as you can imag- ine, even with overlap, a fixed character count per chunk, coupled with a fixed overlap window, will inevitably lead to disruptions in the flow of information, mixing of dis- parate topics, and even sentences being split in the mid- dle of a word. The character splitting approach has abso- lutely no regard for document structure.

- **Sentence-Level Chunking or Recursive Chunking**

Character splitting is a simplistic approach that doesn't take into account the structure of a document at all. By relying solely on a fixed character count, this method of- ten results in sentences being split mid-way or even mid- word, which is not great.

One way to address this problem is to use a recursive chunking method that helps to preserve individual sen- tences. With this method you can specify an ordered list of separators to guide the splitting process. For example, here are some commonly used separators:

"\n\n" - Double new line, commonly indicating para- graph breaks
"\n" - Single new line "." - Period
" " - Space

If we apply the separators listed above in their specified order, the process will go like this. First, recursive chunk- ing will break down the document at every occurrence of a double new line ("\n\n"). Then, if these resulting seg- ments still exceed the desired chunk size, it will further break them down at new lines ("\n"), and so on.

While this method significantly reduces the likelihood of sentences being cut off mid-word, it still falls short of capturing the complex document structure. Documents often contain a variety of elements, such as paragraphs, section headers, footers, lists, tables, and more, all of which contribute to their overall organization. However, the recursive chunking approach outlined above primar- ily considers paragraphs and sentences, neglecting other structural nuances.

- **Optimized Chunking**

This is the technique where you optimized the chunk with given contraints of the token size and how the documnet is layout such as a document can contains table, images and other components. This can be down by converting the documnet to markdown format and then chunking based on the headings, sub- headings. More ways to handle table in dynamic pages where there are huge tables, to handle this cases the table can chunked into smaller tables making sure the headers are stored for all the chunks to keep the semantics of information. This header information is very useful in correctly identifying in interpreting these tables.

Other piece is to handle the images within a document where VLM (Vision Language Model) can be utilzed to extract those information.

Chunking is one of the essential preprocessing steps in any RAG system. The choices you make when you set it up, will influence the retrieval quality, and as a consequence, the overall performance of the system. Here are some consider- ations to keep in mind when designing the chunking step:

Experiment with different chunk sizes: While large chunks may contain more context, they also result in coarse representation, negatively affecting the retrieval precision. Optimal size chunk depends on the nature of your documents, but aim to optimize for smaller chunks without losing important context.

token len	Vectors	success	near miss
500-1000	vector search	90(67%)	9(7%)
500-1000	Hybrid search($\alpha = 0.5$)	99(71%)	0

- Utilize smart chunking strategies: Opt for chunking strategies that allow you to separate text on semantically meaningful boundaries to avoid interrupting the information flow, or mixing content.
- Evaluate the impact of your chunking choices on the overall RAG performance: Set up an evaluation set for your specific use case, and track how your experiments with chunk sizes and chunking strategies impact the overall per- formance. Unstructured streamlines chunking experimenta- tion by allowing you to simply tweak a parameter or two, no matter the documents' type.

Results

For any Rag systems, the first part of the answering any question is the search for the document where the answer ex- ists. To search the underlying documents we were relying on the sematic search using the embeddings from ADA model. As we explore more on the searching capabilities using these embeddings, we have to remember that vector databases are not the panacea of search – they are very good at semantic search, but in many cases, traditional keyword search can yield more relevant results and increased user satisfaction.

In our experiment, search is based purely on semantic search using vectors from ADA model. The semantic search helps to bring the required URL to answer the given query for around 67% of time (top 5) and semantic search tends to nearly miss the urls for 3% of queries i.e. URLs exist within near miss window (top 6-10). To improve the search and get the required URL at top, Hybrid search was a promis- ing direction to improve the searching capabilities. We have seen that from this methodology we are able to convert those near miss to a success in terms for source search. Results in search results table [1-16].

Conclusion

In this paper we talked about the hybrid search and chunk- ing that can be utilized to optimize the RAG system, the con- text of hybrid search as a combination of keyword-based and vector searches. Hybrid search merges the search results of the separate search algorithms and re-ranks the search re- sults accordingly.

In hybrid search, the parameter alpha controls the weight- ing between keyword-based and semantic searches. This pa- rameter alpha can be viewed as a hyperparameter to tune in RAG pipelines to improve the accuracy of search results.

We proceeded with our hypothesis of Hybrid search can boost the search capabilities of QA System and we exper- imented with including sparse embedding and making our search a hybrid search with getting best of both semantic and keyword search. As well as with dynamic chunking(discussed above) we were able to get better chunks which helped in better search and good synthesis for the RAG systems.

References

1. Amati G (2009) BM25 Boston, MA: Springer US 257-260.
2. Robertson S, Zaragoza H (2009) The bm25 algorithm, Foundations and Trends in Information Retrieval.
3. Johnson M (2019) Knn algorithms for semantic search. in Proceedings of the International Conference on Ma- chine Learning.
4. Jeff Johnson, Matthijs Douze, Herve´ Je´gou (2019) Billion-scale similarity search with gpus. IEEE Transac- tions on Big Data 7: 535-547.
5. Taunk K, De S, Verma S, Swetapadma A (2019) A brief review of nearest neighbor algorithm for learning and classification, in 2019 International Conference on In- telligent Computing and Control Systems (ICCS) 1255-1260.
6. Kwiatkowski T, Palomaki J, Redfield O, Collins M, Parikh A, et al. (2019) Natural questions: a benchmark for question answering research, Transactions of the Association of Computa- tional Linguistics.
7. Wang LL, Lo K, Chandrasekhar Y, Reas R, Yang J, et al. (2020) Cord-19: The covid-19 open research dataset. ArXiv.
8. Yang Z, Qi P, Zhang S, Bengio Y, Cohen WW, et al. (2018) Hotpotqa: A dataset for diverse, explainable multi-hop question answering. arXiv preprint arXiv: 1809.09600.
9. Wang Y, Wang L, Li Y, He D, Liu TY (2013) A theo- retical analysis of ndcg type ranking measures, in Con- ference on learning theory 25-54.
10. Rajpurkar P, Zhang J, Lopyrev K, Liang P (2016) Squad: 100,000+ questions for machine comprehension of text, arXiv preprint ar Xiv: 1606.05250.
11. Reddy S, Chen D, Manning CD (2019) Coqa: A con- versational question answering challenge. Transactions of the Association for Computational Linguistics 7: 249-266.
12. Siriwardhana S, Weerasekera R, Wen E, Kalu- arachchi T, Rana R, et al. (2023) Improving the domain adaptation of retrieval augmented genera- tion (rag) models for open domain question answering. Transactions of the Association for Computational Lin- guistics 11: 1-17.
13. AWS-Whitepaper (2021) Hybrid machine learning.
14. Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dry- den, Alexandra Peste (2021) Sparsity in deep learn- ing: Pruning and growth for efficient inference and train- ing in neural networks. J Mach Learn Res 22.
15. Haney D, Gibson D (2023) in Stack Overflow Blog. Ask like a human: Implementing semantic search on Stack Over- flow.
16. https://www.pinecone.io/learn/hybrid-search-intro/.