Deep Learning Application

Chapter 2. Pipeline

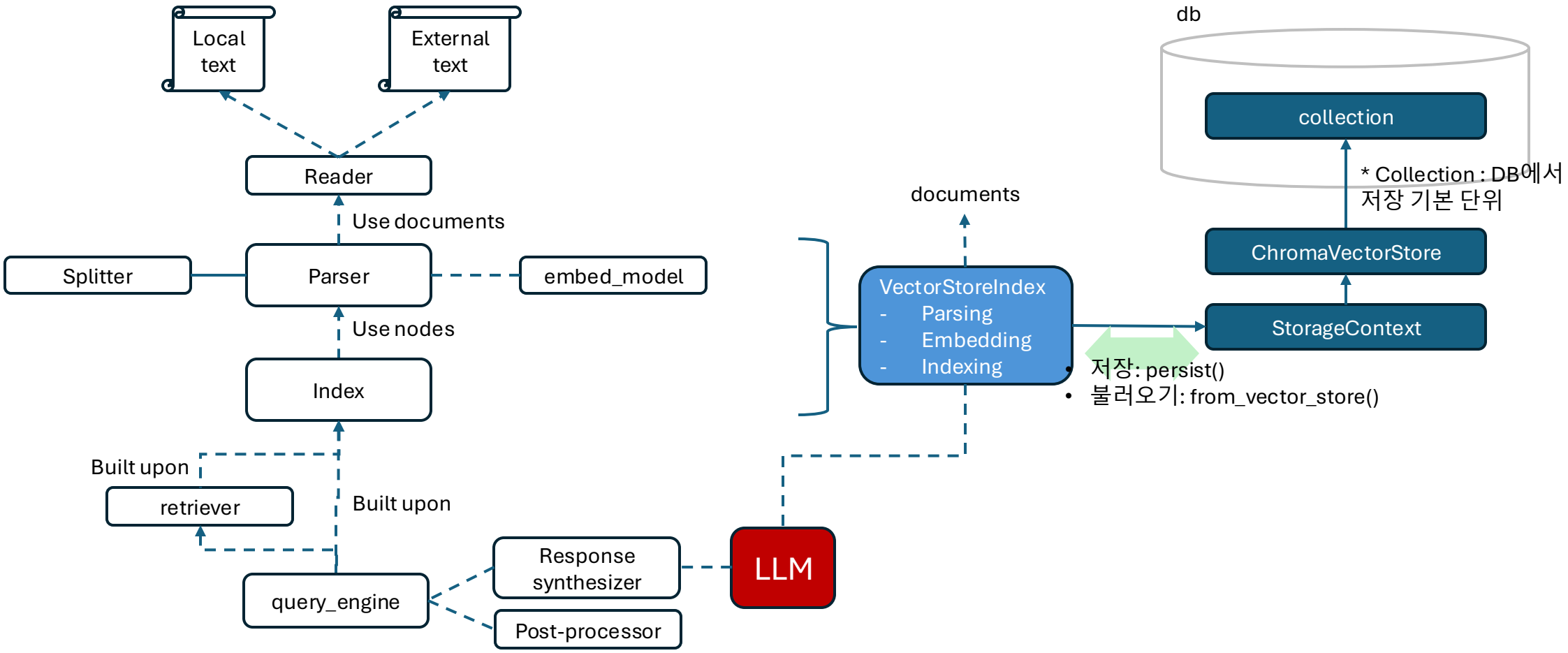
Pipeline

- 데이터 로딩(Data Loading):
  JSON, CSV, PDF와 같은 다양한 형식의 데이터를 로드하고, 초기 전처리 를 수행
- 텍스트 분할(Text Spliting):
  - 로드된 문서를 LLM에 최적화된 크기로 나눔.
  - 문장 단위, 토큰 단위, 의 미 기반 분할 방식을 활용하여 문서의 컨텍스트를 유지하면서도 검색 효율성을 높임.
  - 문서 객체는 청크(Chunk) 단위로 나뉘며, 이후 노드(Node) 객체가 생성. 생성된 노드는 검색, 인덱 싱, 질의 처리 과정에서 활용.
- 인덱싱(Indexing):
  - 벡터 데이터베이스에 데이터를 저장하기 위한 인덱스를 생성.
  - 생성된 인덱스는 검색(Retrieval) 및 유사도 기반 질의 처리에 사용.
- 저장(Storing):
  - 생성된 인덱스를 영구적으로 저장하거나 Pinecone, Weaviate 등 외부 벡터 스토어와 연 동하여 확장성을 높이고 데이터 관리를 지원.
- 쿼리(Query):
  - 사용자의 질의를 처리하고 관련 데이터를 검색. LLM을 활용하여 자연어 형태로 입력된 쿼리의 의미를 정교하게 해석.
- 검색(Retrieval):
  - 인덱스를 통해 가장 관련성 높은 결과를 검색. 검색된 결과는 추가적인 LLM 기반 후처리를 거쳐 더욱 풍부한 응답으로 정제.

Pipeline - Example

- 데이터 로딩 :
  - 영화 리뷰 데이터를 로드. 리뷰 데이터가 JSON 형식으로 구성되어 있다고 가정.
  - 각 리뷰는 문서 객체로 변환. 예를 들어. "이 영화는 매우 지루했습니다. 줄거리가 너무 느렸 어요."라는 리뷰는 하나의 문서 객체로 처리.
- 텍스트 분할
  - 리뷰를 문장 단위로 분할(chunking)하여 각각을 노드 객체로 변환. 예를 들어, 위 리뷰는 다음과 같이 두 개의 노드.
  - Node 1: "이 영화는 매우 지루했습니다."
  - Node 2: "줄거리가 너무 느렸어요."
- 인덱싱
  - 생성된 노드들을 벡터화(vectorization)하여 벡터 데이터베이스에 저장. 이 데이터베이스는 각 문 장의 의미를 기반으로 유사한 내용을 검색할 수 있도록 설계.
- 저장
  - 생성된 벡터 데이터를 영구적으로 저장하도록 구성.
- 질의와 답변 생성
  - 사용자가 "이 영화는 지루한가요?"라는 질문을 입력하면, 벡터 데이터베이스가 '지루함'과 관련된 노드를 검색,
  - "이 영화는 매우 지루했습니다."라는 검색된 문장을 기반으로 LLM은 다음과 같은 답변을 생성. "네, 이 영화는 매우 지루하다고 평가되었습니다."

Visual summary



가상환경구축

- Venv
- Pipenv
- ...

2.2 데이터로딩

- 데이터 리더
  - 파일시스템, 로컬 저장소
  - llama_index.core.SimpleDirectoryReader
- 데이터 커넥터 (?)
  - 다양한 데이터 소스에서 정보를 가져오는 역할
  - 데이터베이스, API, 클라우드스토리지 등과 같은 외부소스
  - llama_index.reader.database.DatabaseReader

Local sources

Reader (class)	What it loads	Official reference
`SimpleDirectoryReader`	Files from a **local** folder; auto-picks file readers by extension	(docs.llamaindex.ai)
`SimpleDirectoryReader` over remote FS	Same API, but mounts a **remote** filesystem (e.g., S3) and reads as if local	(docs.llamaindex.ai)

External/networked sources

Reader (class)	Source type	Official reference
`SimpleWebPageReader`	Web pages / URLs (HTML → text/Markdown)	(docs.llamaindex.ai)
`DatabaseReader`	SQL databases via SQLAlchemy / SQLDatabase	(docs.llamaindex.ai)
`S3Reader`	**Amazon S3** buckets/keys (files or whole prefixes)	(docs.llamaindex.ai)
`GoogleDriveReader`	**Google Drive** files/folders	(docs.llamaindex.ai)
`GoogleDocsReader`	**Google Docs** documents	(docs.llamaindex.ai)
`NotionPageReader`	**Notion** pages / databases	(docs.llamaindex.ai)
`GithubRepositoryReader`	**GitHub** repository contents	(docs.llamaindex.ai)
`SlackReader`	**Slack** channel conversations (time-bounded)	(docs.llamaindex.ai)
`ConfluenceReader`	**Atlassian Confluence** spaces/pages	(docs.llamaindex.ai)
`MicrosoftSharePointReader`	**SharePoint** folders/doc libraries	(docs.llamaindex.ai)
`GCSReader`	**Google Cloud Storage** buckets/objects	(docs.llamaindex.ai)
`AzStorageBlobReader`	**Azure Blob Storage** containers/blobs	(docs.llamaindex.ai)
`BoxReader*`	**Box** files (includes AI-Extract variants)	(docs.llamaindex.ai)

2.2.1 Data Reader

- SimpleDirectoryReader
  - 지정된 폴더에서 파일을 읽어 문서 (Document) 객체로 변환하는 가장 기본적인 데이터 리더
  - 마크다운, PDF, 워드 문서 등 다양한 형식을 지원하며, 이미지와 오디오 파일도 처리할 수 있음.

- [실습] “/ch2/sample_docs”
  - * “.docx” 파일(워드파일) 을 읽기위해서는 Parser가 필요
  - ”pipenv install docx2txt”

```
ch02
└── sample_docs/
    ├── file1.txt       # 텍스트 파일
    ├── file2.docx      # 워드 문서
    ├── file3.pdf       # PDF 문서
    └── sub_sample_docs/
        ├── file4.txt   # 텍스트 파일
        └── file5.md    # 마크다운 파일
```

```python
from llama_index.core import SimpleDirectoryReader

documents = SimpleDirectoryReader("sample_docs").load_data()
```

```python
# Document 리스트의 각 요소에 접근하여 내용 출력
for i, document in enumerate(documents):
    print(f"Document {i+1}:")
    print(document.text)
    print("\n")
```

✓ 0.0s

2.2.1 Data Reader

```
from llama_index.core import SimpleDirectoryReader

documents = SimpleDirectoryReader("sample_docs").load_data()
```

- Documents 리스트에는 여러개의 document 객체가 있고, document 객체의 text 속성에 파일 내용이 할당

```
# Document 리스트의 각 요소에 접근하여 내용 출력
for i, document in enumerate(documents):
    print(f"Document {i+1}:")
    print(document.text)
    print("\n")
```

✓ 0.0s

```
[
    Document(
        id_='54ba572e-9570-4da3-b4d8-62a53f4218a3',
        embedding=None,
        metadata={'file_path': '/Users/raycho-mbp/my_code/LlamaIndex/llama-index-tutorial-ma
        excluded_embed_metadata_keys=['file_name', 'file_type', 'file_size', 'creation_date
        excluded_llm_metadata_keys=['file_name', 'file_type', 'file_size', 'creation_date',
        relationships={},
        metadata_template='{key}: {value}',
        metadata_separator='\n',
        text_resource=MediaResource(
            embeddings=None,
            data=None,
            text='This is the content of file 1.',
            path=None,
            url=None,
            mimetype=None),
        image_resource=None,
        audio_resource=None,
        video_resource=None,
        text_template='{metadata_str}\n\n{content}'),
    Document(id_='720960e5-f930-4a9f-bed3-d7670a6c42d2', embedding=None, metadata={'file_nam
    Document(id_='124d231f-d07c-46a4-b55f-58948c58fb09', embedding=None, metadata={'page_lab
]
```

2.2.1 Data Reader

- 하위폴더 포함

```
documents = SimpleDirectoryReader(
    "sample_docs", recursive=True).load_data()
```

✓ 0.0s

```
# Document 리스트의 각 요소에 접근하여 내용 출력
for i, document in enumerate(documents):
    print(f"Document {i+1}:")
    print(document.text)
    print("\n")
```

2.2.1 Data Reader

- 하위폴더 포함 : recursive

```
documents = SimpleDirectoryReader(
    "sample_docs", recursive=True).load_data()
```

✓ 0.0s

```
# Document 리스트의 각 요소에 접근하여 내용 출력
for i, document in enumerate(documents):
    print(f"Document {i+1}:")
    print(document.text)
    print("\n")
```

- 특정 파일 형식만 로드하기

```
documents = SimpleDirectoryReader(
    "sample_docs",
    required_exts=[".txt", ".pdf"],
    recursive=True).load_data()
```

2.2.1 Data Reader

- 문서 객체의 메타데이터 확인
  - Metadata : 데이터에 대한 추가 정보, 파일경로, 파일명, 크기, 생성 일시 등

```python
# document 리스트의 각 요소에 접근하여 메타데이터를 출력
for i, document in enumerate(documents):
    print(f"Document {i+1} metadata:")
    print(document.metadata)
    print("\n")
```

```
Document 1 metadata:
{
    'file_path': '/external/llama-index-tutorial/ch02/sample_docs/file1.txt',
    'file_name': 'file1.txt',
    'file_type': 'text/plain',
    'file_size': 30,
    'creation_date': '2025-07-04',
    'last_modified_date': '2025-07-04'
}
```

2.2.2 Data Connector

- 데이터 커넥터 :
  - 외부 데이터를 불러오는데 역할
  - 라마인덱스에서는 기본적으로 제공되는 커넥터 외에도, 확장 가능한 다양한 커넥터 생태계를 지원
  - 특히 외부 데이터 소스와의 연동이 필요한 경우에는 별도의 커넥터를 활용하여 연결을 구성
- 라마허브의 커넥터 사용
  - 라마인덱스는 다양한 데이터 커넥터를 기본적으로 지원
  - 라마인덱스에 내장된 데이터 커넥터만으로 실제 활용에 한계가 있을 때는 라마허브 (LlamaHub)에서 제공하는 추가 커넥터를 활용
  - 라마허브: 데이터 커넥터 레지스트리
- [실습] ch02/ch02_practice.ipynb
  - 데이터베이스 리더(DatabaseReader)라는 커넥터를 내려받고 설치
  - 데이터베이스 리더는 SQL 데이터베이스에 쿼리를 실행하고, 결과의 모든 행을 문서로 반환하는 커넥터

2.2.2 Data Connector

```
!pipenv install llama-index-readers-database==0.3.0 -q
```

```
!pipenv install pymysql
```

```python
from sqlalchemy import create_engine
from llama_index.readers.database import DatabaseReader

# MySQL 연결 정보 직접 입력
scheme = "mysql+pymysql"
host = "localhost"
password = "llamaindex"
port = "3306"
user = "root"
dbname = "test_db"

connection_string = f"{scheme}://{user}:{password}@{host}:{port}/{dbname}"
engine = create_engine(connection_string)
reader = DatabaseReader(sql_database=engine)

# 데이터 로드
query = "SELECT * FROM users"
documents = reader.load_data(query=query)

# documents 리스트의 각 요소 출력
for idx, doc in enumerate(documents):
    print(f"Document {idx + 1}:")
    print(f"ID {doc.id_}")
    print(f"Row {doc.text}")
    print("-" * 50)
```

✓ 0.0s

```
Document 1:
ID d442fcdc-cbcd-4228-acb2-517b3d65f305
Row id: 1, name: 홍길동, email: hong@example.com, age: 30
--------------------------------------------------
Document 2:
ID 1be4898c-dedc-435f-9946-3c5dc5c3f254
Row id: 2, name: 김철수, email: kim@example.com, age: 25
--------------------------------------------------
```

Script.sh

```bash
# 터미널에서 실행하세요.

# 실행 권한 부여 : chmod +x ch02/mysql_setup.sh
# 스크립트 실행: ./ch02/mysql_setup.sh

# 1. Homebrew로 MySQL 설치 (이미 설치되어 있다면 생략)
echo "[1] Homebrew로 MySQL 설치 중..."
brew list mysql &>/dev/null || brew install mysql

# 2. MySQL 서버 실행
echo "[2] MySQL 서버 실행 중..."
brew services list | grep mysql | grep started &>/dev/null || brew services start mysql

# 3. MySQL 데이터베이스 및 테이블 생성, 데이터 삽입
echo "[3] DB 및 테이블 생성, 데이터 삽입 중..."

mysql -u root -p'llamaindex' <<EOF
CREATE DATABASE IF NOT EXISTS test_db;
USE test_db;
CREATE TABLE IF NOT EXISTS users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    email VARCHAR(100),
    age INT
);
INSERT INTO users (id, name, email, age) VALUES (1, '홍길동', 'hong@example.com', 30);
INSERT INTO users (id, name, email, age) VALUES (2, '김철수', 'kim@example.com', 25);
EOF

echo "모든 작업 완료."

# brew services stop mysql # background에서 실행 중인 MySQL 서버를 중지
# ps aux | grep mysql
# sudo kill -9 58477sudo chown -R $(whoami) /opt/homebrew/var/mysql
```

2.3 텍스트 분할
2.3.1 문서(Document)와 노드(Node)

- 문서를 효과적으로 관리하고 검색하기위해 텍스트를 적절한 크기로 분할, 검색 및 인덱싱 성능도 향상
- 라마인덱스는 문서를 여러 개의 노드로 분할하여 저장하고 분석하도록 설계
- 문서와 노드는 라마인덱스에서 데이터를 구조화하고 관리하는 핵심 요소
  - Document: 원 시 데이터를 처리 가능한 형태로 변환한 데이터의 기본 단위
  - Node: 문서를 더 작은 단위로 세분화하여 검색 및 분석할 수 있는 기본 단위
- Document
  - PDF, 텍스트 파일, 데이터베이스 쿼리 결과 등 다양한 형태의 데이터를 문서 객체로 변환
- [실습] ch02_practice.ipynb
  - 영화 기생충의 줄거리를 담은 텍스트를 문서 객체로 생성하고, 메타데이터를 추가 하는 예제

```python
from llama_index.core import Document

# 텍스트 데이터를 기반으로 문서 생성
document_text = "영화 '기생충'은 가난한 가족인 기우네가 부유한 박 사장네 집에 하나씩 취업하면서 벌어지는 이야
document = Document(text=document_text)

# 메타데이터 추가
document.metadata = {'author': '영화 해설', 'subject': '기생충 줄거리'}

print(document)
```

2.3 텍스트 분할
2.3.1 문서(Document)와 노드(Node)

- 노드(Node)
  - 노드는 문서를 더 작은 단위로 나눈 데이터 요소, 주로 텍스트 청크(chunk)나 이미지 패치(patch)와 같은 개별적이고 독립적인 정보 단위
  - 문서를 여러 개의 노 드로 분할하면 보다 세부적인 분석이 가능하며, 각 노드는 독자적인 메타데이터를 가질 수 도 있음
  - 문서를 노드 단위로 분할하여 검색 및 인덱싱에 활용하는
- 라마인덱스에서는 SimpleNodeParser를 사용하여 문서를 여러 개의 노드로 쉽게 분할
  - 노드 분할 시에는 Chunk_size 옵션을 통해 각 노드(청크)의 크기를 지정, 예) chunk size= 80: 문서를 80 **Token** 단위로 청킹
  - Llama_index.core.node_parser.SimpleNodeParser

Token?

- In LLMs, a **token** is the unit the model reads and writes—**not necessarily a word**. (BPE/Unigram), so a token can be:

- Sentence: I'm playing basketball today.
  - A plausible tokenization (illustrative) could look like:
  - I + ‘m + " playing” + " basketball” + " today” + .
- Notes:
  - Many tokenizers include the **leading space** inside a token (e.g., " playing").
  - playing stays whole here, but in other contexts it might split: play + ing.
- Sentence: 오늘은 날씨가 참 좋아요!
  - A plausible tokenization (illustrative) could be: 오늘 + 은 + " 날씨” + 가 + " 참” + 좋 + 아요 + !
- Notes:
  - Korean often gets split into **morpheme (형태소) or syllable-level pieces** (좋 + 아요).
  - Spaces may be bundled into the next token (e.g., " 날씨" includes a leading space).
  - Exact splits vary by the model’s tokenizer (GPT vs. LLaMA vs. others).

Category	Class / Name	What it does (one-liner)	Official docs
Basic text splitters	`SimpleNodeParser`	Wrapper that uses a text splitter to turn `Documents` → `TextNodes`.	(docs.llamaindex.ai)
	`SentenceSplitter`	Splits text preferring whole sentences/paragraphs.	(docs.llamaindex.ai)
	`TokenTextSplitter`	Splits by token count with optional overlap.	(docs.llamaindex.ai)
Context-preserving / windowed	`SentenceWindowNodeParser`	One node per sentence **plus** a window of neighboring sentences stored in metadata.	(docs.llamaindex.ai)
	`SlideNodeParser` (SLIDE)	Sliding-window chunker that enriches each chunk with localized context.	(docs.llamaindex.ai)
Semantic / adaptive	`SemanticSplitterNodeParser`	Groups semantically related sentences into nodes (embedding-based).	(docs.llamaindex.ai)
Hierarchical / multi-scale	`HierarchicalNodeParser`	Builds a hierarchy (big → small) of overlapping chunks; returned as a flat list with relationships.	(docs.llamaindex.ai)
Format-aware (file/structure specific)	`MarkdownNodeParser`	Splits using Markdown header/structure logic; keeps header path metadata.	(docs.llamaindex.ai)
	`HTMLNodeParser`	Parses selected HTML tags into nodes (BeautifulSoup-based).	(docs.llamaindex.ai)
	`JSONNodeParser`	JSON-aware splitting using custom JSON logic.	(docs.llamaindex.ai)
	`CodeSplitter`	Code/AST-aware splitter for source files (per language; max size/overlap configurable).	(docs.llamaindex.ai)
	`SimpleFileNodeParser`	Auto-chooses the right parser by **file type**(Markdown/HTML/JSON/etc.).	(docs.llamaindex.ai)
Interop / wrapper	`LangchainNodeParser`	Wraps a LangChain text splitter so you can use it inside LlamaIndex.	(docs.llamaindex.ai)
Topic-driven (example)	`TopicNodeParser`	Topic/statement-oriented parsing to keep coherent topics together.	(docs.llamaindex.ai)

2.3 텍스트 분할
2.3.1 문서(Document)와 노드(Node)

```python
from llama_index.core.node_parser import SimpleNodeParser

parser = SimpleNodeParser(chunk_size=80, chunk_overlap=0)
nodes = parser.get_nodes_from_documents([document])

print("\n생성된 노드들")
for idx, node in enumerate(nodes, start=1):
    node.metadata = {'type': '영화 줄거리', 'genre': '드라마', 'node_id': idx}
    print(f"노드 {idx}: {node.text}")
    print(f"    메타데이터: {node.metadata}")
```

```
생성된 노드들
노드 1: 영화 '기생충'은 가난한 가족인 기우네가 부유한 박 사장네 집에 하나씩 취업하면서 벌어지는 이야기입니다.
    메타데이터: {'type': '영화 줄거리', 'genre': '드라마', 'node_id': 1}
노드 2: 처음에는 평화로워 보이지만, 이들의 거짓말이 쌓이며 긴장감이 점점 고조됩니다.
    메타데이터: {'type': '영화 줄거리', 'genre': '드라마', 'node_id': 2}
노드 3: 박 사장네 집에는 비밀 지하실이 존재하며, 그곳에는 오랫동안 숨어 살던 남자가 있다는 반전이 있습니다.
    메타데이터: {'type': '영화 줄거리', 'genre': '드라마', 'node_id': 3}
노드 4: 이 사실을 알게 된 기우네 가족은 예상치 못한 위기를 맞이하게 됩니다.
    메타데이터: {'type': '영화 줄거리', 'genre': '드라마', 'node_id': 4}
노드 5: 결국 극한 상황에서 벌어지는 사건으로 인해 비극적인 결말로 이어집니다.
    메타데이터: {'type': '영화 줄거리', 'genre': '드라마', 'node_id': 5}
```

2.3 텍스트 분할
2.3.1 문서(Document)와 노드(Node)

- 토큰(Token)
  - 토큰이란 텍스트를 작은 단위로 분할한 결과물, 일반적으로 단어, 형태소(부분 단어), 문자 단위 등으로 구성
  - 분할 과정 에는 토크나이저(Tokenizer)가 사용.
- 라마인덱스는 기본적으로 OpenAI의 CL100k_base 토크나이저를 사용
- [실습] TokenTextSplitter를 활용하여 지정된 텍스트를 80 토큰 단위로 나누는 과정

```python
from llama_index.core.node_parser import TokenTextSplitter
import tiktoken

# cl100k_base 토크나이저 로드
tokenizer = tiktoken.get_encoding("cl100k_base")

token_splitter = TokenTextSplitter(chunk_size=80, chunk_overlap=0)

chunks = token_splitter.split_text(document_text)

for i, chunk in enumerate(chunks):
    token_count = len(tokenizer.encode(chunk))  # 각 청크의 토큰 개수 계산
    print(f"청크 {i+1}: {chunk}\n[토큰 개수: {token_count}]\n")

print(f"총 생성된 청크 개수: {len(chunks)}")
```

```
청크 1: 영화 '기생충'은 가난한 가족인 기우네가 부유한 박 사장네 집에 하나씩 취업하면서 벌어지는 이야기입니다. 처음에는 평화로워 보이지만, 이들의
[토큰 개수: 77]

청크 2: 거짓말이 쌓이며 긴장감이 점점 고조됩니다. 박 사장네 집에는 비밀 지하실이 존재하며, 그곳에는 오랫동안 숨어 살던 남자가 있다는
[토큰 개수: 78]

청크 3: 반전이 있습니다. 이 사실을 알게 된 기우네 가족은 예상치 못한 위기를 맞이하게 됩니다. 결국 극한 상황에서 벌어지는 사건으로 인해 비극적인 결말로 이어집니다.
[토큰 개수: 80]

총 생성된 청크 개수: 3
```

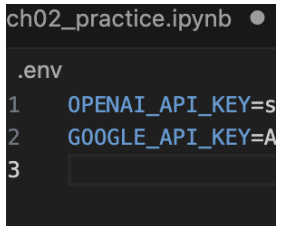
(1) parser = SimpleNodeParser(chunk_size=80, chunk_overlap=0), vs. (2) token_splitter = TokenTextSplitter(chunk_size=80, chunk_overlap=0)

- (1) = *"split + **make Nodes**"* (sentence-aware by default).
  (2) = *"split text by tokens"* (no Nodes unless wrapped).
- **1) Output type**
  - (1) SimpleNodeParser(…) → nodes = parser.get_nodes_from_documents(docs) → list of TextNodes (each has text, metadata, relationships).
  - (2) TokenTextSplitter(…) → chunks = splitter.split_text(doc.text) → list of strings. (To get Nodes, wrap it: SimpleNodeParser(text_splitter=splitter).)
- **2) Boundary logic (with chunk_size=80, chunk_overlap=0)**
  - (1) **Sentence-aware packing**: tries to pack whole sentences without exceeding ~80 tokens.
    Example text with three sentences: 50 tokens, 50 tokens, 30 tokens
    → chunks likely: [S1] (50), [S2] (50), [S3] (30) (keeps sentences intact).
  - (2) **Strict token windows**: cuts at exact token counts.
    Same text → chunk1 = S1 (50) + first 30 tokens of S2 (80), chunk2 = rest of S2 (20) + S3 (30) (50).
    Result: **clean 80-token windows**, but may split sentences in the middle.
- **3) Metadata/graph**
  - (1) Nodes keep **original document metadata** and get **prev/next** relationships; useful for retrieval and context windows.
  - (2) Splitter alone doesn't attach metadata or relationships.

2.3 텍스트 분할
2.3.1 문서(Document)와 노드(Node)

- 노드 단위 검색의 장점
  - 검색 속도와 정확도가 크게 향상
  - chunk_over Lap는 각 청크별로 문자열을 얼마나 겹칠지를 결정하는데, 0으로 설 정하면 각 청크가 서로 겹치지 않도록 청킹
  - 분리된 청크는 각각 별도의 검색 대상이 되어, 보다 빠르고 정밀한 검색 결과를 제공
  - 문서 전체를 한 덩어리로 검색하 는 방식보다 더 빠르고, 필요한 정보에 정확히 도달

- [실습] 문서 단위의 검색 vs. 노드 단위의 검색
  - ../.env 파일 생성
  - OpenAI API Key 생성




```
import os
from dotenv import load_dotenv
load_dotenv()
```

```python
from dotenv import load_dotenv
load_dotenv()

# 한글 답변 설정
llm = OpenAI(api_key=os.environ["OPENAI_API_KEY"], model="gpt-4o-mini", system_prompt="반드시 한국어로 답변하세요.")
# Settings.llm = OpenAI(model="gpt-4o-mini", temperature=0, system_prompt="항상 한국어로 답변하세요.")
```

```python
# 텍스트 데이터를 기반으로 문서 생성
document_text = "영화 '기생충'은 가난한 가족인 기우네가 부유한 박 사장네 집에 하나씩 취업하면서 벌어지는 이야기입니다. 처음에는
document = Document(text=document_text)
```

```python
parser = SimpleNodeParser(chunk_size=80, chunk_overlap=0)
nodes = parser.get_nodes_from_documents([document])
```

```python
# 문서 단위 검색을 위한 전체 문서 인덱스 생성
full_doc_index = VectorStoreIndex.from_documents([document], llm=llm)
```

```python
# 노드 단위 검색을 위한 인덱스 생성
node_index = VectorStoreIndex(nodes, llm=llm)
```

```python
# 검색 비교
query_text = '이 영화의 반전은 무엇인가요?'

## 문서 단위 검색
print("\n문서 단위 검색 결과")
doc_query_engine = full_doc_index.as_query_engine()
doc_response = doc_query_engine.query(query_text)
print(f"문서 검색 응답: {doc_response.response}")
if doc_response.source_nodes:
    for idx, document in enumerate(doc_response.source_nodes, start=1):
        print(f"- 결과 {idx}: {document.node.text}")

## 노드 단위 검색
print("\n노드 단위 검색 결과")
node_query_engine = node_index.as_query_engine()
node_response = node_query_engine.query(query_text)
print(f"노드 검색 응답: {node_response.response}")
if node_response.source_nodes:
    for idx, document in enumerate(node_response.source_nodes, start=1):
        print(f"- 결과 노드 {idx}: {document.node.text}")
```


Abstract

- **Nodes** → atomic, chunked units of your documents.
- **Index** → organizes nodes into a structure (VectorStoreIndex, TreeIndex, ListIndex).
- **Query Engine** → uses the index + LLM(s) to process queries and return answers.

- VectorStoreIndex.from_documents([document], llm=llm) : 임베딩용이 아니라, 나중에 쿼리(query_engine = full_doc_index.as_query_engine()) 시 질의응답 단계에서 사용할 LLM을 지정하는 것

- LlamaIndex 내부적으로는 다음과 같이 구성됩니다 :Settings.embed_model : 문서 벡터화용 임베딩 모델 Settings.llm : 답변을 생성할 때 사용할 LLM

- 현 코드에서는 Settings.embed_model 미지정, 내장된 **기본 embedding 모델 (예: text-embedding-3-small)**을 사용

실행 결과

```
문서 단위 검색 결과
문서 검색 응답: 이 영화의 반전은 박 사장네 집에 있는 비밀 지하실에 오랫동안 숨어 살던 남자가 있다는 사실입니다.
- 결과 1: 영화 '기생충'은 가난한 가족인 기우네가 부유한 박 사장네 집에 하나씩 취업하면서 벌어지는 이야기입니다. 처음에는 평화로워 보이지만, 이들의 거짓말이 쌓이며 긴장감이 점점 고조됩니다. 박 사장네 집에는 비밀 지하실이 존재하며, 그곳에는 오랫동안 숨어 살던 남자가 있다는 반전이 있습니다. 이 사실을 알게 된 기우네 가족은 예상치 못한 위기를 맞이하게 됩니다. 결국 극한 상황에서 벌어지는 사건으로 인해 비극적인 결말로 이어집니다.

노드 단위 검색 결과
노드 검색 응답: 영화의 반전은 박 사장의 집에 비밀 지하실이 존재한다는 사실을 드러냅니다. 그곳에는 오랫동안 숨어 지내던 한 남자가 살고 있었습니다.
- 결과 노드 1: 박 사장네 집에는 비밀 지하실이 존재하며, 그곳에는 오랫동안 숨어 살던 남자가 있다는 반전이 있습니다.
- 결과 노드 2: 결국 극한 상황에서 벌어지는 사건으로 인해 비극적인 결말로 이어집니다.
```

2.3 텍스트 분할
2.3.1 Document와 Node

- 문서 단위 검색
  - 전체 문서에서 답변을 찾기 때문에 특정 문장이 나 세부적인 내용을 추출하는 데 한계, 불필요한 정보가 포함,
  - 전체 리뷰 문서를 분석해야 하므로 검색 속도가 느려짐
- 노드 단위 검색
  - 의미 단위로 나눈 각 청크에 직접 접근, 질의와 가장 관련 있는 노드만 선별하여 반환, 더 정확하고 간 결한 검색 결과를 제공
  - 문서 전체를 일일이 탐색하지 않음. 검색 시간 단축

- * 이분법적으로 판단할 것이 아니라 Chunk 크기의 개념으로 바라볼 필요가 있음

2.3 텍스트 분할
2.3.1 Document와 Node

- 적절한 청킹 사이즈
  - 문서를 적절한 크기의 노드로 나누는 것은 검색 효율성과 정확도를 높이는 핵심 요소
  - 라마인덱스는 다양한 청킹 기법을 제공하며, 문서를 보다 효과적으로 분할하고 검색할 수 있도록 지원
  - 라마인덱스 기본 청킹 크기(chunk size)는 1,024
  - 너무 큰 청크는 불필요한 정보까지 포함하여 검색 정확도를 떨어뜨릴 수 있고, 너무 작은 청크는 문맥이 단절되어 핵심 정보를 파악하기 어려울 수 있음.
  - 청킹 크기는 데이터의 특성과 검색 목적에 맞추어 신중 히 조정해야 할 중요한 요소
  - 검색 성능 평가 지표
    - 평균 신뢰도: 생성된 응답이 원본 문서(출처)와 얼마나 일치하는지 평가
    - 평균 관련성: 생성된 응답이 사용자 질문과 얼마나 관련이 있는지 평가

표 2.2 최적의 청킹 크기를 찾기 위한 실험 결과

청크 크기	평균 응답 시간 (초)	평균 신뢰도	평균 관련성
128	1.55	0.85	0.78
256	1.57	0.9	0.78
512	1.68	0.85	0.85
1024	1.68	0.93	0.9
2048	1.72	0.9	0.89

2.3 텍스트 분할
2.3.2 Token 단위 분할

- 토큰 단위 분할은 문서를 일정한 길이의 토큰(token) 단위로 나누는 방식
  - 장점: 문서를 일정한 크기의 블록으로 나눌 수 있어 전체 문서의 길이가 일 정하게 유지. 따라서 메모리 관리가 편리
  - 단점: 문장이 중간에서 인위적으로 잘릴 수 있으며, 이로 인해 문맥이 단절되거나 원래 의미가 훼손될 수 있음
- [실습] 예제 코드
  - TokenTextSplitter를 사용하여 텍스트를 50토큰 크기로 분할
  - Chunk_ size=50 옵션은 각 청크의 최대 토큰 수를 50개로 설정
  - chunk_over lap=10 옵션은 각 청크가 앞뒤로 10토큰씩 중복되도록 설정
  - 이러한 중첩 구 조는 문장이 청크 경계에서 잘리는 경우에도 문맥이 단절되지 않도록 방지

```
sample_text = "영화 '기생충'은 가난한 가족인 기우네가 부유한 박 사장네 집에 하나

token_splitter = TokenTextSplitter(
    chunk_size=50,
    chunk_overlap=10
)

token_chunks = token_splitter.split_text(sample_text)
print("=== 토큰 기반 분할 결과 ===")
for i, chunk in enumerate(token_chunks):
    print(f"Chunk {i + 1}:", chunk.strip(), "\n")
```

2.3 텍스트 분할
2.3.3 문장 단위 분할

- 문장 단위 분할은 문서를 각 문장을 기준으로 나누는 방식
- 가능한 한 설정된 chunk_size 범위 내에서 최대한 문장을 온전하게 포함하도록 처리
- 토큰 단위 분할보다 문장이 불완전하게 나 뉠 가능성이 낮으며, 문맥의 흐름을 보다 자연스럽게 유지할 수 있는 장점
- 예-1)"영화 '기생충'은 가난한 가족인 기우네가 부유한 박 사장네 집에 하나씩 취업 하면서 벌어지는 이야기입니다."는 50토큰을 초과하므로 문장이 분할되며, 초과된 부분은 다음 청크로 넘어감. 이때 남은 부분인 "벌어지는 이야기입니다."는 자체적으로 완전 한 문장이므로, 다른 문장과 결합하지 않고 독립적인 청크로 유지
- 예-2) "처음에는 평화로워 보이지만, 이들의 거짓말이 쌓이며 긴장감이 점점 고조됩 니다."는 50토큰을 초과하지 않기 때문에 하나의 청크로 유지

```python
from llama_index.core.node_parser.text.sentence import SentenceSplitter

splitter = SentenceSplitter(chunk_size=50, chunk_overlap=0)

sentence_chunks = splitter.split_text(sample_text)
print("=== 문장 기반 분할 결과 ===")
for i, chunk in enumerate(sentence_chunks):
    print(f"Chunk {i + 1}:", chunk.strip(), "\n")
```

```
=== 문장 기반 분할 결과 ===
Chunk 1: 영화 '기생충'은 가난한 가족인 기우네가 부유한 박 사장네 집에 하나씩 취업하면서

Chunk 2: 벌어지는 이야기입니다.

Chunk 3: 처음에는 평화로워 보이지만, 이들의 거짓말이 쌓이며 긴장감이 점점 고조됩니다.

Chunk 4: 그러나 이 영화의 반전은 지하실에서 시작됩니다.
```

2.3 텍스트 분할
2.3.4 의미(Semantic) 단위 분할

- 의미 기반 분할(Semantic Splitting)은 : 문맥의 의미를 고려하여 텍스트를 적절한 단위로 분할하는 방식
  - 의미적으로 연관된 문장들을 함께 유지함으로써, 보다 자연스러운 정보 단위로 텍스트를 구성
- OpenAI 임베딩(OpenAIEmbedding) : 의미 단위 분할은 임베딩을 활용하여 텍스트의 의미적 유사도를 계산

```python
from llama_index.core.node_parser import SemanticSplitterNodeParser
from llama_index.embeddings.openai import OpenAIEmbedding

embed_model = OpenAIEmbedding()

splitter = SemanticSplitterNodeParser(
    buffer_size=1,
    breakpoint_percentile_threshold=95,
    embed_model=embed_model
)

chunks = splitter.sentence_splitter(sample_text)
print("=== 의미 기반 분할 결과 ===")
for i, chunk in enumerate(chunks):
    print(f"Chunk {i + 1}:", chunk.strip(), "\n")
```

Args

buffer_size : *int*
number of sentences to group together when evaluating semantic similarity

embed_model
(BaseEmbedding): embedding model to use

sentence_splitter : *Optional[Callable]*
splits text into sentences

include_metadata : *bool*
whether to include metadata in nodes

include_prev_next_rel : *bool*
whether to include prev/next relationships

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Breakpoint_percentile_threshold: "The percentile of cosine dissimilarity that must be exceeded between a group of sentences and the next to form a node. The smaller this number is, the more nodes will be generated",

2.3 텍스트 분할
2.3.4 의미(Semantic) 단위 분할

- OpenAI 임베딩 외에도 허깅페이스 임베딩 모델을 사용 가능
- Transformers, llama-index-embeddings-huggingface 설치

```
!pipenv install transformers -q
!pipenv install llama-index-embeddings-huggingface -q
```

```
from llama_index.core.node_parser import SemanticSplitterNodeParser
from llama_index.embeddings.huggingface import HuggingFaceEmbedding

embed_model = HuggingFaceEmbedding(model_name="sentence-transformers/all-MiniLM-L6-v2")

splitter = SemanticSplitterNodeParser(
    buffer_size=1,
    breakpoint_percentile_threshold=95,
    embed_model=embed_model
)

chunks = splitter.sentence_splitter(sample_text)
print("=== 의미 기반 분할 결과 ===")
for i, chunk in enumerate(chunks):
    print(f"Chunk {i + 1}:", chunk.strip(), "\n")
```

- OpenAI 임베딩과 비교: 동일
- OpenAI 임베딩: 유료 서비스
- 허깅페이스 기반 임베딩:
  - 로컬 환경에서 실행 가능
  - 추가 비용 없음

```
print(Settings.llm)
print(Settings.embed_model)
```

2.3 텍스트 분할
2.3.4 텍스트 분할 비교

표 2.3 텍스트 분할 방식별 장단점 비교

분할 방식	장점	단점	적합한 도메인
토큰 단위 분할	▪ 일정한 크기로 분할되어 메모리 관리가 용이함 ▪ 대용량 데이터 처리에 효율적	▪ 문맥이 단절되어 의미 왜곡 가능 ▪ 중요한 정보가 누락될 가능성 있음	로그 데이터 처리, 대용량 문서 처리
문장 단위 분할	▪ 문맥이 유지되어 의미 왜곡이 적음 ▪ 자연스러운 흐름 유지	▪ 문장 길이의 불균형으로 메모리 관리 어려움 ▪ 긴 문장은 여전히 문제 발생 가능	뉴스 기사, 일반 문서 처리
의미 단위 분할	▪ 의미 기반 분할로 검색 정확도 향상 ▪ 중요한 내용 강조 가능	▪ 계산 비용 증가 ▪ 모델 성능에 의존	법률 문서, 의학 논문, 학술 자료

- 토큰 단위 분할 :
  - 메모리 관리에 효 율적, 대용량 데이터를 처리할 때 적합, 예: 로그 데이터나 대규모 문서 처리
  - 문맥 정보를 고려하지 않기 때문에 의미가 왜곡되거나, 중요한 문장이 분리되어 정보가 누락 단점.
- 문장 단위 분할
  - 문맥을 유지, 특히 문장의 완성도가 높은 뉴스 기사나 일반 문서의 처 리에 유용
  - 문장 길이가 일정하지 않다는 점, 짧은 문장은 지나치게 작은 청크로 나뉨. 긴 문장은 하나의 청크에 과도한 정보가 담기게 됨.
- 의미 단위 분할
  - 검색 정확도를 높이고, 중요한 내용을 보다 명확하게 강조할 수 있는 방식
  - 법률 문서, 의 학 논문, 학술 자료 등과 같이 정밀한 해석과 높은 검색 정확도가 요구되는 전문 도메인에서 특히 효과적
  - 임베딩 모델을 활용하여 텍스트의 의미를 계산하는 과정을 포함하므로, 일반적인 분할 방식에 비해 계산 비용이 증가, 결과의 품질이 임베딩 모델의 성능에 좌우

2.4 인덱싱
2.4.1 인덱싱이란?

- Index is the retriever-ready data structure that organizes Nodes and knows how to search them.
  - Retriever: given a query, fetches relevant chunks from the store.
    - retriever = index.**as_retriever**(similarity_top_k=5)
    - nodes = retriever.retrieve("your query")
  - + Generator (LLM) : Consumes retrieved chunks + query, and produces the answer.
    - qe = index.as_query_engine(similarity_top_k=5)
    - answer = qe.query("your query")

- 인덱스(Index)
  - 문서 객체로 구성된 특정 형태의 데이터 구조, LLM이 쿼리를 보다 효율적으로 수행할 수 있도록 함.
  - 라마인덱스에서 인덱스는 내부적으로는 문서를 노드(Node) 단위로 분할하여 처리
  - 라마인덱스는 다양한 인덱스 유형을 제공

- **2.4.2 Vector Store Index**
  - 문서를 노드 단위로 분할한 뒤, 각 노드의 텍스트를 벡터화
  - 벡터 임베딩(Vector Embedding): 텍스트의 의미를 수치화하여 벡터로 표현하는 기술
    - 의미적으로 유사한 단 어나 문장은 임베딩 공간에서 물리적으로 가까운 위치에 매핑
    - 예) '고양이' 와 '개'는 유사한 문맥에서 자주 사용되므로, 임베딩 공간에서 가까운 위치에 놓일 수 있음
- 벡터 임베딩 장점
  - 효율적인 검색: 대규모 데이터에서도 빠르게 검색을 수행
  - 의미론적 검색: 단순한 키워드 매칭이 아닌, 텍스트의 의미를 기반으로 유사한 결과를 반환
  - 다양한 쿼리 전략 지원: 벡터 임베딩을 활용해 상위 k개의 결과만 반환하거나, 문맥을 고려한 검색을 수행

- VectorStoreIndex implicitly pipelines: (1) **Chunking** (Documents → Nodes), (2) **Embedding** (Nodes → embeddings), (3) **Indexing** (embeddings → vector store)

2.4 인덱싱
2.4.2 VectorStoreIndex

- [실습]

```python
from llama_index.core import VectorStoreIndex
from llama_index.core import SimpleDirectoryReader

reader = SimpleDirectoryReader('data')
documents = reader.load_data()

# Document 객체를 전달하여 인덱스 생성
index = VectorStoreIndex.from_documents(documents)
```

internally runs a **node parser** (with a default splitter) to create **Nodes** from each `Document`.
- Default chunk size: 1024

```python
# 상위 3개의 결과를 반환
query_engine = index.as_query_engine(similarity_top_k=3)

response = query_engine.query("고양이에게 수분 공급이 중요한 이유는?")

print("[검색된 상위 3개 문서]")

for idx, node in enumerate(response.source_nodes):
    print(f"\n[문서 {idx+1}]\n{node.text}")
print("\n[답변]")

print(response)
```

retrieves the **top-k Nodes**;
`response.source_nodes` are those chunks.

Note that:

```
len(documents)
✓ 0.0s
5
```

2.4 인덱싱
2.4.2 VectorStoreIndex

```python
from llama_index.core import VectorStoreIndex
from llama_index.core import SimpleDirectoryReader

reader = SimpleDirectoryReader('data')
documents = reader.load_data()

# Document 객체를 전달하여 인덱스 생성
index = VectorStoreIndex.from_documents(documents)
```

```python
# 상위 3개의 결과를 반환
query_engine = index.as_query_engine(similarity_top_k=3)

response = query_engine.query("고양이에게 수분 공급이 중요한 이유는?")

print("[검색된 상위 3개 문서]")

for idx, node in enumerate(response.source_nodes):
    print(f"\n[문서 {idx+1}]\n{node.text}")
print("\n[답변]")
```

```
[문서 1]
고양이는 물을 충분히 마셔야 합니다. 수분이 부족하면 신장 문제가 발생할 수 있습니다.
건식 사료보다 습식 사료가 수분 공급에 도움이 됩니다.

[문서 2]
고양이는 육식 동물입니다. 주로 고기, 생선, 그리고 가공된 고양이 사료를 먹습니다.
특히, 단백질이 풍부한 음식을 선호하며, 탄수화물 섭취는 적은 편입니다.

[문서 3]
고양이는 초콜릿, 양파, 마늘 같은 음식은 먹으면 안 됩니다.
특히, 초콜릿에 포함된 테오브로민 성분은 고양이에게 치명적일 수 있습니다.

[답변]
고양이에게 수분 공급이 중요한 이유는 신장 문제를 예방하기 위해서입니다.
```

```python
from llama_index.core import VectorStoreIndex
from llama_index.core import SimpleDirectoryReader
from llama_index.core.node_parser import SentenceSplitter  # node parser

# load documents
documents = SimpleDirectoryReader("data").load_data()

# chunk into Nodes
splitter = SentenceSplitter(chunk_size=80, chunk_overlap=0)
nodes = splitter.get_nodes_from_documents(documents)

# build index over nodes
index = VectorStoreIndex(nodes)
```

```
[문서 1]
고양이는 물을 충분히 마셔야 합니다. 수분이 부족하면 신장 문제가 발생할 수 있습니다.

[문서 2]
특히, 초콜릿에 포함된 테오브로민 성분은 고양이에게 치명적일 수 있습니다.

[문서 3]
고양이는 육식 동물입니다. 주로 고기, 생선, 그리고 가공된 고양이 사료를 먹습니다.

[답변]
To prevent potential kidney issues, it is important for cats to consume an adequate amount of water.
```

Index

Index	Internal structure / data	Retrieval method	Best for	Strengths	Trade-offs
VectorStoreIndex	Nodes + embeddings in a vector store (FAISS/Chroma/Pinecone/etc.)	k-NN semantic similarity (optionally reranking)	General RAG over unstructured text	Scales well; strong semantic recall; ecosystem support	Needs embedding model; can retrieve off-topic without filtering
ListIndex	Simple ordered list of Nodes	Linear scan / simple filtering	Small corpora; order-sensitive data (logs, short transcripts)	Minimal setup; deterministic ordering	Doesn't scale; no semantic search without add-ons
TreeIndex	Hierarchical summaries (children → parent)	Top-down traversal via summaries	Long, structured docs where drill-down helps	Good "overview → detail"; controllable context	Build time for summaries; retrieval path depends on summary quality
SummaryIndex(DocumentSummary)	One (or few) summaries per document	Select doc via summary, then drill into content	Document-level routing before chunk search	Fast doc routing; reduces search space	Coarser granularity; requires second step for passages
KnowledgeGraphIndex	Triples (entity–relation–entity) extracted from text	Graph queries (neighbors, paths) + hybrid retrieval	Entity/relationship questions; reasoning over facts	Structured reasoning; complements vectors	KG extraction cost; coverage depends on extraction quality
SQLStructStoreIndex	Schema + table stats for SQL DBs	SQL generation + execution	Tabular/relational data (analytics, reports)	Accurate numeric/relational answers	Requires DB connectivity and schema grounding
PandasIndex	DataFrame schema + samples	DataFrame ops (pandas) guided by LLM	Local CSV/Parquet analytics	Quick local analytics; no DB needed	Memory-bound; less suitable for very large data
ComposableGraph(meta)	Graph of sub-indices (possibly mixed)	Router decides which sub-index to query	Multi-source or heterogeneous corpora	Modular; per-source tuning	More wiring; router quality matters

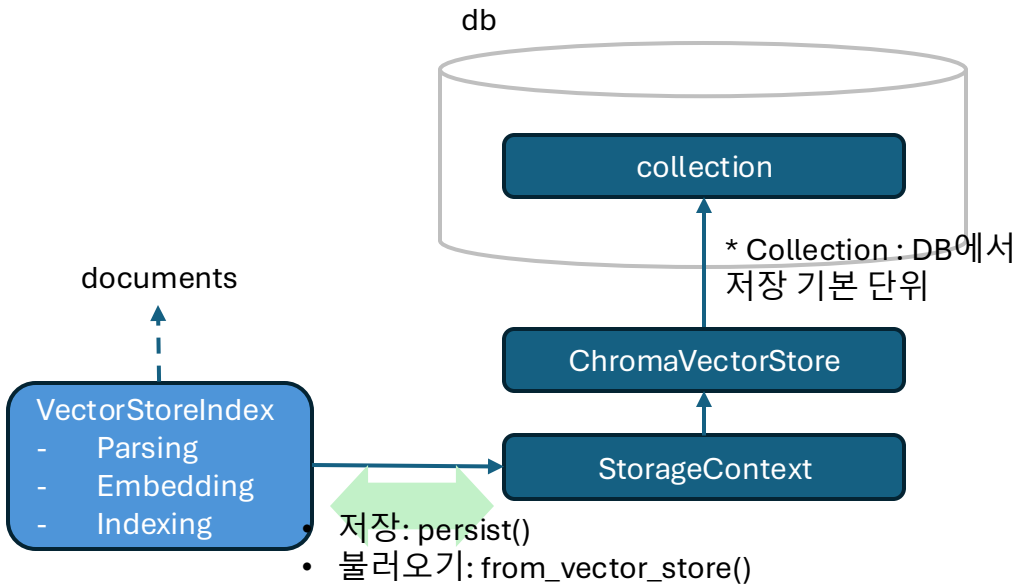
2.5 저장하기

- 문서를 인덱싱하면 쿼리를 실행할 준비가 완료
- 모든 텍스트에 대해 임베딩 을 생성하는 작업은 시간이 오래 걸릴 수 있으며, 특히 외부에서 호스팅되는 LLM 서비스 를 사용하는 경우에는 처리 비용 발생
- 처리 속도를 높이고 비용을 절감하기 위해, 한 번 생성된 임베딩을 저장해 두고 재사용하는 방법
- 라마인덱스는 인덱싱된 데이터를 메모리에 저장. 지속적인 활용 을 위해서는 로컬 스토리지에 임베딩을 저장하는 방식 사용

- 인덱싱된 데이터를 저장하는 가장 간단한 방법은
  - 모든 인덱스 객체에 내장된 **persist()** 메서드를 사용 : 데이터를 지정한 경로의 디스크에 저장
  - 저장된 인덱스를 다시 로드하면 임베딩을 다시 생성하거나 문서를 다시 처리할 필요 없이, 이전에 구축된 인덱스를 그대로 불러와 즉시 쿼리를 실행 가능
- 라마인덱스는 다양한 벡터 스토어(Vector Store)를 지원 : 실습으로 오픈 소스 벡터 저장소인 **크로마(Chroma)**를 사용.
  - Chromada 설치 : pipenv install chromadb -q

2.5 저장하기

```python
import chromadb
from llama_index.core import VectorStoreIndex
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.core import StorageContext

# 크로마 클라이언트 초기화
db = chromadb.PersistentClient(path="./chroma_db")

# 저장된 컬렉션을 다시 가져오기 (또는 생성하기)
chroma_collection = db.get_or_create_collection("quickstart")

# 저장된 크로마 벡터 스토어 설정
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# 벡터 스토어 인덱스에 문서를 저장 (데이터 임베딩 후 저장)
index = VectorStoreIndex.from_documents(documents, storage_context=storage_context)

# 인덱스 데이터를 로컬 디렉터리에 저장
index.storage_context.persist(persist_dir="./index_data")

# --- 이후 다시 데이터를 불러오고 쿼리 수행 ---
# 크로마 클라이언트 다시 초기화
db = chromadb.PersistentClient(path="./chroma_db")

# 저장된 컬렉션을 다시 가져오기
chroma_collection = db.get_or_create_collection("quickstart")

# 저장된 크로마 벡터 스토어 설정
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# 저장된 벡터 데이터를 이용해 인덱스를 다시 로드
index = VectorStoreIndex.from_vector_store(
    vector_store, storage_context=storage_context
)

# 쿼리 엔진 생성 및 쿼리 수행
query_engine = index.as_query_engine()
response = query_engine.query("What is Chroma?")
print(response)
```



2.6 쿼리 (Query)
2.6.1 QueryEngine

- 쿼리 엔진(QueryEngine)
  - 가장 간단한 생성 방법은 인덱스 객체로부터 쿼리 엔진을 직접 생성
  - 쿼리 엔진 : 사용자의 질문(쿼리)을 받아 검색(Retrieval), 후처리 (Postprocessing), 응답 합성(Response Synthesis) 단계를 거쳐 최종 응답을 생성
  - 쿼리가 입력되면 검색 엔진이 인덱스를 통해 관련 문서를 찾고, 후처리 과정을 거쳐 필터링하거나 재정렬한 뒤, 검색된 데이터를 바탕으로 LLM(대규모 언어 모델)을 호출하여 최종 응답을 생성해 반환

- 쿼리 엔진의 주요 역할
  - 검색(Retrieval): 입력된 쿼리와 가장 관련 있는 문서를 인덱스에서 검색. 예) TOP-K Semantic Search
  - 후처리(Postprocessing): 검색된 노드를 선택적으로 재정렬, 변환, 필터링하는 단계. 예) 특정 키워드가 포함되어 있거나, 지정된 메타데이터 조건(예: 영화 유형이 액션, 감독이 '홍길동)을 만족하는 노드만 필터링
  - 응답 합성(Response synthesis): 가장 관련 있는 데이터가 프롬프트와 결합되어 LLM으로 전송

```
query_engine = index.as_query_engine()
response = query_engine.query(
    "고객의 개인 정보를 참고하여 맞춤형 이메일을 작성해 주세요."
)
print(response)
```

✓ 6.3s

2.6 쿼리 (Query)
2.6.2 검색(Retrieval), 2.6.3 후처리(Postprocessing)

- Retrieval: 인덱스에서 가장 적합한 문서를 찾아 반환, Top-K
  - 예) 사용자가 'Llama2의 특징은 무엇인가요?"라고 질문하면, 라마인덱스는 임베딩을 활용해 해당 질문과 가장 관련성이 높은 문서를 벡터 공간에서 검색

```python
from llama_index.core.retrievers import VectorIndexRetriever

retriever = VectorIndexRetriever(
    index=index,
    similarity_top_k=5, # 상위 5개의 결과 반환
)
```

- Postprocessing: 검색된 문서를 재정렬, 변환, 필터링
  - 예) 유사도 점수가 일정 기준 이상인 문서만 선택후, 특정 메타데이터(날짜, 카테고리)를 포함한 문서만 사용

```python
from llama_index.core.postprocessor import SimilarityPostprocessor

postprocessor = SimilarityPostprocessor(similarity_cutoff=0.7)
query_engine = index.as_query_engine(node_postprocessors=[postprocessor])
```

2.6 쿼리 (Query)
2.6.4 응답 합성 (Response synthesis)

- 응답 합성 단계는 검색된 문서와 사용자의 질의를 기반으로 LLM이 최종 응답을 생성하는 과정
  - "compact" 모드:
    - "compact" 모드는 검색된 문서들을 하나로 연결(concatenate)하여 하나의 LIM 호출로 응답을 생성하는 방식.
    - 호출 횟수가 적어 응답 속도가 빠르며, 일반적인 질의응답 시스템에서 널리 사용.
  - "refine" 모드:
    - 검색된 문서들를 순차적으로 읽어가며 응답을 점진적으로 정제 하는 구조. 처음에는 첫 번째 문서를 기반으로 초기 응답을 생성하고, 이후 문서들을 하나씩 읽으며 응답을 보완하거나 수정
    - LLM이 보다 깊이 있는 응답을 생성할 수 있지만, 문서당 하나의 LLM 호출이 발생하므로 호출 비용이 증가
  - "tree_summarize" 모드:
    - 여러 문서를 묶어서 요약한 후, 그 요약 결과를 다시 상위 수준에서 통합 요약하는 트리 구조를 활용
    - 전체적인 맥락을 유지하면서도 정보를 간결하게 요약하고자 할 때 효과적

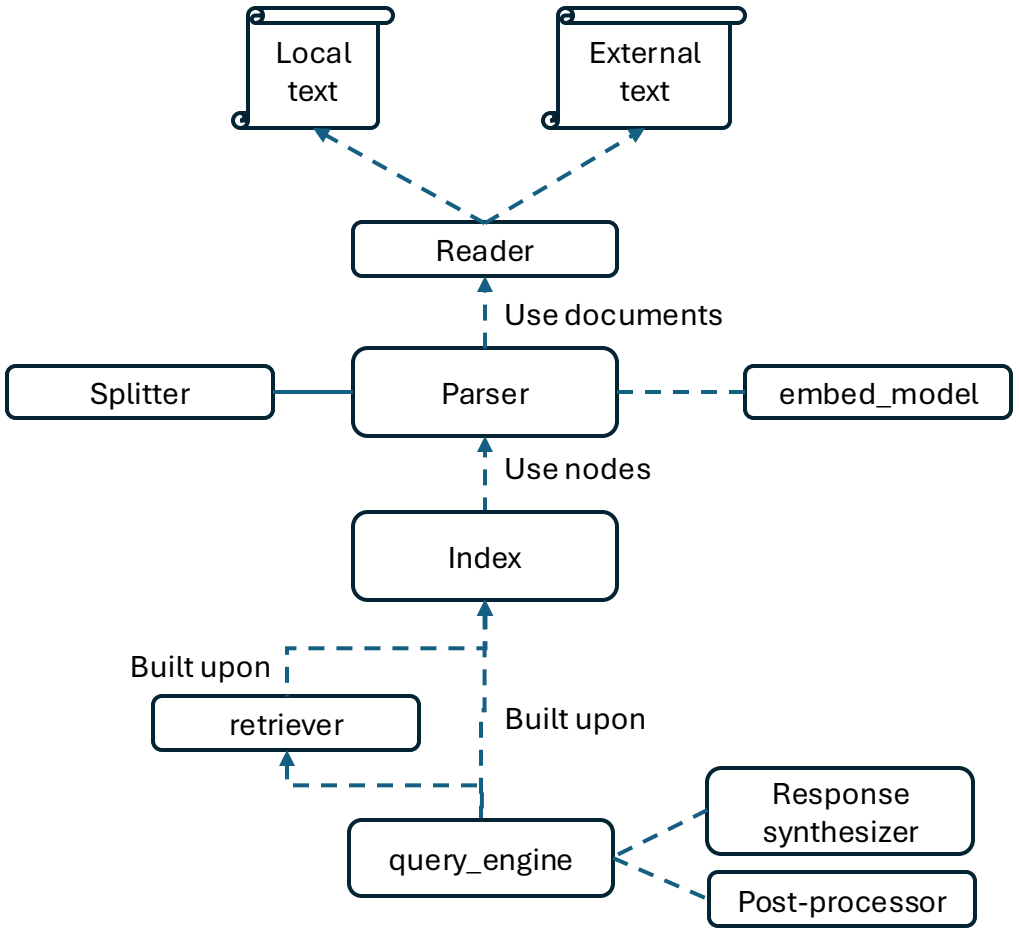
```python
from llama_index.core.response_synthesizers import get_response_synthesizer
from llama_index.core.query_engine import RetrieverQueryEngine

# 응답 합성기 설정
response_synthesizer = get_response_synthesizer(response_mode="compact")

# 쿼리 엔진 구성
query_engine = RetrieverQueryEngine.from_args(
    retriever=index.as_retriever(),
    response_synthesizer=response_synthesizer,
)

# 쿼리 실행
response = query_engine.query("Llama2란?")
```

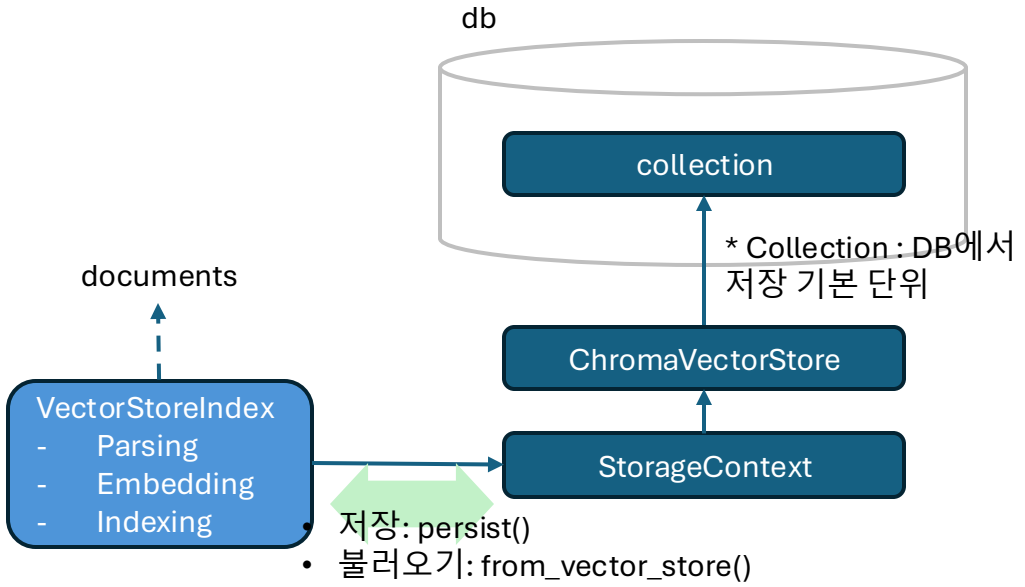
2.6 쿼리 (Query)
2.6.5 커스터마이징 (customizing)

- 쿼리 처리 과정에서 검색기(Retriever), 후처리(Postprocessor), 응답 합성 (Response Synthesizer) 등의 구성 요소를 직접 설정할 수 있는 저수준 구성 API를 제공

```python
from dotenv import load_dotenv
load_dotenv()

documents = SimpleDirectoryReader("data").load_data()

llm = OpenAI(api_key=os.environ["OPENAI_API_KEY"], model="gpt-4o-mini", system_prompt="반드시 한국어로 답변

# 문서를 기반으로 인덱스 생성
index = VectorStoreIndex.from_documents(documents, llm=llm)

# 검색기 설정 (상위 10개의 유사한 결과 반환)
retriever = VectorIndexRetriever(
    index=index,
    similarity_top_k=10,
)

# 응답 합성기 설정
response_synthesizer = get_response_synthesizer()

# 후처리 설정 (유사도 0.7 이상인 노드만 선택)
postprocessor = SimilarityPostprocessor(similarity_cutoff=0.7)

# 쿼리 엔진
query_engine = RetrieverQueryEngine(
    retriever=retriever,
    response_synthesizer=response_synthesizer,
    node_postprocessors=[SimilarityPostprocessor(similarity_cutoff=0.7)],
)

# 쿼리 실행 및 결과 출력
response = query_engine.query("고양이에게 수분 공급이 중요한 이유는?")
print(response)
```

Visual Summary, again. Where is an LLM?





query_engine linked to LLMs

```python
# 1. Load docs
documents = SimpleDirectoryReader("data").load_data()

# 2. Build index (no llm needed here, just embeddings)
index = VectorStoreIndex.from_documents(documents)

# 3. Define LLM for query-time reasoning
llm = OpenAI(
    api_key=os.environ["OPENAI_API_KEY"],
    model="gpt-4o-mini",
    system_prompt="반드시 한국어로 답변하세요."
)

# 4. Build retriever (pulls relevant nodes from index)
retriever = VectorIndexRetriever(index=index, similarity_top_k=5)

# 5. Build response synthesizer with the LLM
response_synthesizer = get_response_synthesizer(llm=llm)

# 6. Connect into a query engine
query_engine = RetrieverQueryEngine(
    retriever=retriever,
    response_synthesizer=response_synthesizer,
)

# 7. Ask a question
response = query_engine.query("고양이에게 수분 공급이 중요한 이유는?")
print(response)
```

Visual Summary, again. Where is an LLM?