Retrieval Augmented Generation – The Process of Improving Output Using External Resources

Sebastian Newberry Endri Islami Parthiv Gajula
April 28, 2025

Abstract

Retrieval-Augmented Generation (RAG) is a framework that combines document retrieval with language generation to produce accurate, evidence-based responses. The process begins by retrieving documents relevant to a user query, which are then used to either extract exact answers or provide context for generation. As the size of the document corpus grows, retrieval becomes increasingly challenging. To address this, retrievers leverage document embeddings stored in vector databases to efficiently compute similarity scores between queries and documents. Advanced models, such as REALM, incorporate a reader component to re-rank retrieved documents based on relevance before passing them to a generator model like GPT. Other approaches, such as RETRO, integrate retrieved document chunks directly into the generation process using cross-attention, enabling tighter alignment between retrieval and generation. Although these methods share the common goal of improving factuality and relevance, they differ in their architectural choices and objectives. These objectives could be things like prioritizing retrieval accuracy, evidence extraction, or seamless integration with generation.

1 Introduction

The objective of this report is to develop a methodology for retrieving relevant information from a corpus of documents and either leveraging it alongside a generation model to produce accurate responses to user prompts, or finding an exact answer in a huge corpus of documents. Before delving into the detailed mechanics of both of these approaches, it is important to establish key terminology that will be referenced throughout the paper.

In the following sections, several important concepts will be defined. **Indexing** refers to the process of organizing and structuring document chunks to enable efficient retrieval. **Retrieval** denotes the method of selecting the most relevant documents from a larger collection based on a given query. A **reader** is defined as a model or algorithm that evaluates the relationship between a query and candidate documents. Finally, a **generator** is a model tasked with creating a coherent response using the information identified during retrieval and reading.

In retrieval systems, two primary encoder architectures are commonly utilized: the bi-encoder and the cross-encoder. A bi-encoder consists of two independently applied BERT models: one encodes the query and the other encodes candidate documents. Each input is processed separately, producing dense vector embeddings that capture the semantic content of either the query or the document. Relevance is then assessed by comparing these embeddings using similarity metrics such as Maximum Inner Product Search (MIPS) or Maximum Cosine Similarity (MCS).

In contrast, a cross-encoder jointly encodes the query and document by concatenating them and processing them together with a single BERT model. Self-attention is computed across both the query and document tokens, allowing for fine-grained interaction between them. A cross-encoder is typically trained to output a single relevance score indicating the degree of match between a query and a document. It can also be trained for more complex tasks, such as predicting the start and end token positions corresponding to an answer span within a document.

Although cross-encoders are highly effective at modeling fine-grained contextual similarities between queries and documents, they are computationally infeasible to use for large-scale retrieval. Evaluating a query against every document in a corpus would require running full self-attention across millions or billions of document-query pairs, which is prohibitively expensive in both time and computational resources. As a result, most retrieval systems rely on bi-encoder architectures for initial retrieval, sacrificing some fine-grained matching ability in exchange for scalability and efficiency.

2 Literature Review

The first significant research on Retrieval-Augmented Generation (RAG) was published in early 2020. During this period, much of the focus in the field was on developing algorithms capable of retrieving and ranking documents ef-

ficiently, while generation remained a secondary concern.

One of the earliest and most influential approaches was **REALM** (Retrieval-Augmented Language Model Pre-Training) [**?**]. REALM employs a bi-encoder architecture, where queries and documents are embedded separately to facilitate efficient retrieval. A key contribution introduced by REALM was the formulation of retrieval as part of a probabilistic framework. Specifically, the probability of producing a correct output y given an input x is expressed by marginalizing over possible retrieved documents z:

$$p(y \mid x) = \sum_{z \in \mathcal{Z}} p(y \mid z, x)\, p(z \mid x)$$

REALM is specifically designed for tasks involving the identification of an exact answer span to a prompt within a document, predicting masked tokens, or finding additional context to be used for Open-Domain Question Answering (Open-QA). To support retrieval, REALM uses a bi-encoder-based **neural knowledge retriever**, embedding queries and documents separately. Both the query encoder and, optionally, the document encoder are trained during pretraining. Because updating the document encoder during training causes the document embeddings to become stale, REALM periodically re-embeds and re-indexes all documents every several hundred training steps to maintain retrieval consistency.

After retrieving the top-k documents for a query, a **knowledge-augmented encoder** is applied to each document separately. This reader model predicts a probability distribution over possible start and end positions for the answer span within each document. The final output is selected as the span with the highest predicted probability across all retrieved documents. For this component, REALM employs a cross-encoder architecture by concatenating the query and document, allowing the model to compute fine-grained contextual interactions between query and document passages.

Later in June and September 2020, two additional algorithms were developed to improve the retrieval component of retrieval-augmented generation frameworks. These algorithms are **ColBERT** (Contextualized Late Interaction over BERT) and **DPR** (Dense Passage Retrieval). Each provides a different approach to optimizing the retrieval probability $p(z \mid x)$ within the RAG framework.

DPR [**?**] is designed to embed large passages into dense vector representations, which can then be efficiently stored and retrieved from a vector database. Compared to REALM, DPR introduces a key innovation: the use of contrastive learning to better supervise the retriever. In DPR, the model is trained to maximize the similarity between a query and its corresponding positive passage while minimizing similarity with negative passages. The paper introduces three methods for sampling negative passages during training:

- **Random negatives:** Randomly selected passages that do not contain the correct answer.
- **BM25 negatives:** Top-ranked passages returned by a BM25 search that match many question tokens but do not contain the answer.
- **Gold negatives:** Passages that serve as positive examples for other questions in the training dataset.

ColBERT [**?**] introduces a new approach to retrieval that differs from traditional dense retrievers. Like other retrieval algorithms, ColBERT uses a bi-encoder architecture where queries and documents are encoded separately using BERT. However, instead of pooling document or query representations into a single vector, ColBERT preserves token-level embeddings for both queries and documents.

Each document token is independently encoded and stored in a compressed vector index. At retrieval time, each query token is encoded separately, and the similarity between a query and a document is computed through a late interaction mechanism. Specifically, for each query token, the maximum similarity with any document token is computed, and the overall score is the product of these maximum similarities:

$$S_{q,d} := \prod_{i \in [|E_q|]} \max_{j \in [|E_d|]} E_{q_i} \cdot E_{d_j}$$

Unlike traditional cross-encoders that jointly attend over both queries and documents, ColBERT focuses on independent token-level matching. While this design allows for highly efficient retrieval over lots of tokens, it also means that ColBERT captures token-level relevance without modeling full cross-token contextualization between the query and document. This means that ColBERT can be bad at understanding context of a sentence and instead often times only understands the meaning of tokens in documents but not exactly the entire context they are used in.

ColBERT also uses the same contrastive learning that DPR uses. ColBERT built on this contrastive learning by using relevance guided supervision. [**?**] This method is intended to solve the problem of finding negative samples to compare against when training ColBERT. It works by finding negative samples in an initial ColBERT model with a sparse vector retrieval algorithm like BM25, then once the ColBERT retriever gets trained once, a new retriever is created that bases its negative samples off of the previous trained retriever. In other words, after a certain batch of samples, a ColBERT retriever model will learn the bad samples, then a

new ColBERT retriever will be created that uses the previous, partially trained retriever to find negative samples.

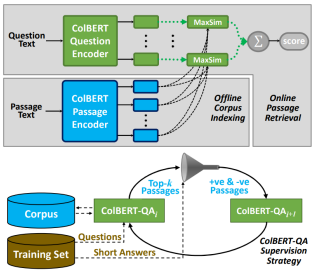

Figure 1: Top: This figure showcases ColBERT token-by-token similarity search across all documents one query token at a time. Bottom: This figure shows ColBERT retrievers being trained based on negative samples coming from previous retrievers.

Later in February 2022, research for one of the first generation focused models came out. This model is called **RETRO** (Retrieval Enhanced Transformer) [**?**]. This model separates an input into chunks, and fetches appropriate top-k documents for each chunk. It uses the traditional bi-encoder architecture to retrieve documents, then passes the original query through a self-attention layer to learn context, then the documents through cross-attention with the learned query embeddings to an autoregressive decoder. This part of the process is called chunked cross attention. During chunked cross attention, subsequent tokens are generated only based on previous tokens just like a GPT model, but documents that were fetched from the previous chunk are used with cross attention. Cross attention occurs with the last token of the previous chunk and first 4 tokens of the next chunk in this specific implementation:

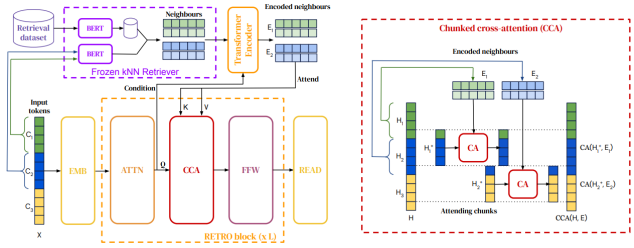

Figure 2: Left: simplified version where a sequence of length $n = 12$ is split into $l = 3$ chunks of size $m = 4$. For each chunk, we retrieve $k = 2$ neighbours of $r = 5$ tokens each. Theretrieval pathway is shown on top. Right: Details of the interactions in the Cca operator. Causality is maintained as neighbours of the first chunk only affect the last token of the first chunk and tokens from the second chunk.

This version of crossed chunk attention allows for retrieving relevant documents, and also generating portions of text based on the next chunk of query tokens, along with documents that are relevant to the previous input. One downside of RETRO is that it isn't compatible with pre-trained LLMs like LLAMA or GPT. In order to use it, both the retrieval part and attention parts of this algorithm must be trained from scratch.

3 Problem Statement

The objective of Retrieval-Augmented Generation (RAG) to generate accurate outputs by leveraging external knowledge retrieved from a large corpus of documents can be mathematically expressed by marginalizing over the possible retrieved documents z as shown in the equation in the previous section:

$$p(y \mid x) = \sum_{z \in \mathcal{Z}} p(y \mid z, x)\, p(z \mid x)$$

Here, $p(z \mid x)$ denotes the probability of retrieving a relevant document z given the input x, while $p(y \mid z, x)$ denotes the probability of generating the correct output y conditioned on both the input and the retrieved document. This formulation separates the retrieval and generation components of the RAG pipeline and emphasizes the importance of jointly optimizing both stages to maximize overall performance.

In practice, retrieval models aim to maximize $p(z \mid x)$ by selecting the most relevant documents from a corpus, while generation models aim to maximize $p(y \mid z, x)$ by producing accurate and contextually appropriate outputs based on the retrieved information.

4 Evaluation Method

4.1 Indexing

Indexing is a foundational step in the Retrieval-Augmented Generation (RAG) pipeline, responsible for transforming raw data into a structured format suitable for efficient retrieval. It begins by extracting and cleaning documents from varied formats—such as PDF, HTML, and Markdown—into plain text. To accommodate the limited context windows of language models, the text is segmented into smaller, semantically meaningful chunks. These chunks are then embedded into vector representations using an embedding model and stored in a vector database for fast similarity-based retrieval during inference (Gao et al.).

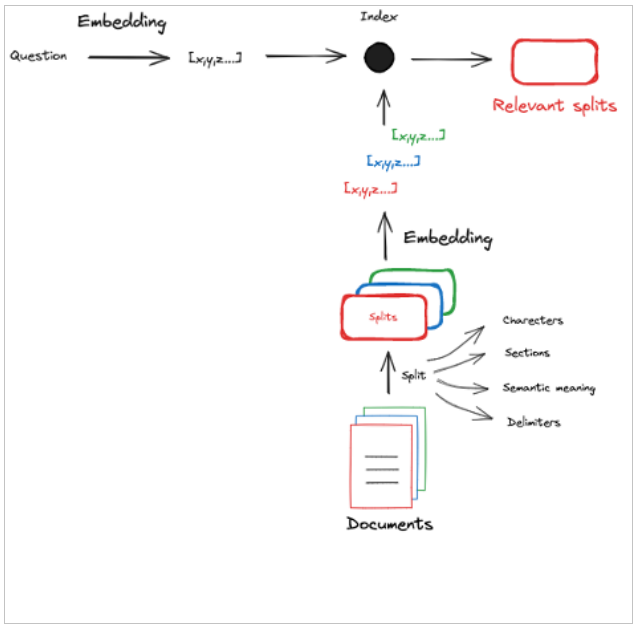

Figure 3

4.2 Splitting

Splitting is a foundational step in the indexing pipeline of Retrieval-Augmented Generation (RAG) systems. It addresses a critical challenge in processing long-form documents—how to preserve semantic meaning while conforming to input length constraints of downstream models. The goal is to divide documents into manageable, semantically coherent chunks that can later be embedded as vector representations for retrieval.

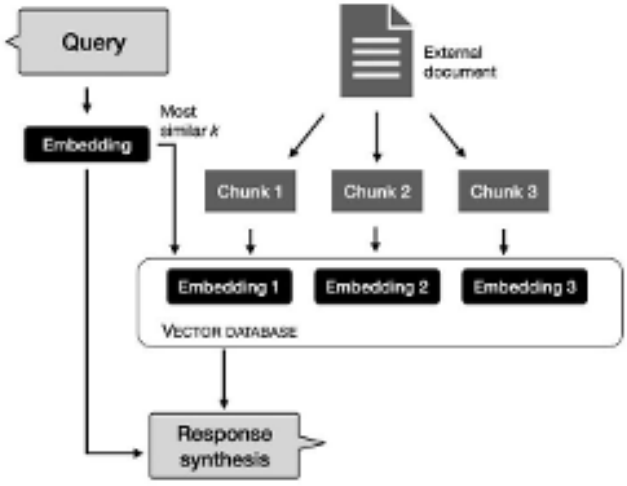

Figure 4

4.2.1 Recursive Character-Based Splitting

A widely adopted technique is the `RecursiveCharacterTextSplitter`, as utilized in frameworks like LangChain. This approach operates by recursively splitting text based on a prioritized list of delimiters such as paragraph breaks (`\n\n`), single line breaks (`\n`), spaces ( ), and finally, at the character level. This recursive logic ensures that the largest possible semantically coherent units are preserved while still adhering to predefined `chunk_size` limits.

For example, a chunk size of 1000 characters with an overlap of 200 characters is a common configuration to maintain continuity between adjacent chunks.

```
# Split Documents
text_splitter =
    RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200
)

splits = text_splitter.split_documents(docs
    )
# Now `docs` contains chunks of text that
    are ready for embedding and indexing
```

This method ensures that important semantic boundaries, such as paragraph or sentence breaks, are respected as much as possible before falling back to more aggressive splitting criteria. As demonstrated in LangChain's *rag-from-scratch* repository, this process results in chunks that are better suited for embedding and semantic search tasks.

4.2.2 Token-Aware Chunking with Overlap

Another popular strategy is chunking by tokens, particularly with overlapping segments. Overlap is essential in preserving contextual information, especially for transformer-based models where the position of a token within its context heavily influences its representation. By ensuring that adjacent chunks share a fixed number of tokens—often referred to as a *sliding window*—the model can capture cross-boundary semantics that might otherwise be lost. This method is particularly useful in scenarios where precision in retrieval is paramount, as it prevents fragmentation of key phrases or ideas that span multiple sentences.

4.2.3 Structure-Aware Chunking

More advanced methods adopt a **structure-aware strategy** that aligns chunks with syntactic units such as sentences or paragraphs. This is especially useful in tasks where fine-grained semantic relationships matter, such as question answering or summarization. In structure-aware models, small text units (e.g., a sentence) are treated as atomic retrieval elements, but their broader meaning is interpreted in the context of surrounding units. Thallys Costalat (2024)

emphasizes that such methods yield higher retrieval precision, as the coherence of the chunk directly correlates with its informativeness during retrieval. Sentence-based chunking, for example, can be extended with "context windows" by appending neighboring sentences, creating composite chunks that balance granularity and context.

4.3 Splitting Strategy Summary

Splitting Strategy	Key Feature	Use Case Scenario
Recursive Character TextSplitter	Flexible-fallback through-delimiters	General-purpose chunking-of-large documents
TokenOverlap Chunking	Preserves continuity across segments	Semantic search with transformers-(e.g., BERT, GPT)
Structure-Aware Chunking	Aligns with linguistic structure	High-precision-retrieval or summarization tasks

Table 1

Each technique addresses a different trade-off between semantic preservation, context integrity, and chunk manageability. Selecting an appropriate method depends on the downstream task, document type, and retrieval sensitivity.

4.4 Embedding

After documents are split into semantically coherent chunks, the next stage in the RAG indexing pipeline is embedding, which transforms textual data into numerical vector representations. These embeddings allow for efficient semantic comparison, search, and retrieval within high-dimensional vector spaces.

4.5 Purpose of Embedding

Embedding serves as a bridge between unstructured textual information and structured numerical computation. By encoding each chunk into a dense vector, embedding models enable similarity comparisons based on vector proximity (e.g., cosine similarity or dot product), rather than relying solely on keyword overlap. This transformation is essential because it allows RAG systems to compare the semantic meaning of queries and documents, rather than just their surface-level lexical content This transformation is essential because it allows RAG systems to compare the semantic meaning of queries and documents, rather than just their surface-level lexical content.

4.6 Types of Representations

There are two principal methods for representing text:

- **Statistical Representations:** Techniques like Bag-of-Words (BoW) and TF-IDF produce sparse vectors, which are simple but often fail to capture semantic nuance.
- **Machine-Learned Embeddings:** Transformer-based models like DPR (Dense Passage Retrieval), ColBERT, RAPTOR, and CONTRIEVER generate dense vectors, which embed semantic meaning into a compact numerical format suitable for neural retrieval methods.

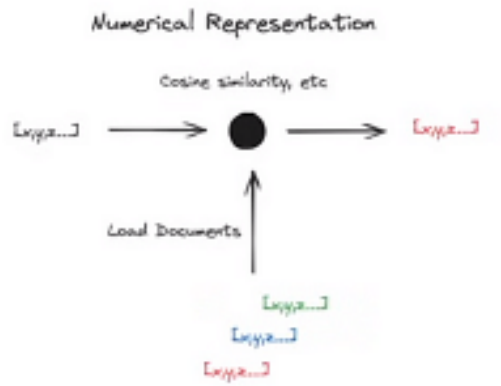

Figure 5

The latter is the standard in RAG systems due to its superior performance in understanding context, paraphrasing, and semantic similarity.

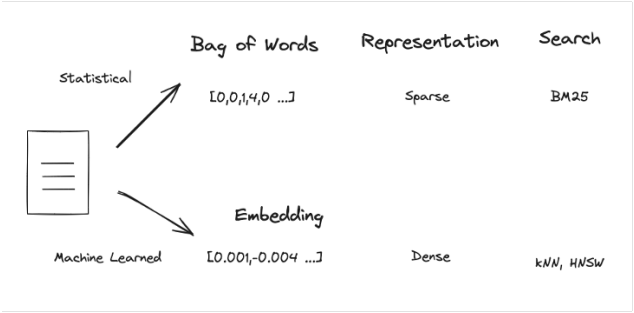

Figure 6

4.7 Common Embedding Models

- **DPR:** Generates a single vector per passage, optimized for full-document semantic similarity.
- **ColBERT:** Computes token-level embeddings, allowing fine-grained interaction between queries and passages, ideal for high-resolution retrieval.
- **RAPTOR:** Leverages hierarchical encoding and retrieval patterns for multi-hop reasoning.
- **CONTRIEVER:** Focuses on unsupervised learning of sentence-level representations, suitable for low-resource or domain-adaptive scenarios.

4.8 Embedding Challenges

1. **High Dimensionality:** Embedding vectors typically exist in hundreds or thousands of dimensions. Searching or comparing vectors in such high-dimensional spaces is computationally intensive. Classical structures like KD-trees degrade rapidly beyond 10 dimensions.
2. **Approximate vs. Exact Search:**
   - **Exact Nearest Neighbor Search** guarantees precision but scales poorly.
   - **Approximate Nearest Neighbor Search (ANNS)**—as used in libraries like FAISS—sacrifices some accuracy for speed and scalability. Tuning these systems is non-trivial and often domain-specific.
3. **Incompatible Representations:** Not all embeddings are interoperable. For instance:
   - **ColBERT** produces embeddings at the token level, optimized for fine-grained matching.
   - **DPR** represents entire passages with a single embedding.

   These differences mean that one cannot interchangeably use embeddings from different models within the same vector store. Each embedding model encodes information at a different granularity and semantic level, affecting retrieval performance and indexing compatibility.

Embedding Models Comparison

Model	Granularity	Search Type	Use Case
DPR	Passage-level	Dense vector search	General-purpose-retrieval
ColBERT	Token-level	Late interaction model	Fine-grained, high-precision retrieval
RAPTOR	Hierarchical	Hybrid	Multi-hop-or layered reasoning
CONTRIEVER	Sentence-level	Dense unsupervised	Domain adaptation

Table 2

5 Experimental Results

6 Conclusion