elastic search labs

BLOG

Advanced RAG techniques part 1: Data processing

Discussing and implementing techniques which may increase RAG performance. Part 1 of 2, focusing on the data processing and ingestion component of an advanced RAG pipeline.

Vector Database | Generative AI

By: **Han Xiang Choong**

On August 14, 2024

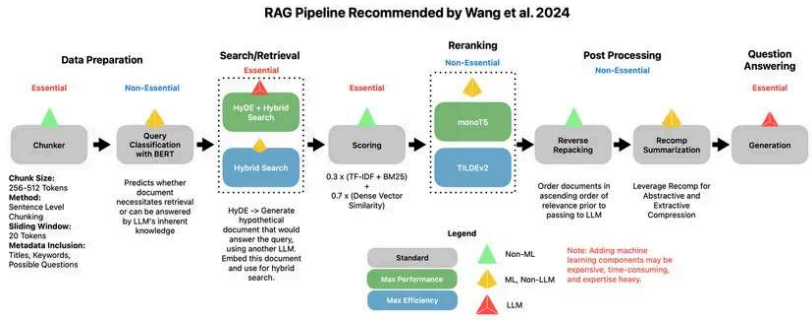
PART OF SERIES

Advanced RAG techniques

This is Part 1 of our exploration into Advanced RAG Techniques. ***Click here for Part 2!***

The recent paper **Searching for Best Practices in Retrieval-Augmented Generation** empirically assesses the efficacy of various RAG enhancing techniques, with the goal of converging on a set of best-practices for RAG.

JUMP TO



The RAG pipeline recommended by Wang and colleagues.

We'll implement a few of these proposed best-practices, namely the ones which aim to improve the quality of search **(Sentence Chunking, HyDE, Reverse Packing)**.

For brevity, we will omit those techniques focused on improving efficiency **(Query Classification and Summarization)**.

We will also implement a few techniques that were not covered, but which I personally find useful and interesting **(Metadata Inclusion, Composite Multi-Field Embeddings, Query Enrichment)**.

Finally, we'll run a short test to see if the quality of our search results and generated answers has improved versus the baseline. Let's get to it!

RAG overview

RAG aims to enhance LLMs by retrieving information from external knowledge bases to enrich generated answers. By providing domain-specific information, LLMs can be quickly adapted for use cases outside the scope of their training data; significantly cheaper than fine-tuning, and easier to keep up-to-date.

Measures to improve the quality of RAG typically focus on two tracks:

Data ingestion

Show more

SHARE

1. Enhancing the quality and clarity of the knowledge base.
2. Improving the coverage and specificity of search queries.

These two measures will achieve the goal of improving the odds that the LLM has access to relevant facts and information, and is thus less likely to hallucinate or draw upon its own knowledge - which may be outdated or irrelevant.

The diversity of methods is difficult to clarify in just a few sentences. Let's go straight to implementation to make things clearer.

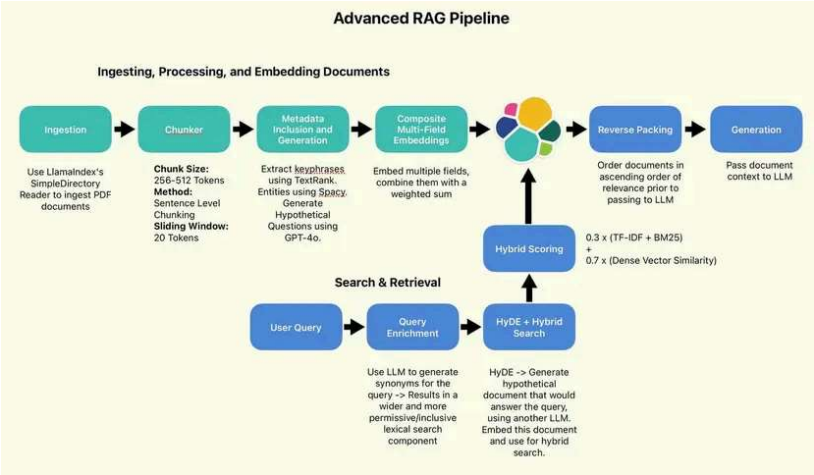


Figure 1: The RAG pipeline used by the author.

Table of contents

Set-up

*All code may be found **in the Searchlabs repo**.*

First things first. You will need the following:

1. An Elastic Cloud Deployment
2. An LLM API - We are using a GPT-4o deployment on Azure OpenAI in this notebook
3. Python Version 3.12.4 or later

We will be running all the code from **the main.ipynb notebook.**

Go ahead and git clone the repo, navigate to supporting-blog-content/advanced-rag-techniques, then run the following commands:

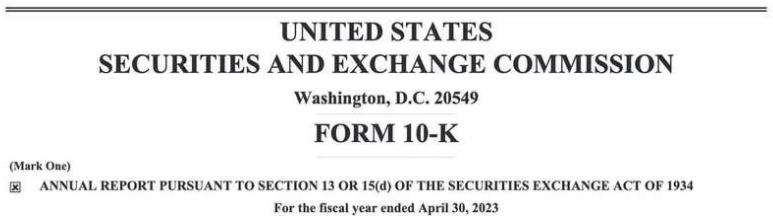
```
# Create a new virtual environment named 'rag_
python -m venv rag_env

# Activate the virtual environment (for Unix-ba
source rag_env/bin/activate

# (For Windows)
.\rag_env\Scripts\activate

# Install packages listed in requirements.txt
pip install -r requirements.txt
```

Once that's done, create a *.env* file and fill out the following fields (Referenced in ***.env.example***). Credits to my co-author, Claude-3.5, for the helpful comments.

```
# Elastic Cloud: Found in the 'Deployment' page
# console
ELASTIC_CLOUD_ENDPOINT=""
ELASTIC_CLOUD_ID=""

# Elastic Cloud: Created during deployment setu
# settings
ELASTIC_USERNAME=""
ELASTIC_PASSWORD=""

# Elastic Cloud: The name of the index you crea
ELASTIC_INDEX_NAME=""

# Azure AI Studio: Found in 'Keys and Endpoint
# OpenAI resource
AZURE_OPENAI_KEY_1=""
AZURE_OPENAI_KEY_2=""
AZURE_OPENAI_REGION=""
AZURE_OPENAI_ENDPOINT=""

# Azure AI Studio: Found in 'Deployments' sect
# resource
```

```
AZURE_OPENAI_DEPLOYMENT_NAME=""

# Using BAAI/bge-small-en-v1.5 because I think
# resource efficiency and performance.
HUGGINGFACE_EMBEDDING_MODEL="BAAI/bge-small-en
```

Next, we'll choose the document to ingest, and place it in the documents folder. For this article, we'll be using the **Elastic N.V. Annual Report 2023**. It's a pretty challenging and dense document, perfect for stress testing our RAG techniques.



UNITED STATES
SECURITIES AND EXCHANGE COMMISSION
Washington, D.C. 20549
FORM 10-K
(Mark One)
☒ ANNUAL REPORT PURSUANT TO SECTION 13 OR 15(d) OF THE SECURITIES EXCHANGE ACT OF 1934
For the fiscal year ended April 30, 2023

Elastic Annual Report 2023

Now we're all set, let's go to ingestion. Open *main.ipynb* and execute the first two cells to import all packages and intialize all services.

Back to top

Ingesting, processing, and embedding documents

Data ingestion

- *Personal note: I am stunned by LlamaIndex's convenience. In the olden days before LLMs and LlamaIndex, ingesting documents of various formats was a painful process of collecting*

esoteric packages from all over. Now it's reduced to a single function call. Wild.

The `SimpleDirectoryReader` will load every document in the `directory_path.` For `.pdf` files, it returns a list of document objects, which I convert to Python dictionaries because I find them easier to work with.

```
# llamaindex_processor.py
from llama_index.core import SimpleDirectoryRe

class LlamaIndexProcessor:
    def __init__(self):
        pass

    def load_documents(self, directory_path):
        '''
        Load all documents in directory
        '''
        reader = SimpleDirectoryReader(input_di
        return reader.load_data()

# main.ipynb
llamaindex_processor=LlamaIndexProcessor()
documents=llamaindex_processor.load_documents(
documents=[dict(doc_obj) for doc_obj in docume
```

Each dictionary contains the key content in the `text` field. It also contains useful metadata such as page number, filename, file size, and type.

```
{
  'id_': '5f76f0b3-22d8-49a8-9942-c2bbab14f63f
  'metadata': {'page_label': '5',
   'file_name': 'Elastic_NV_Annual-Report-Fisc
   'file_path': '/Users/han/Desktop/Projects/t
   'file_type': 'application/pdf',
```

```
        'file_size': 3724426,
        'creation_date': '2024-07-27',
        'last_modified_date': '2024-07-27'},
        'text': 'Table of Contents\nPage\nPART I\nI
        ...
    }
```

Back to top

Sentence-level, token-wise chunking

The first thing to do is reduce our documents to chunks of a standard length (to ensure consistency and manageability). Embedding models have unique token limits (maximum input size they can process). Tokens are the basic units of text that models process. To prevent information loss (truncation or omission of content), we should provide text that does not exceed those limits (by splitting longer texts into smaller segments).

Chunking has a significant impact on performance. Ideally, each chunk would represent a self-contained piece of information, capturing contextual information about a single topic. Chunking methods include word-level chunking, where documents are split by word count, and semantic chunking which uses an LLM to identify logical breakpoints.

Word-level chunking is cheap, fast, and easy, but runs a risk of splitting sentences and thus breaking context. Semantic chunking gets slow and expensive, especially if you're dealing with documents like the 116-page Elastic Annual Report.

Let's choose a middleground approach. Sentence level chunking is still simple, but can preserve context more effectively than word-level chunking

while being significantly cheaper and faster. Additionally, we'll implement a sliding window to capture some of the surrounding context, and alleviate the impact of splitting paragraphs.

```
# chunker.py

import uuid
import re


class Chunker:
    def __init__(self, tokenizer):
        self.tokenizer = tokenizer

    def split_into_sentences(self, text):
        """Split text into sentences."""
        return re.split(r'(?<=[.!?])\s+', text

    def sentence_wise_tokenized_chunk_document
        '''
        1. Split text into sentences.
        2. Tokenize using the provided tokeniz
        3. Build chunks up to the chunk_size l
        4. Create an overlap based on tokens -
        5. Only keep chunks that meet the mini
        '''
        chunked_documents = []

        for doc in documents:
            sentences = self.split_into_senten
            tokens = []
            sentence_boundaries = [0]

            # Tokenize all sentences and keep
            for sentence in sentences:
                sentence_tokens = self.tokeniz
                tokens.extend(sentence_tokens)
                sentence_boundaries.append(len

            # Create chunks
            chunk_start = 0
```

```
            while chunk_start < len(tokens):
                chunk_end = chunk_start + chunk

                # Find the last complete sentence
                sentence_end = next((i for i in
                chunk_end = min(chunk_end, sent

                # Create the chunk
                chunk_tokens = tokens[chunk_sta

                # Check if the chunk meets the
                if len(chunk_tokens) >= min_chu
                    # Create a new document obj
                    chunk_doc = {
                        'id_': str(uuid.uuid4(
                        'chunk': chunk_tokens,
                        'original_text': self.
                        'chunk_index': len(chun
                        'parent_id': doc['id_'
                        'chunk_token_count': le
                    }

                    # Copy all other fields fro
                    for key, value in doc.items
                        if key != 'text' and ke
                            chunk_doc[key] = va

                    chunked_documents.append(ch

                # Move to the next chunk start
                chunk_start = max(chunk_start +

        return chunked_documents

# main.ipynb
# Initialize Embedding Model
HUGGINGFACE_EMBEDDING_MODEL = os.environ.get('H
embedder=EmbeddingModel(model_name=HUGGINGFACE_

# Initialize Chunker
chunker=Chunker(embedder.tokenizer)
```

The `Chunker` class takes in the embedding model's tokenizer to encode and decode text. We'll now build chunks of 512 tokens each, with an overlap of 20 tokens. To do this, we'll split the text into sentences, tokenize those sentences, and then add the tokenized sentences to our current chunk until we cannot add more without breaching our token limit.

Finally, decode the sentences back to the original text for embedding, storing it in a field called `original_text`. Chunks are stored in a field called `chunk`. To reduce noise (aka useless documents), we will discard any documents smaller than 50 tokens in length.

Let's run it over our documents:

```
chunked_documents=chunker.sentence_wise_tokeni
```

And get back chunks of text that look like this:

```
print(chunked_documents[4]['original_text'])

[CLS] the aggregate market value of the ordinar
based on the closing price of the shares of or
october 31, 2022 ( the last business day of th
approximately $ 6. 1 billion. [SEP] [CLS] as o
ordinary shares, par value €0. 01 per share, o
reference portions of the registrant 's defini
023 annual general meeting of shareholders are
...
...
```

Back to top

Metadata inclusion and generation

We've chunked our documents. Now it's time to enrich the data. I want to generate or extract additional metadata. This additional metadata can be used to influence and enhance search performance.

We'll define a `DocumentEnricher` class, whose role is to take in a list of documents (Python dictionaries), and a list of processor functions. These functions will run over the documents' `original_text` column, and store their outputs in new fields.

First, we extract keyphrases using **TextRank**. TextRank is a graph-based algorithm that extracts key phrases and sentences from text by ranking their importance based on the relationships between words.

Next, we'll **generate potential_questions using GPT-4o**.

Finally, we'll **extract entities** using **Spacy**.

Since the code for each of these is quite lengthy and involved, I will refrain from reproducing it here. If you are interested, the files are marked in the code samples below.

Let's run the data enrichment:

```
# documentenricher.py
from tqdm import tqdm

class DocumentEnricher:

    def __init__(self):
        pass
```

```
    def enrich_document(self, documents, proces
        for doc in tqdm(documents, desc="Enricl
            for (processor, field) in processor
                metadata=processor(doc[text_co
                if isinstance(metadata, list):
                    metadata='\n'.join(metadat
                doc.update({field: metadata})

# main.ipynb
# Initialize processor classes
nltkprocessor=NLTKProcessor() // nltk_processor
entity_extractor=EntityExtractor() // entity_e
gpt4o = LLMProcessor(model='gpt-4o') // llm.py

# Initialize LLM
documentenricher=DocumentEnricher()

# Create new fields in the documents - These ar
processors=[
    (nltkprocessor.textrank_phrases, "keyphrase
    (gpt4o.generate_questions, "potential_quest
    (entity_extractor.extract_entities, "entit
    ]

# .enrich_document() will modify chunked_docs :
# To view the results, we'll print chunked_doc
documentenricher.enrich_document(chunked_docs,
```

And take a look at the results:

Keyphrases extracted by TextRank

These keyphrases are a stand-in for the chunk's core topics. If a query has to do with cybersecurity, this chunk's score will be boosted

```
print(chunked_documents[25]['keyphrases'])
```

```
'elastic agent stop', 'agent stop malware',
'stop malware ransomware', 'malware ransomware
'ransomware environment wide', 'environment wi
'wide visibility threat', 'visibility threat d
'sep cl key', 'cl key feature'
```

Potential questions generated by GPT-4o

These potential questions may directly match with user queries, offering a boost in score. We prompt GPT-4o to generate questions which can be answered using the information found in the current chunk.

```
print(chunked_documents[25]['potential_questio
```

```
1. What are the primary functions that Elastic
2. Describe how Logstash contributes to data m
3. List and explain any key features of Logsta
4. How does Elastic Agent enhance environment-
5. What capabilities does Logstash offer for h
6. In what ways does the document suggest that
7. Can you identify any relationships between
8. What implications might the advanced threat
9. Compare and contrast the roles of Elastic A
10. How might the centralized collection abili
```

Entities extracted by Spacy

These entities serve a similar purpose to the keyphrases, but capture organizations' and individuals' names, which keyphrase extraction may miss.

```
print(chunked_documents[29]['entities'])
```

```
'appdynamics', 'apm data', 'azure sentinel',
'microsoft', 'mcafee', 'broadcom', 'cisco',
'dynatrace', 'coveo', 'lucidworks'
```

Back to top

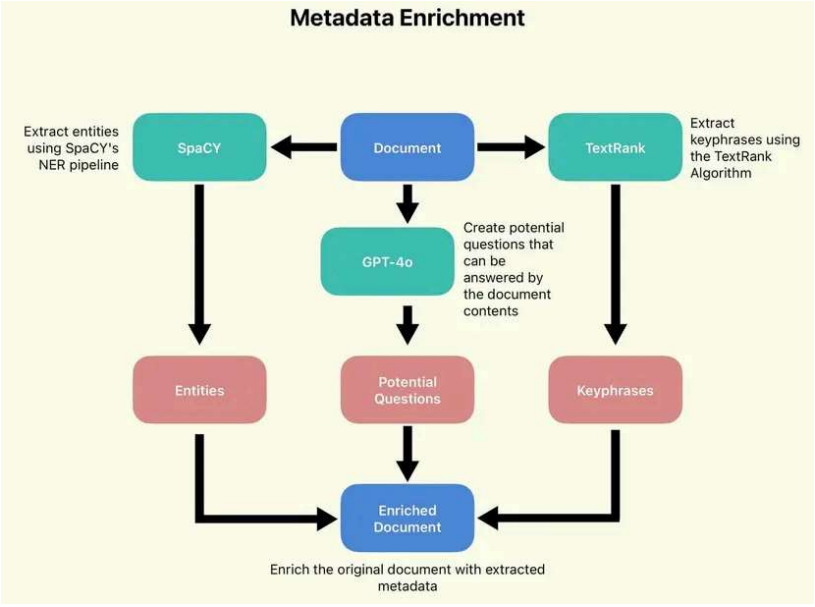
Composite multi-field embeddings

Now that we have enriched our documents with additional metadata, we can leverage this information to create more robust and context-aware embeddings.

Let's review our current point in the process. We've got four fields of interest in each document.

```
{
    "chunk": "...",
    "keyphrases": "...",
    "potential_questions": "...",
    "entities": "..."
}
```

Each field represents a different perspective on the document's context, potentially highlighting a key area for the LLM to focus on.



Metadata Enrichment Pipeline

The plan is to embed each of these fields, and then create a weighted sum of the embeddings, known as a Composite Embedding.

With luck, this Composite Embedding will allow the system to become more context aware, in addition to introducing another tunable hyperparameter from controlling the search behavior.

First, let's embed each field and update each document in place, using our locally defined embedding model imported at the beginning of the main.ipynb notebook.

```
# EmbeddingModel defined in embedding_model.py
embedder=EmbeddingModel(model_name=HUGGINGFACE_

cols_to_embed=['keyphrases', 'potential_questi

embedding_cols=[]
for col in cols_to_embed:
    # Works on text input
    embedding_col=embedder.embed_documents_tex
    embedding_cols.append(embedding_col)
# Works on token input
embedding_col=embedder.embed_documents_token_w
```

```
    embedding_cols.append(embedding_col)
```

Each embedding function returns the embedding's field, which is just the original input field with an `_embedding` postfix.

Let's now define the weightings of our composite embedding:

```
embedding_cols=[
                'keyphrases_embedding',
                'potential_questions_embedding
                'entities_embedding',
                'chunk_embedding']
combination_weights=[
                    0.1,
                    0.15,
                    0.05,
                    0.7
                ]
```

The weightings allow you to assign priorities to each component, based on your usecase and the quality of your data. Intuitively, the size of these weightings is dependent on the semantic value of each component. Since the chunk text itself is by far the richest, I assign a weighting of 70%. Since the entities are the smallest, being just a list of org or person names, I assign it a weighting of 5%. The precise setting for these values has to be determined empirically, on a use-case by use-case basis.

Finally, let's write a function to apply the weightings, and create our composite embedding. We'll delete all the component embeddings as well to save space.

```
from tqdm import tqdm
```

```
def combine_embeddings(objects, embedding_cols
    # Ensure the number of weights matches the
    assert len(embedding_cols) == len(combinat

    # Normalize weights to sum to 1
    weights = np.array(combination_weights) / 

    for obj in tqdm(objects, desc="Combining e
        # Initialize the combined embedding
        combined = np.zeros_like(obj[embedding_

        # Compute the weighted sum
        for col, weight in zip(embedding_cols,
            combined += weight * np.array(obj[

        # Add the new combined embedding to th
        obj.update({primary_embedding:combined

        # Remove the original embedding column
        for col in embedding_cols:
            obj.pop(col, None)

combine_embeddings(chunked_documents, embeddin
```

With this, we've completed our document

which look like this:

```
ıpany, indicate by check mark if the registrant ha
```

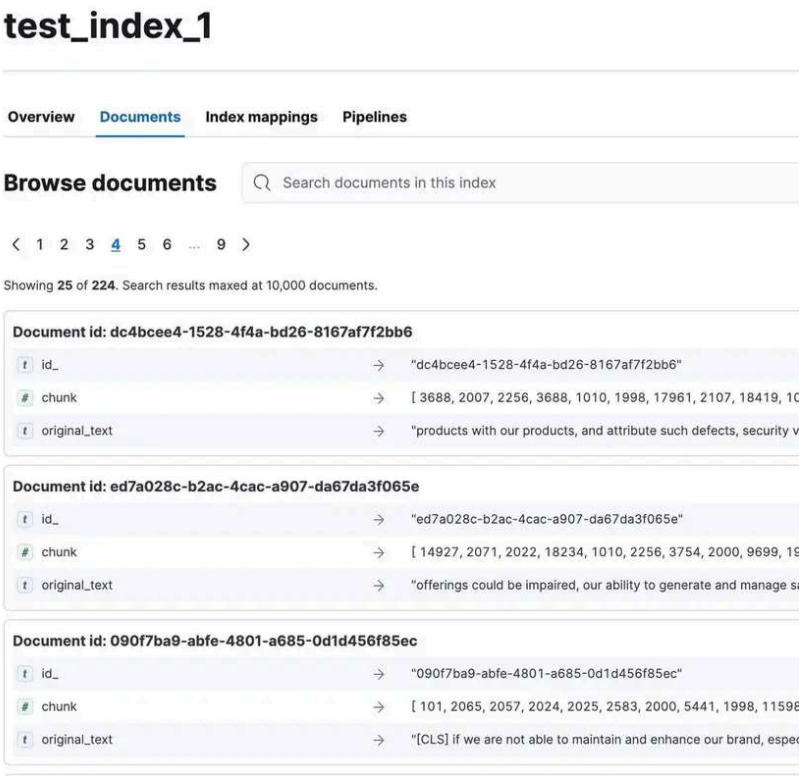
Indexing to Elastic

Let's bulk upload our documents to Elastic Search. For this purpose, I long-ago defined a set of Elastic Helper functions in elastic_helpers.py. It is a very lengthy piece of code so let's sticking to looking at the function calls.

`es_bulk_indexer.bulk_upload_documents` works with any list of dictionary objects, taking advantage of Elasticsearch's convenient dynamic mappings.

```
# Initialize Elasticsearch
ELASTIC_CLOUD_ID = os.environ.get('ELASTIC_CLOU
ELASTIC_USERNAME = os.environ.get('ELASTIC_USEI
ELASTIC_PASSWORD = os.environ.get('ELASTIC_PASS
ELASTIC_CLOUD_AUTH = (ELASTIC_USERNAME, ELASTIC
es_bulk_indexer = ESBulkIndexer(cloud_id=ELASTI
es_query_maker = ESQueryMaker(cloud_id=ELASTIC_

# Define Index Name
index_name=os.environ.get('ELASTIC_INDEX_NAME'


# Create index and bulk upload
index_exists = es_bulk_indexer.check_index_exis
if not index_exists:
    logger.info(f"Creating new index: {index_na
    es_bulk_indexer.create_es_index(es_configur

success_count = es_bulk_indexer.bulk_upload_doc
    index_name=index_name,
    documents=chunked_documents,
    id_col='id_',
    batch_size=32
)
```

Head on over to Kibana and verify that all documents have been indexed. There should be 224 of them. Not bad for such a large document!



Indexed Annual Report Documents in Kibana

Back to top

Cat break

Let's take a break, article's a little heavy, I know.
Check out my cat:

look at how furious she is

Adorable. The hat went missing and I half suspect she stole and hid it somewhere :(

Congrats on making it this far :)

Join me in **Part 2** for testing and evaluation of our RAG pipeline!

Appendix

Definitions

1. Sentence Chunking

- A preprocessing technique used in RAG systems to divide text into smaller, meaningful units.
- *Process:*

1. Input: Large block of text (e.g., document, paragraph)
2. Output: Smaller text segments (typically sentences or small groups of sentences)

- *Purpose:*
  - Creates granular, context-specific text segments
  - Allows for more precise indexing and retrieval
  - Improves the relevance of retrieved information in RAG systems
- *Characteristics:*
  - Segments are semantically meaningful
  - Can be independently indexed and retrieved
  - Often preserves some context to ensure standalone comprehensibility
- *Benefits:*
  - Enhances retrieval precision
  - Enables more focused augmentation in RAG pipelines

2. HyDE (Hypothetical Document Embedding)

- A technique that uses an LLM to generate a hypothetical document for query expansion in RAG systems.
- *Process:*
  1. Input query to an LLM

2. LLM generates a hypothetical document answering the query
3. Embed the generated document
4. Use the embedding for vector search

- *Key difference:*
  - Traditional RAG: Matches query to documents
  - HyDE: Matches documents to documents
- *Purpose:*
  - Improve retrieval performance, especially for complex or ambiguous queries
  - Capture richer semantic context than a short query
- *Benefits:*
  - Leverages LLM's knowledge to expand queries
  - Can potentially improve relevance of retrieved documents
- *Challenges:*
  - Requires additional LLM inference, increasing latency and cost
  - Performance depends on quality of generated hypothetical document

3. Reverse Packing

- A technique used in RAG systems to reorder search results before passing them to the LLM.
- *Process:*

1. Search engine (e.g., Elasticsearch) returns documents in descending order of relevance.
2. The order is reversed, placing the most relevant document last.

- *Purpose:*
  - Exploits the recency bias of LLMs, which tend to focus more on the latest information in their context.
  - Ensures the most relevant information is "freshest" in the LLM's context window.
- *Example:* Original order: [Most Relevant, Second Most, Third Most, ...] Reversed order: [..., Third Most, Second Most, Most Relevant]

4. Query Classification

- A technique to optimize RAG system efficiency by determining whether a query requires RAG or can be answered directly by the LLM.
- *Process:*
  1. Develop a custom dataset specific to the LLM in use
  2. Train a specialized classification model
  3. Use the model to categorize incoming queries
- *Purpose:*
  - Improve system efficiency by avoiding unnecessary RAG processing
  - Direct queries to the most appropriate response mechanism

- *Requirements:*
  - LLM-specific dataset and model
  - Ongoing refinement to maintain accuracy
- *Benefits:*
  - Reduces computational overhead for simple queries
  - Potentially improves response time for non-RAG queries

5. Summarization

- A technique to condense retrieved documents in RAG systems.
- *Process:*
  1. Retrieve relevant documents
  2. Generate concise summaries of each document
  3. Use summaries instead of full documents in the RAG pipeline
- *Purpose:*
  - Improve RAG performance by focusing on essential information
  - Reduce noise and interference from less relevant content
- *Benefits:*
  - Potentially improves relevance of LLM responses
  - Allows for inclusion of more documents within context limits

- *Challenges:*
  - Risk of losing important details in summarization
  - Additional computational overhead for summary generation

6. Metadata Inclusion

- A technique to enrich documents with additional contextual information.
- *Types of metadata:*
  - Keyphrases
  - Titles
  - Dates
  - Authorship details
  - Blurbs
- *Purpose:*
  - Increase contextual information available to the RAG system
  - Provide LLMs with clearer understanding of document content and relevance
- *Benefits:*
  - Potentially improves retrieval accuracy
  - Enhances LLM's ability to assess document usefulness
- *Implementation:*
  - Can be done during document preprocessing

- May require additional data extraction or generation steps

7. Composite Multi-Field Embeddings

- An advanced embedding technique for RAG systems that creates separate embeddings for different document components.
- *Process:*
  1. Identify relevant fields (e.g., title, keyphrases, blurb, main content)
  2. Generate separate embeddings for each field
  3. Combine or store these embeddings for use in retrieval
- *Difference from standard approach:*
  - Traditional: Single embedding for entire document
  - Composite: Multiple embeddings for different document aspects
- *Purpose:*
  - Create more nuanced and context-aware document representations
  - Capture information from a wider variety of sources within a document
- *Benefits:*
  - Potentially improves performance on ambiguous or multi-faceted queries
  - Allows for more flexible weighting of different document aspects in retrieval

- *Challenges:*
  - Increased complexity in embedding storage and retrieval processes
  - May require more sophisticated matching algorithms

8. Query Enrichment

- A technique to expand the original query with related terms to improve search coverage.
- *Process:*
  1. Analyze the original query
  2. Generate synonyms and semantically related phrases
  3. Augment the query with these additional terms
- *Purpose:*
  - Increase the range of potential matches in the document corpus
  - Improve retrieval performance for queries with specific or technical language
- *Benefits:*
  - Potentially retrieves relevant documents that don't exactly match the original query terms
  - Can help overcome vocabulary mismatch between queries and documents
- *Challenges:*
  - Risk of query drift if not carefully implemented

- May increase computational overhead in the retrieval process

Back to top

Elasticsearch is packed with new features to help you build the best search solutions for your use case. Dive into our sample notebooks to learn more, start a free cloud trial, or try Elastic on your local machine now.

⚠ **Report an issue**

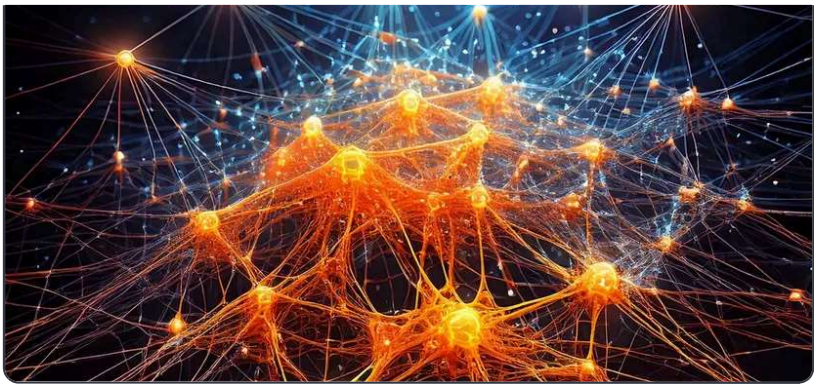
Related content

Vector Database | Generative AI | January 9, 2025

Improving e-commerce search with query profiles in Elastic

Query profiles tackle semantic search challenges in e-commerce. This blog demonstrates how to...

By: **Han Xiang Choong**

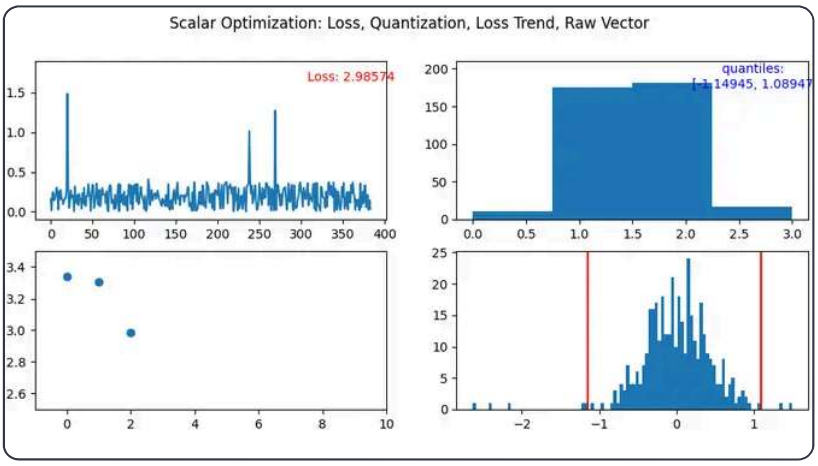


Vector Database Lucene +1 January 7, 2025

Early termination in HNSW for faster approximate KNN search

Learn how HNSW can be made faster for KNN search, using smart early termination strategies.

By: **Tommaso Teofili**



Lucene Vector Database January 6, 2025

Optimized Scalar Quantization: Even Better Binary Quantization

Here we explain optimized scalar quantization in Elasticsearch and how we used it to improve Better...

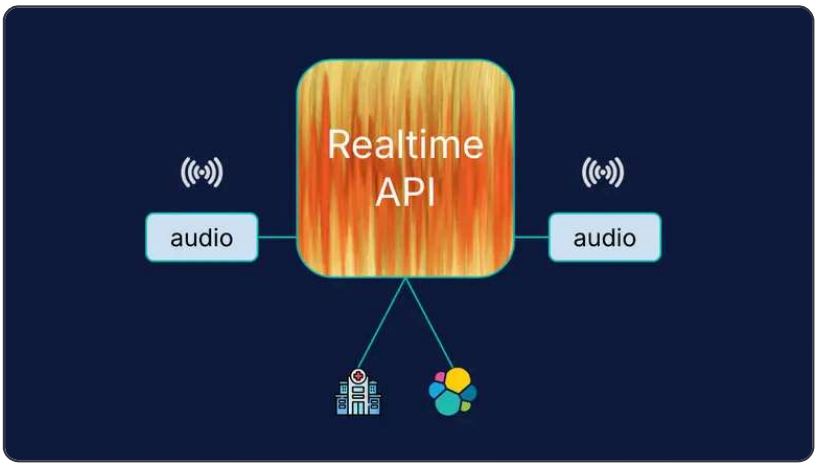
By: **Benjamin Trent**

Generative AI | Vector Database | +1 January 1, 2025

When hybrid search truly shines

Demonstrating when hybrid search is better than lexical or semantic search on their own.

By: **Gustavo Llermaly**



Vector Database December 30, 2024

“Hey Care!” - Speech-to-speech assistant powered by Elastic and OpenAI Realtime API

How we built an AI assistant for caregivers based on speech-to-speech interactions enriched using...

By: **Zing Zai**

Ready to build state of the art search experiences?

Sufficiently advanced search isn't achieved with the efforts of one. Elasticsearch is powered by data scientists, ML ops, engineers, and many more who are just as passionate about search as your are. Let's connect and work together to build the magical search experience that will get you the results you want.

Try it yourself

Subscribe to newsletter

elastic search labs

Elasticsearch Labs is the one-stop destination for developers to learn how to easily utilize Elasticsearch to build advanced search experiences including generative AI, embedding models, reranking capabilities and more.

MENU

Tutorials

Examples

Integrations

Blogs

Search

ADDITIONAL RESOURCES

Elasticsearch API Reference

Elastic.co

LET'S CONNECT

Sitemap RSS