Natural Language Processing (2025), **31**, pp. 1–25
doi:10.1017/nlp.2024.53

CAMBRIDGE UNIVERSITY PRESS

ARTICLE

Maximizing RAG efficiency: A comparative analysis of RAG methods

Tolga Şakar and Hakan Emekci

Applied Data Science, TED Üniversitesi, Ankara, Turkey
Corresponding author: Tolga Şakar; Email: tolga.sakar@tedu.edu.tr

(Received 29 May 2024; revised 17 September 2024; accepted 17 September 2024; first published online 30 October 2024)

Abstract

This paper addresses the optimization of retrieval-augmented generation (RAG) processes by exploring various methodologies, including advanced RAG methods. The research, driven by the need to enhance RAG processes as highlighted by recent studies, involved a grid-search optimization of 23,625 iterations. We evaluated multiple RAG methods across different vectorstores, embedding models, and large language models, using cross-domain datasets and contextual compression filters. The findings emphasize the importance of balancing context quality with similarity-based ranking methods, as well as understanding tradeoffs between similarity scores, token usage, runtime, and hardware utilization. Additionally, contextual compression filters were found to be crucial for efficient hardware utilization and reduced token consumption, despite the evident impacts on similarity scores, which may be acceptable depending on specific use cases and RAG methods.

Keywords: Large Language Models; Vector Databases; Retrieval-Augmented Generation; Contextual Compression; Embedding Models

1. Introduction

The use of goal-oriented large language models (LLMs) (Devlin *et al.* 2019; Brown *et al.* 2020; Chowdhery *et al.* 2022), coupled with diverse LLM-oriented frameworks, is continually broadening the spectrum of AI applications, enhancing the proficiency of LLMs across complex tasks (Wei *et al.* 2022). Contemporary LLMs demonstrate remarkable capabilities, from answering questions about legal documents with latent provenance (Jeong 2023; Nigam *et al.* 2023; Cui *et al.* 2023) to chatbots adept at generating programing code (Vaithilingam *et al.* 2022). However, this increased capability also introduces additional complexities. Emerging LLMs, formidable in conventional text-based tasks, necessitate external resources to adapt to evolving knowledge.

To address this challenge, non-parametric retrieval-based methodologies, exemplified by retrieval-augmented generation (RAG) (Lewis *et al.* 2020), are becoming integral to the latest LLM applications, especially for domain-specific tasks (Vaithilingam *et al.* 2022; Manathunga and Illangasekara 2023; Nigam *et al.* 2023; Pesaru *et al.* 2023; Peng *et al.* 2023; Gupta 2023). The evolution of AI-stack applications underscores the critical role of fine-tuning RAG methods in updating the knowledge base of LLMs (Choi *et al.* 2021; Lin *et al.* 2023; Konstantinos and Pouwelse Andriopoulos and Johan 2023). Retrieval-based applications demand optimization when searching the most relevant passages, or top-K vectors, through semantic similarity search.

Querying multi-document vectors and augmenting LLMs with relevant context introduce dependencies on both time and token limits. The 'bi-encoder' retrieval models (refer to Figure 1) leverage cutting-edge approximate nearest-neighbor search algorithms (Jégou *et al.* 2011).

However, diverse data structures (e.g., multimedia, tables, graphs, charts, and unstructured text) pose additional challenges, including the potential generation of hallucinative responses (Sean *et al.* 2019; Konstantinos and Pouwelse Andriopoulos and Johan 2023) and the risk of surpassing LLM token limits (Roychowdhury *et al.* 2023). Overcoming the hallucination problem and staying within the token generation limit is a difficult task. The tradeoff is that once the retrieval process is complete and top-K documents are obtained by the search algorithm, the total number of tokens sent may exceed the token limit for the specific LLM in use.

On the other hand, constraining the retrieval capabilities may limit the LLM's ability to successfully generate the relevant response based on the restrained context. Optimizing the tradeoff requires an in-depth analysis involving various processes, including experimenting with diverse and hardware-optimized search algorithms (Chen *et al.* 2019; Zhang and He, 2019; Malkov and Yashunin 2020; Johnson *et al.* 2021; Lin *et al.* 2023), applying embedding filters based on similarity score threshold, and routing the LLM inputs and outputs along with the filtered context. These processes play a crucial role in fine-tuning LLM responses by optimizing the retrieval process. In the pursuit of optimization, commonly known vector databases, such as Pinecone (Sage 2023), ChromaDB, FAISS, Weaviate, and Qdrant, find application for various reasons. These databases are classified based on criteria like scalability, ease of use (versatile Application Programming Interface [API] support), filtering options, security, efficiency, and speed (Han *et al.* 2023). The integration of these databases into the LLM operational pipeline not only enhances the overall efficiency and effectiveness of retrieval-based methodologies but also contributes significantly to the optimization of LLM performance.

Optimization initiates with document preprocessing. Chunking the multi-document into smaller paragraphs and overlapping the chunks (sentence tokens) could potentially affect the retrieval process since the embedded documents in the vector space are selected as a candidate context by the search algorithm (Schwaber-Cohen 2023; Zhou *et al.* 2023). Chunks and overlapping chunk hyper-parameters control the granularity of text splitting, making fine-tuning crucial, especially when dealing with documents of a similar or reasonably uniform format (Schwaber-Cohen 2023). In addition to optimizing data preprocessing, there are various known RAG methodologies, such as 'Stuff', 'Refine', 'Map Reduce', 'Map Re-rank', 'Query Step-Down', and 'Reciprocal RAG' (refer to Figures 2.4.1–2.5.3), which significantly impact vectorstore scalability, semantic retrieval speed, and token budgeting. When making an LLM call, RAG methods either feed the retrieved documents as the unfiltered context to the call or apply re-ranking within the retrieved context based on the highest individual similarity for each document vector. More importantly, in multi-document tasks, fine-tuning the context routing in the LLM call sequence can dramatically affect the response generation time, hardware resources, and token budgeting (Nair *et al.* 2023). Despite the recognition of the importance of optimizing RAG processes in numerous papers (Vaithilingam *et al.* 2022; Nair *et al.* 2023; Topsakal and Akinci 2023; Manathunga and Illangasekara 2023; Nigam *et al.* 2023; Pesaru *et al.* 2023; Peng *et al.* 2023; Konstantinos and Pouwelse, Andriopoulos and Johan 2023; Roychowdhury *et al.* 2023), there remains a notable gap in previous work on the methods of optimization. Therefore, this paper aims to illuminate the process of optimizing RAG processes to improve LLM responses.

2. Materials and methodology

2.1 Retrieval-augmented generation

RAG method enables continuous updates and data freshness for question-answering chatbots by retrieving latent documents as provenance, without the need of re-training or fine-tuning the LLMs for domain-specific tasks. First-generation RAG models tackled challenges (Mialon *et al.* 2023) in knowledge-intensive text generation tasks by combining a parametric pretrained sequence-to-sequence BART model (refer to Figure 1) (Sutskever *et al.* 2014; Lewis *et al.* 2019)

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

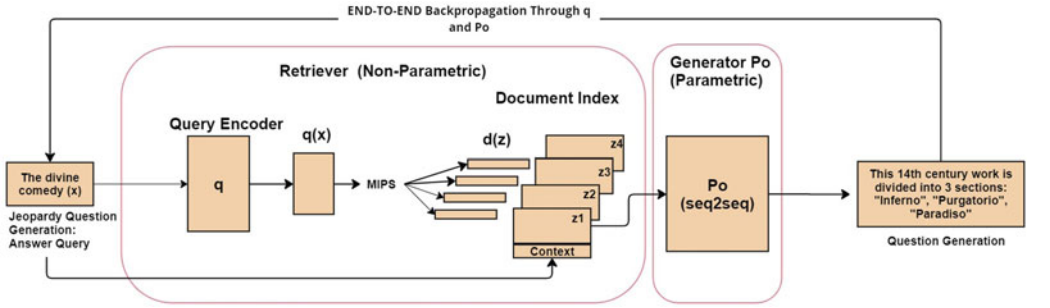

Figure 1. First-generation retrieval-augmented generation (RAG) methodology: Non-parametric RAG with a parametric sequence-to-sequence model. (Refer to RAG for knowledge-intensive natural language processing tasks.

with a non-parametric memory. This memory is a vectorized dense representation of latent documents from Wikipedia, accessed through a pretrained neural retriever (Lewis *et al.* 2020). This integration of a sequence-to-sequence encoder and a top-K document index enables token generation based on latent provenance augmented with context from the query. The 'Retrieval and Generation' sequence efficiently produces output by leveraging both the pretrained sequence-to-sequence model and the non-parametric memory with a probabilistic model (refer to Figure 1).

The non-parametric retriever conditions latent provenance on the input query (x), and subsequently, the parametric sequence-to-sequence model (P_o) is then conditioned on this new information along with the input (x) to generate the output response $P_\theta(y_i|x, z, y_{i-1})$. The combined probabilistic model then marginalizes to approximate the top-K documents (latent documents). This can occur either on a per-output basis, meaning that a single latent document is responsible for all output generation, or different indexed documents are responsible for different output generations. Subsequently, the parameterized θ generates the y_i token based on the top-K context from a previous token, y_{i-1}, x, z, where x is the original input, and z is a retriever document (refer to Figure 1).

Recent advancements in retrieval methods involve prominent embedding models, which are extensively trained on a large corpus of data in diverse languages. Notably, a significant shift has occurred, with a preference for LLMs over traditional models like sequence-to-sequence for token generation in the context of RAG. This transition builds on the foundations laid by first-generation RAG models, expanding the possibilities for more efficient and versatile text generation using latent documents as provenance. The latest RAG methods (refer to Figures 2.4.1–2.5.3) used in LLM-powered chatbots involve much more sophisticated retrieval processes. Using frameworks such as LangChain and LlAma-index, LLMs can now have direct access to SQL, vectorstores, Google search results, and various APIs. This increased capacity in generating more accurate results by having access to concurrent information allows LLMs to expand their knowledge base for domain-specific tasks. Moreover, using embedding filters provided by frameworks, RAG processes can become even more efficient in terms of retrieval speed when querying from vector databases. Embedding filters apply thresholds during the similarity search process to select text vectors (embeddings) that have similarity scores above the threshold while removing the redundant vectors, achieving efficiency in both retrieval speed and token budgeting.

2.2 Vectorstores

Vectorstores, playing a crucial role in RAG-based LLM applications, are distinguished by their proficiency in executing fast and precise similarity searches and retrievals. Unlike traditional databases reliant on exact matches, vector databases assess similarity based on vector distance and embedding filters, enabling the capture of semantic and contextual meaning (Han *et al.* 2023)

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

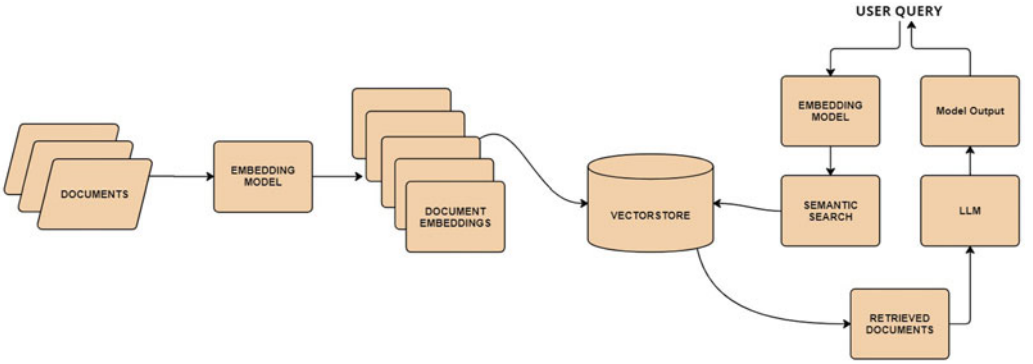

Figure 2. Schema for searching and matching similar context from vector database based on user query.

(refer to Figure 2). This positions them as pivotal components in the ongoing effort to optimize the intricate interplay between language models and retrieval methodologies.

Prominent vector databases, such as FAISS, Pinecone, and ChromaDB, find applications in various domains, each offering key differentiators. FAISS, acknowledged as a leading vector database, excels in high-speed vector indexing for similarity searches. Its memory-efficient search algorithm enables the handling of high-dimensional data without compromising speed and efficiency (Johnson *et al.* 2019). Adding to the strengths of FAISS, Pinecone emerges as a notable choice, providing a high-level API service and delivering comparable performance to FAISS in similarity searches (Sage 2023). However, as a managed service, Pinecone raises scalability concerns. Limitations on the number of queries pose a barrier to large-scale data processing needs, especially for high-volume applications.

Unlike Pinecone, ChromaDB is an open-source vector database that offers more flexibility in terms of scalability and usage. This open-source nature facilitates adaptability to different needs and use cases, making it a compelling option for customization and control over vector database infrastructure. Semantic similarity search is the retrieval of information or data based on semantic relationships that go beyond exact keyword or text matching. It focuses on understanding the contextual and conceptual relationships between words, phrases, or documents. A specific application within semantic similarity search is the exploration of semantic connections between embeddings.

Vector representations (embeddings) of words, phrases, or documents are used for similarity-based search. The process begins by converting documents or text into vectors using an embedding model. Then, similarity search algorithms identify the vectors most similar to the query based on various metrics, such as cosine similarity. In the final step, the output contains the response corresponding to the most similar vectors within the vector space. When calculating the similarity, cosine similarity is especially effective in this context as it measures the cosine of the angle between two vectors, providing a normalized measure that captures the directional similarity between vectors. This property makes it particularly useful for evaluating the similarity of embeddings in semantic search tasks.

Cosine similarity

Cosine similarity measures the cosine of the angle between two vectors. Given two vectors A and B in a vector space:

$$\cos(A, B) = \frac{A \cdot B}{\|A\| \cdot \|B\|} = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \cdot \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

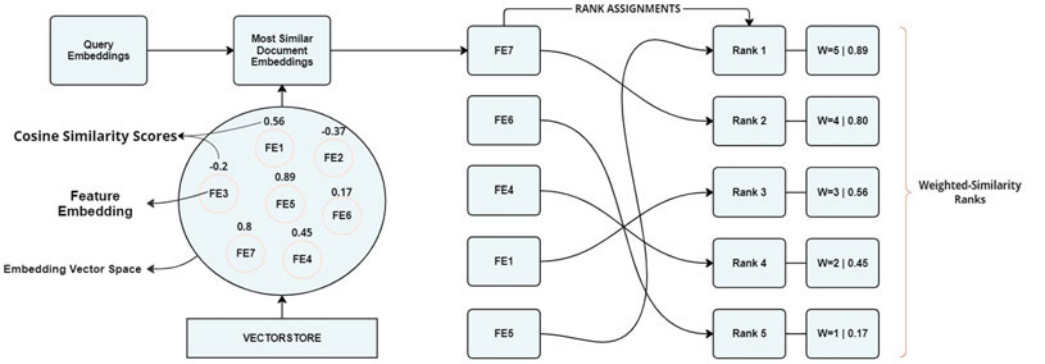

Figure 3. Semantic similarity search process when finding top-K documents in a vector space based on input query to assign score-weighted ranks.

- A and B are vectors in a vector space.
- A_i and B_i are the components of vectors A and B at index i, respectively.
- n is the dimensionality (number of components) of the vectors.
- The numerator $\sum_{i=1}^{n} A_i \cdot B_i$ represents the dot product of vectors A and B.
- The denominators $\sqrt{\sum_{i=1}^{n} (A_i)^2} \cdot \sqrt{\sum_{i=1}^{n} (B_i)^2}$ represent the Euclidean magnitudes (lengths) of vectors A and B, respectively.

2.3 Search algorithms

At the forefront of enhancing retrieval efficiency and speed, search algorithms play a pivotal role. Specifically, tree-based algorithms are widely employed in vector databases to measure the distance between similar top-K vectors. Nearest neighbor search (NNS) (Chen *et al.* 2019; Zhang and He 2019; Malkov and Yashunin 2020; Han *et al.* 2023) identifies the data points in a dataset that are nearest to a particular query point, often based on a distance measure such as Euclidean distance or cosine similarity. Exact closest neighbor search uses methods like linear search or tree-based structures like kd-trees (Bentley 1975; Dolatshah, Hadian, and Minaei-Bidgoli 2015; Ghojogh, Sharifian, and Mohammadzade 2018) to identify the genuine nearest neighbors without approximations. However, the computational complexity of accurate search might be prohibitive for big or high-dimensional data sets.

Approximate NNS (ANNS) (Zhang and He 2019; Christiani 2019; Li and Hu 2020; Singh *et al.* 2021), on the other hand, achieves a compromise between accuracy and efficiency. By adopting index structures such as locality-sensitive hashing (LSH) (Dasgupta, Kumar, and Sarlos 2011), or graph-based techniques, it trades some precision for quicker retrieval. ANNS is especially beneficial in situations involving high-volume or high-dimensional data sets, such as picture retrieval, recommendation systems, and similarity search in massive text corpora. To meet the issues of both precise and ANNS, several methods such as k-d trees, LSH, and tree-based structures are utilized, with tradeoffs to fit specific use cases and computational restrictions.

The process of similarity search begins by transmitting the vector embeddings of the input query to a preexisting vector store, which contains embeddings of various documents (Feature Embeddings) (refer to Figure 3). The initial step involves identifying the most similar vector embeddings within the available vector space. Subsequently, upon locating the most relevant or

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

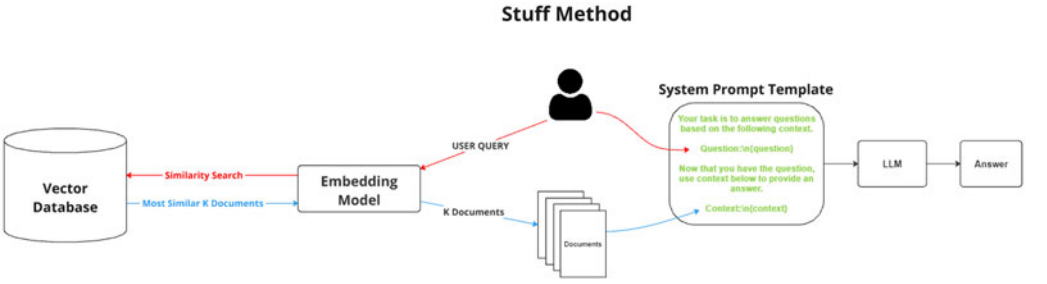

Figure 4. Stuff retrieval-augmented generation method.

top-K documents, these documents are retrieved and ranked based on their individual cosine similarity to the input query (see: Figure 3). This retrieval mechanism is fundamental to every RAG methodology. Consequently, the similarity search and retrieval processes are indispensable and necessitate thorough evaluation and optimization.

2.4 RAG methods

Various RAG methods offer distinct benefits when augmenting LLMs with latent information. Selecting the appropriate RAG method is crucial for optimizing retrieval speed, token budgeting, and response accuracy. RAG methods, such as Stuff, Refine, Map Reduce, and Map Re-rank can directly influence the number of top-K documents, retrieval speed, the number of input tokens used for generation, and response time. Therefore, optimizing the RAG methods to suit the specific task is of utmost importance.

2.4.1 Stuff method

The stuff method is the most straightforward RAG technique for updating the knowledge base of an LLM with latent information (refer to Figure 4). The query received from the user is sent to the existing vector store to search for similar content based on the specified number of documents. These documents are then incorporated into the system prompt template as context, which the LLM uses to generate a response. This method aims to enable LLMs to generate well-structured responses based on all relevant information within the entire context. However, the Stuff method also presents cost-efficiency challenges. Providing an LLM with multiple contexts could potentially increase token usage, as both input and output tokens will be significantly higher unless limited by max token hyperparameter. It is generally an appropriate option for applications where RAG processes do not involve long documents (Nair *et al.* 2023, LangChain n.d.).

Moreover, the context window of the selected LLM plays a crucial role, as certain models have quite a limited context window, such as GPT-3.5-Turbo with only a 4096 context token limit. Therefore, the more documents retrieved from the vector store, the more likely the context window limit will be reached.

2.4.2 Refine method

The refine method, in contrast to the 'Stuff' method, creates an answer by iteratively looping over the input documents and refining its response (refer to Figure 5) (LangChain n.d.). Essentially, after the first iteration, an additional context is generated from LLM call using $document_{n-1}$. The generated context from $document_{n-1}$ is then inserted into the successor system prompt template as additional context, along with the next document context in the iteration ($document_n$), and then with the combined context ($document_{n-1} + document_n$) and the user query, another

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

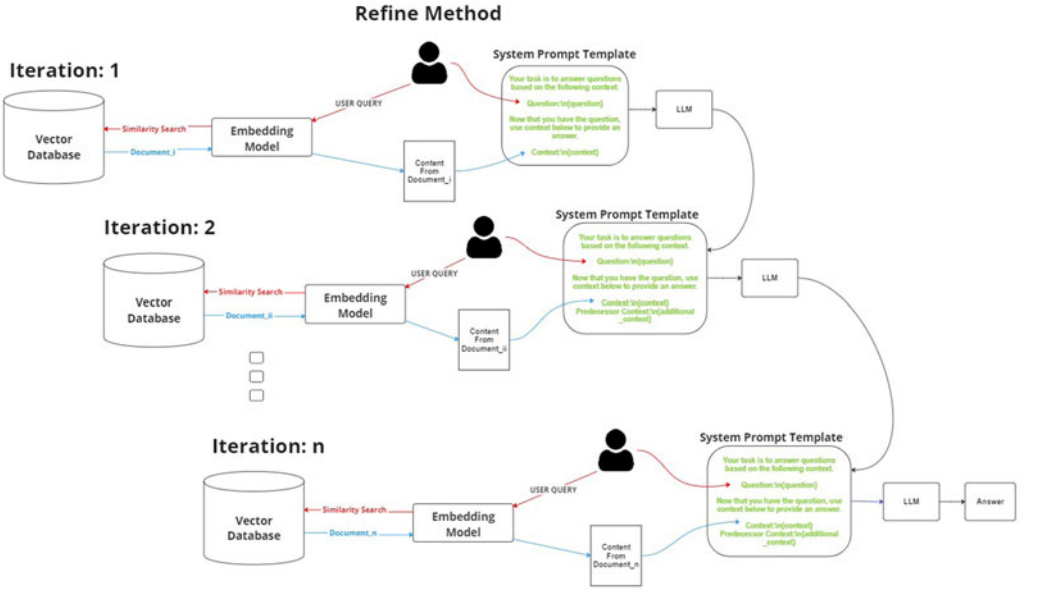

Figure 5. Refine retrieval-augmented generation method.

answer is created. This loop continues until the specified number of top-K documents is reached. Because the refine method only sends a single document to the LLM at a time, it is ideally suited for applications that require the analysis of more documents than the model can accommodate, which addresses the token context window issue. The clear disadvantage is that this method will make significantly more LLM calls, which is not ideal for token budgeting and response time. On the other hand, the final response will be 'refined' due to the enriched context supplied from predecessor LLM calls using numerous documents.

2.4.3 Map-reduce method

In the 'Map Reduce' method, similar to 'Refine', each document is iteratively used to generate a response (refer to Figure 6) (LangChain n.d.). However, one key difference in this method is that, rather than combining predecessor context with the successor, the final responses are 'mapped' together (refer to Figure 6). These mapped responses are then used as the final context when generating the 'reduced' response. In the initial phase of the map-reduce process, RAG method is systematically applied to each document independently (mapping phase), with the resulting output from the method treated as a single document. Subsequently, all newly generated documents are directed to a separate chain designed to consolidate them into a single output (reduce step). The Map Reduce method is suitable for tasks involving short documents, containing only a few pages per document. Longer contexts or long documents might cause the LLM to reach its token context window limit.

Moreover, if the LLM is not limited by a maximum token hyperparameter or is not instructed to provide concise and short answers, the 'reduced' context could also cause token context limit issues, even if not caused by the document itself.

2.4.4 Map re-rank method

This method closely resembles the Map Reduce method but incorporates a filtering application on each $document_n$ based on an assigned cosine similarity score, which ranks each response from

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

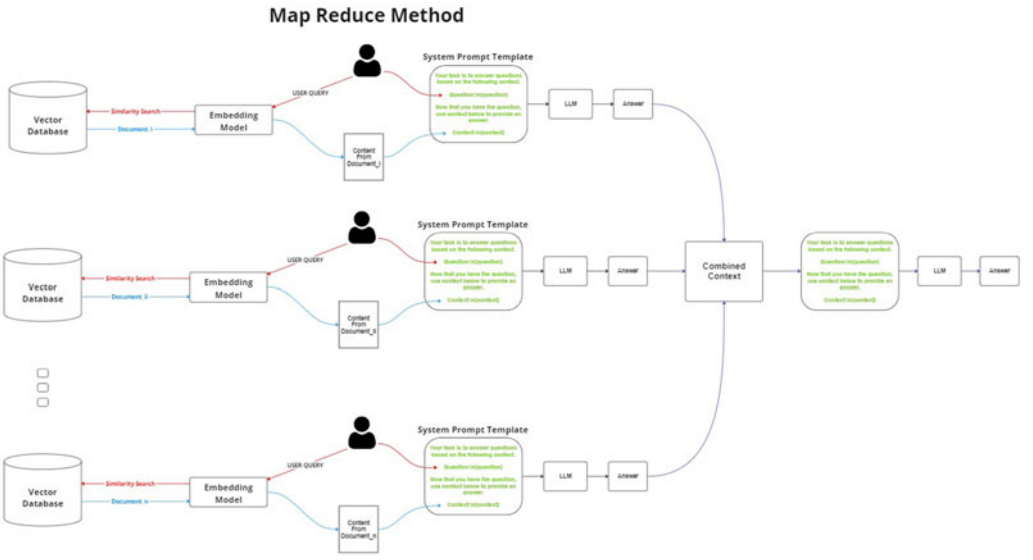

Figure 6. Map Reduce method.

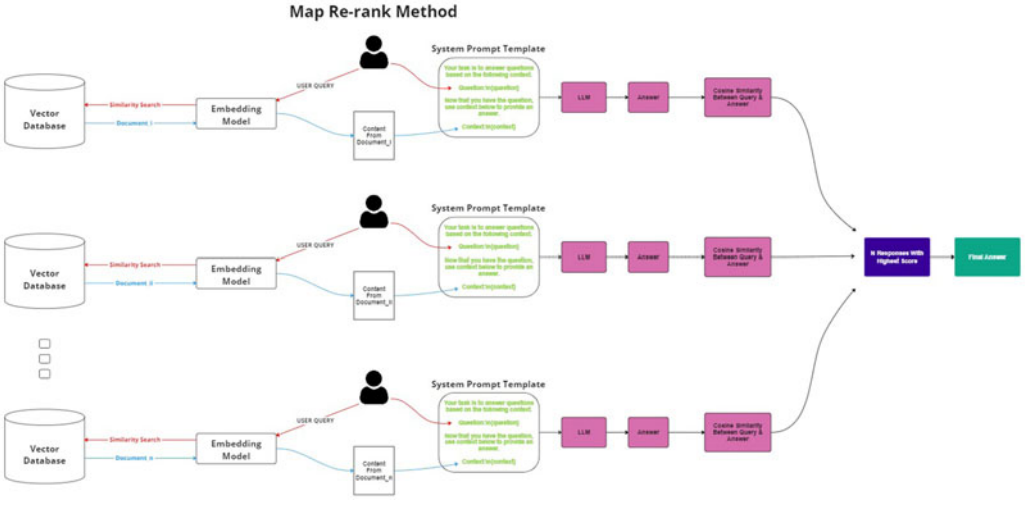

Figure 7. Map re-rank Method.

highest to lowest in terms of similarity to the query (refer to Figure 7) (LangChain n.d.). The map re-rank documents chain initiates a preliminary query on each document, aiming not only to perform a task but also to assign a confidence score to each answer. The answer with the highest score is then provided as the output.

2.5 Advanced RAG

The RAG methods discussed in Section 2.4 (2.4.1–2.4.4) primarily address issues related to the quality of the final response. However, another significant challenge during retrieval stems from the ambiguous nature of queries. This ambiguity can result in relevant information being

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

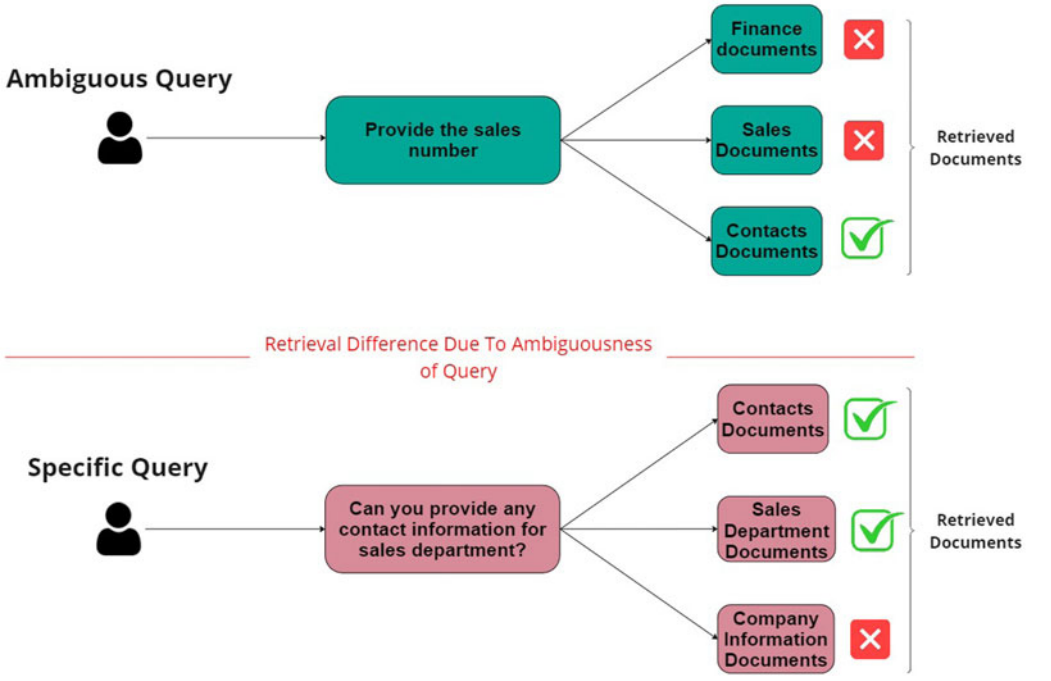

Figure 8. Impact of ambiguous vs specific query on retrieval.

overlooked during retrieval, leading to the inclusion of irrelevant documents as context during generation (LangChain n.d.; Zheng *et al.* 2023; Rackauckas 2024).

To enhance response accuracy and quality, and to prevent irrelevant documents from being inserted into the system prompt as context, more advanced methodologies are needed. These include filtering out irrelevant documents, creating alternative questions based on the original and then ranking each answer based on similarity (similar to the 'Map Re-rank' method), or generating less abstract alternatives of the input query and then proceeding with answer generation. Filtering irrelevant documents by applying similarity score thresholds can potentially exclude irrelevant documents and enable the retriever to insert only relevant documents as context. Moreover, with similarity filtering applied, token usage efficiency is achieved by limiting input token usage during filtering.

Another approach involves populating the input query with less abstract alternative questions to address the ambiguity issue (refer to Figure 8). Attaching a question 'generator chain' could potentially provide more relevant questions that align more closely with the actual intent of the original query. This reduction in abstraction could be achieved by providing a set of instructions for the generator chain to generate questions based on the content available in the vector store. Consequently, the generator chain would not produce irrelevant questions. One drawback of this approach is that each query results in a total of $GC_i + Q_j$ LLM calls.

- GC_i: The number of generator chain calls per user query.
- Q_j: The number of alternative questions generated per user query.

To exemplify, in a scenario where each query is decomposed (generated by the 'generator chain') into four different alternatives, and for each generated question, there will be four standalone RAG chain calls, totaling up to five calls for each query received from the user.

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

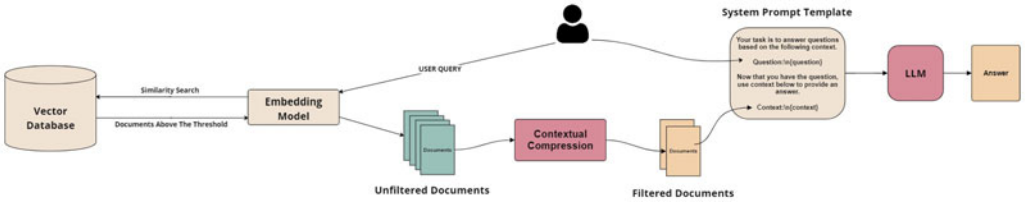

Figure 9. Contextual Compressor method applied on top-K documents.

Alternatively, the generated questions and their follow-up answers could be ranked based on how closely the answers match the original query. While this approach would result in more LLM calls and could potentially address the ambiguity issue and improve response quality, it would also deteriorate token usage efficiency and runtime performance.

2.5.1 Contextual compression

Contextual compression resolves the problem of inserting irrelevant documents caused by the ambiguous nature of a query by compressing retrieved documents based on the cosine similarity score between the query and each document (LangChain n.d.). The Contextual Compression retriever sends queries to the vector store, which initially filters out documents through the Document Compressor. This first step shortens the document list by eliminating irrelevant content or documents based on the specified similarity threshold. Subsequently, if needed, the Redundancy Compressor can be applied as a second step to perform further similarity filtering on the retrieved documents, compressing the context even more (refer to Figure 9).

Selecting an appropriate document chain is essential. Enhancing this process with contextual filters on embeddings, based on similarity scores, can significantly reduce both input and output token generation. However, applying embedding filters necessitates a careful balance between the threshold score in similarity search and the relevancy of the response. As the similarity threshold scores increase, the number of context documents decreases, thereby affecting the relevance and comprehensiveness of the responses.

Utilizing Contextual Compression along with various RAG methods could potentially provide further efficiency in both input token and output token generation.

2.5.2 Query step-down

To address the issue of ambiguity, the 'Query Step-Down' method offers a potential solution by generating variant questions based on the content within the documents (refer to Figure 10) (Zheng *et al.* 2023). This approach can be particularly useful for domain-specific tasks where users may lack sufficient information about the content.

The Query Step-down process initiates a 'generator chain' tasked with formulating questions based on the content available within the documents and the user query. Subsequently, for each generated question, the vector store is employed to retrieve the top-k documents corresponding to each $question_n$. To further optimize this approach, employing diverse RAG methods could enhance various aspects such as response time, token usage, and hardware utilization. Given the high volume of LLM calls made during each conversational transaction, reducing the context retrieved from the vector store through advanced methodologies (e.g., Contextual Compression) could address efficiency concerns effectively.

2.5.3 Reciprocal RAG

Reciprocal RAG, akin to the 'Query Step-Down' method, addresses the issue of ambiguity through a similarity score-based ranking process. Rather than aggregating all generated responses for

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

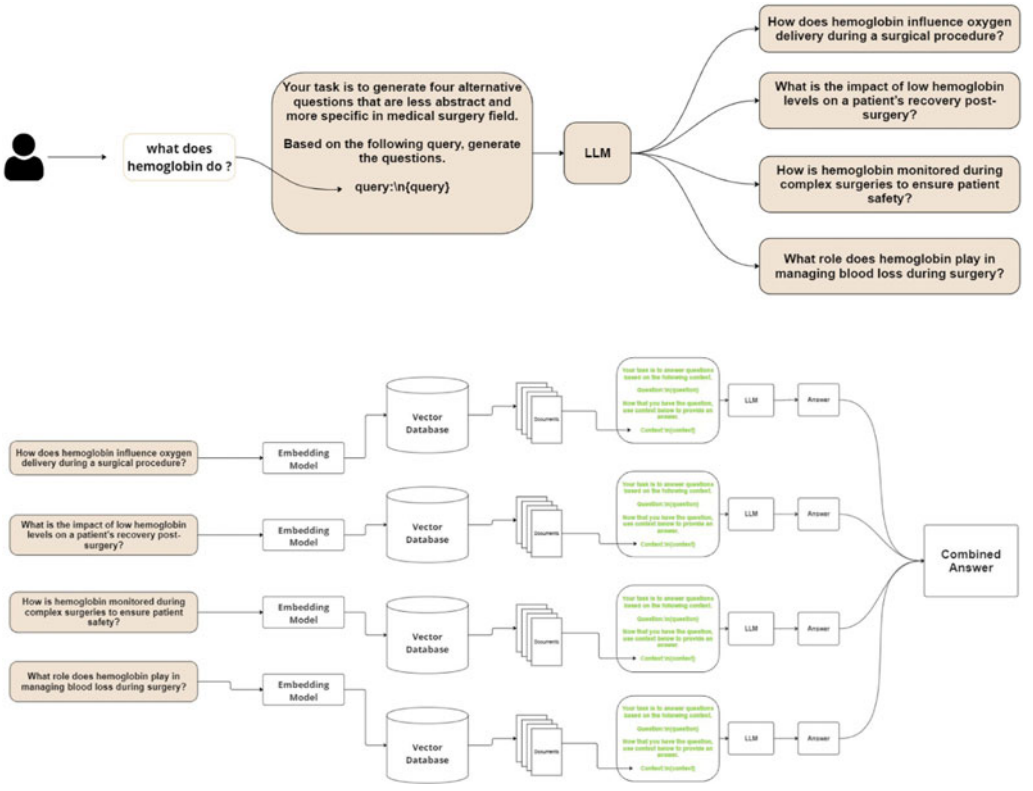

Figure 10. Query Step-Down method.

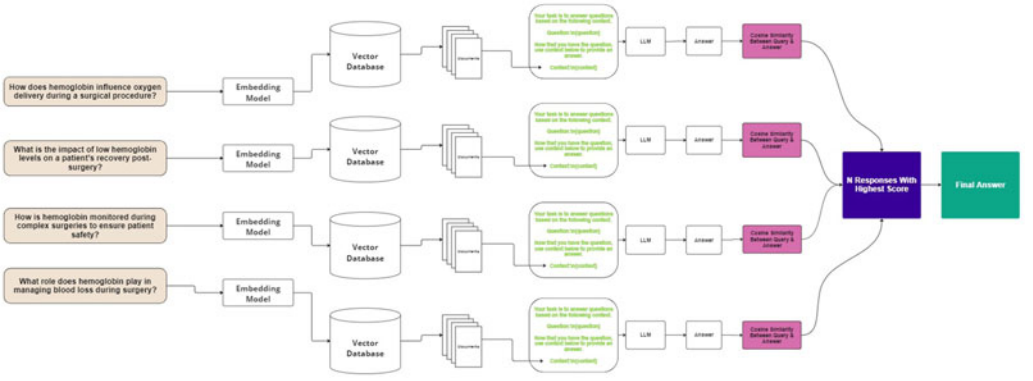

Figure 11. Reciprocal retrieval-augmented generation method.

each produced query to form a final answer, reciprocal RAG employs a ranking mechanism—comparable to 'Map Re-Rank'—to selectively filter and retain only the most pertinent or relevant answers from the pool of possible responses (refer to Figure 11). This method potentially reduces token usage by prioritizing relevance and precision in the final output (Rackauckas 2024).

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

3. Data

This paper utilizes three domain-specific datasets. The first dataset, the Docugami Knowledge Graph RAG dataset, includes 20 PDF documents containing SEC 10Q filings from companies such as Apple, Amazon, Nvidia, and Intel. These 10Q filings cover the period from Q3 2022 to Q3 2023. The dataset comprises 196 questions derived from the documents, with reference answers generated by GPT-4. The second dataset employed is the Llama2 Paper dataset, which consists of the Llama2 ArXiv PDF as the document, along with 100 questions and GPT-4 generated reference answers. The third dataset utilized is the MedQA dataset, a medical examination QA dataset. This dataset focuses on the real-world English subset in MedQA, featuring questions from the US Medical Licensing Examination (MedQA-US), including 1273 four-option test samples.

The significance of utilizing diverse datasets in optimizing RAG processes is rooted in the complex nature of the content within each dataset. These datasets often contain a variety of elements, including numerical values, multilingual terms, mathematical expressions, equations, and technical terminology. Given the distinct nature of these documents, optimizing the RAG processes becomes crucial. This optimization ensures efficient handling of the retrieval process during subsequent analysis. To process the data, we initially split the characters into tokens based on 1000 chunk size along with 100 overlapping chunks for each dataset. We employed the tiktoken encoder and the Recursive Character Split method for recursive splitting, ensuring that splits do not exceed the specified chunk size. The subsequent merging of these splits together completes the data processing steps.

In the next step, a grid search optimization was conducted, exploring different datasets, vector databases, RAG methods, LLMs, Embedding Models, and embedding filter scores. RAG performance was assessed by measuring the cosine similarity between the embedded LLM answer and the reference answer from each dataset and question-answer pairs.

Additionally, various performance metrics were created to monitor parameters such as run time (sec), central processing unit (CPU) usage (%), memory usage (%), token usage, and cosine similarity scores. Token usage calculation was done by applying the following formula: $T_i = (LR_i \times 4)/3$, where T_i is the token usage at the ith iteration, and LR_i is the length of the response generated by LLM for the i^{th} iteration.

4. Results

A comprehensive set of 23,625 grid-search iterations was conducted to obtain the results presented herein. Various embedding models were employed, including OpenAI's flagship embedding model (text-embedding-v3-large), BAAI's (Beijing Academy of Artificial Intelligence) open-source bge-en-small, and Cohere's cohere-en-v3. The LLMs used in this study comprised GPT-3.5, GPT-4o-mini, and Cohere's Command-R. For vectorstores, we utilized ChromaDB, FAISS, and Pinecone. Additionally, seven different RAG methodologies were implemented to complete the trials, and lastly, we deployed a wide range of contextual compression filters. In total, 42.12 million embedding tokens and 18.46 million tokens (combined input and output) were generated by the deployed LLMs. The cumulative runtime for all iterations was approximately 112 uninterrupted hours.

4.1 Similarity score performances

We implemented a range of RAG methodologies, LLMs, embedding models, and datasets to determine which combination would produce the highest similarity score. Among the RAG methods, Reciprocal RAG emerged as the most effective with a %91 similarity (Figure 12), followed by Step-Down (%87), Stuff (%86), and Map Reduce (%85) methodologies. As elaborated in Section 2.5.3, Reciprocal RAG aims to populate input queries based on the content within the documents and

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

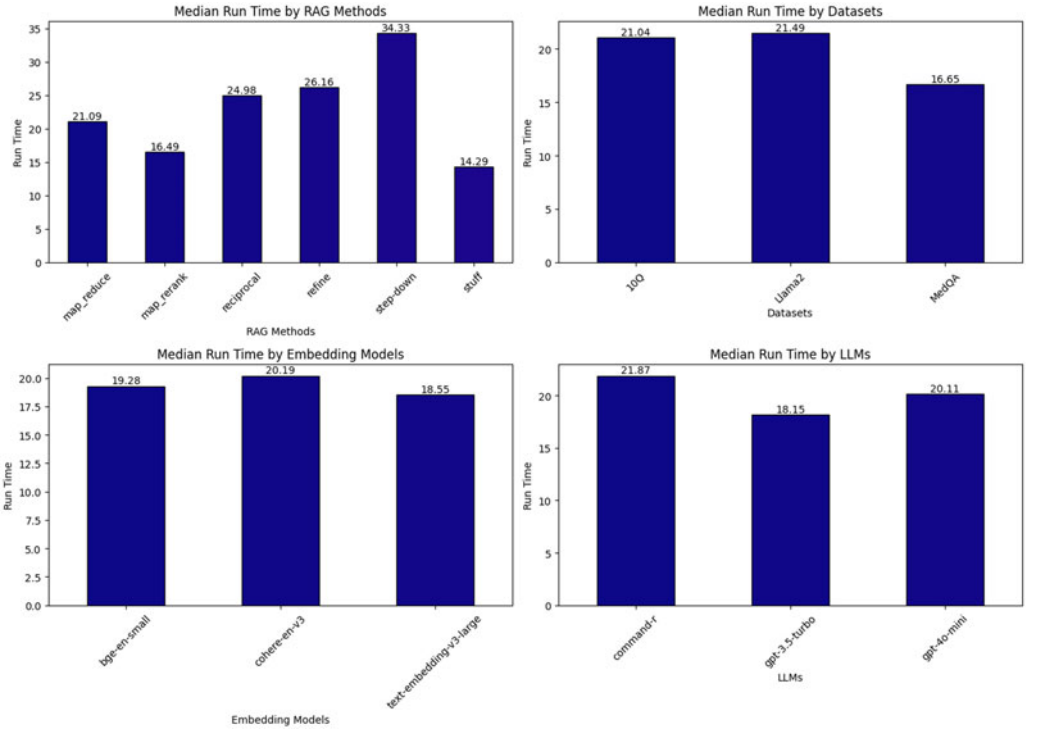

Figure 12. Median run time (sec) comparisons by retrieval-augmented generation methods, datasets, embedding models, and large language models.

the generated alternative questions. Each question is then queried within the vectorstore to generate an answer. Subsequently, all generated answers are filtered based on a desired number of documents or similarity score. We used 50% similarity threshold, which is the default value across all iterations where Reciprocal RAG method is utilized.

This filtering step is particularly advantageous in scenarios where some of the generated questions may be less pertinent to the original query. By omitting the answers and documents retrieved from less-relevant and populated queries (refer to Section 2.5.3), this filtering process ensures that only the most relevant information is retained. This distinguishes Reciprocal RAG from the Step-Down method, which aggregates all answers without filtering (refer to Section 2.5.2), as well as from other methodologies that do not adequately address or aim to resolve query ambiguity.

In the evaluation of LLMs, Cohere's Command-R model stands out as the top performer, achieving an impressive similarity score of 83%, which is 2.4% higher than that of GPT-3.5 and 3.75% higher than GPT-4o-Mini (refer to Figure 12). The significance of employing various LLMs is underscored when tested against different datasets across diverse domains. For instance, 10Q documents, which predominantly contain financial information such as income statements and balance sheets, are largely composed of numerical data. Similarly, LLama2 documents include not only numerical values but also mathematical expressions, notations, and formulas. Consequently, any errors in generation by an LLM when dealing with complex or challenging contexts could substantially impact the similarity score. Furthermore, MedQA documents are replete with technical and multilingual terminology, primarily in Latin. Certain LLMs may not be extensively trained on such documents or may have been trained on a smaller corpus of similar content compared to other datasets. Therefore, evaluating different datasets enables a comprehensive capability

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

Table 1. Similarity scores, run time, and token usage for various retrieval-augmented generation methods across different datasets. The datasets used in this study differ in complexity and domain specificity, thus the results are separated and evaluated separately

Dataset	RAG method	Similarity scores		Run time (sec)		Token usage	
		Median	Std	Median	Std	Median	Std
10Q	Reciprocal	0.971	±0.015	24.56	±2.51	3187	±411
10Q	Stuff	0.895	± 0.188	14.93	± 2.48	937	± 642.57
10Q	Step-Down	0.890	± 0.029	33.46	± 8.85	5527	± 926
10Q	Map Reduce	0.873	± 0.190	24.25	± 8.40	896	± 609
10Q	Refine	0.836	± 0.088	29.74	± 11.04	2298	± 940
10Q	Map Rerank	0.801	± 0.278	18.74	± 3.48	308	± 365
Llama2	Reciprocal	0.930	± 0.105	24.81	± 3.04	2707	± 689
Llama2	Step-Down	0.927	± 0.061	34.94	± 2.61	4308	± 1189
Llama2	Stuff	0.905	± 0.200	15.05	± 3.81	890.66	± 1147
Llama2	Map Reduce	0.892	± 0.207	25.89	± 13.64	1190	± 1240
Llama2	Refine	0.830	± 0.135	29.16	± 15.28	1982	± 1509
Llama2	Map Rerank	0.827	± 0.310	17.62	± 3.90	418	± 400
MedQA	Step-Down	0.678	± 0.045	33.01	± 3.97	4393	± 600
MedQA	Reciprocal	0.673	± 0.183	19.97	± 5.55	703	± 629
MedQA	Stuff	0.641	± 0.284	13.48	± 0.86	155	± 192
MedQA	Refine	0.609	± 0.095	25.18	± 5.42	3174	± 558
MedQA	Map Reduce	0.569	± 0.288	17.66	± 1.58	220	± 275
MedQA	Map Rerank	0.148	± 0.312	15.21	± 0.60	56	± 76

comparison of various LLMs, particularly in tasks such as accurately handling numerical values, mathematical expressions, and multilingual terminology.

In the evaluation of embedding models, BAAI's 'Bge-en-small' model demonstrated remarkable performance, achieving a median similarity score of 94%. This score is notably 20.5% higher than Cohere's flagship embedding model 'Cohere-en-v3' and 22% higher than OpenAI's flagship embedding model 'Text-embedding-v3-large' (refer to Figure 12). The significance of embedding models is paramount, extending beyond the initial conversion of documents into dense vector representations (embeddings) to encompass the retrieval of semantically meaningful text from the vector space (vectorstore). Embedding models trained for complex tasks—such as capturing semantic relationships between textual and numerical data, mathematical expressions, or multilingual texts—introduce additional considerations for enhancing the processes and efficiency of RAG systems.

Upon all possible iterations and datasets, 'Reciprocal RAG' achieved the highest similarity score (Median score: 97.1%, Std: ±0.015) across both Dokugami's 10Q and Llama2 datasets, yet not the lowest run time (sec) or token usage (refer to Table 1). The tradeoff between response accuracy, run time, and token usage plays crucial significance when designing RAG process for specific tasks.

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

As a result, we concluded that in RAG-based applications where response accuracy is paramount, Reciprocal RAG yields the best performances. RAG processes that might require the highest possible response accuracy could involve financial, insurance, and research-related documents. On the other hand, in applications where response time and token usage are more significant, such as building chatbots for high volume usages, 'Stuff' method could be utilized, since it yielded the 70.5% lower token usage, along with 38.9% faster response time, while only giving up 7.2% response accuracy compared to 'Reciprocal RAG' (refer to Table 1).

Additionally, in terms of minimizing token usage, the 'Map Re-rank' method demonstrated exceptional performance. It achieved similarity scores of 83.0% and 82.7% with the Llama2 and 10Q documents, respectively, while generating only 418 and 308 output tokens.

All RAG methods exhibited significantly lower similarity scores—some even yielding the poorest results—when applied to the MedQA dataset. This outcome can be primarily attributed to the dataset's nature, which involves challenging question-and-answer pairs within a highly specialized domain. MedQA documents focus on medical surgery content, where abstract or nuanced questions can substantially alter the expected answers, presenting a more formidable challenge for RAG-based applications. Consequently, methods designed to address ambiguity yielded higher similarity scores, with 'Step-Down' achieving 67.8% and 'Reciprocal' attaining 67.3% (refer to Table 1), compared to other methodologies that do not address such ambiguity. Moreover, 'Map Re-rank' yielded the lowest score on MedQA dataset with only 14.8% similarity. These results underscore the importance of retrieving the correct documents with sufficient context, highlighting that filtering context for LLMs prior to generation is not of even greater significance than the quality context. The results demonstrate that even with less content if it is irrelevant, the performance is adversely affected.

4.2 Hardware utilization

Another critical aspect of this research is to assess how various LLMs, embedding models, datasets, and RAG methods impact hardware consumption, including CPU (%) usage, memory (%) usage, and runtime. Hardware utilization becomes increasingly significant when deploying RAG-based applications for high-volume usage.

Further, we evaluated hardware utilization across three categories: runtime (in seconds), which measures the speed of obtaining an answer; CPU usage (percentage), which indicates the proportion of the workload handled by the CPU; and memory usage, which assesses the memory consumption during retrieval. Additionally, we considered the impact of different LLMs, embedding models, RAG methods, and datasets on these metrics.

We observed that the 'Step-Down' method resulted in the highest median run time of 34.33 s, whereas the 'Stuff' method exhibited the lowest run time at 14.29 s (refer to Figure 13). As detailed in Table 1, the difference in similarity scores between the top-performing methods is only 7.2% compared to the 'Stuff' method. Additionally, the 'Stuff' method demonstrated a 71.7% improvement in token usage efficiency relative to the top-performing method. Despite being the simplest approach and not addressing query ambiguity, the 'Stuff' method proves to be the most efficient, albeit with marginally lower similarity scores.

Furthermore, the comparative results revealed that among the embedding models, Cohere's 'Cohere-en-v3' exhibited the slowest median run time of 20.19 s, whereas OpenAI's 'Text-embedding-v3-large' demonstrated the fastest median run time of 18.55 s, achieving a 0.91% improvement in run time efficiency (refer to Figure 13). The inclusion of embedding models in the experiment was intended to assess which model could achieve the most rapid run time. This aspect is crucial for high-volume and high-scale applications with even denser vectorstores.

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

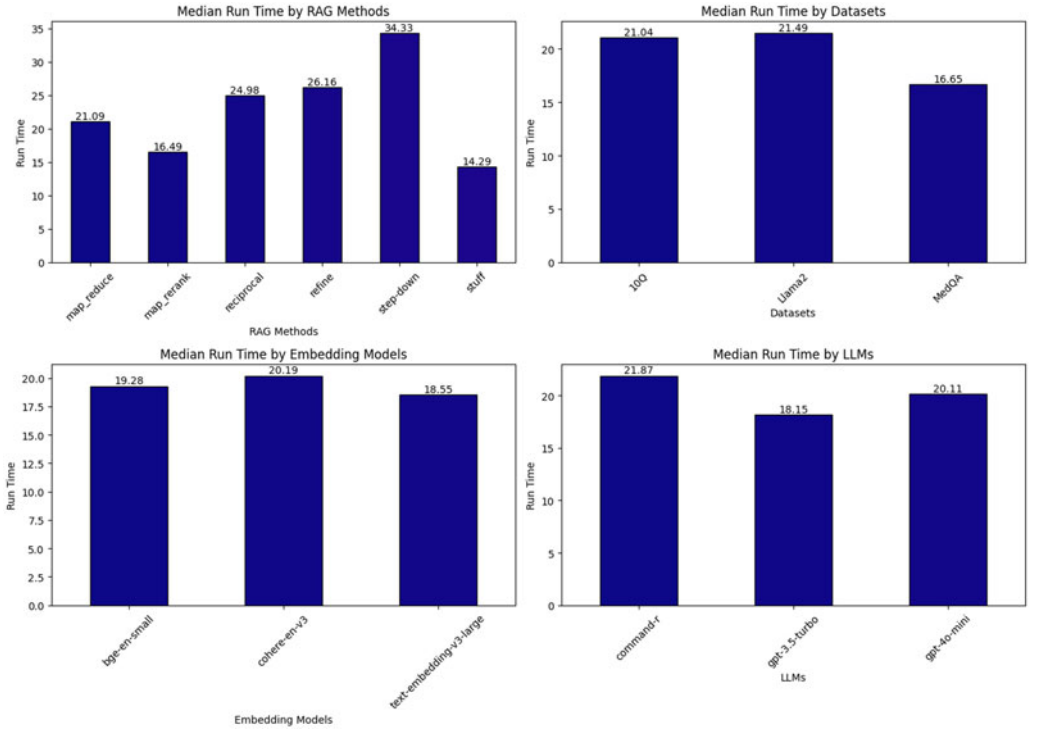

Figure 13. Median run time (sec) comparisons by retrieval-augmented generation methods, datasets, embedding models, and large language models.

Additionally, we assessed the run time differences among various LLMs. The OpenAI GPT-3.5-Turbo model emerged as the top performer in terms of generation time, with a median run time of 18.15 s. This model achieved generation times that were 9.7% and 17.0% faster compared to GPT-4o-Mini and Command-R, respectively (refer to Figure 13). Achieving rapid generation times is particularly crucial for applications such as educational chatbots, where swift responses are needed to accommodate lower attention spans. Moreover, the tradeoff between run time and response accuracy becomes more pronounced as tasks or domains vary. In cases where the primary goal is to mitigate response hallucination or address query ambiguity, generation time may be of lesser concern. Conversely, chatbot applications designed for roles such as information desks or promotional purposes, such as university promotion, may tolerate certain levels of response hallucination in favor of minimizing response time.

We also evaluated the performance of different embedding models and LLM combinations in terms of hardware utilization. GPT-3.5-Turbo, when paired with 'Bge-en-small', achieved the fastest run time of 17.55 $\pm$5.92 s, though it did not exhibit the lowest CPU utilization percentage (refer to Table 2). Generally, GPT-3.5-Turbo is identified as the fastest in response time, but the choice of embedding model also significantly impacts this efficiency. The difference in run time for GPT-3.5-Turbo, when used with various embedding models, is 3.7% between the fastest and slowest configurations (refer to Figure 14). Considering CPU and memory usage–critical factors for cloud-deployed applications handling high-volume requests–we concluded that the 'Text-embedding-v3-large' embedding model when used with the GPT-3.5-Turbo LLM, results in the lowest CPU % usage. This configuration sacrifices only 3.7% of run time compared to the fastest setup, which is GPT-3.5-Turbo paired with 'Bge-en-small' (refer to Table 2).

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

Table 2. Utilization metrics for various embedding models and large language models

LLM	Embedding Model	Run time (sec)		CPU usage		Memory usage	
		Median	Std	Median	Std	Median	Std
gpt-3.5-turbo	bge-en-small	17.55	±5.92	2.95	±8.31	0.0	±0.5904
gpt-3.5-turbo	cohere-en-v3	18.11	± 6.09	2.70	± 4.326	0.0	± 0.5590
gpt-3.5-turbo	text-embedding-v3-large	18.28	± 5.60	1.40	± 4.759	0.0	± 0.2674
gpt-4o-mini	bge-en-small	18.67	± 10.17	3.20	± 9.45	0.0	± 0.2645
gpt-4o-mini	text-embedding-v3-large	19.87	± 9.68	3.19	± 9.147	0.1	± 0.2247
command-r	text-embedding-v3-large	20.41	± 14.09	3.95	± 7.907	0.1	± 0.2827
command-r	bge-en-small	21.80	± 17.36	3.20	± 7.912	0.1	± 0.2492
gpt-4o-mini	cohere-en-v3	22.07	± 10.42	3.09	± 3.683	0.1	± 0.797
command-r	cohere-en-v3	25.64	± 18.42	2.00	± 3.964	0.0	± 0.3547

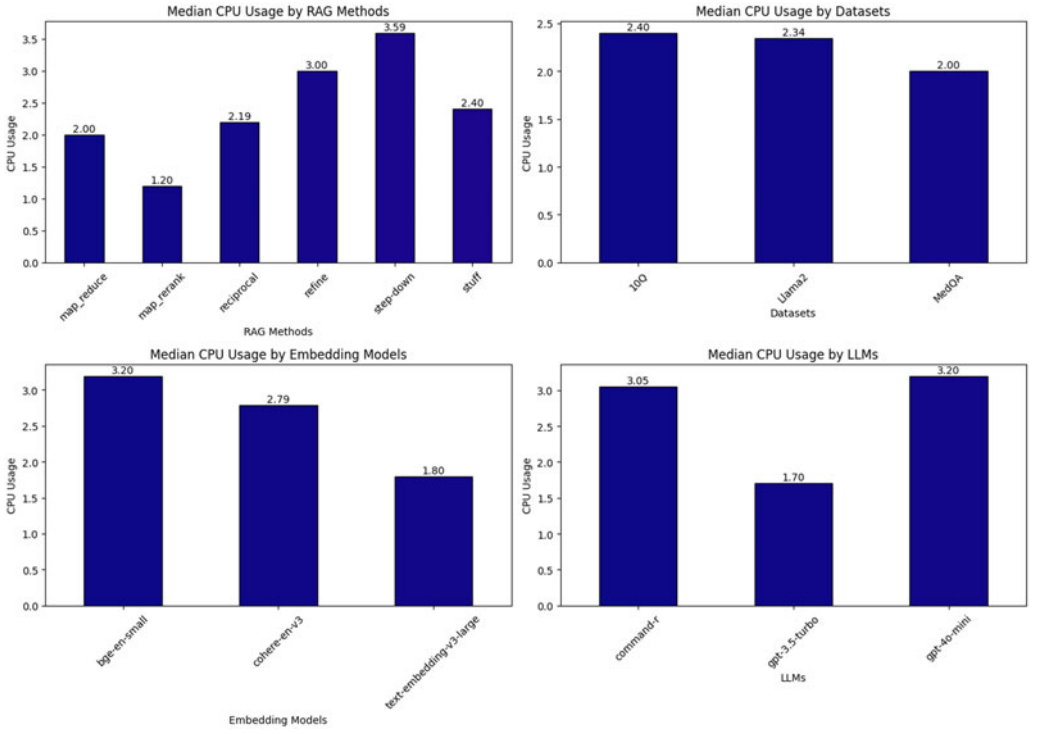

Figure 14. Median CPU (%) usage comparisons by retrieval-augmented generation methods, datasets, embedding models, and large language models.

4.3 Vector databases

Another aspect of our evaluation involved identifying which combination of vectorstores and embedding models provides the lowest run time while also minimizing hardware utilization. This

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

Table 3. Performance metrics for vectorstore systems with different embedding models

Vectorstore	Embedding model	Run time		CPU usage		Memory usage	
		median	std	median	std	median	std
ChromaDB	text-embedding-v3-large	18.34	±6.21	1.50	±1.93	0.00	±0.20
	bge-en-small	19.15	±13.998	1.98	±2.20	0.00	±0.432
	cohere-en-v3	19.22	±13.528	2.65	±2.59	0.00	±0.485
FAISS	bge-en-small	19.40	±11.33	9.75	±10.178	0.10	±0.35
	text-embedding-v3-large	19.75	±12.22	6.80	±10.89	0.10	±0.365
	cohere-en-v3	33.60	±7.69	3.54	±1.044	0.15	±0.936
Pinecone	bge-en-small	19.64	±9.59	1.70	±7.66	0.10	± 0.52
	text-embedding-v3-large	21.85	±8.31	1.79	±8.38	0.10	±0.65
	cohere-en-v3	24.835	±9.88	2.84	±8.68	0.15	±0.62

configuration is crucial for achieving cost-efficiency in high-volume, scalable, and cloud-deployed applications. Notably, maintaining scalability of the vectorstore without significantly compromising performance is essential, as the primary goal of RAG is to expand the vectorstore with additional documents. The performance metrics displayed in Table 3 underscore the significance of selecting the appropriate vectorstore and embedding model combination, as variations in run time, CPU usage, and memory usage are evident across different setups. For instance, ChromaDB paired with the 'Text-embedding-v3-large' model exhibits a median run time of 18.34 ± 6.21 s, alongside a median CPU usage of 1.50% and virtually negligible memory usage.

Conversely, the combination of FAISS and the 'Cohere-en-v3' model results in a higher median run time of 33.60 ±7.69 s and an increased memory usage of 0.15 ±0.936%, along with a relatively high CPU demand compared the remaining configurations. These discrepancies highlight that a tailored approach in configuring vectorstores and embedding models is essential for optimizing performance parameters critical to RAG tasks. Thus, identifying the best configuration can significantly enhance processing efficiency, resource management, and overall system responsiveness, which are crucial for high-volume and large-scale RAG-based applications. Upon further examination of various RAG methodologies and vectorstores, we observed that the 'Stuff' method, when combined with the Pinecone vectorstore, achieved the lowest run time, averaging 12.14 ±1.136 s (refer to Figure 15). Conversely, despite being one of the top performers in terms of similarity scores, the 'Step-Down' method consistently resulted in the slowest run times, regardless of the vectorstore configuration.

Additionally, the discrepancy between response accuracy and hardware utilization becomes more pronounced when considering CPU (%) and Memory (%) usages, as displayed in Figure 15 for both the Reciprocal and Step-Down RAG methodologies. Although both methodologies demonstrate impressive performances in similarity scores (refer to Table 1), they are also concluded to be the most demanding in terms of hardware utilization. These results signify the importance of selecting a suitable configuration for RAG applications, which utterly depends on the use case and available resources. These findings underscore the critical importance of selecting an appropriate configuration for RAG applications, which should be determined based on the specific use case and available resources. We also observed that across all RAG methodologies, ChromaDB yielded the lowest run time, CPU (%), and memory (%) consumption (refer to

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

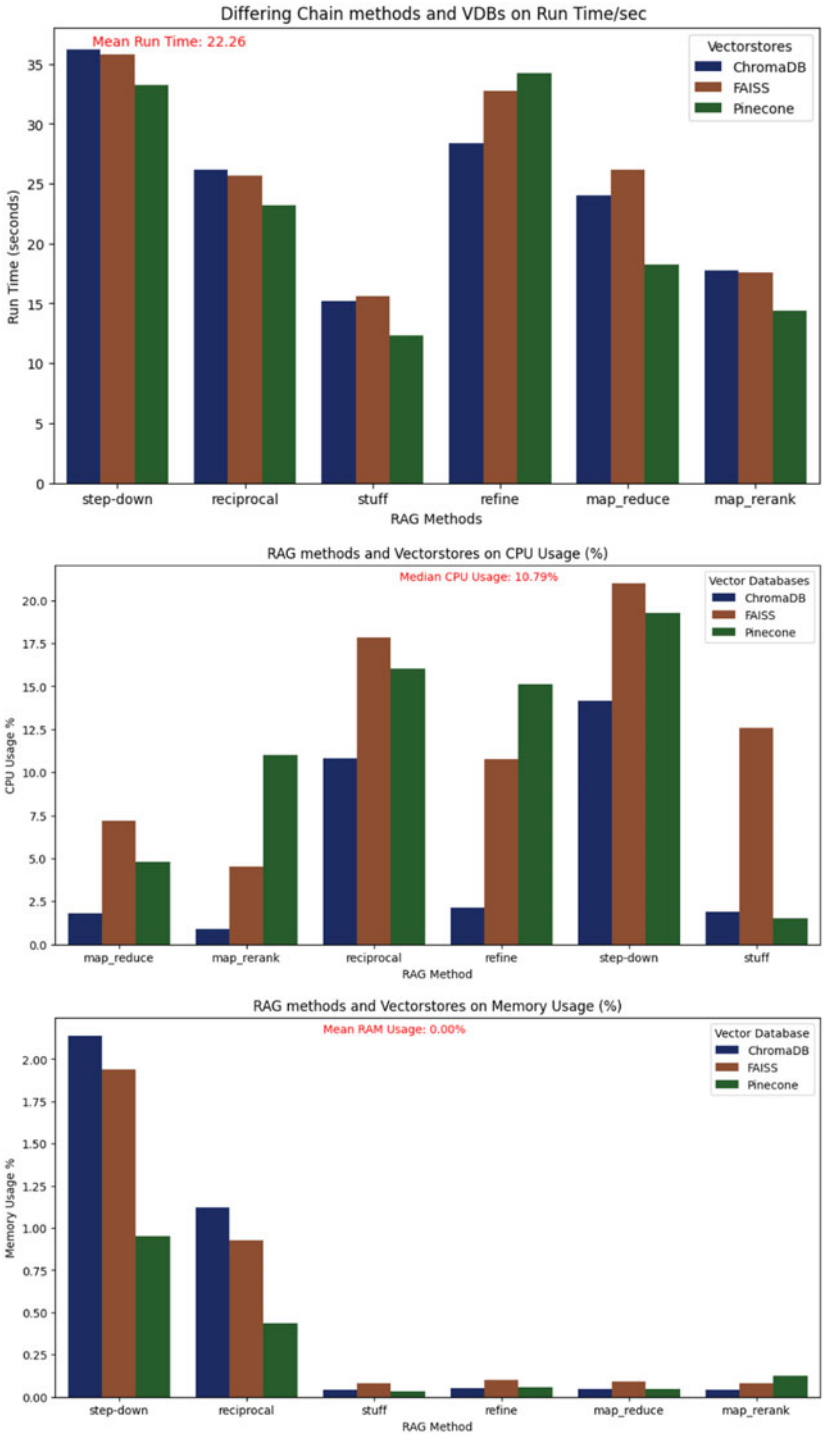

Figure 15. Median run time (sec), CPU, and memory usage comparisons by retrieval-augmented generation methods on different vectorstores.

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

Table 4. Performance metrics for vectorstore systems

	Run time		CPU usage		Memory usage	
Vectorstore	median	std	median	std	median	std
ChromaDB	18.635112	±9.733899	1.800	±2.133639	0.0	±0.319043
FAISS	19.888412	±11.792252	7.300	±10.447708	0.1	±0.459941
Pinecone	22.582000	±9.277844	2.495	±8.196688	0.1	±0.598845

Table 4) with 18.63 ±9.73, 1.80 ±2.13, and 0.0 ±0.31, run time (sec), CPU (%), and memory (%) usage, respectively.

Considering the above factors, the optimal configuration of the RAG method, LLM, embedding model, and vector store is highly dependent on the specific use case. However, the combination of the 'Stuff' method, 'GPT-3.5-Turbo' as the LLM, 'ChromaDB' vector store, and 'Bge-en-small' embedding model demonstrated better performance, exhibiting lower median scores in runtime, CPU usage (%), and memory usage (%).

4.4 Embedding filters

To enhance each RAG methodology, we applied Contextual Compression (see Section 2.5.1) to each retrieval process. This approach aimed to reduce token usage, lower run times, and improve similarity score performance. Due to the delicate balance between filters—such as similarity and redundancy thresholds—and the similarity between document embeddings in vectorspace and user queries, we conducted a grid search over a range of filters on each RAG method and different datasets. Starting from 0.5 for similarity and redundancy scores to 0.9, we used a step value of 0.1 to derive the results presented below (refer to Table 5).

As highlighted in Table 5, we achieved significant efficiency in token usage and runtime across various RAG methods. With the 'Map Reduce' method, we observed an 8.99% reduction in token usage and a 7.2% decrease in runtime. Conversely, the 'Map Re-rank' method did not show improvements in token usage or runtime; however, it concluded with lower standard deviations. Moreover, the 'Reciprocal' method demonstrated substantial efficiency with a 12.5% reduction in token usage and a 4.40% decrease in runtime. Similarly, the 'Refine' method resulted in an impressive 18.6% reduction in token usage and a 3.05% improvement in runtime. Additionally, the 'Step-Down' method achieved an 8.04% reduction in token usage and a notable 13.98% improvement in runtime. Lastly, the 'Stuff' method did not show a median improvement in token usage; however, it achieved a 13.18% reduction in standard deviation and a 1.39% improvement in runtime.

While filtering out document embeddings below a certain threshold value, some valuable information might not retrieved by the retriever. This issue is particularly pronounced when similarity scores are evaluated after applying the contextual compression filters. Our analysis indicates that across all RAG methods, the similarity score deteriorates, emphasizing the tradeoff between response accuracy and resource management once again. Specifically, in the 'Map Reduce' method, the similarity score deteriorated by 4.7%, in 'Map Re-rank' by 7.89%, in 'Reciprocal' by 7.69%, in 'Refine' by 7.59%, and in 'Stuff' by a notable 17.44% (refer to Table 6).

As discussed in Section 2.5.1, contextual compression filters out irrelevant documents by evaluating document embeddings based on their similarity scores relative to the user query. If certain document embeddings contain both valuable and irrelevant information, they may be filtered out due to a lower overall similarity score resulting from the redundant information within.

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

Table 5. Comparison of Contextual Compression on median and standard deviation of token usage, run time, and score

	RAG Method	Token usage		Run time		Score	
		Median	Std	Median	Std	Median	Std
CC^{-}	map_reduce	602.67	±964.04	21.09	±10.50	0.85	± 0.28
	map_rerank	60.00	±346.94	16.49	±3.45	0.76	±0.35
	reciprocal	3132.00	±570.94	24.98	±3.87	0.91	±0.16
	refine	1768.00	±1314.15	26.16	±12.61	0.79	±0.15
	step-down	4436.00	±938.43	34.33	±6.03	0.87	±0.12
	stuff	606.00	±909.19	14.29	±2.93	0.86	±0.26
CC^{+}	map_reduce	548.44	±813.37	19.57	±6.81	0.81	±0.24
	map_rerank	64.48	±267.78	16.89	±1.35	0.70	±0.28
	reciprocal	2738.00	±743.04	23.88	±3.00	0.84	± 0.25
	refine	1439.01	±1153.74	25.36	±2.21	0.73	± 0.24
	step-down	4079.28	±1187.14	29.53	±3.10	0.78	±0.19
	stuff	670.00	±789.30	14.09	±1.43	0.71	±0.21

Note: CC^{+} Contextual Compression method applied

Table 6. Comparison of run time, score, and similarity threshold

Dataset	CC^{-} Score		CC^{+} Similarity threshold		CC^{+} Redundancy threshold		CC^{+} Score	
	Median	Std	Median	Std	Median	Std	Median	Std
10Q	0.865	±0.200	0.80	±0.100	0.80	±0.100	0.821	±0.189
Llama2	0.871	±0.217	0.70	±0.100	0.80	±0.100	0.791	±0.346
MedQA	0.605	±0.291	0.50	±0.100	0.60	±0.100	0.510	±0.170

Note: CC^{+} Contextual Compression method applied

Consequently, the retrieved document may lack sufficient information to provide adequate context for the LLM to generate a relevant answer. Furthermore, applying a redundancy threshold adds another layer of filtering, potentially excluding additional documents and limiting the LLM's context to generate a comprehensive response. This issue, where documents contain dispersed and mixed information, significantly deteriorates the response accuracy performance when contextual compression is applied. As observed in Table 6, we conclude that the optimal configuration of both similarity and redundancy threshold filters varies significantly depending on the quality of the documents or whether there is a disparity of information within documents.

Our findings indicate that each dataset possesses distinct optimal filter thresholds, beyond which performance significantly deteriorates due to sufficient information not being available at higher thresholds when generating adequate responses. For the 10Q dataset, we determined that a similarity threshold of 80% and a redundancy threshold of 80% would limit the deterioration in similarity score to 5.08%. In the case of the Llama2 dataset, we concluded that the optimal threshold values are 70% for similarity and 80% for redundancy. Lastly, for the MedQA dataset, a lower

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

threshold value was required to mitigate deterioration, with the best configuration being 50% for similarity and 60% for redundancy, resulting in a 15.7% reduction in similarity score performance (refer to Table 6). Beyond these optimal filtering values, or thresholds, we observed a significant decline in the similarity score. This decline is attributable to either an insufficient number of relevant documents or the omission of critical information buried within other irrelevant document embeddings. (refer to Table 6).

In essence, contextual compression should be employed to achieve a balance between token usage, run-time, and similarity score. However, this balance must be carefully fine-tuned to minimize the deterioration of the similarity score.

5. Conclusion

In addressing the existing gap in research on the optimization of RAG processes, this paper embarked on a comprehensive exploration of various methodologies, particularly focusing on RAG methods, vector databases/stores, LLMs, embedding models, and datasets. The motivation stemmed from the recognized significance of optimizing and improving RAG processes, as underscored by numerous studies (Vaithilingam *et al.* 2022; Nair *et al.* 2023; Topsakal and Akinci 2023; Manathunga and Illangasekara 2023; Nigam *et al.* 2023; Pesaru *et al.* 2023; Peng *et al.* 2023; Konstantinos and Pouwelse Andriopoulos and Johan 2023; Roychowdhury *et al.* 2023).

Through a comprehensive grid-search optimization encompassing 23,625 iterations, we conducted a series of experiments to evaluate the performance of various RAG methods (Map Re-rank, Stuff, Map Reduce, Refine, Query Step-Down, Reciprocal). These experiments involved different vectorstores (ChromaDB, FAISS, and Pinecone), embedding models (Bge-en-small, Cohere-en-v3, and Text-embedding-v3-large), LLMs (Command-R, GPT-3.5-Turbo, and GPT-4o-Mini), datasets (Dokugami 10Q, LLama2, and MedQA), and contextual compression filters (Similarity and Redundancy Thresholds).

Our findings highlight the significance of optimizing the parameters involved in developing RAG-based applications. Specifically, we emphasize aspects such as response accuracy (similarity score performance), vectorstore scalability, hardware utilizations; run-time efficiency, CPU(%), and Memory(%) usages. These considerations are crucial under various conditions, including different embedding models, diverse datasets across various domains, different LLMs, and various RAG methodologies.

The results of our grid-search optimization highlight the significance of context quality over similarity-based ranking processes or other methods that aggregate all responses iteratively as the final output. Specifically, context quality demonstrates greater importance than simply applying similarity-based methods, which may result in only marginal improvements in similarity scores by excluding less relevant context from the rankings. Additionally, the discussion on the distinction between ambiguity and specificity in user queries (see Section 2.5) further emphasizes the need for increased focus on this methodology. Methods addressing the 'ambiguity' issue yield higher similarity scores (refer to Table 1) compared to those that do not address it. However, a higher similarity score incurs a cost, creating a tradeoff between response accuracy (represented by the similarity score) and resource management, which includes run-time, token usage, and hardware utilization (see Sections 4.1–4.4). To address this issue, our objective was to determine the optimal configurations that balance similarity score performance with run-time efficiency, vectorstore scalability, token usage, as well as CPU and memory usage. The results revealed nuanced performances across different iterations.

In our evaluations for RAG methods, The 'Reciprocal' method emerged as a particularly effective approach, demonstrating higher similarity score compared to other methods, achieving up to 91% (see Figure 13 and Table 1). However, this method also led to an increased number of

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

LLM calls, resulting in greater token usage and extended run time, as it addresses the query-ambiguity problem (refer to Section 2.5.3, Figure 14). In contrast, although the 'Stuff' method does not address query-ambiguity issue, it exhibited an acceptably lower similarity score performance compared to that of the 'Reciprocal' RAG method while only sacrificing 7.2% performance, while cutting down token usage by 71.3% and improving run time by 33.89%(see Table 1 and Figure 14). Moreover, we also observed that, when inadequate context is retrieved, regardless of any post-retrieval ranking processes, the response accuracy significantly deteriorates across all datasets (refer to Table 1), especially with 'Map Re-rank' method and MedQA dataset.

In consideration of hardware utilization, our analysis reveals that GPT-3.5-Turbo, when employed with the 'Bge-en-small' embedding model, demonstrated a median run-time performance of 17.55 ± 5.92 s, alongside a CPU usage of $2.95\% \pm 8.31$. Conversely, when integrated with the 'Text-embedding-v3-large' embedding model, we noted a marginal 3.99% reduction in run-time, paired with a significant 52.5% enhancement in CPU usage efficiency for GPT-3.5-Turbo. This underscores the importance of selecting the optimal configuration of embedding models and LLMs to achieve scalability improvements in high-volume RAG-based applications.

Furthermore, we conclude that vectorstores (vector databases) also play a crucial role in optimizing hardware utilization. Notably, the ChromaDB vectorstore demonstrated superior performance in terms of runtime, CPU, and memory usage across all iterations with different embedding models and vectorstores. Specifically, when employed with the 'Text-embedding-v3-large' embedding model, ChromaDB achieved a runtime as low as 18.34 ± 6.21 s, and a CPU usage of $1.50\% \pm 1.93$. These results put forth another aspect of developing RAG-based applications with more scalable and stable configurations in order to expand the capabilities of RAG processes.

In contextual compression evaluations, we observed that the tradeoff between similarity score performance and runtime, along with token usage, necessitates a tailored optimization process due to the sensitive nature of similarity search. Inadequate context can significantly deteriorate similarity scores due to the application of higher filters (refer to Table 6) where we can observe the evident reduction in similarity score performances, such as 5.08% for 10Q, 9.18% for Llama2 and 15.7% for MedQA datasets. Conversely, identifying the most optimal configuration can substantially reduce both token usage and runtime. This highlights the necessity of deploying contextual compression for specific use cases where slight deteriorations in similarity scores can be tolerated in exchange for lower runtime and token usage (refer to Tables 5 and 6).

In light of our findings, it is evident that optimizing RAG-based applications is essential, given the critical role that parameters such as similarity score performance, run-time efficiency, vectorstore scalability, token usage, and CPU and memory utilization play in the effectiveness of various tools, AI chatbots, and AI agents. Consequently, we emphasize the importance of further research into optimizing RAG processes, which could lead to significant advancements in performance.

Supplementary material. The supplementary material for this article can be found at https://doi.org/10.1017/nlp.2024.53.

References

Andriopoulos K. and Johan P. (2023). Augmenting LLMs with knowledge: a survey on hallucination prevention. *arXiv*. https://doi.org/10.48550/arXiv.2309.16459.

Bentley J.L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM* **18**, 509–517. https://doi.org/10.1145/361002.361007.

Brown T., Mann B., Ryder N., Subbiah M., Kaplan J.D., Dhariwal P. and Neelakantan A. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems* **33**, 1877–1901

Chen W., Chen J., Zou F., Li Y.-F., Lu P. and Zhao W. (2019). RobustiQ: a robust ANN search method for billion-scale similarity search on GPUs. *Proceedings of the 2019 International Conference On Multimedia Retrieval* (pp. 132–140). https://doi.org/10.1145/3323873.3325018.

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

Choi J., **Jung E.**, **Suh J. and Rhee W.** (2021). Improving bi-encoder document ranking models with two rankers and multi-teacher distillation. In *Proceedings of the 44th International ACM SIGIR Conference On Research and Development in Information Retrieval*, 2192–2196. https://doi.org/10.1145/3404835.3463076

Chowdhery A., **Narang S.**, **Devlin J.**, **Bosma M.**, **Mishra G.**, **Roberts A. and Barham P.** (2023). PaLM: scaling language modeling with pathways. *Journal of Machine Learning Research* **24**, 1–113.

Christiani T. (2019). Fast locality-sensitive hashing frameworks for approximate near neighbor search. In Amato G., Gennaro C., Oria V. and Radovanović M. (eds), *Similarity Search and Applications*. Springer International Publishing, pp. 3–17. https://doi.org/10.1007/978-3-030-32047-81.

Cui J., **Li Z.**, **Yan Y.**, **Chen B. and Yuan L.** (2023). ChatLaw: open-source legal large language model with integrated external knowledge bases. https://doi.org/10.48550/arXiv.2306.16092.

Dasgupta A., **Kumar R. and Sarlos T.** (2011). Fast locality-sensitive hashing, In Proceedings of the 17th ACM SIGKDD International Conference On Knowledge Discovery and Data Mining, 1073–1081. https://doi.org/10.1145/2020408.2020578,

Devlin J., **Chang M.-W.**, **Lee K. and Toutanova K.** (2019). BERT: pre-training of deep bidirectional transformers for language understanding. *arXiv*. https://doi.org/10.48550/arXiv.1810.04805.

Dolatshah M., **Hadian A. and Minaei-Bidgoli B.** (2015). Ball*-tree: efficient spatial indexing for constrained nearest-neighbor search in metric spaces. https://doi.org/10.48550/arXiv.1511.00628. arXiv: 1511.00628.

Ghojogh B., **Sharifian S. and Mohammadzade H.** (2018). Tree-based optimization: a meta-algorithm for metaheuristic optimization, arXiv: 1809.09284.

Gupta Utkarsh (2023). GPT-investAR: enhancing stock investment strategies through annual report analysis with large language models. *SSRN Electronic Journal*, arXiv: 2309.03079.

Han Yikun, **Liu Chunjiang and Wang Pengfei** (2023). A comprehensive survey on vector database: storage and retrieval technique, Challenge. *arXiv*, 2023-10-18.

Jégou H., **Douze M.**, **Schmid C.** (2011). Product quantization for nearest neighbor search. *IEEE Transactions On Pattern Analysis and Machine Intelligence* **33**, 117–128. https://doi.org/10.1109/TPAMI.2010.57.

Jeong Cheonsu (2023). A study on the implementation of generative AI services using an enterprise data-based LLM application architecture. *arXiv*. https://doi.org/10.48550/arXiv.2309.

Johnson J., **Douze M. and Jégou H.** (2021). Billion-scale similarity search with GPUs. *IEEE Transactions On Big Data* **7**, 535–547.

LangChain. (n.d.) *MapRerankDocumentsChain documentation*. langChain API documentation. Available at: https://api.python.langchain.com/en/latest/chains/langchain.chains.combine_documents.map_rerank.MapRerankDocumentsChain.html#langchain.chains.combine_documents.map_rerank.MapRerankDocumentsChain.

LangChain. (n.d.) *Contextual compression documentation*. langChain API documentation. Available at: https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/contextual_compression/.

LangChain. (n.d.) *MapReduceDocumentsChain documentation*. langChain API documentation. Available at: https://api.python.langchain.com/en/latest/chains/langchain.chains.combine_documents.map_reduce.MapReduceDocumentsChain.html#langchain.chains.combine_documents.map_reduce.MapReduceDocumentsChain.

LangChain. (n.d.) *RefineDocumentsChain documentation*. langChain API documentation. Available at: https://api.python.langchain.com/en/latest/chains/langchain.chains.combine_documents.refine.RefineDocumentsChain.html#langchain.chains.combine_documents.refine.RefineDocumentsChain.

LangChain. (n.d.) *StuffDocumentsChain documentation*. langChain API documentation, Available at: https://api.python.langchain.com/en/latest/chains/langchain.chains.combine_documents.stuff.StuffDocumentsChain.html#langchain.chains.combine_documents.stuff.StuffDocumentsChain.

Lewis M., **Liu Y.**, **Goyal N.**, **Ghazvininejad M.**, **Mohamed A.**, **Levy O.**, **Stoyanov V. and Zettlemoyer L.** (2019). BART: denoising sequence-to-sequence pre-training for natural language generation, Translation, and and Comprehension. *arXiv*, 1910.13461.

Lewis P., **Perez E.**, **Piktus A.**, **Petroni F.**, **Karpukhin V.**, **Goyal N. and Küttler H.** (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, **33**, Curran Associates, Inc, pp. 9459–9474.

Li L. and Hu Q. (2020). Optimized high order product quantization for approximate nearest neighbors search. *Frontiers of Computer Science* **14**, 259–272.

Lin Jimmy, **Pradeep Ronak**, **Teofili Tommaso and Xian Jasper** (2023). Vector search with openAI embeddings: lucene is all you need. *arXiv*, https://doi.org/10.48550/arXiv.2308.14963 August 28, 2023.

Malkov Y. A. and Yashunin D. A. (2020). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions On Pattern Analysis and Machine Intelligence* **42**, 824–836.

Manathunga S. S. and Illangasekara Y. A. (2023). Retrieval augmented generation and representative vector summarization for large unstructured textual data in medical education. *arXiv*, arXiv: 2308.00479.

Mialon G., **Dessí R.**, **Lomeli M.**, **Nalmpantis C.**, **Pasunuru R.**, **Raileanu R.**, **Rozière B.**, (2023). Augmented language models: a survey. *arXiv*,

https://doi.org/10.1017/nlp.2024.53 Published online by Cambridge University Press

Nair I., **Somasundaram S.**, **Saxena A. and Goswami K.** (2023). Drilling down into the discourse structure with LLMs for long document question answering. *arXiv*, 14593–14606, 2311.13565.

Nigam S. K., **Mishra S. K.**, **Mishra A. K.**, **Shallum N. and Bhattacharya A.** (2023). Legal question-answering in the Indian context: efficacy, challenges, and potential of modern AI models. *arXiv*, preprint arXiv: 2309.14735.

Peng R., **Liu K.**, **Yang P.**, **Yuan Z. and Li S.** (2023). Embedding-based retrieval with LLM for effective agriculture information extracting from unstructured data. *arXiv*, 2308.03107, https://arxiv.org/abs/2308.03107

Pesaru A., **Gill T. and Tangella A.** (2023). AI assistant for document management using lang chain and pinecone. *International Research Journal of Modernization in Engineering Technology and Science*.

Rackauckas Z. (2024). Rag-fusion: a new take on retrieval-augmented generation. *International Journal On Natural Language Computing* **13**, 37–47, arXiv preprint arXiv: 2402.03367.

Roychowdhury S., **Alvarez A.**, **Moore B.**, **Krema M.**, **Gelpi M. P.**, **Rodriguez F. M.**, **Rodriguez A.**, **Cabrejas J. R.**, **Serrano P. M.**, **Agrawal P. and Mukherjee A.** (2023). Hallucination-minimized Data-to-answer Framework for Financial Decision-makers, *arXiv*, 2311.07592.

Sage A. (2023). Great algorithms are not enough —, pinecone, Retrieved December 19, 2023.

Schwaber-Cohen R. (2023). Chunking strategies for LLM applications, Pinecone, Retrieved December 13, 2023,

Singh A., **Subramanya S. J.**, **Krishnaswamy R. and Simhadri H. V.** (2021). FreshDiskANN: a fast and streaming similarity search, *arXiv*, preprint arXiv: 2105.09613.

Sutskever Ilya, **Vinyals Oriol and Le Quoc V.** (2014). Sequence to sequence learning with neural networks, *arXiv*, 1409.3215. arXiv.

Topsakal O. and Akinci T. C. (2023). Creating large language model applications utilizing langChain: a primer on developing LLM apps fast. *International Conference On Applied Engineering and Natural Sciences* 1050–1056.

Vaithilingam P., **Zhang T. and Glassman E. L.** (2022). Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*

Wei J., **Wang X.**, **Schuurmans D.**, **Bosma M.**, **Ichter B.**, **Xia F.**, **Chi E.**, **Le Q. V. and Zhou D.** (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* **35**, 24824–24837.

Zhang M. and He Y. (2019). GRIP: Multi-Store Capacity-Optimized High-Performance Nearest Neighbor Search for Vector Search Engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pp. 1673–1682.

Zheng H. S., **Mishra S.**, **Chen X.**, **Cheng H. T.**, **Chi E. H.**, **Le Q. V. and Zhou D.** (2023). Take a step back: evoking reasoning via abstraction in large language models. *arXiv*, preprint arXiv: 2310.06117.

Zhou X., **Li G. and Liu Z.** (2023). Llm as dba. arXiv preprint arXiv:2308.05481.

Cite this article: Şakar T and Emekci H (2025). Maximizing RAG efficiency: A comparative analysis of RAG methods. *Natural Language Processing* **31**, 1–25. https://doi.org/10.1017/nlp.2024.53