💡 What Is Indexing in RAG?

Indexing = preparing your raw documents so they can be **searched and retrieved** later based on a user query. It's like building a search engine that feeds relevant chunks to an LLM.

You **DON'T** just throw documents at an LLM. You **preprocess, chunk, embed, and store** them smartly. That's called **indexing**.

📦 The 5 Stages of Indexing

Each step is essential. Skipping or half-doing any of these will kill your RAG system quality.

✅ Steps:

1. **Load documents**
2. **Preprocess (cleaning & formatting)**
3. **Chunk documents**
4. **Embed each chunk into a vector**
5. **Store in a vector database**

🛠️ Full Code-Based Guide (Step-by-Step)

We'll use:

- langchain for abstraction
- HuggingFaceEmbeddings for open-source embedding
- FAISS for local vector storage (super fast)

✅ 1. Load Raw Documents

```
from langchain.document_loaders import DirectoryLoader


# Load PDFs, TXTs, MDs, etc.
```

```
loader = DirectoryLoader(
    "./my_documents",  # folder with files
    glob="**/*.pdf",   # or **/*.txt, **/*.md, etc.
    show_progress=True
)

documents = loader.load()
print(f"Loaded {len(documents)} documents")
```

👉 **Each document** is a LangChain object:
Document(page_content="actual text", metadata={"source": "filename"})

✅ 2. Preprocess the Text (Optional but Smart)

You can clean up stuff like footers, navigation bars, empty spaces.

```
def cleanText(documents):
    cleaned = []
    for doc in documents:
        text = doc.page_content.strip().replace("\n", " ")
        doc.page_content = text
        cleaned.append(doc)
    return cleaned

documents = cleanText(documents)
```

✅ 3. Chunk the Documents

LLMs choke on large context. We split text into manageable overlapping chunks.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

```
textSplitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,      # tokens or characters
    chunk_overlap=200,    # overlap for context
)

chunks = textSplitter.split_documents(documents)
print(f"Generated {len(chunks)} chunks")
```

👉 Every chunk is still a Document object with its own text + metadata.

✅ 4. Create Embeddings (Vector Representation)

Use a pre-trained model to turn each chunk into a high-dimensional vector.

```
from langchain.embeddings import HuggingFaceEmbeddings

embeddingModel = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2"
)
```

This converts your string → vector like:

"How does login work?" → [0.23, -0.47, 0.81, ...]

✅ 5. Store Embeddings in Vector DB (Indexing)

We'll use FAISS (local) to build and save the index.

```
from langchain.vectorstores import FAISS

vectorStore = FAISS.from_documents(chunks, embeddingModel)
```

Save to disk for reuse

vectorStore.save_local("rag_index")

🔥 Your RAG system now has a searchable vector index ready to go.

💬 Retrieval Preview (How It's Used Later)

When user asks a question:

```
loadedIndex = FAISS.load_local("rag_index", embeddingModel)
query = "How to reset password?"

results = loadedIndex.similarity_search(query, k=3)

for doc in results:
   print(doc.metadata["source"])
   print(doc.page_content)
```

🧠 Full Picture of Indexing Pipeline

```
graph TD
A[Raw Documents] --> B[Load]
B --> C[Preprocess (Clean)]
C --> D[Chunking]
D --> E[Embedding]
E --> F[Vector Store (Index)]
```

🧩 Bonus: Metadata Handling

You can attach metadata during load or chunking:

Document(page_content="...", metadata={

```
  "source": "faq.pdf",
  "page": 4,
  "section": "Login"
})
```

This helps you later filter or re-rank based on section, file, date, etc.

💀 Things You Should NOT Do

- Don't use raw long documents directly → LLMs can't handle them.
- Don't skip chunk overlap → it breaks context between sections.
- Don't compute embeddings at retrieval time → do it once and cache.

✅ TL;DR — Final Checklist for Indexing in RAG

Step	What You Do
Load	Read PDFs, docs, etc
Preprocess	Clean up junk text
Chunk	Split into 500–1000 token segments with overlap
Embed	Convert each chunk to a dense vector
Store	Save all vectors + metadata in FAISS (or Pinecone, etc)
