IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

A Survey Of Vector-Indexed And Graph-Based Approaches To Retrieval-Augmented Generation (RAG)

[1]Aryan Dhalpe, [2]Dr. Vina M. Lomte, [3]Piyush Savale, [4]Aditya Diwakar, [5]Gaurav Kantak

[1]Student, [2]Head of Department, [3]Student, [4]Student,[5]Student,

[1-5]Department of Computer Engineering, RMD Sinhgad School of Engineering, Warje, Pune–411058, India

Abstract:

This survey explores the various vector-indexed and graph-based approaches employed in Retrieval-Augmented Generation (RAG) systems, a paradigm that enhances large language models (LLMs) by incorporating external knowledge through retrieval mechanisms. RAG systems enable LLMs to generate more accurate, contextually aware, and informative responses by integrating external documents or structured data during the generation process. We delve into two main techniques: vector-indexed retrieval, which leverages high-dimensional vector embeddings and methods like Hierarchical Navigable Small World Graphs (HNSW) for efficient approximate nearest neighbor search, and graph-based retrieval, which utilizes knowledge graphs to store and query relational data. We examine the key stages involved in these approaches, including data preprocessing, embedding generation, search algorithms, and integration with LLMs. Furthermore, we address the challenges of handling large-scale data and unstructured information. The paper provides a comprehensive overview of the current methods, their strengths, and the limitations of RAG.

Index Terms – GenAI , Retrieval-Augmented Generation (RAG) , Large Language Model (LLM) , Graph-based RAG.

Introduction

Retrieval-Augmented Generation (RAG) represents a significant advancement in natural language processing (NLP) by combining the strengths of generative models and retrieval-based systems. Traditional large language models (LLMs) rely on pre-trained knowledge encoded in their parameters, which can be limited, especially for knowledge-intensive tasks that require up-to-date or domain-specific information. RAG systems address this limitation by incorporating external knowledge sources during the text generation process. By retrieving relevant documents or passages from large databases or structured knowledge graphs, RAG systems enable LLMs to generate more accurate, contextually informed, and factually grounded responses.

In this paper, we survey the key approaches to vector-indexed and graph-based retrieval in the context of RAG, focusing on how these methods facilitate the integration of external knowledge into generative workflows.

The scope of this paper is confined to the retrieval and generation of text-based data.`

I. RECENT TECHNOLOGICAL ADVANCES

Recent advances in LLM techniques, such as Chain-of-Thought Prompting [6], have shown that eliciting reasoning from the model during the generation process can significantly improve performance, especially on complex tasks. These advances have inspired the development of more efficient LLM architectures and retrieval mechanisms. Models like GPT-4 [5] and Llama [1,2] have significantly improved their token limits, allowing them to process much larger context windows compared to previous models. This enhancement enables RAG systems to retrieve and incorporate a larger amount of external text data in a single query,

improving the quality and relevance of the generated responses. By handling more context in one pass, these models can maintain better coherence across longer documents and more complex retrieval scenarios, thus improving the performance of RAG systems, particularly in tasks that require the integration of extensive background knowledge.

These models are pivotal in facilitating the integration of large-scale external knowledge sources into generative workflows, ensuring that RAG systems remain effective and efficient across a range of knowledge-intensive applications.

Moreover, cutting-edge models like Gemini [4], which integrate multimodal capabilities, suggest exciting new possibilities for expanding RAG's utility beyond traditional text-based applications. Multimodal RAG systems could, for example, incorporate images, videos, or other non-textual data alongside traditional text-based queries, broadening the scope of tasks that RAG systems can handle—from complex visual question answering to more sophisticated content generation across modalities.

Despite these advances, challenges remain in integrating structured and unstructured data in real-time, which is a central focus of RAG systems. As demonstrated in models like BERT [3], effective handling of structured data is possible, thanks to its ability to capture rich, contextualized representations of text. Similarly, we should aim to handle unstructured data effectively to maximize data ingestion efficiency and ensure high-quality retrieval in RAG systems.

II. Foundations of Retrieval-Augmented Generation (RAG)

RAG Overview:

Retrieval-augmented generation (RAG) is a paradigm that enhances the capabilities of large language models (LLMs) by incorporating external knowledge sources during the generation process. Rather than relying solely on pre-trained model weights, RAG systems integrate a retrieval component that fetches relevant documents or passages from a large knowledge base and provides these as context for generating responses. This hybrid approach enables models to generate more accurate, informative, and contextually aware text, especially for knowledge-intensive tasks that require up-to-date or domain-specific information. The core mechanics of RAG were introduced and formally defined in prior works, such as the Retrieval-augmented Generation for Knowledge-Intensive NLP Tasks paper [7], which laid the groundwork for how retrieval-based methods can be coupled with generative models to improve performance across tasks like question answering and summarization.

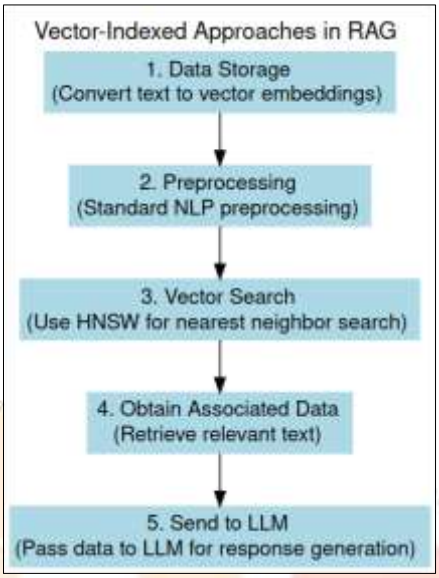
Key Techniques:

The retrieval component of RAG systems typically employs two main methods for sourcing relevant information: **nearest neighbor search** and **token-based retrieval**. Nearest neighbor search involves finding documents or passages that are closest in semantic space to the query, often based on vector embeddings derived from pre-trained language models. These embeddings are compared using similarity metrics, such as cosine similarity, to identify the most relevant information. Token-based retrieval methods, on the other hand, select documents based on direct matching of query tokens to indexed document tokens, making it a more traditional keyword-based retrieval approach. For large-scale retrieval, methods such as approximate nearest neighbor (ANN) search are commonly used to accelerate the process of finding the closest matching documents, as demonstrated in works like **Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs** [9]. In RAG, the retrieved documents or passages are then passed to the generative model, which produces contextually relevant text by conditioning the generation process on both the input query and the retrieved information.

III. VECTOR-INDEXED APPROACHES IN RAG

Key Steps:

1. **Data Storage**: Convert and store text data as vector embeddings.
2. **Preprocessing**: Standard NLP preprocessing (tokenization, stop word removal, etc.).
3. **Vector Search**: Use HNSW to perform efficient approximate nearest neighbor search in high-dimensional vector space.
4. **Obtain Associated Data**: Retrieve the relevant text associated with the closest embedding.
5. **Send to LLM:** Pass the retrieved text and user query to an LLM for response generation.



This is the most common approach to building a RAG application. In this approach, we store the data that the user wants the LLM (Large Language Model) to read and answer in the form of vector embeddings. We then use **vector-indexed search** to retrieve the relevant text from the database.

Data Storage

The text data provided by the user must be converted into **text embeddings** for efficient storage. Text embeddings are generated by passing the text through embedding models, such as Ada-002. The model's output consists of vector embeddings—dense, real-valued numerical arrays that capture semantic information about the objects they represent.

Vector embeddings are essentially **ordered collections of numbers**, and they store semantic relationships between words. These vectors exist in **high-dimensional space**, typically ranging from several hundred to a few thousand dimensions. In such a high-dimensional space, text data is stored in a way that words with similar meanings are positioned close to each other, while words with different meanings are positioned farther apart.

For example, words like **"car," "vehicle,"** and **"taxi"** will be close in this vector space, while words like **"and"** or **"the"** will be far apart.

Preprocessing

Preprocessing consists of standard natural language processing (NLP) operations, such as tokenization, lowercasing, stop word removal, and special character handling.

- **Tokenization**: Breaking a larger text into smaller units (tokens).
- **Lowercasing**: Converting text to lowercase to ensure uniformity.
- **Stop Word Removal**: Eliminating irrelevant words (such as "is," "the," etc.) that don't contribute much to the meaning.
- **Special Character Handling**: Removing symbols, markup language formatting, etc.

Once the text is preprocessed, we convert it into **vector embeddings** using a suitable model. Embedding generator models typically handle tokenization, encoding, and embedding generation.

- **Tokenization**: The text is split into smaller units (tokens).
- **Self-attention Mechanism**: The model determines the strength of the relationship between each word and every other word in the sequence, allowing it to capture dependencies regardless of their

distance in the text. This mechanism, introduced in the Transformer model by Vaswani et al. [8], is key to enabling parallel computation and long-range context understanding.

- **Encoding**: A transformer-based model creates the text encoding using neural networks. BERT, in particular, introduced highly effective bidirectional encoders, significantly improving the quality of contextual understanding in language models [3].

The output of the embedding model is a **vector embedding** that captures the semantic meaning of the text in a high-dimensional space.

Storage of Vector Embeddings

These vector embeddings need to be stored in a specific **database** designed to handle the storage and retrieval of high-dimensional vectors. Popular databases for storing vector embeddings include **FAISS**, **Pinecone**, and **ChromaDB**, among others.

Vector Search

During **vector search**, we retrieve the most relevant pieces of information (stored as text embeddings) from the high-dimensional vector space. The process involves finding the **nearest neighbors** of the query vector in this embedding space.

As the number of vectors grows, the search time can become exponentially expensive. In this case, one of the most popular, effective, and efficient **Approximate Nearest Neighbor Search (ANNS)** techniques is **HNSW (Hierarchical Navigable Small World)**, introduced in the paper *"Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs"* [9].

HNSW: Hierarchical Navigable Small World Graphs

HNSW takes advantage of the **small-world** property of graphs, which has two key characteristics:

1. **Local Clustering**: Nodes are connected to a small number of neighbors that are close in the graph.
2. **Long-range Connections**: Despite having only a few neighbors, each node has a small number of long-range connections that enable fast "shortcuts" to distant parts of the graph.

Hierarchical Structure

HNSW uses a **multi-layered graph** structure:

- The lower layers are denser, while the upper layers become more sparse.
- The **bottom-most layer** contains all the vectors, and each higher layer contains a subset of the previous layer, allowing for faster navigation.

Performing the Search

Each node in the multi-layer graph represents a data point with a text embedding. The search begins at the **topmost layer** and moves downwards until reaching the lowest layer.

At each layer:

- The algorithm greedily moves towards the **most similar node** to the query.
- Once the best candidate is found at the current layer, the search moves downward to the next layer.
- As we move down, the graph becomes denser, and the search becomes more precise.

This process is described as the **zoom-out phase** (moving through lower-density layers) and the **zoom-in phase** (navigating through higher-density layers) in the HNSW paper [9].

Obtain Associated Data

Once the closest embedding is found, it typically corresponds to a document or text chunk in a large corpus. The next step is to **retrieve** the actual text data associated with the closest embedding.

This can include a full document, a passage, or a section of text relevant to the user query.

Send Retrieved Data to the LLM

The retrieved text is then passed, along with the user's original query, to the **LLM (Large Language Model)**. The LLM reads both the user query and the relevant associated text and responds in the designated question-answer format or other task-specific formats (e.g., summarization, completion, etc.).

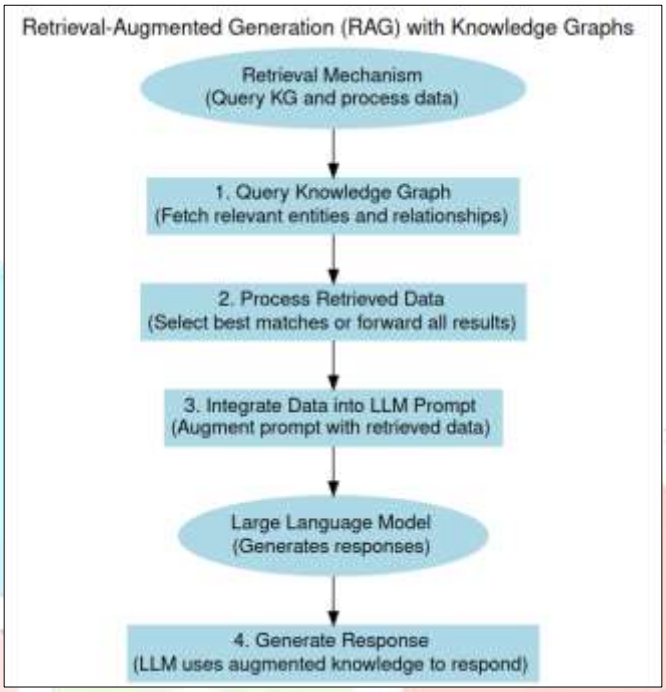
IV. RETRIEVAL-AUGMENTED GENERATION (RAG) WITH KNOWLEDGE GRAPHS

Key Steps:

1. **Query the Knowledge Graph** to fetch relevant entities and relationships.
2. **Process the retrieved data** (either by forwarding all results or selecting the best matches).
3. **Integrate the retrieved data** into the prompt for the LLM.
4. **Generate a response** using the LLM, now augmented with relevant external knowledge.

In a **retrieval-augmented generation (RAG)** system, two main components are at play:

1. **The Retrieval Mechanism**, which retrieves external data that the language model (LLM) has not been pre-trained on.
2. **The LLM**, which generates responses based on the retrieved data and the user's query.



The system operates by feeding relevant portions of this external data to the LLM during the prompt phase. For instance, in a question-answering scenario, the retrieved information supplements the LLM's knowledge, providing it with up-to-date, domain-specific, or factual knowledge that the model may not inherently possess.

Knowledge Graphs for Data Storage and Retrieval

A **knowledge graph** is a data structure that stores information in the form of **nodes** and **edges**. Each node represents an **entity** (such as an object, person, or concept), while the edges define the **relationships** between these entities.

For example:

- **Entity 1**: Gold (Node)
- **Entity 2**: Jewellery (Node)
- **Relationship**: "Used in" (Edge)

Thus, the knowledge graph could represent the relationship between these two entities as follows:

- **Node 1**: Gold
- **Node 2**: Jewellery
- **Edge**: "Used in"

The main advantage of using a knowledge graph for storage is that it reflects how humans store and recall information. Just as we relate concepts and recall facts based on their associations (e.g., "Gold is used in Jewellery"), a knowledge graph captures this relational structure, making it easier to traverse and retrieve related data.

By storing related data together and connecting it via edges, knowledge graphs enable efficient **traversal**. When a particular piece of information is needed, it is usually located in close proximity to the relevant entity within the graph, reducing the computational cost of search queries.

Retrieving Data from the Knowledge Graph

The retrieval phase in a RAG system involves querying the knowledge graph to fetch relevant information. This is done using **graph query languages** or **graph databases**, which are designed to perform pattern matching and pathfinding across nodes and edges. Examples of graph query languages include **Cypher** (used in Neo4j), **SPARQL** (used in RDF-based systems), and other query systems tailored for graph-based data.

The process of query execution typically involves:

1. **Pattern Matching**: Identifying nodes and edges that match the query's conditions.
2. **Pathfinding**: Traversing the graph to find relationships between entities, particularly when the query involves multiple nodes and edges.
3. **Returning Results**: The result of a query can be a list of matching nodes, a set of relationships between entities, or a subgraph representing a pattern of connections. The data returned is usually structured in a machine-readable format such as **JSON** or **RDF**.

For instance, a query asking for "What is gold used in?" might return a list of related entities like "Jewellery", "Electronics", and "Coins", along with their associated relationships (e.g., "Gold is used in Jewellery").

Integrating Retrieved Data into the LLM for Response Generation

Once the query results are returned from the knowledge graph, the next step is to decide how to utilise this information in the **prompting** process for the LLM. The retrieved data can be processed in a few different ways:

- **Forwarding All Retrieved Results**: You could send the entire result set (e.g., a list of nodes and relationships) along with the user's query to the LLM. This ensures the model has access to all the information, though it may increase complexity or noise in the prompt.
- **Selecting the Best Matches**: Alternatively, the system could select the most relevant or highly ranked results from the knowledge graph based on a scoring mechanism, ensuring that only the most relevant information is provided to the LLM.

Generating Responses with the LLM

Once the LLM receives the **user's query** along with the relevant retrieved data, it can use this external knowledge—information that was previously **unknown** to the model—to generate a response. The retrieval-augmented approach allows the LLM to generate more **contextually accurate** and **knowledge-rich** responses than it would otherwise be able to generate based on its pre-trained knowledge alone.

The LLM, now equipped with relevant factual data from the knowledge graph, uses this context to produce an output. The generated answer can then be a more precise, contextually informed response to the user's query.

V. CHALLENGES IN DATA HANDLING FOR RAG SYSTEMS

Large Data Intake

One of the key challenges in implementing a Retrieval-Augmented Generation (RAG) system is **handling large volumes of data**. The system may struggle to **ingest** large datasets quickly, particularly when those datasets must be processed in real-time. The process of **embedding creation**—which transforms raw data into vector embeddings—can be **computationally expensive** and time-consuming.

When large datasets are suddenly ingested:

- The system's **performance may degrade**, with **slower ingestion times** and **increased processing overhead**.
- **Batch processing** may be required to mitigate the performance impact, but this may introduce additional delays.
- **Overall system efficiency** may decline due to the increased load on the ingestion pipeline and the resource-intensive process of embedding generation.

To manage this challenge, scalable systems are needed to efficiently handle large data volumes while maintaining a balance between data intake speed and computational resources.

Handling Unstructured Data

Another major challenge in RAG systems is **working with unstructured data**. Unstructured data, such as **PDFs, emails, and web pages**, often contains a variety of complex elements—tables, charts, images, and different formatting—that make it difficult to store and retrieve efficiently. The challenges include:

- **Data Storage**: Unstructured data may not be stored in a structured format, causing issues with organizing and indexing.
- **Semantic Loss**: When the data is not properly structured, the **original order and context** of the information may be lost. This can lead to incomplete or inaccurate retrievals, especially if the relationships between different pieces of information (e.g., tables, figures, and accompanying text) are not preserved.
- **Formatting Complexity**: Unstructured data comes in varied formats, and processing it requires additional steps like **text extraction**, **cleaning**, and **standardization** to make it suitable for the RAG system.

The difficulty of preserving semantic meaning, dealing with noisy or complex data formats, and ensuring that relevant context is captured during retrieval presents significant obstacles in building robust RAG systems. Overcoming these challenges is crucial for improving the overall efficiency and effectiveness of RAG, particularly in real-time knowledge retrieval and response generation.

VI. *Conclusion*

Retrieval-Augmented Generation (RAG) systems represent a significant advancement in natural language processing by enhancing large language models (LLMs) with external knowledge retrieval. By integrating relevant, up-to-date, and domain-specific information, RAG systems enable LLMs to generate more accurate, informative, and contextually aware responses, particularly for knowledge-intensive tasks. Techniques such as vector-indexed search, approximate nearest neighbor algorithms like HNSW, and the integration of knowledge graphs are fundamental to improving the efficiency and quality of data retrieval in RAG systems.

Despite their powerful capabilities, RAG systems face challenges in handling large-scale data intake, especially with unstructured data like PDFs, web pages, and images. Addressing these challenges requires ongoing improvements in data preprocessing, storage solutions, and retrieval mechanisms. Moreover, advancements in LLMs, including techniques like chain-of-thought prompting and multimodal integration, continue to expand the potential of RAG systems, making them more efficient and capable of solving complex tasks across diverse domains.

Future research in RAG systems will likely focus on optimizing data retrieval and integration processes, handling real-time data ingestion, and improving the scalability of systems to manage large datasets. As these systems evolve, they are poised to become even more powerful tools for a wide range of applications, including question answering, summarization, and complex reasoning tasks, driving advancements in both academic research and practical applications.

VII. REFERENCES

[1] Touvron, Hugo, et al. “Llama: Open and Efficient Foundation Language Models.” arXiv.Org, 27 Feb. 2023, arxiv.org/abs/2302.13971.

[2] Touvron, Hugo, Louis Martin, et al. “Llama 2: Open Foundation and Fine-Tuned Chat Models.” arXiv.Org, 19 July 2023, arxiv.org/abs/2307.09288.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

[4] Gemini Team Google. [2312.11805] Gemini: A Family of Highly Capable Multimodal Models, arxiv.org/abs/2312.11805.

[5] OpenAI, et al. “GPT-4 Technical Report.” arXiv.Org, 4 Mar. 2024, arxiv.org/abs/2303.08774.

[6] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2024. Chain-of-thought prompting elicits reasoning in large language models. In

Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22). Curran Associates Inc., Red Hook, NY, USA, Article 1800, 24824–24837.

[7] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20). Curran Associates Inc., Red Hook, NY, USA, Article 793, 9459–9474.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17). Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[9] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. IEEE Trans. Pattern Anal. Mach. Intell. 42, 4 (April 2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473