L3-M6: Building a Simple Knowledge Base - Data Chunking, Embeddings, Vector Databases, and Knowledge Sources

Introduction: The Foundation of Retrieval-Augmented Generation (RAG)

The power of Large Language Models (LLMs) is undeniable, yet their knowledge is inherently limited by their training cutoff date and the specific corpus they were trained on. To bridge this gap and enable LLMs to interact with proprietary, up-to-date, or domain-specific information, the paradigm of **Retrieval-Augmented Generation (RAG)** has emerged as a critical architectural pattern [1]. A RAG system fundamentally relies on a robust and well-structured **Knowledge Base (KB)**. This module delves into the core technical components required to construct such a KB, focusing on the transformation of raw data into a searchable, semantic index.

The process of building a simple knowledge base involves four critical stages: 1. **Data Ingestion and Pre-processing:** Acquiring and cleaning raw data. 2. **Data Chunking:** Strategically dividing large documents into smaller, meaningful units. 3. **Embedding Generation:** Converting text chunks into high-dimensional numerical vectors. 4. **Vector Database Storage:** Indexing and storing these vectors for efficient similarity search.

This module will provide a detailed, technical examination of the latter three stages, which form the backbone of the semantic search capability essential for RAG.

Chapter 1: Data Chunking Strategies for Semantic Retrieval

Data Chunking is the process of breaking down large documents, such as PDFs, articles, or code repositories, into smaller, manageable segments, or "chunks." This step is paramount in RAG because the size and quality of the retrieved context directly impact the LLM's ability to generate an accurate and relevant response. An overly large chunk can introduce irrelevant noise, while an overly small chunk can lose necessary context.

1.1 The Chunking Dilemma: Context vs. Noise

The goal of chunking is to find the optimal balance between **context preservation** and **noise reduction**. A good chunk should be self-contained and semantically coherent, meaning it can be understood without excessive reliance on surrounding text, while remaining small enough to be efficiently processed by both the embedding model and the LLM's context window.

1.2 Primary Chunking Methodologies

Several methodologies exist for chunking, each with its own trade-offs in terms of implementation complexity and retrieval performance.

Chunking Strategy	Description	Advantages	Disadvantages
Fixed-Size Chunking	Splits text into segments of a predefined character or token count, often with a small overlap.	Simple to implement; predictable chunk size.	Often breaks semantic boundaries; can split a single sentence or idea.
Document-Based Chunking	Splits based on inherent document structure (e.g., paragraphs, sentences, sections, chapters).	Preserves natural semantic boundaries; high coherence.	Chunk sizes are highly variable; requires structured documents (e.g., Markdown, HTML).
Recursive Chunking	Attempts to split text using a sequence of delimiters (e.g., `\n\n`, `\n`, `.`,), recursively trying smaller delimiters if a chunk is still too large.	Highly effective at preserving structure; adapts to different text formats.	More complex to implement; requires careful ordering of delimiters.
Semantic Chunking	Uses an embedding model to identify semantic shifts in the text, clustering similar sentences into chunks.	High semantic coherence; chunks are highly relevant to the core idea.	Computationally expensive (requires an initial embedding pass); slower ingestion.
Parent-Child Chunking	Stores two versions of the data: small chunks for retrieval and larger "parent" chunks for context passed to the LLM.	Combines precise retrieval with rich context for generation.	Requires double the storage; more complex retrieval logic.

1.3 Implementing Recursive Character Text Splitter

The **Recursive Character Text Splitter** is a widely adopted and effective technique. It operates by iterating through a list of separators, attempting to split the text on the first separator. If the resulting chunks are still too large, it recursively applies the next separator in the list to those chunks.

```python
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Define a list of separators, ordered from largest to smallest semantic break
separators = [
    "\n\n",  # Paragraphs
    "\n",    # Newlines
    " ",     # Spaces
    "",      # Characters
]

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    separators=separators,
    length_function=len,
    is_separator_regex=False,
)

# Example: split a large document
# chunks = text_splitter.create_documents([long_text])
```

The `chunk_overlap` parameter is crucial. Overlap ensures that the context from the end of one chunk is carried over to the beginning of the next, preventing the loss of information at the chunk boundaries.

Chapter 2: Embeddings: The Language of Vectors

Once the raw data is segmented into coherent chunks, the next step is to transform this symbolic data (text) into a numerical format that can be mathematically processed. This is the role of **Embeddings**.

2.1 Definition and Purpose of Text Embeddings

A **text embedding** is a dense, low-dimensional vector of floating-point numbers that represents the semantic meaning of a piece of text (a word, sentence, or chunk). The key property of embeddings is that texts with similar meanings are mapped to vectors that are close to each other in the high-dimensional vector space.

This proximity is typically measured using a distance metric, most commonly **Cosine Similarity**.

$$\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

A cosine similarity value close to 1 indicates high semantic similarity, while a value close to 0 indicates low similarity.

2.2 The Role of Embedding Models

Embeddings are generated by specialized **Embedding Models**, which are typically deep neural networks (often transformer-based) trained on massive text corpora. These models learn to map text to a vector space where semantic relationships are preserved.

Embedding Model Characteristics	Description
Dimensionality	The length of the resulting vector (e.g., 384, 768, 1536). Higher dimensions often capture more nuance but increase storage and computational cost.
Context Window	The maximum input size the model can process to generate a single embedding. Directly impacts the maximum effective chunk size.
Training Data	The corpus the model was trained on (e.g., general web data, scientific papers, code). Determines the model's domain expertise.
Performance Metrics	Evaluated on benchmarks like the Massive Text Embedding Benchmark (MTEB), which measures retrieval and classification accuracy.

For example, a model might map the phrase "a large canine" and "a big dog" to vectors that are closer together than the vector for "a small cat." This numerical representation is what allows the RAG system to perform a **semantic search** rather than a simple keyword match.

Chapter 3: Vector Databases: The Semantic Index

Once the data chunks are converted into vectors, they must be stored and indexed in a way that allows for extremely fast and efficient similarity search. This is the function of a **Vector Database (Vector DB)**.

3.1 Key Differences from Traditional Databases

Traditional relational databases (SQL) and NoSQL databases are optimized for exact matches and structured queries (e.g., `SELECT * FROM users WHERE age > 30`). Vector

databases, conversely, are optimized for **Approximate Nearest Neighbor (ANN)** search, which is essential for finding the most semantically similar vectors to a given query vector.

Feature	Traditional Database (e.g., PostgreSQL)	Vector Database (e.g., Pinecone, Weaviate)
Primary Indexing	B-trees, Hash tables	Hierarchical Navigable Small Worlds (HNSW), Inverted File Index (IVF)
Query Type	Exact match, structured query (SQL)	Similarity search (ANN), vector operations
Data Type	Structured records (rows, columns, JSON)	High-dimensional vectors (embeddings)
Optimization Goal	Transaction speed, data integrity	Low-latency similarity search, high recall

3.2 Approximate Nearest Neighbor (ANN) Algorithms

Since exact nearest neighbor search in high-dimensional space is computationally prohibitive (the "curse of dimensionality"), Vector DBs rely on ANN algorithms. These algorithms sacrifice a small amount of accuracy (recall) for massive gains in speed (latency).

The most popular ANN algorithm is **Hierarchical Navigable Small Worlds (HNSW)**. HNSW builds a multi-layer graph structure where: 1. The top layers contain fewer nodes, acting as "express lanes" for long-distance searches. 2. The bottom layers contain all data points, allowing for precise local searches.

A query vector navigates this graph, starting at the top layer and moving down, quickly finding the neighborhood of the closest vectors.

3.3 Vector Database Components

A functional Vector DB typically manages three pieces of information for each entry: 1. **The Vector (Embedding):** The numerical representation of the chunk. 2. **The Metadata:** Structured information about the chunk (e.g., source document name, author, date, section title). This is used for pre-filtering (e.g., "only search documents

from 2024"). 3. **The Original Text:** The raw text chunk itself, which is retrieved and passed to the LLM.

Chapter 4: Creating Knowledge Sources: The Ingestion Pipeline

The final step in building the knowledge base is the design and execution of the **Ingestion Pipeline**, which orchestrates the processes described in the previous chapters. This pipeline transforms raw documents into the structured vector index.

4.1 Step-by-Step Ingestion Process

The ingestion pipeline can be broken down into a series of distinct, sequential operations:

Step	Operation	Technical Component	Output
1. Load	Read and parse the raw data from its source (e.g., file system, cloud storage, API).	Document Loaders (e.g., LangChain DocumentLoaders)	A list of `Document` objects.
2. Split	Apply a chunking strategy to break the documents into smaller, semantically coherent pieces.	Text Splitters (e.g., RecursiveCharacterTextSplitter)	A list of text chunks.
3. Embed	Pass each text chunk through the chosen embedding model to generate a vector representation.	Embedding Model (e.g., OpenAI, Cohere, open-source models)	A list of high-dimensional vectors.
4. Store	Insert the vector, the original text chunk, and any associated metadata into the Vector Database.	Vector Database Client (e.g., Pinecone, ChromaDB, Weaviate client)	A searchable vector index.

4.2 Practical Application: Building a Simple KB with Open-Source Tools

To illustrate, consider a simple Python-based ingestion script using popular open-source libraries:

```
 # Assuming necessary libraries are installed: pip install langchain openai
pydantic chromadb

import os
from langchain.document_loaders import TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma

# Configuration (replace with your actual API key and data path)
# os.environ["OPENAI_API_KEY"] = "YOUR_API_KEY"
DATA_PATH = "./source_documents/technical_manual.txt"

# 1. Load the document
loader = TextLoader(DATA_PATH)
documents = loader.load()

# 2. Split the document
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=100
)
chunks = text_splitter.split_documents(documents)

print(f"Original document split into {len(chunks)} chunks.")

# 3. Embed and 4. Store
# Initialize the embedding model (using a hypothetical model for demonstration)
embeddings = OpenAIEmbeddings()

# Initialize and persist the vector store (using ChromaDB for local storage)
vector_db = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings,
    persist_directory="./chroma_db"
)

print("Knowledge Base ingestion complete. Vector index created.")
```

This script demonstrates the creation of a persistent vector index (`./chroma_db`) that can be queried later by a RAG application. The combination of the `TextLoader` , `RecursiveCharacterTextSplitter` , and the `Chroma` vector store encapsulates the entire KB creation process.

Conclusion: The Semantic Bridge

Building a simple knowledge base is a foundational skill in the field of advanced AI applications, particularly for implementing effective RAG systems. The process is a disciplined exercise in data engineering and semantic representation.

The key takeaways from this module are: * **Chunking is Context Engineering:** The choice of chunking strategy (e.g., Fixed-Size, Recursive, Parent-Child) directly determines the quality of the context retrieved, balancing the need for semantic coherence with the constraints of the LLM's context window. * **Embeddings are Semantic Maps:** Text embeddings transform symbolic language into a mathematical space where proximity equals semantic similarity, enabling the core functionality of RAG. * **Vector Databases are Specialized Indices:** Vector DBs utilize specialized indexing algorithms like HNSW to perform low-latency Approximate Nearest Neighbor searches, making the knowledge base practical for real-time applications.

Mastery of these concepts allows practitioners to transform raw, unstructured data into a powerful, searchable asset, significantly enhancing the utility, accuracy, and relevance of Large Language Model applications.

References

[1] Lewis, P., Perez, E., Piktus, A., Petroni, F., Piktus, A., Petroni, F., ... & Kiela, D. (2020). **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks**. *Advances in Neural Information Processing Systems*, 33, 9459-9474. https://arxiv.org/abs/2005.11401

[2] Databricks Community. (2025). **Mastering Chunking Strategies for RAG**. https://community.databricks.com/t5/technical-blog/the-ultimate-guide-to-chunking-strategies-for-rag-applications/ba-p/113089

[3] Stack Overflow Blog. (2023). **An intuitive introduction to text embeddings**. https://stackoverflow.blog/2023/11/09/an-intuitive-introduction-to-text-embeddings/

[4] Microsoft Learn. (2024). **Understanding Vector Databases**. https://learn.microsoft.com/en-us/data-engineering/playbook/solutions/vector-database/

[5] Weaviate. (2025). **Chunking Strategies to Improve Your RAG Performance**. https://weaviate.io/blog/chunking-strategies-for-rag