QA-RAG: Leveraging Question and Answer-based Retrieved Chunk Re-Formatting for Improving Response Quality During Retrieval-augmented Generation

Kaushik Roy *, Yuxin Zi, Chathurangi Shyalika, Renjith Prasad, Sidhaarth Murali, Vedant Palit, Amit Sheth *

Posted Date: 4 July 2024

doi: 10.20944/preprints202407.0376.v1

Keywords: Retrieval-augmented Generation; Information Retrieval; Large Language Models

Article

QA-RAG: Leveraging Question and Answer-based Retrieved Chunk Re-Formatting for Improving Response Quality During Retrieval-augmented Generation

Kaushik Roy [1,*], Yuxin Zi [1,*], Chathurangi Shyalika [1,*], Renjith Prasad [1,*], Sidhaarth Murali [2,*], Vedant Palit [3,*], and Amit Sheth [1,*]

[1] Artificial Intelligence Institute, University of South Carolina
[2] National Institute of Technology Karnataka, Surathkal
[3] Indian Institute of Technology, Kharagpur

Abstract: The use of Retrieval-augmented generation (RAG) using large language models (LLMs) has shown potential for addressing issues such as hallucinations and inadequately contextualized responses. A pivotal stage in the RAG process involves a retriever for retrieving chunks based on semantic similarity with the query. In this study, we advocate for and provide experimental evidence supporting integrating and maintaining questions and answers (QA) formatted databases to improve retrieved-context representations and response quality. Our experiments evaluate our approach on benchmark RAG datasets using standard evaluation metrics and provide comparative analyses against state-of-the-art retrieval methods, showing the potential of our approach.

Keywords: Retrieval-augmented Generation; Information Retrieval; Large Language Models

1. Introduction

The rise of large language models (LLMs) and their remarkable natural language processing capabilities have led to the development of various information-retrieval and response (IRR) systems such as information assistance chatbots, search systems, and question-answering systems [1]. LLMs possess a large knowledge base encoded within their extensive parametric memory, acquired through training on enormous volumes of various corpora such as OpenWebTextCorpus and The Pile [2,3]. A typical IRR pipeline consists of a query, relevant information retrieval, and a response. Consequently, in the age of LLMs, several IRR pipelines involve a query as input and an LLM-generated answer to the query by drawing on the relevant information contained within its parametric knowledge. However, in real-world IRR use cases, the query often requires information that the LLM cannot provide, for example, about events that happened beyond the year an LLM was trained. Consider, for example, an LLM-powered fact inquiry system – if such a system inquires about current leaders of countries as of 2024 and the LLM that it relies on is trained in 2021, the system might respond with wrong answers [4]. Retrieval-augmented generation (RAG) has emerged as a promising solution for enabling LLMs to access externally sourced information, thus providing the additional context necessary for accurate and high-quality responses [5]. RAG is a well-studied problem in the NLP literature, and several robust and rigorously evaluated implementations exist, some even incorporated within mature production-grade software [6–9]. As a result, the rapid incorporation and testing of RAG methods in academia and industry has revealed significant open issues. A large part of the issue and the focus of this work, is the retrieval mechanism from external knowledge sources, specifically ensuring that the retrieved information is relevant to the query. Figure 1 shows a complicated retrieval scenario in which a RAG-based IRR system could retrieve irrelevant information for a given query.

 doi:10.20944/preprints202407.0376.v1

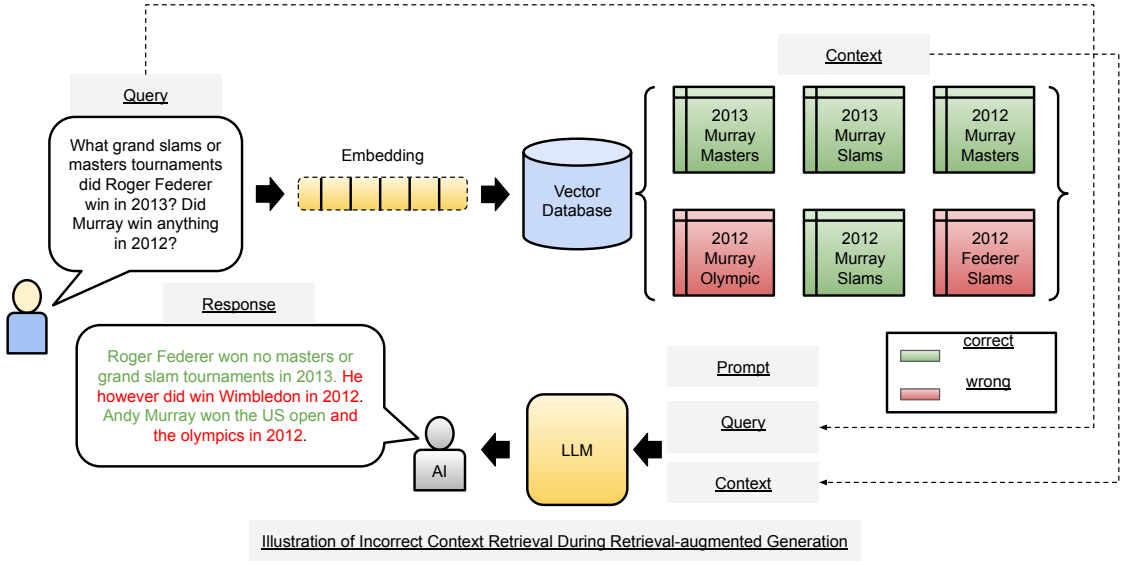

Figure 1. The figure demonstrates examples of irrelevant contexts retrieved by a potentially faulty retriever. The query pertains to tournaments won by two tennis players in different years. In this illustration, the green boxes represent accurate information relevant to the query, while the red boxes represent incorrect information unrelated to the query (irrelevant context). The query asks about the player's information about *grandslams and masters tournaments* in the year *2013*; however, the response contains information about the year *2012* and tournaments that are neither grand slams nor masters tournaments, such as the Olympics.

At the core of the retrieval mechanism in current implementations of RAG-based systems is comparing the query against external information using vector-similarity. As a preprocessing step, collections of external information (assumed not part of the LLM's parametric memory) are stored as vectorized entries in a vector database - vectorization is done using a neural network-based model, and a collection refers to a set of text fragments such as a paragraph or a set of sentences. These text fragments are commonly referred to as chunks. After this preprocessing phase, the input query is vectorized and compared against the text fragments at runtime using vector similarity. The most similar fragments are retrieved as external information considered relevant to the query. However, past literature has established various issues with vector-similarity-based retrieval, specifically towards irrelevant context retrieval, and many fixes have been proposed to mitigate such issues [10]. We discuss these fixes in the background and methods section (Sections 2 and 3. In this work, we introduce a novel *plug-in*, that can be applied to any existing fix, inspired by the concept of Frequently Asked Questions (FAQ) pages. When an inquiry or query is already on an FAQ page, it is easily accessed and leveraged to produce an accurate response. The challenge lies in generating FAQ-style representations of collections and text fragments, and the main contribution of our work is to address this challenge. We refer to our proposed method as QA-RAG.

The rest of the paper is organized as follows: We provide a survey of general categories of current RAG methods, describing the methods, their benefits, and shortcomings. We introduce the proposed QA-RAG method to address the shortcomings of current methods towards better RAG outcomes. We then provide details of the QA-RAG method, our experimental setup for demonstrating its effectiveness, and the results. We end with a discussion of the results and conclude with a general summary and future directions. Our main contributions are the QA-RAG methodology and how it can be used as a plug-in over any existing RAG method to boost the RAG method's performance.

 doi:10.20944/preprints202407.0376.v1

2. Current RAG Methods

Although Section 1 gives an intuitive justification for exploring QA representations, in this section, we provide a formal treatment of commonly used retriever methods, including an explanation of their benefits and shortcomings, followed by arguing for our proposed method in its ability to overcome these shortcomings. We first formalize retriever components regarding its main modules and submodules to make it easier to cover the related work and how it relates to these modules and submodules. Let x denote an input sequence, i.e., a query, and let y denote the response. In RAG, let obtaining a response given a query be denoted by a function $y = \mathbf{f}(x, x_c = \mathbf{g}(\mathcal{C}, x))$, where x_c denotes additional query-relevant context retrieved using a retriever $\mathbf{g}$, from external information sources, denoted by $\mathcal{C}$. Given this notation, we now delve into the various implementations of retriever fixes within RAG frameworks mentioned in Section 1.

As mentioned in Section 1, central to the retrieval mechanism is comparing the query against text fragments from external information sources, also known as "chunks". The process of obtaining such chunks is known as "chunking". In current implementations, chunks are typically arbitrarily defined basic units of information that the external information sources, $\mathcal{C}$, are composed of, such as sentences, paragraphs, or a fixed sequence of words (a typical sequence length is 250 words). The retriever methods that we will cover differ often in their chunking strategy. Consequently, retriever methods also differ in how the query-relevant context is obtained $x_c = \mathbf{g}(\mathcal{C}, x))$ is obtained using the retriever function g. Therefore, the chunking strategy and the specifics of the retriever function will define the broad categories of retriever methods we will cover. While several implementations exist in current and past works, our categorization presents a representative set [11].

We will illustrate the methods using pseudocode snippets, where $\mathbf{g}$ will correspond to the function with the definition def g(x, C), and the chunking strategy is illustrated by the function with the definition def chunker(C). Since the retriever mechanism for obtaining x_c, $\mathbf{g}$ consists of several steps before the actual retrieval step, such as vectorizing the query and chunks, we illustrate the retriever submodule contained within $\mathbf{g}$, as the static function call: retrieve_top_k(vector(query),vector(chunks),<K>). The results of the retriever modules are passed as context into $\mathbf{f}$ to obtain the final generation output.

2.1. Vanilla Retrieval (VaRetr)

VaRetr involves using a brute force vector similarity-based calculation between the query x and all the chunks in $\mathcal{C}$. Such a method can be optimized for efficient retrieval on millions of chunks using an approximate nearest-neighbor implementation. Listing 1 illustrates a Pythonic pseudocode for a typical VaRetr implementation. The vector module employs a neural network-based method to embed text into a high dimensional vector space (e.g., BERT-based sentence-transformer embeddings [12].)

Listing 1: Code structure for VaRetr.

```
def chunker(C):
    """
    splits C into chunks of size n_words
    """
    #set n words
    n_words = <N>
    return split_N(C,N)

def g(x, C):
    """
    vanilla retriever function
    """
    #set top k
    top_k = <K>
```

 doi:10.20944/preprints202407.0376.v1


```
    #obtain chunks
    chunks = chunker(C,N)
    #get query relevant chunks,
    #uses nearest neighbor-based retrieval
    x_c = retrieve_top_k(vector(x),vector(chunks),<K>)
    return x_c
```

2.2. Query Rewriting and Fusion (QRFusion)

An LLM is first employed to rewrite the query x using a prompt that often seeks to clarify the query x (e.g., "rewrite the query for clarity"). Any LLM can be used, but the specific LLM significantly affects the clarity of the rewritten query. The vanilla retrieval mechanism illustrated in Listing 1 is then carried out for both the original and rewritten query. The returned contexts from each of the vanilla retriever is then fused to obtain x_c. There is no change to the chunker function compared to the VaRetr case listed in Listing 1. Illustrating the two retrievals separately as g1(x, C) and g2(x, C), the pseudocode is modified to get the Listing 2 as follows (Note that the chunker not shown as it is assumed the same as the VaRetr case):

Listing 2: Code structure for QRFusion.

```
def g1(x, C):
    """
    vanilla retriever function for original query
    """
    #set top k
    top_k = <K>
    #obtain chunks
    chunks = chunker(C,N)
    #get query relevant chunks,
    #uses nearest neighbor-based retrieval
    return retrieve_top_k(vector(x),vector(chunks),<K>)

def g2(x_new, C):
    """
    vanilla retriever function for rewritten query
    """
    #set top k
    top_k = <K>
    #obtain chunks
    chunks = chunker(C,N)
    #get query relevant chunks,
    #uses nearest neighbor-based retrieval
    return retrieve_top_k(vector(x_new),vector(chunks),<K>)

def g(x, C):
    """
    retriever function to return fused context
    """
    #return fused context
    x_c = g1(x, C) + g2(x_new, C)
    return x_c
```

 doi:10.20944/preprints202407.0376.v1

QRFusion: Benefits and Shortcomings

The benefit of this method is that an LLM can be leveraged to enhance the clarity of the main set of inquiries in the query x. However, much is left to trial and error-based configuration of the right prompt for query rewriting, often based on the specific downstream application [13].

2.3. Hybrid Fusion (HFusion)

Hybrid fusion retrievers use a combination of vector-similarity-based search and keyword-similarity-based search. The specific type of keyword search employed significantly affects the contextual relevance of the combined results. The keyword-based retrieval mechanism and the vanilla method illustrated in Listing 1 are then carried out on the query x. The returned contexts from each retriever are then fused to obtain x_c. In our illustration, we use the function keyword to denote a keyword extractor model (e.g., keyBERT) [14]. Others have also employed simpler mechanisms for keyword searches, such as a TF-IDF representation with a BM25 search or a text constituency parser, i.e., using the parsed syntax elements such as noun phrases and verb phrases contained in the sentence structure as keywords [15,16]. Illustrating the two retrievals separately as g1(x, C) and g2(x, C), we get the following pseudocode:

Listing 3: Code structure for HFusion.

```
def g1(x, C):
    """
    keyword retriever function for original query
    """
    #set top k
    top_k = <K>
    #obtain chunks
    chunks = chunker(C,N)
    #get query relevant chunks,
    #uses keyword-based retrieval
    return retrieve_top_k(keywords(x),keywords(chunks),<K>)

def g2(x, C):
    """
    vanilla retriever function for rewritten query
    """
    #set top k
    top_k = <K>
    #obtain chunks
    chunks = chunker(C,N)
    #get query relevant chunks,
    #uses nearest neighbor-based retrieval
    return retrieve_top_k(vector(x_new),vector(chunks),<K>)

def g(x, C):
    """
    retriever function to return fused context
    """
    #return fused context
    x_c = g1(x, C) + g2(x, C)
```

 doi:10.20944/preprints202407.0376.v1


```
    return x_c
```

HFusion: Benefits and Shortcomings

The benefits of the method illustrated in Listing 3, are that more relevant content with broader coverage can be retrieved; however, at the same time, a lack of attention to the keywords used could introduce noisy content as part of x_c.

2.4. Sentence Window Retriever (SWRetr)

In this type of retriever, The chunking is carried out by splitting $\mathcal{C}$ into fragments of one-or-two sentences. Several sentence tokenizers can be employed to chunk text into sentence fragments. Due to the size of the chunks, the retriever step in sentence window retrieval is often carried out using parallel processing, in addition to employing nearest neighbor searches to avoid inefficiency in processing many sentences during retrieval [17]. The chunker(C) function is used to split $\mathcal{C}$ into sentences, and the retriever mechanism is then similar to Listing 1. Listing 4 shows the pseudocode.

Listing 4: Code structure for SWRetr.

```
def chunker(C):
    """
    splits C into sentences of size n
    """
    #set n words
    n_sentences = <N>
    return sentence_split_N(C,N)

def g(x, C):
    """
    vanilla retriever function
    """
    #set top k
    top_k = <K>
    #obtain chunks
    chunks = chunker(C,N)
    #get query relevant chunks,
    #uses parallelized nearest neighbor-based retrieval
    x_c = retrieve_top_k(vector(x),vector(chunks),<K>)
    return x_c
```

***SWRetr*: Benefits and Shortcomings**

This method provides excellent query accuracy to context vector-based distance search. However, determining the size of the sentence window is subject to extensive trial and error, often dependent on the specifics of the downstream application.

2.5. Metadata Retrieval (MRetr)

Metadata retrievers use a combination of vector-similarity-based search filtered by metadata contained in the query. The specific type of metadata employed significantly affects the contextual relevance of the retrieved results. The metadata filters are applied to the results of the vanilla retriever illustrated in Listing 1. The filtered context is stored in x_c. In our illustration, we use the function metadata to denote a metadata extractor model. Like HFusion, metadata can be keywords, TF-IDF representations, or a text constituency parser-based syntax structure such as noun and verb phrases. More advanced forms of Metadata, such as topics using topic modeling or related entity links using

 doi:10.20944/preprints202407.0376.v1

entity linking from knowledge graphs such as Wikidata, have also been tested in existing implementations [18]. Illustrating the basic retriever and filter separately as g(x, C) and meta_filter(chunks), respectively, we get the following pseudocode:

Listing 5: Code structure for MRetr.

```
def meta_filter(x_c):
    """
    returns meta data filtered chunks
    """
    #set metadata type
    metadata_type = <meta_data>
    return filter(x_c,metadata_type)

def g(x, C):
    """
    vanilla retriever function
    """
    #set top k
    top_k = <K>
    #obtain chunks
    chunks = chunker(C,N)
    #get query relevant chunks,
    #uses parallelized nearest neighbor-based retrieval
    x_c = retrieve_top_k(vector(x),vector(chunks),<K>)
    #filter the chunks and return
    x_c = meta_filter(x_c)
    return x_c
```

***MRetr*: Benefits and Shortcomings**

The retrieval illustrated in 5 can significantly reduce noisy retrieval. Still, downstream task knowledge is required to determine the suitable set of metadata to use, and including too many types of metadata can even exacerbate noisy retrieval.

2.6. Response Controls

The previous subsections in this section have talked about representative categories of retriever and chunking variations. It is also possible to apply controls on the response generation function **f** to improve RAG outcomes. Past work has used a second LLM as a "judge" to refine the response and even placed hard constraints on the phrases allowed in the response [19–21]. In this work, our primary focus is on the retrieval side. However, we also run experiments with response controls and provide a comparison with and without such controls.

Motivation for QA-RAG

Generally speaking, the representative set of retrieval methods discussed above depends on manual engineering efforts in deciding hyperparameters such as window sizes, keyword models, metadata specifics, and prompt designs. At the core of the issue is the lack of a generic way to capture the *essential* semantics of documents in $\mathcal{C}$ required for response synthesis given query x. Our proposed method leverages LLMs based on simple fixed prompts to generate question-and-answer (QA) representations that can be applied to the query x and retrieved content x_c obtained using any retriever mechanism. The core thesis of our method is the expectation that converting the query and context into QA sets makes it easier for the LLM to "read" information in the context x_c

 doi:10.20944/preprints202407.0376.v1

and appropriately relate it to the query x, ultimately improving response quality. Furthermore, the proposed method is downstream-task-agnostic and hyperparameter-efficient.

3. Materials and Methods

3.1. Dataset Details and Response Performance Evaluation

We experiment with two question-answering datasets: NarrativeQA and QASPER[22,23]. NarrativeQA contains question-answer pairs based on the full texts of books and movie transcripts, totaling 1,572 documents. It requires a comprehensive understanding of entire narratives, testing the model's ability to comprehend longer texts. Performance is measured using BLEU (B-1, B-4), ROUGE (R-L), and METEOR (M) metrics [24–26]. The QASPER dataset includes 5,049 questions across 1,585 NLP papers, probing for information within the full text. Answer types include Answerable/Unanswerable, Yes/No, Abstractive, and Extractive. Performance is measured using standard accuracy, precision, recall, and F1 scores. We also experiment with randomized Wikipedia article-based question-answering to consider performance without potential dataset biases. A random set of articles is chosen, and a random sequence of questions and answers is generated using an LLM, which is treated as ground truth. We report results on this randomized dataset using BLEU (B-1, B-4), ROUGE (R-L), and METEOR (M) metrics.

3.2. Proposed Method: QA-RAG

3.2.1. The QA-Plug-In

As mentioned in Section 2, our proposed method acts as a *plug-in* on the query x and retrieved context x_c and can be used with any retriever mechanism. We now describe the details of our plug-in.

Query Plugin

The query plugin works by processing the query x into an elaborated set of subquestions that further elucidate the context of the question. The specific prompt is shown in Listing 6 as follows:

Listing 6: QA Plug-in for query.

```
decomp_prompt(x) = f"""

                    Consider the following query

                    ---- QUERY ----

                    {x}

                    Your task is to decompose the query
                    into subqueries by
                    making sure to follow the instructions below:

                    ---- INSTRUCTIONS ---
                    1. Ensure each subquery is fully
                    self-contained and mentions
                    at least one proper noun.
                    2. Make sure to only include information in
                    the subquery that is
                    explicitly mentioned in the original query.
                    3. If the query is already
                    fully self-contained,
```

 doi:10.20944/preprints202407.0376.v1


```
respond with the original query.
4. Provide your response
in JSON Format as follows:

    {{"Decomposed Queries":
    ["query1","query2", ....,, ]}}

"""
```

Context Plugin

The context plugin works by processing the retrieved context x_c into an elaborated set of questions and answers that further elucidate the contents of the context retrieved. The specific prompt is shown in Listing 7 as follows:

Listing 7: QA Plug-in for context.

```
cexpand_prompt(x_c) = f"""

Consider the following context

---- Context ----

{x_c}

Your task is to convert the context
into a set of questions and answers by
making sure to follow the instructions below:

---- INSTRUCTIONS ---
1. Ensure each question is fully
self-contained and mentions
at least one proper noun.
2. Make sure to only include information in
the context that is
explicitly mentioned in the original context.
3. Provide your response
in JSON Format as follows:

    {{"Questions and Answers":
    "your response"}}

"""
```

3.3. QA-RAG Psuedocode

QA-RAG employs the appropriate plug-ins on the query x and context x_c before response synthesis using the function $\mathbf{f}(x, x_c)$. We will first introduce the pseudocode without any response controls and then cover modifications with response controls.

QA-RAG Psuedocode Without response controls

The pseudocode without response controls is shown in Listing 8 as follows:

 doi:10.20944/preprints202407.0376.v1

Listing 8: QA-RAG Psuedocode.

```
#global instantiation of some language model,
#e.g., llama3-8b
llm = LLM()

def elaborate_x(x):
    """
    elaborates query into
    set of questions using the query plug-in
    """
    #use global llm within local scope
    global llm
    #return formatted questions using question plug-in
    return llm.execute_prompt(qdecomp_prompt(x))

def convert_x_c(x_c):
    """
    converts x_c in a set of QAs using the context plug-in
    """
    #use global llm within local scope
    global llm
    #return QA-formatted context using context plug-in
    return llm.execute_prompt(cexpand_prompt(x_c))

def g(x, C):
    """
    vanilla retriever function
    """
    #set top k
    top_k = <K>
    #obtain chunks
    chunks = chunker(C,N)
    #get query relevant chunks,
    #uses parallelized nearest neighbor-based retrieval
    x_c = retrieve_top_k(vector(x),vector(chunks),<K>)
    #filter the chunks and return
    x_c = meta_filter(x_c)
    #apply context-plugin
    return convert_x_c(x_c)

def f(x, x_c):
    """
    synthesizes response using llm, given query and context
    """
    #use global llm within local scope
    global llm
    #apply query plug-in
    e_x = elaborate_x(x)
    #response prompt
    response_prompt = f"""
```

 doi:10.20944/preprints202407.0376.v1


```
            Consider the following query

            ---- Query ----

            {x}

            Consider also the following
            expanded set of queries relevant
            to the query

            -- Query-relevant Expanded Queries --

            {e_x}

            Below are some
            relevant questions and answers
            that can help answer the query: {x}

            -- Query-relevant Questions and Answers --

            {x_c}


            Your task is to respond to the query {x}
            by considering
            the query-relevant expanded queries
            and query-relevant questions and answers by
            making sure to follow the instructions below:

            ---- INSTRUCTIONS ---
            1. Do not make up information and
             Ensure your response is to the point.
            2. Provide your response
            in JSON Format as follows:

                {{"Response":
                "your response"}}

        """
        #return llm response
        return llm.execute_prompt(response_prompt)

def QA_RAG(x, C):
    """
    main method
    """
    #get query context
    x_c = g(x, C)
    #synthesize response
```

 doi:10.20944/preprints202407.0376.v1


```
        response = f(x, x_c)
        print (response)
```

QA-RAG Psuedocode With response controls

In the modified version of QA-RAG with response controls, we perform an additional check using a specialized LLM-based validator to verify whether the response makes sense given the original query and QA-formatted context. This added measure prevents the LLM from hallucinating without adequate context. Note that all responses are also controlled for the correct format (i.e., JSON format using the json_parse function). Only the modified parts, i.e., the function (def valid_response)that includes LLM-based validator for the response control version of QA-RAG is illustrated in Listing 9.

Listing 9: QA-RAG Psuedocode with Response Control

```
def valid_response(x, x_c, response):
        """
        applies response control to prevent hallucination
        """
        control_prompt = f""" Consider the
        following query and context

        --- Query and Context ---
        Query: {x}
        Context: {x_c}

        Given this query and context,
        judge if the response is
        the correct response to the query.
        Answer only using "yes" or "no".
        return your answer in the following JSON format:

        {{"Yes or no Response": "your response"}}

        """

def QA_RAG(x, C):
        """
        main method
        """
        #get query context
        x_c = g(x, C)
        #synthesize response
        response = f(x, x_c)
        is_valid_response = json_parse(valid_response(
        x, x_c, response
        ))["Yes or no Response"]
        if "yes" in is_valid_response:
            print (response)
        else:
            print ("I do not know how to response")
```

 doi:10.20944/preprints202407.0376.v1

Impelmentation Note.

All the pseudocode presented so far is meant to provide a high-level description of the algorithmic approach for the methods covered. The accompanying software shows all the implementation details (e.g., the specific LLM-based response capture functions for JSON parsing, etc.).

3.4. Experimental Setup

For each method covered in Section 2, we compare against our proposed method, both without and with response control, referred to as Type1 and Type2, respectively. We also include the current state-of-the-art RAG method, RAPTOR, which uses a unique hierarchical context organization before retrieval is carried out [27]. We experiment with the open-source LLMs llama3-8b, llama3-70b, mixtral-8x7b, and gemma-7b implemented using the GROQ API [1]. Figure 2 shows an overview of our experimental setup, displayed on top of Figure 1 to maintain clarity in illustrating various RAG components.

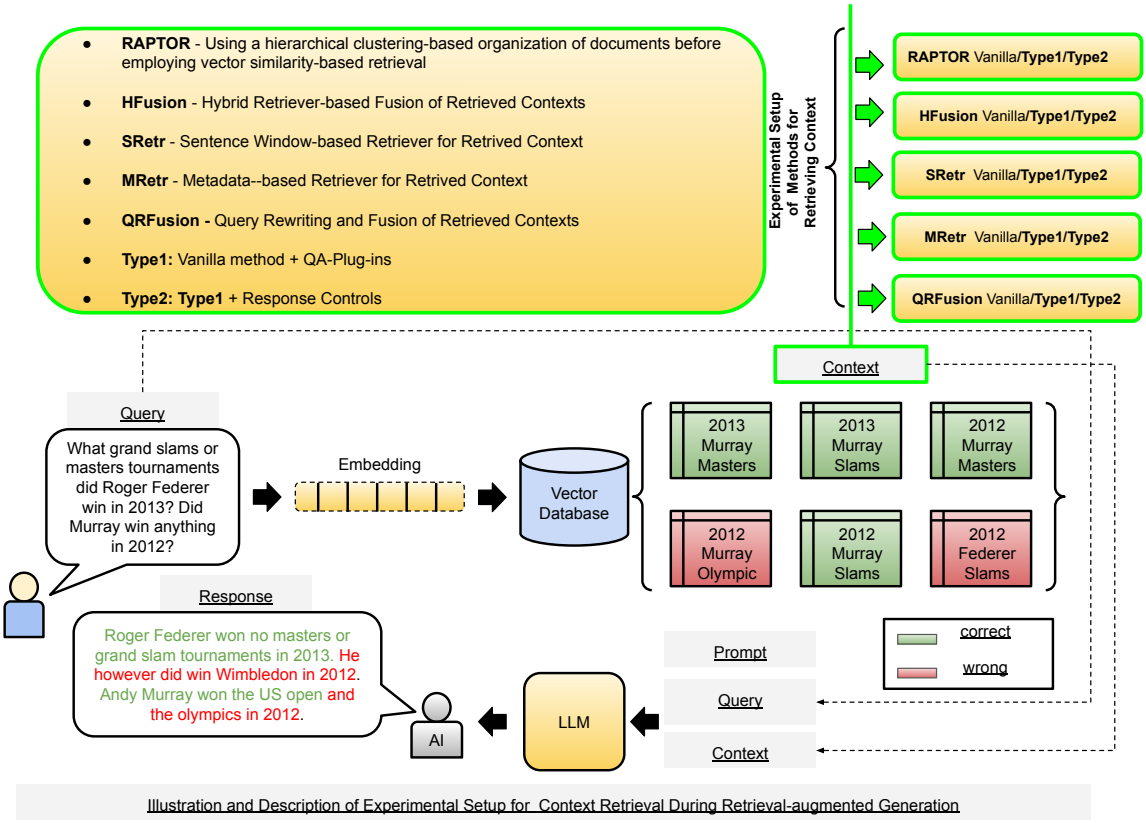

Figure 2. Illustration of the various retriever methods used and compared in our experiments during RAG. The experiments are carried out on benchmark datasets and evaluated using standard metrics (details in Section 3).

3.5. Qualitative Analysis of QA-Plug-ins

Here, we provide the qualitative evaluation of various LLM's ability to convert text into questions to understand their robustness in providing QA-Plug-in capabilities. This qualitative evaluation was conducted using a set of 20 manually curated generation outputs. The evaluation metric used is **hits@<n**, which refers to prompting the LLMs and correcting its ambiguous references based on manually verifying LLM outputs iteratively, in this case, iterating up to n times. Table 1 compares different open-source LLM models used in our experimentation. We find that overall, LLMs perform

[1] https://console.groq.com/docs/models

 doi:10.20944/preprints202407.0376.v1

reasonably well at generating questions from text. All further results are reported using the llama3-70b model.

Table 1. Qualitative Evaluation of QA-Plug-in abilities of various open-source LLMs

Evaluation Metric	llama3-8b	llama3-70b	mixtral-8x7b	gemma-7b
hits@1	87%	**89**%	85%	78%
hits@<5	98%	**99**%	98%	95%

4. Results

4.1. Results on NarrativeQA Dataset

Table 2 reports the results on the NarrativeQA dataset. We report the scores BLEU-1 (**B1**), BLEU-2 (**B2**), ROUGE-L-f1 (**RL**), and METEOR (**M**), where **V** refers to the vanilla implementation of a method, **Type1** refers to the vanilla implementation with the QA-plug-in, and **Type2** refers to Type1 with response controls. The ground truth is the response generated by the LLM given the ground truth context/evidence from the dataset.

Table 2. Results on Narrative QA

Method	$B1_{V/Type1/Type2}$	$B4_{V/Type1/Type2}$	$RL_{V/Type1/Type2}$	$M_{V/Type1/Type2}$
RAPTOR	**0.68/0.92/1.33**	**0.31/0.59/1.12**	**0.76/0.86/0.9**	**0.64/0.89/0.92**
HFusion	0.40/0.59/1.15	0.09/0.31/0.91	0.26/0.41/0.96	0.26/0.29/0.94
SRetr	0.07/0.17/0.51	0.05/0.2/0.32	0.4/0.43/0.46	0.29/0.3/0.45
MRetr	0.20/0.29/0.55	0.04/0.25/0.39	0.37/0.40/0.54	0.20/0.30/0.54
QRFusion	0.11/1.0/1.33	0.39/0.43/1.11	0.33/0.97/0.98	0.18/0.93/0.99

4.2. Results on Random Wikipedia Dataset

Table 3 reports the results on the random Wikipedia dataset. We report the scores BLEU-1 (**B1**), BLEU-2 (**B2**), ROUGE-L-f1 (**RL**), and METEOR (**M**), where **V** refers to the vanilla implementation of a method, **Type1** refers to the vanilla implementation with the QA-plug-in, and **Type2** refers to Type1 with response controls. The ground truth is the response generated by the LLM given the ground truth context/evidence from the dataset.

Table 3. Results on Randomized Wikipedia Dataset

Method	$B1_{V/Type1/Type2}$	$B4_{V/Type1/Type2}$	$RL_{V/Type1/Type2}$	$M_{V/Type1/Type2}$
RAPTOR	**0.86/0.89/1.0**	**0.70/0.71/0.92**	**0.72/0.87/0.9**	**0.78/0.88/0.98**
HFusion	0.60/0.88/1.15	0.82/0.91/0.92	0.63/0.70/0.96	0.77/0.82/0.92
SRetr	0.12/0.47/0.51	0.09/0.32/0.35	0.31/0.33/0.43	0.27/0.30/0.34
MRetr	0.35/0.44/0.58	0.29/0.47/0.55	0.21/0.32/0.34	0.27/0.29/0.44
QRFusion	0.71/0.72/1.3	0.73/0.89/1.12	0.77/0.79/0.9	0.78/0.83/0.98

4.3. Results on QASPER Dataset

Table 4 reports the results on the QASPER dataset. We report the scores Accuracy (**Acc**), Precision (**PR**), Recall (**Recall**), and f1-measure (**F1**), where **V** refers to the vanilla implementation of a method, **Type1** refers to the vanilla implementation with the QA-plug-in, and **Type2** refers to Type1 with response controls. The ground truth is the response captured from the output of the LLM given the ground truth context/evidence from the dataset.

 doi:10.20944/preprints202407.0376.v1

Table 4. Results on QASPER Dataset.

Method	$Acc_{V/Type1/Type2}$	$PR_{V/Type1/Type2}$	$Recall_{V/Type1/Type2}$	$F1_{V/Type1/Type2}$
RAPTOR	**0.54/0.81/0.94**	**0.8/0.81/0.89**	**0.63/0.89/0.96**	**0.83/0.91/0.92**
HFusion	0.43/0.46/0.93	0.31/0.38/0.96	0.36/0.41/0.98	0.35/0.47/0.92
SRetr	0.27/0.4/0.60	0.3/0.75/1.0	0.27/0.4/0.46	0.4/0.42/0.57
MRetr	0.32/0.46/0.5	0.33/0.66/0.8	0.30/0.47/0.52	0.24/0.30/0.56
QRFusion	0.2/0.8/0.95	0.96/0.98/1.0	0.2/0.90/0.96	0.33/0.78/0.99

5. Discussion

Across the Tables 2, 3, and 4, we consistently see improvements across all metrics when using the QA-plug-in, even more so when response controls are applied. This result isn't entirely surprising – In the vanilla implementations, the retrieved context often consists of many bad characters (due to initial document loading errors) or other noisy elements, making it hard for the LLM to synthesize a coherent response by consuming this context. When this context is transformed into coherent QA pairs, and the initial query elaborated using decomposed questions, the LLM naturally can synthesize better responses. Thus, our proposed method is intuitive, simple, and demonstrably improves the RAG pipeline outcomes across various datasets. Among the individual methods experimented with, RAPTOR performs the best, and the performance is further enhanced by augmentation with our proposed QA-RAG method.

6. Conclusion

In this work, we demonstrated a novel methodology for improving RAG outcomes called QA-RAG. We show the efficacy of QA-RAG across a comprehensive experimental setup compared to competitive baselines across benchmark datasets, using standard and well-known evaluation metrics. Although QA-RAG is an intuitive step towards enhancing the representation of retrieved information, we aim to theoretically analyze support for such a representation by formulating RAG as knowledge base entailment over a knowledge base of questions and answers. This effort can also help us classify the general complexity class of RAG, thus opening avenues for solution inspirations from interdisciplinary branches in computer science and mathematics.

Author Contributions: "Conceptualization, Kaushik Roy, Yuxin Zi, and Amit Sheth; methodology, Kaushik Roy, Yuxin Zi, Chathurangi Shyalika, and Renjith Prasad, and Amit Sheth; software, Kaushik Roy, Yuxin Zi, Chathurangi Shyalika, Renjith Prasad, Sidhaarth Murali, and Vedant Palit; validation, Kaushik Roy, Yuxin Zi, Chathurangi Shyalika, Renjith Prasad, Sidhaarth Murali, Vedant Palit, and Amit Sheth; formal analysis, Kaushik Roy, and Yuxin Zi; writing—original draft preparation, Kaushik Roy, Yuxin Zi, Chathurangi Shyalika, and Renjith Prasad, and Amit Sheth; writing— Kaushik Roy, Yuxin Zi, Chathurangi Shyalika, and Renjith Prasad, and Amit Sheth; project administration, Amit Sheth; funding acquisition, Amit Sheth. All authors have read and agreed to the published version of the manuscript."

Funding: "This research is partially supported by NSF Award 2335967 EAGER: Knowledge-guided neurosymbolic AI with guardrails for safe virtual health assistants [28–31]. "

.

Data Availability Statement: "All experiments are run on publicly available datasets. The randomized Wikipedia dataset is available in the accompanying software package. The software package, including source code, data assets, and results tables, will be made publicly available at the following link: Software-Package"

References

1. Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; others. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* **2023**.
2. Gokaslan, A.; Cohen, V. OpenWebText Corpus. https://shorturl.at/9K9wq, 2019.
3. Gao, L.; Biderman, S.; Black, S.; Golding, L.; Hoppe, T.; Foster, C.; Phang, J.; He, H.; Thite, A.; Nabeshima, N.; others. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027* **2020**.

 doi:10.20944/preprints202407.0376.v1

4. Roberts, M.; Thakur, H.; Herlihy, C.; White, C.; Dooley, S. To the cutoff... and beyond? a longitudinal perspective on LLM data contamination. The Twelfth International Conference on Learning Representations, 2023.
5. Asai, A.; Min, S.; Zhong, Z.; Chen, D. Retrieval-based language models and applications. Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 6: Tutorial Abstracts), 2023, pp. 41–46.
6. Borgeaud, S.; Mensch, A.; Hoffmann, J.; Cai, T.; Rutherford, E.; Millican, K.; Van Den Driessche, G.B.; Lespiau, J.B.; Damoc, B.; Clark, A.; others. Improving language models by retrieving from trillions of tokens. International conference on machine learning. PMLR, 2022, pp. 2206–2240.
7. Gao, T.; Yen, H.; Yu, J.; Chen, D. Enabling Large Language Models to Generate Text with Citations. *arXiv preprint arXiv:2305.14627* **2023**.
8. Zirnstein, B. Extended context for InstructGPT with LlamaIndex, 2023.
9. Topsakal, O.; Akinci, T.C. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. International Conference on Applied Engineering and Natural Sciences, 2023, Vol. 1, pp. 1050–1056.
10. Zhou, K.; Ethayarajh, K.; Card, D.; Jurafsky, D. Problems with Cosine as a Measure of Embedding Similarity for High Frequency Words. Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), 2022, pp. 401–423.
11. Gao, Y.; Xiong, Y.; Gao, X.; Jia, K.; Pan, J.; Bi, Y.; Dai, Y.; Sun, J.; Wang, H. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* **2023**.
12. Reimers, N.; Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), 2019, pp. 3982–3992.
13. Chan, C.M.; Xu, C.; Yuan, R.; Luo, H.; Xue, W.; Guo, Y.; Fu, J. Rq-rag: Learning to refine queries for retrieval augmented generation. *arXiv preprint arXiv:2404.00610* **2024**.
14. Song, M.; Feng, Y.; Jing, L. Utilizing BERT intermediate layers for unsupervised keyphrase extraction. Proceedings of the 5th International Conference on Natural Language and Speech Processing (ICNLSP 2022), 2022, pp. 277–281.
15. Kadhim, A.I. Term weighting for feature extraction on Twitter: A comparison between BM25 and TF-IDF. 2019 international conference on advanced science and engineering (ICOASE). IEEE, 2019, pp. 124–128.
16. Metzler, D.P.; Haas, S.W.; Cosic, C.L.; Wheeler, L.H. Constituent object parsing for information retrieval and similar text processing problems. *Journal of the American Society for Information Science* **1989**, *40*, 398–423.
17. Nguyen, D.H.; Le, N.K.; Nguyen, M.L. Exploring Retriever-Reader Approaches in Question-Answering on Scientific Documents. Asian Conference on Intelligent Information and Database Systems. Springer, 2022, pp. 383–395.
18. Yu, W. Retrieval-augmented generation across heterogeneous knowledge. Proceedings of the 2022 conference of the North American chapter of the association for computational linguistics: human language technologies: student research workshop, 2022, pp. 52–58.
19. Zheng, L.; Chiang, W.L.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E.; others. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems* **2024**, *36*.
20. Ahmed, K.; Chang, K.W.; Van den Broeck, G. A pseudo-semantic loss for autoregressive models with logical constraints. *Advances in Neural Information Processing Systems* **2024**, *36*.
21. Zhang, H.; Kung, P.N.; Yoshida, M.; Broeck, G.V.d.; Peng, N. Adaptable Logical Control for Large Language Models. *arXiv preprint arXiv:2406.13892* **2024**.
22. Kočiskỳ, T.; Schwarz, J.; Blunsom, P.; Dyer, C.; Hermann, K.M.; Melis, G.; Grefenstette, E. The narrativeqa reading comprehension challenge. *Transactions of the Association for Computational Linguistics* **2018**, *6*, 317–328.
23. Dasigi, P.; Lo, K.; Beltagy, I.; Cohan, A.; Smith, N.A.; Gardner, M. A Dataset of Information-Seeking Questions and Answers Anchored in Research Papers. Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021, pp. 4599–4610.
24. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.J. Bleu: a method for automatic evaluation of machine translation. Proceedings of the 40th annual meeting of the Association for Computational Linguistics, 2002, pp. 311–318.

 doi:10.20944/preprints202407.0376.v1

25. Lin, C.Y. Rouge: A package for automatic evaluation of summaries. Text summarization branches out, 2004, pp. 74–81.
26. Banerjee, S.; Lavie, A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization, 2005, pp. 65–72.
27. Sarthi, P.; Abdullah, S.; Tuli, A.; Khanna, S.; Goldie, A.; Manning, C.D. RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval. The Twelfth International Conference on Learning Representations, 2023.
28. Sheth, A.; Gaur, M.; Roy, K.; Faldu, K. Knowledge-intensive language understanding for explainable AI. *IEEE Internet Computing* **2021**, *25*, 19–24.
29. Sheth, A.; Gaur, M.; Roy, K.; Venkataraman, R.; Khandelwal, V. Process Knowledge-Infused AI: Toward User-Level Explainability, Interpretability, and Safety. *IEEE Internet Computing* **2022**, *26*, 76–84.
30. Sheth, A.; Roy, K.; Gaur, M. Neurosymbolic Artificial Intelligence (Why, What, and How). *IEEE Intelligent Systems* **2023**, *38*, 56–62.
31. Sheth, A.; Roy, K. Neurosymbolic Value-Inspired Artificial Intelligence (Why, What, and How). *IEEE Intelligent Systems* **2024**, *39*, 5–11.