

CORAG: A Cost-Constrained Retrieval Optimization System for Retrieval-Augmented Generation

Ziting Wang
Nanyang Technological University
Singapore
ziting001@e.ntu.edu.sg

Haitao Yuan
Nanyang Technological University
Singapore
haitao.yuan@ntu.edu.sg

Wei Dong
Nanyang Technological University
Singapore
wei_dong@ntu.edu.sg

Gao Cong
Nanyang Technological University
Singapore
gaocong@ntu.edu.sg

Feifei Li
Alibaba Group
China
lifeifei@alibaba-inc.com

ABSTRACT

Large Language Models (LLMs) have demonstrated remarkable generation capabilities but often struggle to access up-to-date information, which can lead to hallucinations. Retrieval-Augmented Generation (RAG) addresses this issue by incorporating knowledge from external databases, enabling more accurate and relevant responses. Due to the context window constraints of LLMs, it is impractical to input the entire external database context directly into the model. Instead, only the most relevant information, referred to as “chunks”, is selectively retrieved. However, current RAG research faces three key challenges. **First, existing solutions often select each chunk independently, overlooking potential correlations among them. Second, in practice, the utility of chunks are “non-monotonic”, meaning that adding more chunks can decrease overall utility. Traditional methods emphasize maximizing the number of included chunks, which can inadvertently compromise performance. Third, each type of user query possesses unique characteristics that require tailored handling—an aspect that current approaches do not fully consider.**

To overcome these challenges, we propose a cost-constrained retrieval optimization system *CORAG* for retrieval-augmented generation. We employ a Monte Carlo Tree Search (MCTS)-based policy framework to find optimal chunk combinations sequentially, allowing for a comprehensive consideration of correlations among chunks. Additionally, rather than viewing budget exhaustion as a termination condition, we integrate budget constraints into the optimization of chunk combinations, effectively addressing the non-monotonicity of chunk utility. Furthermore, by designing a configuration agent, our system predicts optimal configurations for each query type, enhancing adaptability and efficiency. Experimental results indicate an improvement of up to 30% over baseline models, underscoring the framework’s effectiveness, scalability, and suitability for long-context applications.

PVLDB Reference Format:

Ziting Wang, Haitao Yuan, Wei Dong, Gao Cong, and Feifei Li. CORAG: A Cost-Constrained Retrieval Optimization System for Retrieval-Augmented Generation. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights

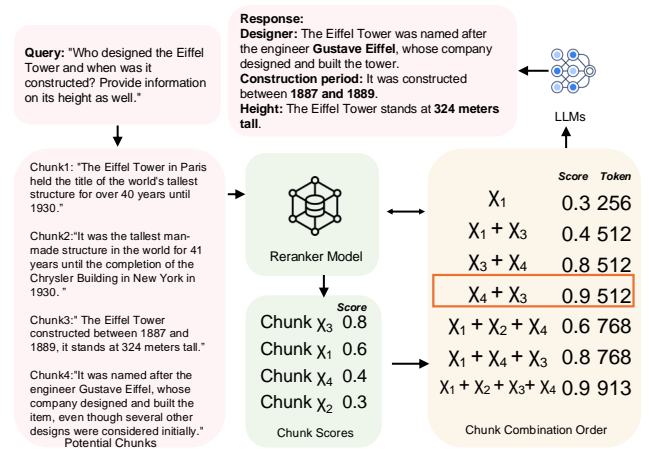


Figure 1: Example of chunks combination order.

1 INTRODUCTION

Although LLMs have demonstrated exceptional capabilities in generation tasks, they often struggle with accessing up-to-date information, which can lead to hallucinations [10, 38]. To address these challenges, RAG has emerged as a crucial solution. By integrating external data sources into LLM, RAG can provide more accurate, relevant, and up-to-date information. Nowadays, RAG has been widely studied in the context of LLMs especially for tasks requiring update external knowledge such as question answering task [2, 22, 29], medical information retrieval [1, 32], and time series analysis [12, 26, 40]. External data sources are often extremely large, making it impractical to input them directly into the LLM. To address this issue, data is typically split into disjoint chunks and stored in a vector database, and then users query the most useful chunks to construct prompts for LLMs. Therefore, designing efficient and accurate structures and algorithms to search for the most relevant chunks has become a prominent research topic and has been widely studied in both the database [39, 48] and machine learning communities [2, 35, 43].

licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

However, there are three key challenges in the existing approaches.

Challenge 1: Correlations between chunks. Currently, two primary methods are used to identify the most relevant chunks. The first approach formulates the problem as a approximate k-nearest neighbor (AKNN) task [41, 45], where each chunk is assigned a score, and the approximate top- k chunks ranked by score are selected. The second approach clusters the chunks, returning all chunks within the most relevant clusters in response to a query [22, 29]. However, both methods overlook potential correlations between chunks: the first approach disregards correlations entirely, while the second approach accounts for them only superficially by treating all chunks within each cluster as equally relevant. As a result, when multiple chunks convey similar or overlapping information, these methods introduce substantial redundancy in the selected chunks.

For example, as illustrated in Figure 1, when querying the height and history of the Eiffel Tower, if each chunk is treated independently, a greedy method would select chunks χ_3 and χ_1 since they have the top two scores. However, both chunks only provide historical information, which is insufficient to fully address the query. To better address the query, it is necessary to include a chunk with constructor’s name, such as χ_4 . On the other hand, the clustering approach would return all of χ_1, χ_2, χ_3 , and χ_4 , resulting in redundancy. An optimal solution would instead select χ_3 and χ_4 , as they provide the required information without redundancy. Additionally, research [11, 19, 42] has shown that the order of chunks influences LLM performance, a factor that existing methods also overlook. Following the example of the Eiffel Tower, when chunks χ_3 and χ_4 are selected, placing χ_4 first yields a higher score compared with the reverse order will have a better performance. However, determining the optimal chunk combination order is a challenging task since both of them require a search space growing exponentially with the number of available chunks. In this paper, we further demonstrate that this problem is NP-hard (see Section 2.1).

Challenge 2: Non-monotonicity of utility. Current solutions operate on the assumption that including more chunks will always yield better final results. Specifically, in the AKNN-based approach, exactly k chunks are selected deterministically each time. In the clustering-based approach, a distance threshold between clusters and the query is set, and all clusters within this threshold are returned. Both of them return as many chunks as possible. However, in practice, the utility of chunks is not monotonic. More specifically, excessive chunks can dilute key information by adding marginally relevant content, creating noise that reduces clarity. Additionally, conflicting or nuanced differences across chunks may confuse the model, lowering response quality. For example, as illustrated in Figure 1, when χ_3 and χ_4 are selected, adding the chunk χ_1 decreases utility, highlighting that utility scores are often non-monotonic in practice.

Challenge 3: Diversity of queries: User queries come in different types, each requiring its own ranking strategy due to their unique characteristics [47]. In current RAG systems, the utility scores of chunks often are determined by the assigned reranker model. So far, various reranker models exist, but we observe that their performance varies significantly across different query types, and no

single fixed reranker model consistently outperforms the others across all query variations (see our experiments in Section 6.3.4 for more details). Current methods [20, 46] typically rely on static reranker models for ranking chunks, lacking the flexibility to adapt to varying query contexts.

Problem Statement: Is there a RAG system that fully considers correlations between chunks and the non-monotonicity of utility while being adaptable to all types of queries?

1.1 Our Contributions

In this paper, we answer this question in the affirmative, by proposing a novel MCTS based policy tree framework to optimize chunk retrieval in RAG systems. In summary, our contributions can be summarized as follows:

- We propose the first RAG framework that considers the chunk combination order for the RAG task. Instead of considering each chunk independently or at the cluster level, we use MCTS to help search the optimal chunk combination order sequentially. The high-level idea is as follows: First, we initialize the root node. Then, in an iterative process, we expand the tree by selecting the highest utility node and computing its expended nodes’ utilities. After each expansion, we update the utilities throughout the entire policy tree. During this process, the decision at each iteration depends on the chunks already selected, allowing us to fully consider the correlations between chunks. Moreover, MCTS reduces the exponential search space to linear, and we apply parallel expansion techniques to further enhance computational efficiency. With such designs, we address *Challenge 1*.
- In contrast to prior RAG frameworks that consider the exhaustion of the budget as one of termination conditions, we propose a novel formulation wherein budget constraints are integrated into the process of optimizing chunk combinations to fully consider the non-monotonicity of utility of chunks thereby addressing *Challenge 2*. Moreover, by prioritizing high-relevance, low-cost chunks and factoring in token length, we further reduce computational costs.
- We propose a contrastive learning-based agent that dynamically adjusts MCTS configurations per query, adapting reranker models and configurations to the specific query domain. This approach tailors retrieval for dynamic, domain-specific queries with flexibility and robustness, addressing *Challenge 3*.
- Additionally, we conducted comprehensive experiments, comparing our framework with several state-of-the-art methods. The results validate the effectiveness, efficiency, and scalability of our approach, also showing a performance improvement of 30% over the baseline.

2 PRELIMINARIES

In this section, we first introduce the definitions of some key concepts in Section 2.1, such as chunks and chunk combination order. Next, we give the NP-hard proof of the chunk order optimization problem. At last, we discuss the related work in Section 2.3.

2.1 Key Concepts

RAG & Chunks. RAG is an effective method for improving the performance of generation models by retrieving relevant context

from an external corpus. In this approach, the corpus is first divided into smaller, manageable units called *chunks*, which are stored in a vector database. Therefore, we can give a formal definition of chunk as follows:

Definition 2.1 (Chunk). Let C represent a corpus of documents, and a chunk χ is defined as a contiguous block of text extracted from C . Formally, a chunk χ consists of a sequence of tokens (t_1, t_2, \dots, t_n) , where each t_i is a token from C and the size n is set by users.

In the RAG system, each chunk is embedded into a vector representation using an embedding model, which captures the chunk’s semantic meaning and enables the retrieval of contextually similar content. When a new query is received, the vector database performs a similarity search to identify the chunks that are most semantically relevant to the query. These retrieved chunks are then passed to a generator (e.g., a large language model) to produce a final response based on the retrieved content. Specifically, the more tokens a chunk contains, the higher the cost incurred by the generator. Thus, we define the cost of a chunk as $\text{cost}(\chi) = |\chi|$, which equals to the number of tokens in the chunk.

Chunk Combination Order. In the RAG system, the retrieval result from the vector database may include multiple chunks. However, due to input limitations of the generation model, using all of these chunks is impractical. Therefore, it is necessary to select an optimal subset of chunks, known as a *chunk combination*, that fits within a given cost budget. Additionally, the order of the chunks within the combination significantly impacts the performance of the generation model. The goal is to identify the chunk combination order with the optimal order, formally defined as follows:

Definition 2.2 (Optimal Chunk Combination Order Selection). Let $\{\chi_1, \chi_2, \dots, \chi_k\}$ be a set of potential chunks, \mathcal{B} be the cost budget, and $\Phi = \langle \chi_{\phi_1}, \dots, \chi_{\phi_m} \rangle$ represent a potential chunk combination order, where each χ_{ϕ_i} is a chunk, and the index ϕ_i indicates its position Φ . Let $U(\Phi)$ be the utility score assigned by the reranker model, which may be arbitrary or composite. Our objective is to find the chunk combination order that maximizes the utility score while adhering to the cost constraint of feeding them into the LLMs to generate the final response, i.e., searching for

$$\hat{\Phi} = \arg \max_{\Phi} U(\Phi) \quad \text{s.t.} \quad \sum_{\chi_i \in \Phi} \text{cost}(\chi_i) \leq \mathcal{B} \quad (1)$$

2.2 Proof of NP-hard

To demonstrate that chunk combination order selection is NP-hard, we reduce the Maximum Weighted Hyperclique Problem (MWHP) to it. Since MWHP is NP-hard, we show that any MWHP instance can be transformed into a Chunk Combination Optimization instance in polynomial time.

2.2.1 Problem definition of MWHP. Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E}, w_1, w_2)$, where \mathcal{V} is the set of vertices, \mathcal{E} is the set of hyperedges, where each contains a subset of \mathcal{V} . $w_1 : v \rightarrow \mathbb{R}$ and $w_2 : e \rightarrow \mathbb{R}$ are weight functions assigning a weight to each vertex and hyperedge, respectively. Given a subset of vertices $\mathcal{V}' \subseteq \mathcal{V}$, we say a hyperedge e belongs to \mathcal{V}' , i.e., $e \in \mathcal{V}'$, if \mathcal{V}' covers all vertices of e . The objective is to find k vertices maximizing the sum

of the weights of these vertices and their covered hyperedges:

$$\arg \max_{\mathcal{V}' \subseteq \mathcal{V}, |\mathcal{V}'| = k} \left(\sum_{v \in \mathcal{V}'} w_1(v) + \sum_{e \in \mathcal{V}'} w_2(e) \right). \quad (2)$$

2.2.2 Reduction process. We now construct a corresponding Chunk Combination Optimization Problem instance from the given MWHP instance. For each node $v \in \mathcal{V}$, we create a corresponding chunk χ_v . We define its cost $\text{cost}(\chi_v) \equiv 1$. Then, a chunk combination order Φ corresponds to a subset of vertices of \mathcal{V} , which is denoted as $\mathcal{V}(\Phi) \subseteq \mathcal{V}$. We define its utility as

$$U(\Phi) = \sum_{v \in \mathcal{V}(\Phi)} w_1(v) + \sum_{e \in \mathcal{V}(\Phi)} w_2(e). \quad (3)$$

Finally, we set $\mathcal{B} = k$ and our objective is

$$\arg \max_{\Phi} U(\Phi) \quad \text{s.t.} \quad \sum_{\chi_i \in \Phi} \text{cost}(\chi_i) = |\Phi| \leq k. \quad (4)$$

Denote Φ^* as the solution of (4), then, it is obvious $\mathcal{V}(\Phi^*)$ is the solution of (2) the reduction can be done in time of $O(|\mathcal{V}| \cdot |\mathcal{E}|)$.

Please note that a precondition of this reduction is that, in our Chunk Combination Optimization Problem, we allow the reranker model to be arbitrary, meaning the utility scores can also be assigned arbitrarily. The complexity of finding the optimal chunk combination order can be significantly reduced if certain assumptions are made about the reranker. For instance, if the reranker does not consider correlations and simply sums the utility scores of individual chunks linearly, each chunk could then be evaluated independently. However, in this paper, we address the most general case, making no assumptions about the reranker model.

2.3 Related Work

2.3.1 Retrieval-augmented Generation. RAG[14, 20] is widely used for handling knowledge-intensive NLP tasks. In a typical RAG pipeline, a dense-embedding-based retriever searches for relevant information from an external database, which is then used by the LLM during the generation process. To improve this pipeline, some studies[5, 18, 22, 35] have focused on adjusting retrievers to suit the generation needs of LLMs better, developing multi-step retrieval methods, and filtering out irrelevant information. Although there are many advanced retrievers[8, 9, 15, 16, 27, 34], it’s more promising to optimize the retriever and LLM together in an end-to-end process[25, 31]. For example, the research[30] has focused on training retrievers and LLMs together, either simultaneously or in stages. However, this requires surrogate loss for optimization and complicates the training pipeline, especially when the embedding database needs to be re-indexed frequently which will bring high compute costs. Therefore, methods such as [5] decompose complex, multi-step queries into smaller sub-intents to improve response comprehensiveness without frequently re-indexing. However, these approaches often overlook the critical role of chunk combination order, which can significantly impact the overall response quality of LLMs. To the best of our knowledge, this paper is the first approach to consider chunk combination order within the RAG task.

2.3.2 Reranking for RAG. Reranking methods are crucial for enhancing retrieval performance within the RAG pipeline [43, 44, 51].

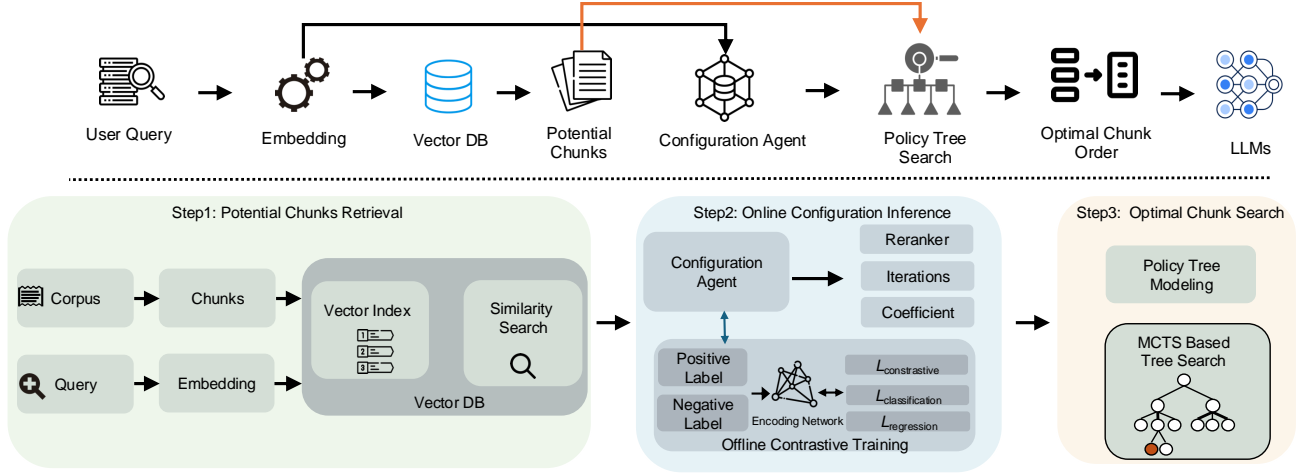


Figure 2: The architecture overview of CORAG system

Traditional reranking approaches [33, 50] typically rely on mid-sized language models, such as BERT or T5, to rank retrieved contexts. However, these models often struggle to capture semantic relationships between queries and contexts, especially in zero-shot or few-shot settings. Therefore, recent research [43] highlights the potential of instruction-tuned LLMs to improve context reranking by more accurately identifying relevant contexts, even in the presence of noise or irrelevant information. Despite these advancements, the full capacity of LLMs for reranking in RAG systems remains underutilized. In particular, studies have shown that chunk arrangement can impact LLM performance [19], emphasizing the need to consider chunk combination order in RAG tasks. However, existing models are not well-suited for cases where optimal retrieval requires specific sequences or combinations of chunks, rather than isolated chunks. Hence, future research is needed to better leverage LLMs for arranging chunks more effectively in response to queries within the RAG framework.

2.3.3 Reinforcement Learning for Large Language Models. Recently, reinforcement learning (RL) has been increasingly utilized in various data management and RAG tasks. The RL technique can enable large language models to improve their generation ability by leveraging external knowledge sources, such as search engines [13, 23]. In particular, the human feedback [4, 36, 37] can be integrated to help models produce more accurate and contextually relevant responses through the RL framework. In addition, some query optimization approaches [17, 21, 49] further refine retrieval processes, allowing model performance to inform query adjustments and ultimately enhance downstream task outcomes. In this work, we apply a lightweight RL technique MCTS to optimize the chunk combination order searching progress in the RAG system. We also introduce a configuration agent to guide the MCTS search process. To our best knowledge, this is the first approach to addressing this specific problem.

3 SYSTEM OVERVIEW

As previously mentioned, existing RAG frameworks face three key challenges: how to fully consider correlations between chunks and

the in-monotonicity of the utility of chunk combination order, and adapt to diverse query domains. These challenges result in reduced relevancy of the outputs. To address these issues, we introduce *CORAG*, a system designed to retrieve the optimal chunk combination while taking into account the query domain and user budget. As the most important component of our system, we introduce the Optimal Chunk Combination Search model. This model employs MCTS based policy tree to perform sequential searches of chunk combination order under a cost constraint, allowing us to fully consider the correlations between chunks (Challenge 1) as well as the non-monotonic nature of utility of chunk combination orders (Challenge 2). Additionally, we propose a Configuration Inference module that recommends the optimal MCTS configuration and reranker tailored to various query domains, thereby addressing the challenge 3. Below, we give a brief descriptions for these two modules.

Optimal Chunk Combination Search: A straightforward approach to considering chunk correlations involves retrieving potential chunks from a vector database (as shown in step 1 in Figure 2) and exhaustively exploring all possible chunk combinations. However, this method incurs significant latency and computational costs. To mitigate this, we construct a policy tree (as shown in step 2), reframing the optimal chunk combination search as a node search problem within the tree. Specifically, the root node of the policy tree represents an initial empty state, and each child node corresponds to a specific combination of chunks. For example, if the root node has a child node representing chunk χ_1 , one of its child nodes might represent the combination $\chi_1 + \chi_2$, while another could represent $\chi_1 + \chi_3$.

We design a search algorithm based on MCTS to address this problem. Unlike traditional MCTS, our approach expands the node with the highest utility in each iteration, simultaneously evaluating all possible child nodes. Additionally, we account for both cost and budget constraints during the policy tree search process. Node utility is calculated by balancing exploration with cost control, optimizing for both efficiency and accuracy.

Configuration Inference: A simple solution for configuration tuning is to enumerate every possible configuration or reranker

and compute the results in parallel, and then select the optimal configuration. However, this would result in impractical costs for the RAG system. To optimize the configuration (i.e., the number of iterations, cost coefficient, and exploration coefficient) for the policy tree search process, we introduce a configuration agent that dynamically generates configurations based on the query domain. To ensure the model’s effectiveness, we employ a contrastive learning approach that uses positive and negative label pairs: positive labels correspond to query embeddings from the same optimal reranker, while negative labels come from different optimal reranker. A joint loss function is used to simultaneously optimize both the regression (for parameter tuning) and contrastive learning (to enhance label differentiation).

Summary. The pipeline of our framework is shown in Figure 2. We first generate an embedding for the input query, which is then used to retrieve potential chunks from the vector database. These query embeddings are also fed into the configuration agent, which dynamically generates the optimal MCTS configuration based on the query domain. Using this optimal configuration, we can search in the policy tree to determine the optimal chunk combination and order from the retrieved potential chunks. Finally, this optimal chunk combination is used to construct the final prompt for the LLMs.

4 CHUNK COMBINATION RETRIEVAL

As previously discussed, the order in which chunks are combined significantly impacts the efficiency of prompt construction in LLMs. Enumerating all possible orders of chunk combinations is not feasible due to the vast number of potential combinations, particularly when the scenario involves a large number of chunks. In this section, we present a novel method that achieves a good trade-off between efficiency and accuracy in searching for the optimal chunk combination order problem. In Section 4.1, we model the problem as searching the optimal node within a policy tree (Section 4.1). Then, we propose an MCTS-based algorithm to address this node search problem (Section 4.2).

4.1 Policy Tree Search Modeling

To approach the optimal combination order, the first step is to find a data structure that enables efficient enumeration of all possible combination orders. A natural choice is a tree, allowing us to explore all potential answers by traversing from the root to the leaf nodes. **Policy Tree.** As illustrated in Figure 3, we construct a policy tree to represent all potential orders of chunk combinations sourced from the vector database. Specifically, the root node symbolizes the initial state without any chunk, with each subsequent node depicting a selected chunk from the potential ones. Thus, a child node emerges from its parent by selecting the next available chunk from the queue of potential chunks and incorporating it into the sequence established by the ancestor node. For instance, if a node represents the chunk combination order $\{\chi_1\}$, then a child node might embody a subsequent combination order such as $\{\chi_1, \chi_2\}$, $\{\chi_1, \chi_3\}$, or $\{\chi_1, \chi_4\}$. Accordingly, we define the policy tree formally as follows:

Definition 4.1 (Policy Tree). Given a query q and a set of potential chunks $\{\chi_1, \chi_2, \dots, \chi_n\}$, we construct a policy tree T . The root

node of T represents the initial state, devoid of any chunks. Each subsequent non-root node embodies a chunk set, achieved by incorporating a newly selected chunk from the remaining potential chunks into the sequence at its parent node. This process sequentially constructs an ordered chunk combination in each non-root node and our objective is to find the node with the highest utility score.

Within the policy tree, our goal is to select a node that encompasses ordered chunks offering the highest benefit at the lowest cost. To accomplish this, we need to devise a utility calculation function to evaluate the trade-off between benefit and cost. This function is quantified through what we define as the “node utility”, described as follows.

Node Utility. The utility metric comprises two components: the benefit derived from selecting the chunk combination and the cost associated with using the chunk as a prompt in LLMs. Specifically, the benefit is quantified with LLMs, which can measure the similarity between the selected chunks and the query. In particular, we denote it as the node value V . Next, we further use the Upper Confidence Bound (UCB)[3] algorithm to balance exploitation (node value $V(v_i)$) and exploration (search count $N(v_i)$) for a given node v_i . Regarding cost, we consider the token cost as defined in Section 2 and measure it by the proportion of the current chunk combination’s cost relative to the total allocated budget \mathcal{B} . Therefore, the node utility is defined as follows:

Definition 4.2 (Node Utility). Given a policy tree and a cost budget \mathcal{B} , the utility of a non-root node is defined as:

$$U(v_i) = \frac{V(v_i)}{N(v_i)} + c \sqrt{\frac{\ln N}{N(v_i)}} - \lambda \frac{\text{cost}(v_i)}{\mathcal{B}} \quad (5)$$

where $V(v_i)$ is the estimated benefit value of the chunk combination at node v_i determined by a trained model, $N(v_i)$ is the count of visits to node v_i , promoting exploration of less frequented nodes, and N is the total number of visits across all nodes in the policy tree, ensuring a balance between exploration and exploitation. In addition, $\text{cost}(v_i)$ denotes the token cost for node v_i , \mathcal{B} is the total token budget, c moderates the exploration-exploitation trade-off, and λ serves as a penalty factor for the cost to enhance cost-efficiency.

Optimal Node Selection Modeling. Building on the defined node utility, the task of selecting an optimal chunk combination order, as outlined in Section 2, is reformulated as optimal node selection within the policy tree T . Given a budget constraint \mathcal{B} , the objective is to identify the node $v_i \subseteq T$ that maximizes the utility $U(v_i)$, while ensuring that the total cost associated with v_i does not exceed \mathcal{B} . Formally, this is represented as:

$$\hat{v}_i = \arg \max_{v_i \subseteq T} \left(\frac{V(v_i)}{N(v_i)} + c \sqrt{\frac{\ln N}{N(v_i)}} - \lambda \frac{\text{cost}(v_i)}{\mathcal{B}} \right) \quad (6)$$

where $V(v_i)$ is the estimated benefit of the chunk combination at node v_i , and $\text{cost}(v_i)$ represents its associated cost. This formulation enables selecting chunks that maximize utility within the given budget.

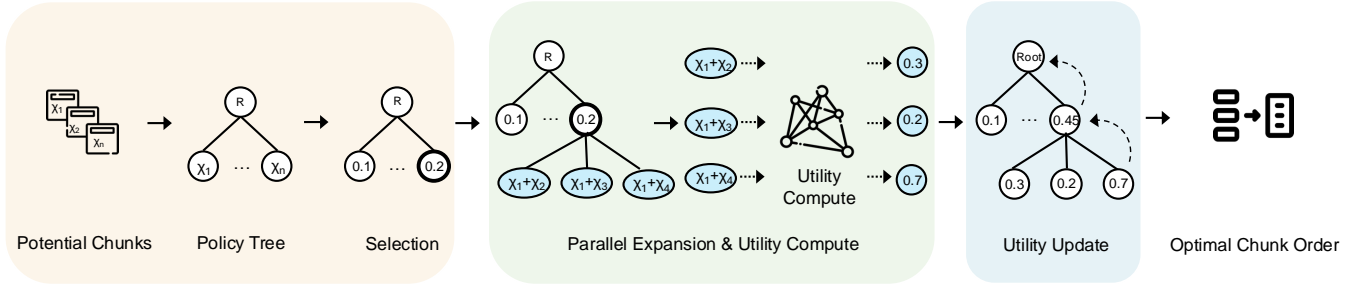


Figure 3: Workflow of MCTS Based Chunk Optimization

Algorithm 1 MCTS-Based Policy Tree Search

```

1: Input:  $q$ : a query;  $D$ : a set of document nodes;  $\mathcal{B}$ : total budget
2: Initialize root node  $v_0$  with query  $q$ 
3: while within computational budget do
4:    $v = \text{NodeSelection}(v_0, \mathcal{B})$ 
5:    $\text{UtilityUpdate}(v, \mathcal{B})$ 
6: end while
7: return the best node combination

```

4.2 MCTS-Based Policy Tree Search

Motivation. Enumerating all nodes in the policy tree will locate the optimal node but result in high computational costs. To address this, a straightforward approach is to apply the greedy strategy, navigating the tree iteratively from the root. In each iteration, the child node with the highest benefit is selected, continuing until the budget is exhausted. However, this method is likely to lead to suboptimal results. For example, the benefit of χ_1 may be slightly higher than χ_2 , but the benefit of $\chi_2 + \chi_3$ could greatly exceed that of $\chi_1 + \chi_3$. In this case, the greedy approach could lead to a suboptimal result. Therefore, it is essential to revisit high-benefit parent nodes. Meanwhile, we need to reduce the exploration of low-benefit nodes.

To achieve our objectives, we propose an MCTS-based policy tree search method designed to efficiently select and rank combinations of chunks. This approach explores the space of potential chunk orders iteratively, optimizing a given query within specified budget constraints.

Overview. The MCTS-based strategy is outlined in Algorithm 1. We begin by initializing the root node of the policy tree using the input query. While the computational budget is not exhausted, we iteratively perform two key steps: *Node Selection* and *Utility Update*. Once the iteration limit or budget is reached, we halt the process and conduct a recursive search within the tree to find the node with the highest utility. Unlike traditional MCTS strategies that often focus solely on the root node, our method also considers promising middle-layer nodes to maximize chunk combination utility.

Detailed Explanation of Key Functions. We further explain the two key functions as follows:

1. **Node Selection (Algorithm 2).** We recursively choose the node with the highest utility, which is most likely to lead to an optimal chunk combination. Specifically:

Algorithm 2 NodeSelection

```

1: function NODESELECTION( $v_0, \mathcal{B}$ )
2:   Input:  $v_i$ : a tree node of the policy tree;  $\mathcal{B}$ : remaining budget;  $R$ : predicted reranker
3:   while not fully expanded and within budget do
4:      $[V(v_1), V(v_2), \dots, V(v_k)] = R(\text{chunks}(v_i))$ 
5:     for all child nodes  $v_j$  of  $v_i$  do
6:        $\text{Utility}(v_j) = \frac{V(v_j)}{N(v_j)} + c\sqrt{\frac{\ln N}{N(v_j)}} - \lambda \frac{C(v_j)}{\mathcal{B}}$ 
7:     end for
8:     Select node  $v$  with the largest utility value
9:     if  $v$  is not fully expanded then
10:      Expand  $v$  by adding new nodes
11:     else
12:      NodeSelection( $v, \mathcal{B}$ )
13:     end if
14:   end while
15: return  $v$ 
16: end function

```

Algorithm 3 UtilityUpdate

```

1: function UTILITYUPDATE( $v, \mathcal{B}$ )
2:   Input:  $v$ : a tree node;  $\mathcal{B}$ : remaining budget
3:   Compute utility  $F(v)$ 
4:   Update  $v$ 's utility:  $w_i = w_i + F(v)$ 
5:   Update  $v$ 's visit count:  $n_i = n_i + 1$ 
6:   Propagate the utility and visit count updates up the tree
7: end function

```

- **Selection:** We identify the node v with the maximum utility value. If v has not been expanded yet, we generate all possible child nodes and incorporate them into the policy tree. If v is already expanded, we select the descendant node with the highest utility for further exploration.
- **Expansion:** Upon selecting the node v with the highest utility, we expand it by generating all potential child nodes. Each child represents a new possible chunk combination order. Our method employs parallel expansion, which computes and evaluates multiple child nodes simultaneously. This parallelism leverages the value network's ability to process multiple combinations at a

similar computational cost as a single node, enhancing search efficiency and breadth.

- **Computing Utility:** We compute the utility values for each new child node using a utility formula. The reranker model R processes multiple chunk combinations in parallel, yielding:

$$[V(v_1), V(v_2), \dots, V(v_k)] = R(v_1, v_2, \dots, v_k) \quad (7)$$

Here, R assesses the ordered chunk combinations, and we prioritize the node v_j with the highest utility for continued exploration. This approach balances utility and computational cost, accelerating convergence by broadening search space coverage.

- **Cost Estimation:** During node expansion, we estimate the token cost $C(v_j)$ of each selected node v_j using word length as a proxy. To manage computational expenses effectively, we ensure these cost estimates do not surpass the available budget. Final cost are verified through tokenization before selecting optimal nodes.

2. **Utility Update (Algorithm 3).** Once a node is expanded and its costs estimated, we perform the Utility Update as follows:

- **Utility Computation:** We calculate the utility $F(v)$ for the node v and update its cumulative utility w_i and visit count n_i .
- **Propagation:** These updates propagate upward through the tree, adjusting the utility of v and its parent nodes. This process ensures that the tree structure accurately reflects the most promising paths, guiding future search efforts.

5 CONFIGURATION AGENT

After addressing the efficient handling of chunk correlations within user budget, the remaining task is designing a system that adapts to each query’s domain. The MCTS process involves several critical configurations, including reranker selection, the number of iterations, the exploration coefficient, and the cost coefficient. Optimally setting these configurations is particularly challenging across diverse query types. To tackle this, we propose a configuration agent framework that predicts the optimal reranker and configuration for each query. In this section, we first present the agent framework in Section 5.1, followed by an overview of the model learning pipeline in Section 5.2.

5.1 Model Framework

Motivation. To address Challenge 3, which requires adapting to each query’s domain and recommending the optimal configuration, a straightforward solution would be to employ an MLP classifier to assign each query to its optimal reranker. However, preliminary experiments indicate that the performance of MLP classification is suboptimal. Upon further analysis, we observed that queries of similar types tend to share the same optimal reranker and configuration. Therefore, leveraging a Siamese Network with contrastive learning to bring queries of the same class closer while pushing those from different classes further apart is a more viable approach.

Figure 4 provides an overview of our configuration agent, which consists of two main modules responsible for transforming the input into feature maps. First, the input embedding module generates embeddings for the input queries. Subsequently, the encoding network processes these embeddings to produce feature maps, which are then utilized to derive various configurations for the MCTS setup.

The following sections will detail each component and explain the design rationale behind them.

(1) Input Query Embedding. To effectively capture the factors from various queries, given the diverse query types such as explicit facts, implicit facts, interpretable rationales, and hidden rationales [47], we employed the BGE-M3 [6] embedding model to generate embeddings for each query. These embeddings enhance the capacity of the learning framework by mapping similar query types to the same reranker class. Represented in a 1024-dimensional space, the embeddings capture essential semantic features, enabling efficient comparison and classification by the encoding network. This step helps improve retrieval relevance across different query types. Additionally, optimal reranker annotation embeddings are also generated using the same embedding model, including its unique features and associated metadata, allowing the model to align queries with the optimal reranker.

(2) Feature Map Generation with Encoding Network. In order to optimize the reranker selection task and recommend the optimal reranker for each query across different query types, we utilize a encoding network to efficiently learn representations that are useful for both classification and configuration prediction. We use a Siamese network consists of three fully connected layers to accomplish this. It processes input query embeddings of dimension $d = 1024$ and learns both a classification output and MCTS configuration predictions (i.e. iterations, and λ). The encoding network’s branches share weights, each applying linear transformations followed by RELU activations. Sequentially, the first hidden layer reduces the dimension to 512, the second to 256, and the third to 128. The final output layer provides a classification prediction specifying the optimal reranker for each query and a regression output for predicting the most effective MCTS configurations, guiding the search process. The classification output identifies the best reranker for each query, while the regression output determines the optimal MCTS configuration settings.

5.2 Joint Training

In this section, we outline the training pipeline for developing the configuration agent. As depicted in Figure 4, we implement three joint training tasks to enhance the model’s learning efficiency. The first two tasks involve classification and regression to respectively select the optimal reranker and predict the best values for MCTS hyperparameters. Additionally, we incorporate a contrastive learning approach to further refine the learning process.

5.2.1 Classification and regression losses. Given the predicted reranker label y_{pred} and its corresponding actual optimal reranker y_{true} , the classification loss L_{cla} is computed as follows:

$$L_{\text{cla}}(\theta) = F_{\text{cla}}(y_{\text{pred}}, y_{\text{true}}) \quad (8)$$

where F_{cla} represents the cross-entropy loss between the predicted and actual reranker labels. This loss function aids in accurately classifying the optimal reranker for each query. Similarly, the regression loss L_{reg} is defined as:

$$L_{\text{reg}}(\theta) = F_{\text{reg}}(p_{\text{pred}}, p_{\text{true}}) \quad (9)$$

where F_{reg} is the mean squared error (MSE) between the predicted and actual MCTS parameters p_{pred} and p_{true} . This metric ensures

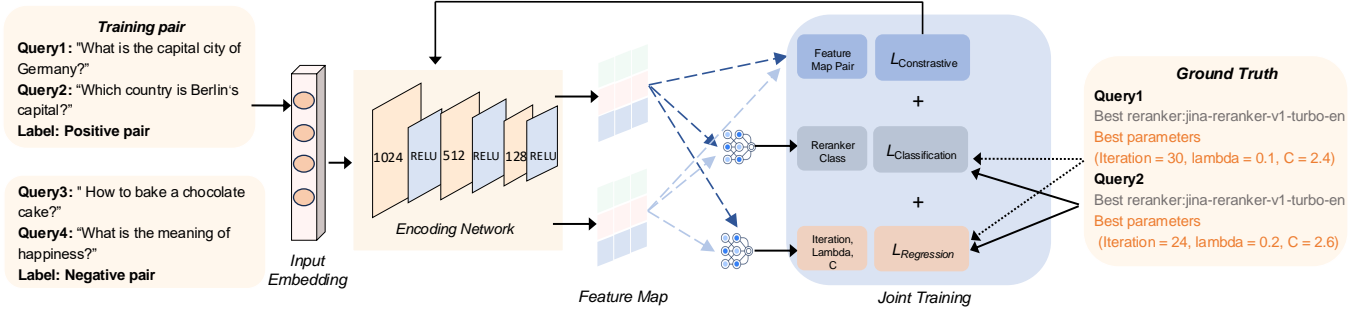


Figure 4: Overview of Configuration Agent

the precise prediction of MCTS configurations, including the iteration count and λ .

5.2.2 Contrastive Learning. To efficiently distinguish between different query domains and recommend the optimal configuration for each query, we utilize contrastive learning to bring queries of the same domain closer together while pushing apart embeddings from different reranker classes.

Contrastive pairs preparation. To prepare the training dataset, we must identify the optimal reranker and configuration for each query. In this study, the most suitable reranker and corresponding configurations for each query are determined through extensive experimentation with various setups. Subsequently, query pairs are generated based on these optimal reranker annotations. Positive pairs are formed from queries that share the same optimal reranker, promoting minimal distance between their embeddings in the feature space. Conversely, negative pairs are composed of queries with different rerankers, where the goal is to maximize the distance between their embeddings. Since some rerankers perform similarly on certain queries, we select only cases with a ROUGE-L difference exceeding 10% to form our training dataset.

Contrastive loss. As illustrated in Figure 4, for a given positive pair (x_i, x_i^+) and a negative pair (x_j, x_j^-) , we first generate their corresponding feature maps with the encoding model. These feature maps are then utilized to compute the contrastive loss L_{con} . In particular, this process can be formulated as follows:

$$L_{con}(\theta) = F_{con}(f_{\theta}(x_i), f_{\theta}(x_i^+)) + F_{con}(f_{\theta}(x_j), f_{\theta}(x_j^-)) \quad (10)$$

where $f_{\theta}(x)$ represents the embedding function, and F_{con} is the similarity function applied to both types of pairs: positive pairs (with similar rerankers) and negative pairs (with different rerankers). This loss function is designed to ensure that queries with the same reranker are brought closer together in the embedding space, while those with different rerankers are distanced.

5.2.3 Whole training process. Finally, the total loss function L_{total} is the combination of the contrastive, classification, and regression losses as follows:

$$L_{total}(\theta) = L_{con}(\theta) + L_{cla}(\theta) + L_{reg}(\theta) \quad (11)$$

In particular, the contrastive loss $L_{con}(\theta)$ encourages the embeddings of queries with the same optimal reranker to be close together, while pushing apart the embeddings of queries with different rerankers. The classification loss $L_{cla}(\theta)$ aids the model in correctly identifying the reranker using cross-entropy, and the regression loss $L_{reg}(\theta)$ minimizes the error in predicting the optimal MCTS configuration.

Remark. Once the total loss L_{total} is calculated, the network parameters θ are updated using gradient descent with a learning rate η . This optimization process is repeated across multiple epochs E and batches, ensuring that both reranker selection and parameter prediction are improved over time.

6 EXPERIMENTS

The experimental study intends to answer the following questions:

- **RQ1** How effective is our *CORAG* for the cost-constrained RAG pipeline compared to other methods?
- **RQ2** How efficient is *CORAG* with varying chunk sizes?
- **RQ3** What are the bottlenecks of the current RAG?
- **RQ4** How scalable is *CORAG* with varying dataset sizes?
- **RQ5** What is the effectiveness of each design in *CORAG*?

6.1 Experiment Setting

Environment. We integrate our system with the popular RAG framework LlamaIndex¹. The experiments are run on a Linux server with an Intel Core i7-13700K CPU (12 cores, 24 threads, 5.3 GHz), 64GB RAM, and a 1 TiB NVMe SSD. The configuration agent module is implemented in PyTorch 2.0 and trained on an NVIDIA RTX 4090 GPU with 24GB VRAM.

Table 1: Statistics of datasets used in the experiments.

Dataset	#train	#dev	#test	#p
MSMARCO	502,939	6,980	6,837	8,841,823
Wiki	3,332	417	416	244,136

¹<https://www.llamaindex.ai/>

Datasets. To evaluate the performance of *CORAG* across diverse scenarios, we conduct experiments on two distinct datasets with differing task focuses: (1) *WikiPassageQA*[7] is a question-answering benchmark containing 4,165 questions and over 100,000 text chunks, aimed at evaluating passage-level retrieval. (2) *MARCO*[24] is a comprehensive dataset tailored for natural language processing tasks, primarily emphasizing question answering and information retrieval. As shown in Table 1, both *WikiPassageQA* and *MARCO* provide high-quality question and passage annotations, making them suitable benchmarks for assessing retrieval effectiveness. In our experiments, we prompt LLMs to generate ground truth answers for each dataset. For instance, if we use Llama3 to evaluate *CORAG*’s performance, we also prompt Llama3 to generate the ground truth in the same experimental setting for fairness and alignment with the features of the LLMs.

Baselines. We compare the performance of *CORAG* with two typical RAG baselines:

- **RAPTOR**[29]: RAPTOR constructs a hierarchical document summary tree by recursively embedding, clustering, and summarizing text chunks, enabling multi-level abstraction. This approach aligns with the clustering-based methods discussed in Section 1. We finish the tree construction within the limit of budget.
- **NaiveRAG**: This is a basic method for retrieving relevant chunks. First, candidate chunks are retrieved from the vector database based on vector similarity search, followed by ranking them using a reranker model. This approach is the type of AKNN method mentioned in Section 1. To meet the cost constraint, we employ the greedy budget allocation strategy, retrieving chunks until the budget is fully exhausted.

In addition, we remove the configuration agent from our method as a baseline to evaluate its impact on the performance of *CORAG*, referring to this version as *CORAG w/o Agent*. At last, we implement a method called *CORAG Upper* to establish an upper bound by exploring all possible chunk combinations and selecting the optimal order. Due to the large number of potential combinations, we limit the exploration to combinations with fewer than six chunks in *CORAG Upper* case.

Remark. Other methods, such as GraphRAG [22], depend significantly on frequent invocations of LLMs to summarize chunks and construct indexes, incurring substantial costs (e.g., billions of tokens) that exceed our strict cost constraints. Consequently, these methods are not feasible for addressing our problem. For a fair comparison, we exclude these types of RAG methods in the experiment.

Hyper-parameter Settings: The hyper-parameters for *CORAG* are automatically determined by the configuration agent, while *NaiveRAG* does not require any hyper-parameters. For other baseline methods, we ensure consistency by using identical hyper-parameters for fair comparisons. Specifically, we set the exploration coefficient to 2.4, the number of iterations to 10, and the cost coefficient λ to 0.1. Preliminary experiments indicate that this configuration optimizes baseline performance. Further ablation studies will also follow to validate these settings.

Learning Parameter Setting: In our method, the configuration agent is trained using contrastive learning. The hyper-parameters

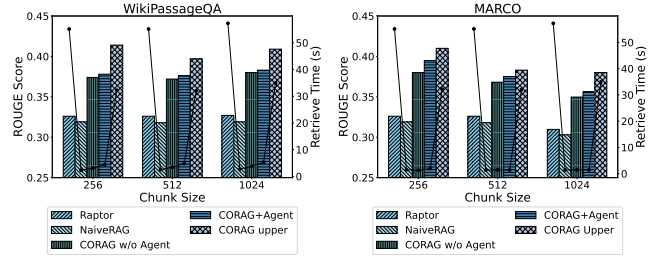


Figure 5: Efficiency Comparison

used during this process include a margin for contrastive loss (margin=1.0), learning rate (lr=0.001), batch size (32), number of epochs (num_epochs=60), and the embedding model (i.e., BAAI/bge-m3[6]).

Evaluation Metrics. We assess effectiveness by comparing the Rouge scores between the ground truth answers and the generated responses, using Rouge-1, Rouge-2, and Rouge-L as evaluation metrics. To evaluate efficiency, we measure the latency required to answer a query using different methods.

6.2 Performance Comparison

6.2.1 RQ1: ROUGE Comparison. As shown in Table 2, we compare *CORAG* with several baselines across different datasets, primarily using *WikiPassageQA* and *MARCO*. The evaluations are conducted on three different chunk sizes, utilizing ROUGE-1, ROUGE-2, and ROUGE-L metrics to assess the improvements in responses generated by the LLM due to our retrieval method. *CORAG* demonstrate a substantial improvement of approximately 25% compared to mainstream RAG approaches such as *NaiveRAG* and *RAPTOR*. As expected, *CORAG* does not exceed the upper bound, which represents an extreme scenario where all possible combination orders are exhaustively enumerated, which is clearly inefficient and impractical. In summary, *CORAG* outperforms baselines, enhancing retrieval relevancy while pruning the search space effectively.

6.2.2 RQ2: Efficiency Evaluation. As shown in Figure 5, since *CORAG* is based on the tree search algorithm, the agent assists in predicting the optimal reranker and parameters for a given query. Therefore, it is crucial to evaluate the impact of different chunk sizes and datasets on retrieval optimization task efficiency. We tested efficiency using various datasets and chunk sizes, observing that *NaiveRAG*, which uses a traditional retrieval approach, achieved shorter retrieval times but lower ROUGE scores. *CORAG upper* performs well in terms of ROUGE, but its efficiency is significantly reduced due to exploring the entire search space. Similarly, *RAPTOR*, which leverages an external LLM for summarization, exhibited poor efficiency. In contrast, our *CORAG* approach strikes a balance between efficiency and retrieval relevance, achieving an effective trade-off.

6.2.3 RQ3: Performance Breakdown. We present a performance breakdown of our baseline *NaiveRAG*, to highlight the bottlenecks in the current RAG system. To address the challenge of searching for the optimal chunk combination order, implementing it with *NaiveRAG* requires the following steps: (a) obtaining the query embedding, (b) retrieving potential chunk combinations, (c) reranking

Table 2: ROUGE Comparison on WikiPassage QA and MARCO Datasets

Method	LLM Type	WikiPassage QA									MARCO								
		256			512			1024			256			512			1024		
		R1	R2	RL	R1	R2	RL	R1	R2	RL	R1	R2	RL	R1	R2	RL	R1	R2	RL
Raptor	Llama3-8B	0.338	0.154	0.316	0.322	0.147	0.301	0.335	0.159	0.305	0.386	0.208	0.356	0.393	0.213	0.366	0.338	0.154	0.316
NaiveRAG	Llama3-8B	0.337	0.149	0.312	0.321	0.142	0.297	0.334	0.158	0.309	0.398	0.203	0.369	0.395	0.213	0.368	0.337	0.149	0.312
CORAG upper	Llama3-8B	0.447	0.275	0.426	0.426	0.262	0.406	0.444	0.273	0.423	0.435	0.235	0.414	0.425	0.229	0.397	0.447	0.275	0.426
CORAG w/o Agent	Llama3-8B	0.390	0.212	0.364	0.372	0.202	0.347	0.388	0.221	0.362	0.401	0.212	0.374	0.393	0.216	0.372	0.390	0.212	0.364
CORAG	Llama3-8B	0.423	0.223	0.392	0.403	0.212	0.373	0.409	0.219	0.378	0.413	0.224	0.382	0.405	0.219	0.376	0.411	0.219	0.380
CORAG	Mixtral8*7B	0.357	0.158	0.325	0.382	0.167	0.351	0.401	0.198	0.367	0.408	0.199	0.378	0.399	0.194	0.369	0.403	0.193	0.373
CORAG	Phi-2 2.7B	0.351	0.137	0.317	0.318	0.117	0.298	0.308	0.108	0.288	0.335	0.109	0.305	0.325	0.103	0.301	0.333	0.108	0.303

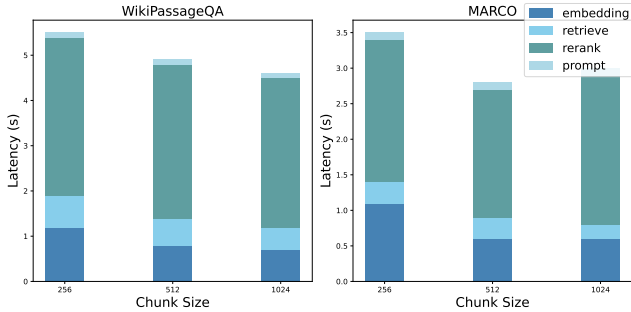


Figure 6: Performance Breakdown

the potential chunk combinations, and (d) prompt refining. The average latency for each of these steps is reported in Figure 6, under the same experimental settings as before.

The results reveal that the baseline suffers significantly from the reranking process. In some cases, particularly with smaller chunk sizes, the reranking process contributes more than half of the overall latency. The slow reranking process is due to two key reasons: (1) The large number of potential chunk combination orders. For example, as we simultaneously consider both the chunk order and the combination, given 50 potential chunks, there can be over 50! possible chunk combination orders. (2) The reranker directly processes raw information using transformer models[28], which minimizes information loss but produces a substantial time penalty. When the reranker is provided with a huge number of chunk combination orders, the efficiency might perform worse significantly.

Therefore, designing a system that can efficiently utilize the reranker while searching for the optimal chunk combination order is crucial for improving RAG performance.

6.2.4 RQ4: Scalability Evaluation. CORAG demonstrates exceptional scalability, particularly when dealing with large datasets such as WikiPassageQA and MARCO, which contain a high volume of passages. By segmenting each passage into 256-token chunks, the chunk count can easily scale to 100k or more. Despite this substantial increase in data volume, our retrieval time only increased by 10% compared to traditional methods, showcasing our system’s ability to manage large-scale retrieval tasks efficiently. Notably, our system outperforms the *CORAG upper* approach, requiring just one-tenth of the retrieval time, while still delivering competitive

ROUGE scores. This efficient balance between performance and computational overhead highlights the system’s capacity to prune the search space effectively, ensuring fast retrieval even in expansive datasets. As a result, our approach is well-suited for scenarios where both large-scale data processing and high retrieval accuracy are crucial.

Table 3: Performance comparison with varying budgets

Budget	Methods	R1	R2	RL	Cost
1024	Raptor	0.338	0.154	0.316	1024
	NaiveRAG	0.337	0.149	0.312	1024
	CORAG w/o Agent	0.390	0.212	0.364	831
	CORAG	0.411	0.219	0.380	769
	CORAG upper	0.447	0.275	0.426	800
2048	Raptor	0.339	0.162	0.318	2048
	NaiveRAG	0.338	0.157	0.314	2048
	CORAG w/o Agent	0.392	0.223	0.366	921
	CORAG	0.425	0.235	0.395	845
	CORAG upper	0.449	0.289	0.429	923
8192	Raptor	0.341	0.171	0.321	8192
	NaiveRAG	0.341	0.165	0.316	8192
	CORAG w/o Agent	0.394	0.235	0.369	901
	CORAG	0.427	0.249	0.398	911
	CORAG upper	0.451	0.305	0.432	923

6.3 RQ5: Ablation Study

6.3.1 Ablation study for different Budget. As shown in Table 3, we evaluate CORAG as a cost-constrained system, examining the impact of varying budgets on overall performance. Using the MARCO dataset, we set budget limits at 1024, 2048, and 8192 tokens and evaluated the results with ROUGE. CORAG consistently outperforms all baselines across these budget levels. Notably, CORAG’s average token cost remains below each budget limit, indicating that chunk utility is not monotonic, as highlighted in Challenge 2. With a higher budget, CORAG benefits from an expanded search space, enabling the inclusion of more relevant information without merely increasing the number of chunks.

6.3.2 Ablation Study for Different Exploration Coefficients. As illustrated in Figure 7, we perform an ablation study to evaluate the impact of varying exploration coefficients on system performance, specifically testing the C values of 0, 1, 2, and 3. The results indicate that an exploration coefficient of around 2 provides the best

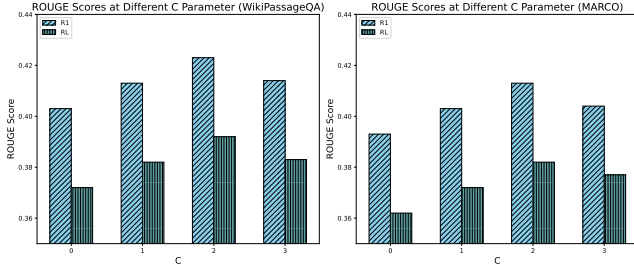


Figure 7: ROUGE Comparison between different C

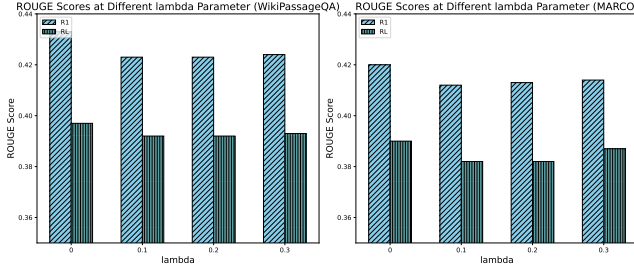


Figure 8: ROUGE Comparison between different lambda

performance, striking an optimal balance between exploration and exploitation within the search process. This balance enables the system to effectively uncover relevant information while maintaining a focus on high-potential chunks, ultimately leading to improved RAG response. In contrast, both lower and higher exploration coefficients led to suboptimal results, either due to insufficient exploration or excessive diffusion of focus. These findings emphasize the critical role of the exploration coefficient in the performance of the *CORAG* search process and highlight the importance of careful parameter tuning.

Table 4: Performance comparison with varying rerankers

ChunkSize	Reranker	R1	R2	RL
256	v1-turbo	0.412	0.216	0.379
	v2-base-multi	0.413	0.221	0.380
	bge-m3	0.425	0.230	0.395
	bge-large	0.431	0.238	0.401
	bge-base	0.421	0.232	0.390
	gte-base	0.424	0.232	0.395
512	v1-turbo	0.366	0.173	0.333
	v2-base-multi	0.367	0.177	0.334
	bge-m3	0.368	0.177	0.336
	bge-large	0.362	0.177	0.332
	bge-base	0.375	0.185	0.344
	gte-base	0.364	0.181	0.335
1024	v1-turbo	0.269	0.093	0.240
	v2-base-multi	0.270	0.094	0.243
	bge-m3	0.270	0.094	0.243
	bge-large	0.265	0.092	0.236
	bge-base	0.265	0.092	0.237
	gte-base	0.270	0.094	0.242

6.3.3 Ablation Study for Different Cost Coefficients. As illustrated in Figure 8, we perform an ablation study to evaluate the impact of varying cost coefficients on system performance, specifically testing values of 0, 0.1, 0.2, and 0.3. The results show that introducing the cost coefficient in the utility led to a slight decrease in ROUGE scores. This decrease occurs because, without cost constraints, *CORAG* tends to produce longer outputs, albeit at the expense of cost efficiency. However, despite the slight reduction in ROUGE scores, the decline remains within 5%, which is acceptable. These results highlight the importance of tuning the cost coefficient effectively to balance output richness and cost constraints, further emphasizing the role of our configuration agent in enabling efficient configuration tuning for optimal *CORAG* performance.

6.3.4 Ablation Study on Different Rerankers. To evaluate the impact of different rerankers on retrieval performance, we conduct an ablation study using six widely recognized reranker models: *jina-reranker-v1-turbo-en*, *jina-reranker-v2-base-multilingual*, *bge-reranker-v2-m3*, *bge-reranker-large*, *bge-reranker-base*, and *gte-multilingual-reranker-base*. These rerankers are evaluated on the MARCO dataset with the llama3-8B model, configured with a fixed cost coefficient of 0.1, an exploration coefficient of 2.4, and a budget limit of 1024.

The results in Table 4 reveal variations in performance across different rerankers, highlighting the importance of careful reranker selection to optimize RAG system performance under specific operational constraints. Among the rerankers, *gte-multilingual-reranker-base* and *bge-reranker-large* demonstrate consistently strong performance on QA tasks, suggesting that these reranker models have high efficacy in capturing relevant information across different QA queries. We observe that as the chunk size increases in the ablation study, each individual reranker yields lower performance than agent recommended reranker towards different queries. This indicates that the configuration agent effectively leverages reranker diversity, dynamically adjusting configurations to improve retrieval results. The configuration agent’s ability to recommend a better configuration for the reranker selection and parameters adaptively underscores its importance in maximizing RAG system performance, particularly under constraints such as a limited budget.

6.4 Case Study

Figure 9 presents three examples to illustrate the retrieval quality comparison between *CORAG* and the traditional NaiveRAG method, focusing on why our approach outperforms baseline methods. Due to its straightforward top-*k* retrieval and reranking, NaiveRAG often misses essential information relevant to the query’s intent, as it frequently retrieves chunks based on keyword matching rather than relevance to the query. For instance, with the query “*Is bougainvillea a shrub?*”, NaiveRAG retrieves content containing matching keywords but fails to provide the actual classification of bougainvillea. In contrast, *CORAG*’s chunk combination strategy retrieves context that includes bougainvillea’s category, enabling the LLM to give a more accurate response. In another case, NaiveRAG retrieves terms and legal clauses containing “*oxyfluorfen*” but lacks understanding of the query’s intent, while *CORAG* provides context linking oxyfluorfen to its use case in cotton, which requires logical relationships between chunks that NaiveRAG’s vector similarity search

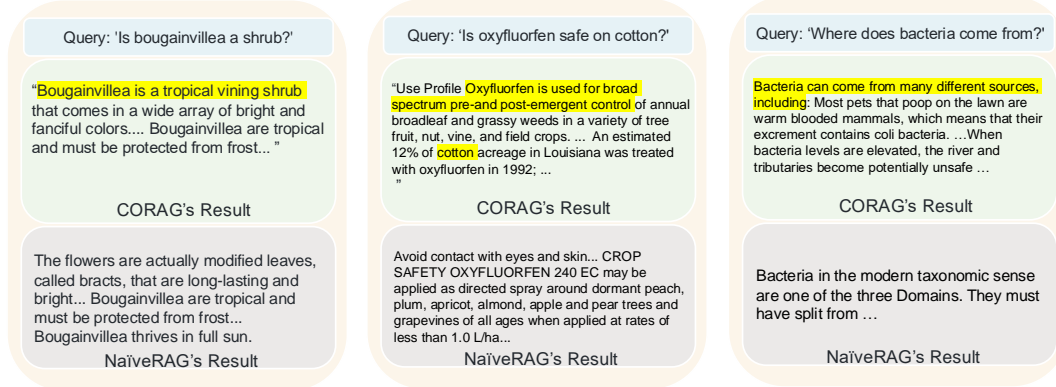


Figure 9: Case Study for CORAG

cannot capture. Finally, for the query “Where does bacteria come from?”, NaïveRAG retrieves chunks with the keyword ‘bacteria’ but does not address its origin, whereas CORAG supplies a more complete response, including sources of bacteria and conditions for their proliferation. These cases illustrate that CORAG excels in retrieving logically connected information, making it more effective than NaïveRAG for queries requiring more than simple keyword matching.

7 INSIGHTS AND FUTURE DESIGN CHOICES ON RAG

7.1 Shortcomings of Current RAG

We provide an analysis of current RAG systems revealing performance challenges across the Retrieve (S1), Augment (S2), and Generation (S3) phases.

S1: Retrieval Overhead Current RAG systems often utilize LLMs for summarization and indexing structures, overlooking the high computational costs associated with external LLMs, thus escalating compute expenses. Model-based rerankers, while improving relevance during retrieval, introduce notable latency, which can impede efficiency in latency-sensitive contexts. Cost-effective index construction and reranking optimization are essential to balance efficiency and performance.

S2: Augmentation Overhead Post-retrieval techniques, such as optimized chunk combination ordering, enhance context relevancy but demand additional computation. Pruning strategies that minimize the search space and refine combination order are critical for balancing computational cost and augmented context relevancy. Efficient chunk combination optimization, emphasizing order and coherence, is vital for reducing costs and enhancing retrieval performance.

S3: Generation Overhead Effective prompt engineering for optimal chunk combinations requires significant computational resources. Query-specific prompt refinement and compression are crucial to reduce overhead while maintaining input relevance and conciseness. Adaptive strategies that handle diverse query types and domain-specific requirements ensure prompt efficiency without compromising output quality.

7.2 Design Choices

To address the challenges identified, the following design choices aim to optimize the performance of RAG systems:

P1: Co-Design of Retrieval and Reranking Processes In CORAG, parallel expansion in tree search accelerates query processing by enabling concurrent retrieval and reranking, significantly reducing latency. Future optimizations could address bottlenecks by eliminating stage-specific delays, further enhancing ranking efficiency. This co-design approach efficiently manages chunk combination order, improving the ranking process and relevance scoring.

P2: Optimization of Tree Structure and Search Iterations Results indicate that shorter policy tree height enhances search efficiency by reducing computational overhead, especially advantageous for large datasets. Minimizing tree height in tree-based searches improves the search speed for contextually relevant chunks, significantly lowering latency and computational costs. This optimization approach enhances RAG system performance across extensive datasets.

P3: Dynamic Prompt Engineering Selecting rerankers based on query type and using adaptable prompt templates improve retrieval relevance for LLMs. Dynamic prompt structures that align with query intent and domain-specific contexts maintain output quality within resource constraints. This adaptive approach to prompt engineering achieves an effective balance between efficiency and retrieval quality, addressing the dynamic nature of RAG system queries.

8 CONCLUSION

Considering the non-monotonicity of the chunk’s utility, correlation between the chunks, and the diversity of the different query domain, we propose a learning-based retrieval optimization system CORAG. Initially, we model the chunk combination orders as a policy tree and employ MCTS to explore this policy tree, aiming to identify the optimal chunk combination order. We introduce a configuration agent that accurately predicts the optimal configuration and reranker for the given query. Additionally, we also design a parallel query expansion strategy to expand multiple nodes in each iteration. Experimental results demonstrate that our method significantly outperforms state-of-the-art methods within constrained cost limits and also shows notable efficiency.

REFERENCES

- [1] Mohammad Alkhalaf, Ping Yu, Mengyang Yin, and Chao Deng. 2024. Applying generative AI with retrieval augmented generation to summarize and extract key clinical information from electronic health records. (2024), 104662.
- [2] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. (2023).
- [3] Peter Auer. 2002. Using confidence bounds for exploitation-exploration trade-offs. 3, Nov (2002), 397–422.
- [4] Tom B Brown. 2020. Language models are few-shot learners. (2020).
- [5] Chi-Min Chan, Chunpu Xu, Ruibin Yuan, Hongyin Luo, Wei Xue, Yike Guo, and Jie Fu. 2024. Rq-rag: Learning to refine queries for retrieval augmented generation. (2024).
- [6] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation. arXiv:2402.03216 [cs.CL]
- [7] Daniel Cohen, Liu Yang, and W. Bruce Croft. 2018. WikiPassageQA: A Benchmark Collection for Research on Non-factoid Answer Passage Retrieval. abs/1805.03797 (2018). arXiv:1805.03797
- [8] Florin Cuconasu, Giovanni Trappolini, Federico Siciliano, Simone Filice, Cesare Campagnano, Yoelle Maarek, Nicola Tonellotto, and Fabrizio Silvestri. 2024. The Power of Noise: Redefining Retrieval for RAG Systems. 719–729.
- [9] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. 209–218.
- [10] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2023. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. arXiv:2311.05232 [cs.CL]
- [11] Ziyang Jiang, Xueguang Ma, and Wenhui Chen. 2024. LongRAG: Enhancing Retrieval-Augmented Generation with Long-context LLMs. arXiv:2406.15319 [cs.CL]
- [12] Ming Jin, Shiyu Wang, Lintao Ma, Zhixuan Chu, James Y Zhang, Xiaoming Shi, Pin-Yu Chen, Yuxuan Liang, Yuan-Fang Li, Shirui Pan, et al. 2023. Time-llm: Time series forecasting by reprogramming large language models. (2023).
- [13] Angeliki Lazaridou, Elena Gribovskaya, Wojciech Stokowiec, and Nikolai Grigorev. 2022. Internet-augmented language models through few-shot prompting for open-domain question answering. (2022).
- [14] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. 33 (2020), 9459–9474.
- [15] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. 2015. A diversity-promoting objective function for neural conversation models. (2015).
- [16] Xinze Li, Zhenghao Liu, Chenyan Xiong, Shi Yu, Yu Gu, Zhiyuan Liu, and Ge Yu. 2023. Structure-Aware Language Model Pretraining Improves Dense Retrieval on Structured Data. (2023).
- [17] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. 2024. Optimizing llm queries in relational workloads. (2024).
- [18] Yanming Liu, Xinyue Peng, Xuhong Zhang, Weihao Liu, Jianwei Yin, Jiannan Cao, and Tianyu Du. 2024. RA-ISF: Learning to Answer and Understand from Retrieval Augmentation via Iterative Self-Feedback. (2024).
- [19] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2021. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. (2021).
- [20] Yuanjie Lyu, Zhiyu Li, Simin Niu, Feiyu Xiong, Bo Tang, Wenjin Wang, Hao Wu, Huanyong Liu, Tong Xu, and Enhong Chen. 2024. Crud-rag: A comprehensive chinese benchmark for retrieval-augmented generation of large language models. (2024).
- [21] Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting for retrieval-augmented large language models. (2023).
- [22] Microsoft. 2024. GraphRAG. <https://microsoft.github.io/graphrag/>
- [23] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. (2021).
- [24] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. MS MARCO: A Human Generated Machine Reading Comprehension Dataset. abs/1611.09268 (2016). arXiv:1611.09268
- [25] Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Le Yan, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, et al. 2023. Large language models are effective text rankers with pairwise ranking prompting. (2023).
- [26] Chidakh Ravuru, Sagar Srinivas Sakhinana, and Venkataramana Runkana. 2024. Agentic Retrieval-Augmented Generation for Time Series Analysis. arXiv:2408.14484 [cs.AI]
- [27] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. 2004. Simple BM25 extension to multiple weighted fields. 42–49.
- [28] Devendra Singh Sachan, Mike Lewis, Mandar Joshi, Armen Aghajanyan, Wen-tau Yih, Joelle Pineau, and Luke Zettlemoyer. 2023. Improving Passage Retrieval with Zero-Shot Question Generation. arXiv:2204.07496 [cs.CL]
- [29] Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. 2024. RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval.
- [30] Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. 2023. Replug: Retrieval-augmented black-box language models. (2023).
- [31] Devendra Singh, Siva Reddy, Will Hamilton, Chris Dyer, and Dani Yogatama. 2021. End-to-end training of multi-document reader and retriever for open-domain question answering. 34 (2021), 25968–25981.
- [32] Karan Singhal, Shekoofeh Azizi, Tao Tu, S. Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, Perry Payne, Martin Seneviratne, Paul Gamble, Chris Kelly, Nathaneal Scharli, Aakanksha Chowdhery, Philip Mansfield, Blaise Agüera y Arcas, Dale Webster, Greg S. Corrado, Yossi Matias, Katherine Chou, Juraj Gottweis, Nenad Tomasev, Yun Liu, Alvin Rajkomar, Joelle Barral, Christopher Semturs, Alan Karthikesalingam, and Vivek Natarajan. 2022. Large Language Models Encode Clinical Knowledge. arXiv:2212.13138 [cs.CL]
- [33] Manveer Singh Tamber, Ronak Pradeep, and Jimmy Lin. 2023. Scaling Down, LiT-ing Up: Efficient Zero-Shot Listwise Reranking with Seq2seq Encoder-Decoder Models. (2023).
- [34] Andrew Trotman, Antti Puurula, and Blake Burgess. 2014. Improvements to BM25 and language models examined. 58–65.
- [35] Shutong Wang, Xin Xu, Mang Wang, Weipeng Chen, Yutao Zhu, and Zhicheng Dou. 2024. RichRAG: Crafting Rich Responses for Multi-faceted Queries in Retrieval-Augmented Generation. (2024).
- [36] Zheng Wang, Bingzheng Gan, and Wei Shi. 2024. Multimodal Query Suggestion with Multi-Agebm25g from Human Feedback. arXiv:2402.04867 [cs.IR]
- [37] Jeff Wu, Long Ouyang, Daniel M Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Christiano. 2021. Recursively summarizing books with human feedback. (2021).
- [38] Ziwei Xu, Sanjay Jain, and Mohan Kankanahalli. 2024. Hallucination is Inevitable: An Innate Limitation of Large Language Models. arXiv:2401.11817 [cs.CL]
- [39] Siqiao Xue, Danrui Qi, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, Wang Zhao, Fan Zhou, Hong Yi, Shaodong Liu, Hongjun Yang, and Faqiang Chen. 2024. Demonstration of DB-GPT: Next Generation Data Interaction System Empowered by Large Language Models. arXiv:2404.10209 [cs.AI]
- [40] Jiexia Ye, Weiqi Zhang, Ke Yi, Yongzi Yu, Ziyue Li, Jia Li, and Fuguee Tsung. 2024. A Survey of Time Series Foundation Models: Generalizing Time Series Representation with Large Language Model. arXiv:2405.02358 [cs.LG]
- [41] Ziqi Yin, Shanshan Feng, Shang Liu, Gao Cong, Yew Soon Ong, and Bin Cui. 2024. LIST: Learning to Index Spatio-Textual Data for Embedding based Spatial Keyword Queries. (2024).
- [42] Tan Yu, Anbang Xu, and Rama Akkiraju. 2024. In Defense of RAG in the Era of Long-Context Language Models. arXiv:2409.01666 [cs.CL]
- [43] Yue Yu, Wei Ping, Zihan Liu, Boxin Wang, Jiaxuan You, Chao Zhang, Mohammad Shoeybi, and Bryan Catanzaro. 2024. RankRAG: Unifying Context Ranking with Retrieval-Augmented Generation in LLMs. (2024).
- [44] Hamed Zamani and Michael Bendersky. 2024. Stochastic RAG: End-to-End Retrieval-Augmented Generation through Expected Utility Maximization. arXiv:2405.02816 [cs.CL]
- [45] Hailin Zhang, Yujing Wang, Qi Chen, Ruiheng Chang, Ting Zhang, Ziming Miao, Yingyan Hou, Yang Ding, Xupeng Miao, Haonan Wang, et al. 2024. Model-enhanced vector index. 36 (2024).
- [46] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, and Bin Cui. 2024. Retrieval-augmented generation for ai-generated content: A survey. (2024).
- [47] Siyun Zhao, Yuqing Yang, Zilong Wang, Zhiyuan He, Luna K. Qiu, and Lili Qiu. 2024. Retrieval Augmented Generation (RAG) and Beyond: A Comprehensive Survey on How to Make your LLMs use External Data More Wisely. arXiv:2409.14924 [cs.CL]
- [48] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2024. Chat2Data: An Interactive Data Analysis System with RAG, Vector Databases and LLMs. (2024).
- [49] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A learned query rewrite system using monte carlo tree search. 15, 1 (2021), 46–58.
- [50] Honglei Zhuang, Zhen Qin, Rolf Jagerman, Kai Hui, Ji Ma, Jing Lu, Jianmo Ni, Xuanhui Wang, and Michael Bendersky. 2022. RankT5: Fine-Tuning T5 for Text Ranking with Ranking Losses. arXiv:2210.10634 [cs.IR]
- [51] Shengyao Zhuang, Bing Liu, Bevan Koopman, and Guido Zuccon. 2023. Open-source large language models are strong zero-shot query likelihood models for document ranking. (2023).