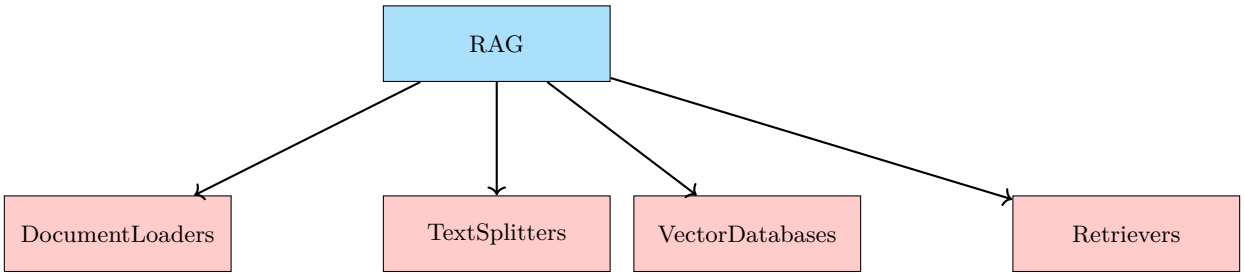
Retrieval Augment Generation

RAG is a technique that combines information retrieval with language generation, where a model retrieves relevant documents from a knowledge base and then uses them as context to generate accurate and grounded responses.

Benefits of using RAG

1. Use of up-to-date information
2. Better privacy
3. No limit of document size



1 Document Loaders

Document loaders are components in LangChain used to load data from various sources into a standardized format (usually as `Document` objects), which can then be used for chunking, embedding, retrieval, and generation.

```
Document(
    page_content="The actual text content",
    metadata={"source": "filename.pdf", ...}
)
```

`page_content`: This field contains the main textual data that has been extracted from the source document. It represents the actual content that will be used for downstream tasks such as chunking or embedding.

`metadata`: This is a dictionary that holds additional information about the document, such as its source, filename, author, or other descriptive details. It helps provide context but is not used directly as the main content.

1.1 LangChain's TextLoader

```python
from langchain_community.document_loaders import TextLoader

loader = TextLoader('cricket.txt', encoding='utf-8')

docs = loader.load()

print(type(docs))

print(len(docs))

print(docs[0])
```

Why Use TextLoader Instead of Raw Text?

- **Standardized Document Format:** This format is required by most downstream components in LangChain (e.g., text splitters, retrievers, vector stores).
- **Automatic Metadata Attachment:** It automatically attaches useful metadata (e.g., source filename, file path, etc.) to each document. This is useful for tracing, debugging, or retrieval filtering.

1.2 PyPDF

```python
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader('dl-curriculum.pdf')

docs = loader.load()

print(docs)
```

Limitation
While `PyPDFLoader` is effective for extracting text from digitally-created PDF files, it has a significant limitation when dealing with **scanned PDFs**. Scanned PDFs often contain images of text rather than actual text data, which means `PyPDFLoader` cannot extract meaningful content from them. In such cases, Optical Character Recognition (OCR) tools like `Tesseract` or libraries such as `unstructured` or `pdfplumber` are more appropriate.

1.3 Recommended PDF Loaders by Use Case

Use Case	Recommended Loader
Simple, clean PDFs	`PyPDFLoader`
PDFs with tables/columns	`PDFPlumberLoader`
Scanned/image PDFs	`UnstructuredPDFLoader` or `AmazonTextractPDFLoader`
Need layout and image data	`PyMuPDFLoader`
Want best structure extraction	`UnstructuredPDFLoader`

1.4 Directory Loader

Directory Loader is a document loader that lets you load multiple documents from a directory (folder or files)

Glob Pattern	What It Loads
`"**/*.txt"`	All `.txt` files in all subfolders
`"*.pdf"`	All `.pdf` files in the root directory
`"data/*.csv"`	All `.csv` files in the `data/` folder
`"**/*"`	All files (any type, all folders)

** = recursive search through subfolders

```python
from langchain_community.document_loaders import DirectoryLoader, PyPDFLoader

# Load all PDF files in the 'books' directory
loader = DirectoryLoader(
    path='books',
    glob='*.pdf',
    loader_cls=PyPDFLoader
)

docs = loader.load()

# Display the content of the first document
print(docs[0].page_content)
```

1.4.1 Difference Between load() and lazy_load()

`load()` immediately loads and returns all documents as a list. It's suitable for small or moderate amounts of data.

`lazy_load()` returns a generator instead of a list. This is useful when processing large datasets, as it avoids loading everything into memory at once.

Analogy:
Imagine you go to a library to read 100 books.

- `load()`: You carry all 100 books to your table at once. You now have access to all books immediately, but it's heavy and takes a lot of space.
- `lazy_load()`: Instead of carrying all at once, you ask the librarian to bring one book at a time as you finish each one. This takes less space and memory, but is slightly slower if you need many books quickly.

When to use:

- Use `load()` for small to medium datasets where speed and immediate access are important.
- Use `lazy_load()` for large collections where memory efficiency is a priority.

1.5 WebBase Loader

WebBaseLoader is a document loader in LangChain used to load and extract text content from web pages (URLs). It uses Beautiful Soup under the hood to parse HTML and extract visible text. **Limitations:**

Doesn't handle JavaScript-heavy pages well

```python
from langchain_community.document_loaders import WebBaseLoader

loader = WebBaseLoader("https://en.wikipedia.org/wiki/Artificial_intelligence
    ↪ ")
docs = loader.load()

print(docs[0].page_content)
```

1.6 CSV Loader

CSVLoader is used to load tabular data from a `.csv` file into LangChain as a list of `Document` objects. Each row in the CSV is typically treated as a separate document.

Example Usage:

```python
from langchain_community.document_loaders import CSVLoader

loader = CSVLoader(file_path='sample.csv')

docs = loader.load()

print(docs[0].page_content)
```

2 Text Splitter

2.1 Length Based Text Splitter

Length-Based Text Splitter is a utility in LangChain that splits large pieces of text into smaller chunks based on a fixed character length. This is useful when dealing with models that have context window limits, such as LLMs that can only process a few thousand tokens at a time.

This splitter does not consider semantic structure like sentences or paragraphs — it simply splits the text wherever the maximum character limit is reached.

Key Parameters:

- `chunk_size` – the maximum size of each text chunk.
- `chunk_overlap` – number of characters that overlap between consecutive chunks.

```python
from langchain_text_splitters import CharacterTextSplitter

text_splitter = CharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50,
    length_function=len
)

chunks = text_splitter.split_text("your long text goes here...")

print(chunks[0])
```

2.2 Text Structure Based

Text Structure Based Splitter, such as the `RecursiveCharacterTextSplitter`, attempts to preserve the natural structure of the text while splitting. It works in a hierarchical manner:

- First, it tries to split by paragraphs (e.g., using double line breaks).
- If the chunk is still too large, it tries to split by single lines.
- If necessary, it falls back to splitting by characters to ensure the chunk meets the specified token or character length.

This method is ideal when preserving the logical flow and context of the text is important, such as in articles, documents, or essays.

```python
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50
)

chunks = text_splitter.split_text("your long, structured document text")

print(chunks[0])
```

2.3 Document-Structure Based Text Splitting

Document-Structure Based Text Splitting is designed for content that does not follow natural language structure—such as source code, configuration files, or markdown documents.

Unlike standard text splitting, this method uses document-aware logic to chunk content based on syntactic elements rather than sentence or paragraph boundaries.

Common Use Cases:

- Programming code (Python, Java, etc.)
- Markdown files
- Logs or configuration files

Separators Used: The splitter uses a predefined list of separators in decreasing priority, such as:

- `"\n\n"` — double newlines (e.g., block-level)
- `"\n"` — single newlines (line-by-line)
- `" "` — spaces
- `""` — as last fallback (individual characters)

These separators ensure that content is split as logically as possible while meeting chunk size constraints.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

code_splitter = RecursiveCharacterTextSplitter(
    language = language.PYTHON,
    chunk_size=300,
    chunk_overlap=30,
)

chunks = code_splitter.split_text("def my_function():\n    print('Hello World
    ↪ ')\n# more code here...")

print(chunks[0])
```

2.4 Semantic Meaning Based

Semantic Meaning Based Text Splitting uses language models or embeddings to split text based on its **meaning**, rather than raw character count or structure.

This method identifies semantically coherent chunks by embedding sections of the text and grouping similar ideas together. It is particularly useful when working with:

- Long, unstructured documents (e.g., academic papers, reports)
- Dialogue or conversational transcripts
- Content where context continuity is critical

Instead of blindly chopping text at a fixed length or at newlines, this method aims to keep related sentences or concepts together in a chunk for better downstream understanding.

Example Usage:

```
from langchain_text_splitters import SemanticChunker
from langchain.embeddings import OpenAIEmbeddings

# Initialize embedding model
embeddings = OpenAIEmbeddings()

# Initialize semantic splitter
splitter = SemanticChunker(embeddings)

chunks = splitter.split_text("Long paragraph of meaningful content goes here
    ↪ ...")

print(chunks[0])
```

3 Vector Stores

A vector store is a system designed to store and retrieve data represented as numerical vectors.

Key Features:

- **Storage -** Ensures that vectors and their associated metadata are retained, whether in-memory for quick lookups or on-disk for durability and large-scale use.
- **Similarity Search -** Helps retrieve the vectors most similar to a query vector.
- **Indexing -** Provide a data structure or method that enables fast similarity searches on high-dimensional vectors.

Use Cases

- Semantic Search
- RAG
- Recommender Systems
- Image or Multimedia Search

3.1 Vector Database

A full-fledged database system designed to store and query vectors. Offers additional "database-like" features:

- Distributed architecture for horizontal scaling
- Durability and persistence (replication, backup or restore)
- Metadata handling (schemas, filters)
- Potential for ACID or near ACID guarentees

3.2 Code

The Code is available in a separate Jupyter Notebook (Vector_stores.ipynb)