리트리버

🔍 리트리버(Retriever)의 역할

리트리버는 질문(Query)을 입력받으면,

지식베이스(예: 문서, DB, 벡터 스토어)에서

가장 관련 있는 문서나 문단을 찾아주는 모듈입니다.

- 입력: 사용자 질문 (예: “MongoDB에서 $lookup은 어떻게 쓰나요?”)
- 출력: 관련 있는 문서의 일부 (예: `$lookup` 설명이 포함된 기술 문서)

이렇게 찾은 결과가 **LLM의 프롬프트에 함께 전달**되어

모델이 문서를 참고해 답을 생성합니다.

🧩 리트리버 구조

```
[사용자 질문] → [Retriever] → [관련 문서] → [Generator(LLM)]
                                  ↓
                            [최종 답변 생성]
```

예를 들어 LangChain이나 LlamaIndex 같은 프레임워크에서는 이렇게 구성됩니다:

- **Retriever**: 벡터DB에서 관련 문서 검색 (예: FAISS, Pinecone, Chroma 등)
- **Generator**: 검색된 문서를 참고해 LLM이 답변 생성

⚙️ 리트리버의 내부 동작

리트리버는 보통 **벡터 임베딩(embedding)** 기반으로 작동합니다.

1. **문서 임베딩**
   - 모든 문서를 벡터 형태로 변환해 벡터 DB에 저장합니다.

     예: `"MongoDB $lookup"` → `[0.21, -0.13, 0.77, ...]`
2. **질문 임베딩**
   - 사용자의 질문도 같은 방식으로 벡터로 변환합니다.
3. **유사도 검색 (Similarity Search)**
   - 코사인 유사도(Cosine Similarity) 등으로 문서 벡터와 질문 벡터를 비교.
   - 가장 비슷한 문서들을 N개(top_k) 반환합니다.

리트리버 알고리즘은 대부분 벡터 스토어 기반의 리트리버를 그대로 사용합니다. 하지만 여러 가지 리트리버를 살펴보고 실험해 가장 검색 품질을 높일 수 있는 방법을 찾아보기 바랍니다.

희소 리트리버, 밀집 리트리버

“**희소 리트리버(Sparse Retriever)**”와 “**밀집 리트리버(Dense Retriever)**”는

RAG에서 **문서를 검색하는 방식의 차이**를 기준으로 나눈 두 가지 리트리버 유형입니다.

한마디로 정리하면 다음과 같습니다:

> 🧠 희소(Sparse): 단어 기반 검색
> 💪 **밀집(Dense)**: 의미 기반 검색

⚙️ 1. 개념 요약 비교

구분	희소 리트리버 (Sparse Retriever)	밀집 리트리버 (Dense Retriever)
검색 기준	단어(토큰) 일치	문장의 의미(벡터 유사도)
대표 알고리즘	BM25, TF-IDF	DPR, Sentence-BERT, OpenAI Embedding
표현 방식	"단어 → 희소 벡터"	"문장 → 밀집 벡터"
특징	대부분 0으로 채워진 긴 벡터 (ex. 30,000차원 중 대부분 0)	낮은 차원의 연속적인 실수 벡터 (ex. 768차원)
장점	빠르고 해석이 쉬움, 전통 검색 시스템과 호환성 높음	의미적으로 유사한 문장 검색 가능 ("동의어"도 인식 가능)
단점	단어가 다르면 매칭 실패 ("집" vs "주택")	임베딩 품질과 학습 데이터에 성능이 의존함
예시	Elasticsearch, Lucene, BM25	FAISS, Pinecone, Chroma

🧠 2. 희소 리트리버 (Sparse Retriever)

💬 원리

희소 리트리버는 문서를 **단어의 빈도 기반**으로 표현하는 전통적인 정보 검색(IR) 방식입니다.

각 문서는 단어의 등장 횟수에 따라 **희소 벡터(Sparse Vector)** 형태로 표현됩니다.

예를 들어, `"MongoDB is fast"`라는 문장은 다음과 같이 벡터화됩니다.

→ `[mongoDB:1, is:1, fast:1, ...]`

💡 대표 알고리즘

- **TF-IDF (Term Frequency - Inverse Document Frequency)**

  자주 등장하지만 전체 문서에서 너무 흔한 단어의 가중치를 낮추는 방식입니다.

- **BM25**

  TF-IDF를 개선한 방식으로, 문서 길이 보정 등을 추가하여 정확도를 높인 알고리즘입니다.

✅ 장점

- 단어 일치 정확도가 높습니다.
- 인덱싱과 검색 속도가 빠릅니다.
- 검색 결과를 해석하기 쉽습니다. ("이 단어가 일치했기 때문에 검색됨")

❌ 단점

- 의미적 유사성을 반영하지 못합니다.

  예를 들어 "자동차"와 "차량"은 완전히 다른 단어로 인식됩니다.

🧬 3. 밀집 리트리버 (Dense Retriever)

💬 원리

밀집 리트리버는 문장 전체의 의미를 **임베딩 벡터(embedding vector)** 로 변환한 뒤,

벡터 간 유사도(코사인 유사도 등) 를 계산하여 관련 문서를 찾습니다.

즉, 단어의 일치 여부가 아니라 **의미의 유사성**을 기준으로 검색합니다.

💡 대표 알고리즘

- **DPR (Dense Passage Retrieval)** — Facebook AI (2020)
- **Sentence-BERT (SBERT)** — 문장 단위 의미 임베딩
- **OpenAI Embeddings (text-embedding-3-small/large)**
- **E5, Cohere Embeddings 등**

✅ 장점

- 동의어나 유사한 의미의 문장을 잘 인식합니다.

  예: "차량 정비 방법" ↔ "자동차 수리 가이드"

- 문맥 기반 검색이 가능합니다.

❌ 단점

- 임베딩 모델이 필요하여 학습 비용이 발생합니다.
- 인덱스 생성과 검색 속도가 BM25보다 느립니다.
- 검색 근거를 해석하기 어렵습니다. (“왜 검색되었는지”를 직관적으로 파악하기 힘듦)

🔀 4. 혼합형 (Hybrid Retriever)

최근에는 **희소 리트리버와 밀집 리트리버를 결합한 Hybrid Retriever** 방식이 많이 사용됩니다.

```
Score = α × Dense_Score + (1−α) × Sparse_Score
```

이 방식은 다음과 같은 장점을 가집니다.

- 키워드가 정확히 일치하는 경우에는 희소 리트리버의 강점을 활용합니다.
- 단어가 다르지만 의미가 유사한 경우에는 밀집 리트리버의 강점을 활용합니다.

대표 구현체로는 **LangChain의 EnsembleRetriever**,

또는 **Elasticsearch + FAISS 결합 구조**가 있습니다.

🧪 5. 간단한 예시

사용자 질문	희소 리트리버 결과	밀집 리트리버 결과
“자동차 수리 방법 알려줘”	`자동차 정비 매뉴얼.pdf`	`차량 관리 가이드.docx`
“MongoDB 집계 파이프라인 설명”	`$lookup 사용법.md`	`Aggregation Framework 개요.txt`

같은 의미의 질문이라도 밀집 리트리버는 **의미적으로 가까운 문서**를 찾아냅니다.

🧩 6. 정리 요약

구분	희소 리트리버	밀집 리트리버
핵심 기준	단어 일치	의미 유사도
특징	빠르고 해석이 쉬움	의미 기반 검색
구현 예	BM25, TF-IDF	DPR, SBERT, OpenAI Embeddings
사용 예	뉴스 검색, 위키 검색	챗봇, RAG QA 시스템
최적 조합	Hybrid (Sparse + Dense)	실제 RAG에서 가장 많이 사용됨

벡터 스토어 기반 리트리버는 벡터 스토어에 구현된 유사도 검색이나 MMR과 같은 검색 메서드를 사용하여 벡터 스토어 내의 텍스트를 검색합니다.

벡터 스토어를 생성하고 as_retriever() 메서드로 호출하면 Chroma나 FAISS 같은 벡터 스토어에 종속된 리트리버를 얻을 수 있습니다.

LangChain에서 `Retriever`를 사용할 때,

`search_type`과 `search_kwargs`는 **검색 전략과 검색 동작을 세밀하게 제어**하는 핵심 파라미터입니다.

이 두 개념을 정확히 이해하면, **RAG 성능과 검색 품질**을 크게 개선할 수 있습니다.

아래에서 단계적으로 정리하겠습니다.

🧠 1. 기본 구조 요약

LangChain에서 벡터스토어(VectorStore) 객체는 다음처럼 **retriever**로 변환할 수 있습니다.

```
retriever = vectorstore.as_retriever(
    search_type="similarity",
```

```
    search_kwargs={"k": 3}
)
```

즉,

- `search_type`: 어떤 **검색 방식(전략)** 을 사용할지 결정
- `search_kwargs`: 검색 시 **세부 파라미터(k값 등)** 를 설정

⚙️ 2. `search_type` — 검색 방식 선택

`search_type`은 벡터스토어에서 **유사 문서 검색 전략**을 지정합니다.

기본적으로 다음 네 가지가 자주 사용됩니다:

search_type	설명
`"similarity"`	기본값. **질문 벡터와 가장 유사한 문서 k개**를 반환합니다.
`"similarity_score_threshold"`	유사도 점수가 특정 임계값 이상인 문서만 반환합니다.
`"mmr"` (Maximal Marginal Relevance)	**유사도 + 다양성 균형 검색.** 비슷한 문서만 몰리지 않도록 조정합니다.
`"similarity_limit"` (일부 백엔드 한정)	특정 거리 이하의 결과만 제한적으로 반환합니다.

🔍 (1) similarity (기본)

가장 단순하고 자주 쓰이는 방식입니다.

```
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 3}
)
```

- `k`: 상위 몇 개 문서를 반환할지 지정 (기본값은 4)
- 예: “MongoDB $lookup 사용법” → 코사인 유사도 기준으로 가장 가까운 3개 문서 반환

장점: 빠르고 단순함

단점: 비슷한 내용의 문서가 중복되어 나올 수 있음

🔍 (2) similarity_score_threshold

유사도가 일정 기준 이상인 문서만 반환합니다.

```
retriever = vectorstore.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"score_threshold": 0.8, "k": 5}
)
```

- `score_threshold`: 최소 유사도 점수 (0~1)
- `k`: 최대 반환 개수 (단, 기준 미달이면 더 적게 반환됨)

장점: 의미 없는 문서(유사도 낮은 문서)를 거를 수 있음

단점: 너무 높게 설정하면 결과가 없을 수도 있음

🔍 (3) mmr (Maximal Marginal Relevance)

비슷한 문서가 중복되지 않도록, **유사성과 다양성**을 동시에 고려합니다.

```
retriever = vectorstore.as_retriever(
    search_type="mmr",
    search_kwargs={"k": 5, "fetch_k": 20, "lambda_mult": 0.5}
)
```

- `fetch_k` : 우선 상위 몇 개 문서를 후보로 가져올지 (기본값 20)(초기에 MMR 알고리즘에 전달할 문서 수를 지정하며, 기본값은 20입니다. 먼저 MMR 알고리즘에 따라 20개의 문서를 검색하고, 그중에서 다양성을 고려해서 k개의 문서를 걸러내는 작업을 한 번 더 합니다.)
- `lambda_mult` : 유사도(1) vs 다양성(0) 가중치 조정 (기본값 0.5)
  - `1.0` → 유사도만 중시
  - `0.0` → 다양성만 중시

장점: 비슷한 문서가 여러 개 중복될 때 유용함

단점: 계산량이 조금 많음

🔍 (4) similarity_limit (일부 구현체 한정)

유사도가 특정 거리 이하인 결과만 반환합니다.

예: FAISS에서는 L2 거리가 기준이 될 수 있습니다.

(단, 모든 VectorStore에서 지원되지는 않습니다.)

⚙️ 3. `search_kwargs` — 검색 매개변수

`search_kwargs`는 검색 시 동작을 미세 조정하는 파라미터 딕셔너리입니다.

`search_type`에 따라 사용할 수 있는 키가 달라집니다.

search_type	주요 search_kwargs	설명
`"similarity"`	`k`	반환할 문서 개수
`"similarity_score_threshold"`	`k` , `score_threshold`	문서 수와 최소 유사도 기준
`"mmr"`	`k` , `fetch_k` , `lambda_mult`	다양성 조정용 파라미터
(기타)	`filter`	특정 메타데이터 필터링 (예: `{ "source": "manual" }`)

🔧 예시 1 — 단순 유사도 검색

```
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 3}
)
```

→ 상위 3개 문서 반환

🔧 예시 2 — 유사도 점수 기준 필터

```
retriever = vectorstore.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"score_threshold": 0.8, "k": 10}
)
```

→ 유사도가 0.8 이상인 문서만 반환, 최대 10개

🔧 예시 3 — MMR(다양성 포함)

```
retriever = vectorstore.as_retriever(
    search_type="mmr",
    search_kwargs={"k": 5, "fetch_k": 20, "lambda_mult": 0.5}
)
```

→ 20개 중 후보를 뽑고, 유사도와 다양성을 반영해 최종 5개 선택

💡 4. 함께 사용할 수 있는 고급 옵션

옵션	설명
`filter`	문서의 메타데이터 기준으로 필터링 가능 (예: 특정 소스만 검색)
`callbacks`	검색 과정 로그를 실시간으로 출력할 때 사용
`max_marginal_relevance_search`	`.similarity_search_with_relevance_scores()` 등과 조합 가능

🧩 5. 실제 예시 코드 (FAISS + OpenAIEmbeddings)

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()
vectorstore = FAISS.load_local("faiss_index", embeddings)

# 다양한 검색 방식 시도
retriever_default = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 3}
)

retriever_mmr = vectorstore.as_retriever(
    search_type="mmr",
    search_kwargs={"k": 5, "fetch_k": 15, "lambda_mult": 0.6}
)
```

🧾 정리 요약

항목	설명
search_type	검색 알고리즘 선택 (`similarity` , `mmr` , `similarity_score_threshold`)
search_kwargs	각 알고리즘의 세부 파라미터 (k, score_threshold, lambda_mult 등)
기본값	`"similarity"` , `{"k": 4}`
추천 조합	RAG 시스템에서는 다양성을 위해 `"mmr"` 방식 권장

ConfigurableField

LangChain의 `ConfigurableField`는 **체인이나 리트리버의 설정값(search_type, search_kwargs 등)을 코드 실행 시점에 동적으로 변경**할 수 있도록 설계된 **"동적 구성(Dynamic Configuration)" 메커니즘**입니다.

즉,

> search_type이나 search_kwargs를 .as_retriever()에서 고정하지 않고,
>
> `.invoke()` 호출 시점에 바꿀 수 있게 해주는 기능입니다.

아래에서 단계별로 설명하겠습니다.

🧩 1. 기본 개념 — ConfigurableField란?

`ConfigurableField`는 LangChain의 구성요소(예: Chain, Retriever, Tool 등)에

런타임에 변경 가능한 설정 필드를 정의할 때 사용하는 데코레이터입니다.

일반적으로 체인이나 리트리버는 초기화할 때 설정값이 고정됩니다.

하지만 `ConfigurableField`를 사용하면,

이 값을 `.invoke()` 또는 `.run()`을 호출할 때마다 동적으로 바꿀 수 있습니다.

🔧 예시 개념

```
from langchain.schema import ConfigurableField

class MyRetriever:
    search_type: str = ConfigurableField(default="similarity")
    k: int = ConfigurableField(default=3)
```

이렇게 정의된 경우,

`.invoke()` 실행 시점에 다음처럼 매개변수를 바꿀 수 있습니다.

```
retriever.invoke("MongoDB $lookup", config={"configurable": {"search_type": "mmr", "k": 5}})
```

→ 실행 시점에서 `search_type="mmr"`, `k=5`로 덮어쓰기 됩니다.

⚙️ 2. 왜 필요한가?

보통 `.as_retriever()`로 리트리버를 만들 때는 `search_type`, `search_kwargs`가 고정됩니다.

```
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 3}
)
```

하지만 이런 경우에는 **검색 방식이나 반환 개수(k)** 를

나중에 코드 실행 중에 바꿀 수 없습니다.

예를 들어 이런 상황이 있을 수 있습니다:

- "요약 모드"에서는 `k=2`
- "분석 모드"에서는 `k=8`
- "다양성 우선 모드"에서는 `mmr`
- "정확도 우선 모드"에서는 `similarity`

이때 매번 리트리버를 새로 만들 필요 없이,

ConfigurableField를 사용하면 한 번 만든 객체로 여러 설정을 동적으로 바꿀 수 있습니다.

🧠 3. LangChain 내 동작 원리

LangChain 내부에서는 `ConfigurableField`가 다음과 같이 작동합니다:

- 체인이나 리트리버 클래스 내부에서 `ConfigurableField`로 선언된 필드는
  `config={"configurable": {...}}` 인자를 통해 `.invoke()` 시점에 오버라이드 가능.
- 오버라이드된 설정은 **해당 실행 컨텍스트 내에서만 적용**되고,
  이후에는 기본값으로 되돌아갑니다 (즉, 영구 변경은 아님).

🧪 4. 실제 예시 (FAISS 리트리버)

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

embeddings = OpenAIEmbeddings()
vectorstore = FAISS.load_local("faiss_index", embeddings)
retriever = vectorstore.as_retriever(
    search_type="similarity",
```

```
    search_kwargs={"k": 3}
)

qa = RetrievalQA.from_chain_type(
    llm=ChatOpenAI(model="gpt-4o-mini"),
    retriever=retriever
)

# 기본 설정: similarity, k=3
print(qa.invoke("MongoDB $lookup 사용법이 뭐야?"))

# 실행 시점에서 동적으로 변경
print(qa.invoke(
    "MongoDB의 join과 $lookup의 차이점은?",
    config={"configurable": {"search_type": "mmr", "search_kwargs": {"k": 5, "lambda_mult": 0.5}}}
))
```

결과

- 첫 번째 호출 → similarity 기반, k=3 문서 검색
- 두 번째 호출 → mmr 기반, k=5 문서 검색 (유사도 + 다양성 조합)

문서압축기

"문서 압축기(Document Compressor)"는 RAG(Re-trieval Augmented Generation) 시스템에서 **검색된 문서의 양을 줄이면서도 핵심 정보를 보존하기 위한 장치**입니다.

즉, LLM이 처리하기 쉬운 형태로 문서를 "요약·선택·가공"하는 역할을 합니다.

🧠 1. 등장 배경

RAG 시스템의 일반적인 흐름은 다음과 같습니다:

```
사용자 질문 → Retriever → 관련 문서(예: 10개) → LLM 생성
```

하지만 현실적으로:

- 검색된 문서가 너무 많거나,
- 각 문서가 길어서 토큰 제한을 초과하거나,
- LLM이 중요하지 않은 내용을 함께 보게 되어 품질이 떨어질 때가 있습니다.

이 문제를 해결하기 위해 나온 개념이 바로 **문서 압축기(Document Compressor)** 입니다.

⚙️ 2. 문서 압축기의 역할

문서 압축기는 **Retriever가 반환한 문서 집합을 LLM이 보기 좋은 형태로 정제**합니다.

주된 역할은 다음과 같습니다.

역할	설명
요약(Summarization)	각 문서의 핵심만 남겨 토큰 수를 줄임
선택(Filtering)	질문과 관련 없는 문서는 제거
부분 발췌(Chunk Trimming)	긴 문서의 일부분만 추출
결합(Merging)	여러 문서의 중복된 내용을 하나로 합침

즉, 문서 압축기는

> "Retriever 결과 → LLM 입력" 사이의 중간 전처리 필터
>
> 라고 이해하면 됩니다.

🧩 3. LangChain에서의 문서 압축기 구조

LangChain에는 기본적으로 `DocumentCompressorPipeline`이 존재하며,

다음과 같은 구조로 동작합니다.

```
Retriever
  ↓
Compressor (문서 압축기)
  ↓
LLM (생성)
```

LangChain에서는 `ContextualCompressionRetriever`라는 클래스로 이 과정을 감쌉니다.

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor

compressor = LLMChainExtractor.from_llm(ChatOpenAI(model="gpt-4o-mini"))
retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=vectorstore.as_retriever()
)
```

이제 검색된 문서는 **LLMChainExtractor**를 통해 자동으로 "요약·정제"되어 전달됩니다.

📦 4. 문서 압축기의 주요 유형

유형	설명	예시
LLM 기반 요약형	LLM이 각 문서를 요약하여 압축	`LLMChainExtractor`, `LLMDocumentCompressor`
Score 기반 필터형	질문과의 유사도 점수를 기준으로 상위 문서만 선택	`ScoreThresholdDocumentCompressor`
길이 제한형	토큰 수가 너무 긴 문서를 자르거나 줄임	`EmbeddingsClippingCompressor`
혼합형	위 방법들을 조합하여 단계별로 압축	`DocumentCompressorPipeline`

🔍 5. 동작 예시

(1) 일반 리트리버

```
retriever = vectorstore.as_retriever(search_kwargs={"k": 5})
docs = retriever.get_relevant_documents("Kafka consumer offset 관리 방법")
print(len(docs))  # 5개 문서
```

(2) 압축 리트리버 추가

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor

compressor = LLMChainExtractor.from_llm(ChatOpenAI(model="gpt-4o-mini"))
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=retriever
)

compressed_docs = compression_retriever.get_relevant_documents("Kafka consumer offset 관리 방법")
print(len(compressed_docs))  # 예: 2~3개로 압축
```

→ LLMChainExtractor가 각 문서를 요약하고 불필요한 문장을 제거하여

LLM이 이해하기 쉽게 줄여줍니다.

📉 6. 문서 압축기의 장점

항목	설명
🔷 **토큰 절감**	입력 길이를 줄여 비용과 속도 절감
🔷 **정확도 향상**	불필요한 문맥 제거로 LLM 집중도 상승
🔷 **중복 제거**	비슷한 내용의 문서 병합 또는 생략
🔷 **확장성**	다른 압축기 조합으로 품질 조정 가능

⚠️ 7. 주의할 점

주의사항	설명
정보 손실 위험	요약 과정에서 중요한 세부 내용이 빠질 수 있음
모델 의존성	LLM이 압축을 잘못 수행하면 오히려 왜곡 가능
시간 증가 가능성	압축 단계에 LLM 호출이 추가되므로 약간의 지연이 생김

💡 8. 실제 적용 예시

```
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI
from langchain.retrievers.document_compressors import LLMChainExtractor
from langchain.retrievers import ContextualCompressionRetriever

# 1) 기본 리트리버 생성
retriever = vectorstore.as_retriever(search_kwargs={"k": 8})

# 2) 문서 압축기 연결
compressor = LLMChainExtractor.from_llm(ChatOpenAI(model="gpt-4o-mini"))
compression_retriever = ContextualCompressionRetriever(
    base_retriever=retriever,
    base_compressor=compressor
)

# 3) QA 체인 구성
qa_chain = RetrievalQA.from_chain_type(
    llm=ChatOpenAI(model="gpt-4o-mini"),
    retriever=compression_retriever
)

# 4) 질의
response = qa_chain.invoke({"query": "Kafka의 offset commit 전략은 어떻게 관리되나요?"})
print(response["result"])
```

→ 이 구조는 “RAG + 문서 압축기” 형태로

성능과 효율을 동시에 높이는 대표적인 패턴입니다.

🧾 9. 정리 요약

항목	내용
정의	검색된 문서에서 핵심 내용만 추려 LLM에 전달하는 구성요소
역할	요약, 필터링, 길이 제한, 중복 제거 등

항목	내용
대표 클래스	`LLMChainExtractor` , `ContextualCompressionRetriever`
장점	토큰 절감, 품질 향상, 중복 제거
주의점	과도한 요약 시 정보 손실 가능

요약하자면,

> "문서 압축기"는 RAG의 리트리버가 가져온 방대한 문서를
> **요약·선별하여 LLM이 효율적으로 이해할 수 있게 만드는 중간 필터**입니다.

🧩 문서 압축기(Document Compressor) 종류별 정리

LangChain의 문서 압축기는 주로 다음 네 가지 계열로 구분됩니다:

분류	대표 클래스	핵심 역할
① **LLM 기반 요약형**	`LLMChainExtractor` , `LLMDocumentCompressor`	LLM이 직접 문서를 요약·정제
② **점수 기반 필터형**	`ScoreThresholdDocumentCompressor`	질문과의 유사도가 낮은 문서를 제거
③ **길이 제한형**	`EmbeddingsClippingCompressor` , `TokenClippingCompressor`	너무 긴 문서의 일부만 잘라냄
④ **혼합형 파이프라인형**	`DocumentCompressorPipeline`	여러 압축기를 순서대로 적용

① LLM 기반 요약형

1️⃣ LLMChainExtractor

가장 널리 쓰이는 압축기입니다.

🔷 개념

- LLM을 이용해 각 문서를 **요약(summarize)** 하거나 **질문과 관련된 문장만 추출(extract)** 합니다.
- Retriever가 반환한 문서를 LLM이 재검토하여 "질문에 필요한 내용만 남기는" 방식입니다.

🔷 동작 방식

1. 각 문서와 사용자의 질문을 함께 LLM에게 전달
2. "이 질문에 관련된 부분만 남겨줘"라는 프롬프트를 통해 핵심만 추출
3. 압축된 결과를 최종 LLM 입력으로 전달

🔷 예시 코드

```
from langchain.retrievers.document_compressors import LLMChainExtractor
from langchain.chat_models import ChatOpenAI

compressor = LLMChainExtractor.from_llm(ChatOpenAI(model="gpt-4o-mini"))
```

🔷 장점

- 질문 중심으로 문맥을 유지하면서 불필요한 내용을 제거
- 압축 후에도 의미 왜곡이 적음 (LLM이 문맥 이해 기반으로 요약하기 때문)

🔷 단점

- LLM 호출이 추가되어 속도와 비용이 증가
- LLM이 부정확한 요약을 할 경우 정보 손실 가능

2️⃣ LLMDocumentCompressor

LLM이 **전체 문서를 한 번에 요약**하거나 **중요도를 재조정**하는 방식입니다.

🔷 특징

- `LLMChainExtractor`가 "부분 발췌"라면,
  `LLMDocumentCompressor`는 "전체 요약"에 가깝습니다.
- 긴 문서를 LLM이 한 번에 압축해 "간략 버전"으로 만들어냅니다.

🔷 예시

```
from langchain.retrievers.document_compressors import LLMDocumentCompressor
compressor = LLMDocumentCompressor.from_llm(ChatOpenAI(model="gpt-4o"))
```

🔷 장점

- 매우 긴 문서를 짧게 만들어 토큰 절감 효과 큼
- 전체 내용을 요약할 때 적합 (예: 기술 문서, 보고서 등)

🔷 단점

- 세부 정보가 손실될 가능성이 높음
- 질의와 직접적 관련성이 낮아질 수 있음

---

② 점수 기반 필터형

3️⃣ ScoreThresholdDocumentCompressor

🔷 개념

- 각 문서의 **질문과의 유사도 점수(similarity score)** 를 계산하고,
  임계값(threshold) 이하인 문서는 제거합니다.
- "LLM을 쓰지 않고도 빠르게 압축"할 수 있는 경량 압축기입니다.

🔷 동작 원리

1. 질문과 문서 임베딩 간의 유사도 계산
2. `score_threshold` 이상인 문서만 남김

🔷 예시 코드

```
from langchain.retrievers.document_compressors import ScoreThresholdDocumentCompressor

compressor = ScoreThresholdDocumentCompressor(
    threshold=0.8,  # 0~1 사이 (높을수록 엄격)
)
```

🔷 장점

- 속도가 빠르고 비용이 없음 (LLM 호출 X)
- 불필요한 문서를 손쉽게 걸러낼 수 있음

🔷 단점

- 단순 유사도 기준이라 문맥은 고려하지 않음
- 중요한 문서라도 임베딩 스코어가 낮으면 제외될 수 있음

---

③ 길이 제한형

4️⃣ EmbeddingsClippingCompressor

- 임베딩 벡터 차원 기준으로 문서를 "잘라내는" 압축기입니다.
- 일부 LangChain 백엔드에서 제공되는 경량화 기능입니다.

> (이 압축기는 주로 내부적으로 사용되며 직접 사용하는 경우는 적음)

5️⃣ TokenClippingCompressor

- 각 문서의 토큰 길이를 측정하고,
- LLM이 허용 가능한 최대 토큰 수를 넘지 않도록 "앞부분만 남기거나 잘라내는" 방식입니다.

🔷 예시

```
from langchain.retrievers.document_compressors import TokenClippingCompressor

compressor = TokenClippingCompressor(max_tokens=1000)
```

🔷 장점

- 초대형 문서를 단순히 토큰 수 기준으로 제한 가능
- 빠르고 간단함

🔷 단점

- 의미 단위로 자르지 않기 때문에, 문맥이 끊어질 수 있음

---

④ 혼합형 (파이프라인형)

6️⃣ DocumentCompressorPipeline

🔷 개념

- 여러 압축기를 **순서대로 조합해 체인 형태로 실행**합니다.
- 예를 들어 "유사도 필터 → LLM 요약 → 토큰 제한" 순으로 처리 가능.

🔷 예시 코드

```
from langchain.retrievers.document_compressors import (
    LLMChainExtractor, ScoreThresholdDocumentCompressor, DocumentCompressorPipeline
)
from langchain.chat_models import ChatOpenAI

compressor = DocumentCompressorPipeline(
    transformers=[
        ScoreThresholdDocumentCompressor(threshold=0.75),
        LLMChainExtractor.from_llm(ChatOpenAI(model="gpt-4o-mini"))
    ]
)
```

🔷 장점

- 다양한 압축 전략을 결합하여 균형 잡힌 품질 확보
- 상황에 따라 단계별 조정 가능

🔷 단점

- 속도가 느리고 구성 복잡도 증가

📊 5. 압축기별 비교 요약표

유형	주요 클래스	방식	LLM 사용	속도	정확도	적합한 상
요약형	`LLMChainExtractor`	질문 관련 부분만 추출	✅	느림	높음	질문 중심
요약형	`LLMDocumentCompressor`	전체 문서 요약	✅	느림	중간	문서 요약 리
필터형	`ScoreThresholdDocumentCompressor`	유사도 기준 필터링	❌	빠름	중간	빠른 검색,
길이형	`TokenClippingCompressor`	토큰 수 기준 자르기	❌	매우 빠름	낮음	대용량 텍 리
파이프라인형	`DocumentCompressorPipeline`	여러 압축기 조합	✅/❌	중간~느림	높음	고품질 RA

💡 6. 활용 조합 예시

목적	추천 압축기 조합
빠른 검색 + 기본 품질	`ScoreThresholdDocumentCompressor`
질문 중심 압축 (정확도 우선)	`LLMChainExtractor`
긴 문서 요약 (리포트 등)	`LLMDocumentCompressor`
최적 품질 RAG 구성	`DocumentCompressorPipeline` (Score + LLM 요약 결합)
메모리 제약 환경	`TokenClippingCompressor`

앙상블 리트리버

🧠 1. 개념 요약

앙상블 리트리버(Ensemble Retriever) 는

여러 종류의 리트리버를 **조합(ensemble)** 하여

각각의 장점을 살리고 단점을 보완하는 리트리버입니다.

즉,

> "하나의 검색 방식으로는 부족하니,
> 여러 리트리버의 결과를 합쳐 더 좋은 검색 결과를 만들자"
> 라는 개념입니다.

LangChain에서는 이 개념을 `EnsembleRetriever` 클래스로 제공합니다.

⚙️ 2. 동작 구조

앙상블 리트리버는 내부적으로 여러 리트리버를 병렬로 호출하여

그 결과들을 하나로 통합합니다.

```
사용자 질문
  ↓
[Retriever #1] (예: BM25)
[Retriever #2] (예: FAISS)
[Retriever #3] (예: Chroma)
  ↓
결과 결합 및 점수 가중 평균
  ↓
```

```
최종 결과 반환
```

이렇게 여러 검색기의 결과를 **가중치 기반으로 결합(weighted combination)** 하여

최종적으로 "가장 유용한 문서들"을 반환합니다.

🧩 3. 왜 필요한가?

각 리트리버는 강점이 다릅니다:

리트리버	강점	약점
희소 리트리버 (BM25)	키워드 정확히 일치하는 문서에 강함	동의어·의미 유사 문장엔 약함
밀집 리트리버 (FAISS, Chroma)	의미 기반 검색에 강함	정확한 키워드 일치엔 약함
메타데이터 리트리버	필터링 조건 검색에 강함	문맥 이해는 약함

따라서 하나의 리트리버만 쓰면 "놓치는 정보"가 생깁니다.

앙상블 리트리버는 이런 **검색 편향을 줄이고 리콜(recall)을 높이기 위해** 사용됩니다.

⚙️ 4. LangChain에서의 기본 구조

LangChain의 `EnsembleRetriever`는 다음처럼 사용합니다:

```
from langchain.retrievers import EnsembleRetriever
from langchain.retrievers import BM25Retriever
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

# 희소 리트리버 (BM25)
bm25_retriever = BM25Retriever.from_texts(
    ["MongoDB is fast", "Kafka offset management", "Redis single-thread concurrency"]
)

# 밀집 리트리버 (FAISS)
embeddings = OpenAIEmbeddings()
faiss_db = FAISS.from_texts(
    ["MongoDB aggregation pipeline", "Kafka consumer commit", "Redis DECR atomicity"],
    embeddings
)
faiss_retriever = faiss_db.as_retriever(search_kwargs={"k": 3})

# 앙상블 리트리버 생성
ensemble_retriever = EnsembleRetriever(
    retrievers=[bm25_retriever, faiss_retriever],
    weights=[0.4, 0.6]  # 각 리트리버의 가중치
)

# 검색
results = ensemble_retriever.get_relevant_documents("Kafka offset commit strategy")
for i, doc in enumerate(results):
    print(f"--- 결과 {i+1} ---\n{doc.page_content}\n")
```

⚖️ 5. 결합 방식

앙상블 리트리버는 단순히 결과를 합치는 것이 아니라,

각 리트리버의 점수를 정규화(normalize) 한 뒤

가중 평균(weighted average) 으로 최종 점수를 계산합니다.

```
final_score = w1 * score_from_retriever1 + w2 * score_from_retriever2 + ...
```

이 방식 덕분에:

- 특정 리트리버에 너무 치우치지 않고,
- 의미 기반과 키워드 기반 결과를 균형 있게 섞을 수 있습니다.

🔍 6. 예시 시나리오

상황	추천 조합	설명
기술 문서 검색	BM25 + FAISS	키워드(명령어, 함수명) + 의미 유사도 모두 활용
논문 검색	FAISS + Chroma	의미 중심 검색 강화
질의응답 챗봇	BM25 + Dense + MetadataRetriever	키워드, 문맥, 조건 기반 검색 결합
뉴스 검색	BM25 + TF-IDF + Dense	제목 기반 + 의미 기반 결합

📊 7. 성능 비교 (예시)

리트리버	정확도(Precision)	재현율(Recall)
BM25	0.85	0.60
Dense (FAISS)	0.75	0.82
앙상블 (0.4, 0.6)	**0.83**	**0.84**

→ 두 리트리버를 조합하면 **정확도와 재현율 모두 상승**하는 경향이 있습니다.

⚙️ 8. 파라미터 설명

매개변수	설명
`retrievers`	결합할 리트리버 리스트
`weights`	각 리트리버의 가중치 (합이 1이 되도록 설정 권장)
`normalize`	각 리트리버의 점수를 0~1로 정규화할지 여부 (기본값 True)
`k`	최종 반환할 문서 개수

💡 9. 장점과 단점

항목	장점	단점
🎯 **정확도**	여러 검색 전략을 결합해 검색 품질 향상	설정 잘못하면 오히려 혼선
⚡ **유연성**	리트리버별 가중치로 맞춤 조정 가능	각 리트리버 실행 비용 증가
🧠 **확장성**	추가 리트리버 쉽게 결합 가능	검색 속도는 느려질 수 있음

🧩 10. 확장 응용 — ContextualCompressionRetriever와 결합

앙상블 리트리버는 보통 **문서 압축기(Document Compressor)** 와 함께 사용됩니다.

예를 들어:

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor
from langchain.chat_models import ChatOpenAI
```

```
compressor = LLMChainExtractor.from_llm(ChatOpenAI(model="gpt-4o-mini"))
compression_retriever = ContextualCompressionRetriever(
    base_retriever=ensemble_retriever,
    base_compressor=compressor
)
```

이렇게 하면:

1. 여러 리트리버 결과를 결합하고
2. LLM으로 문서를 요약하여
3. 최종적으로 "가장 핵심적인 문서만" 남깁니다.

긴 문맥 재정렬

🧠 1. 개념 요약

긴 문맥 재정렬(Long Context Reordering) 이란,

리트리버가 가져온 여러 문서(청크, passages)를

> "LLM이 이해하기 쉬운 순서로 재배치(Reorder)"하는 과정
>
> 을 말합니다.

즉,

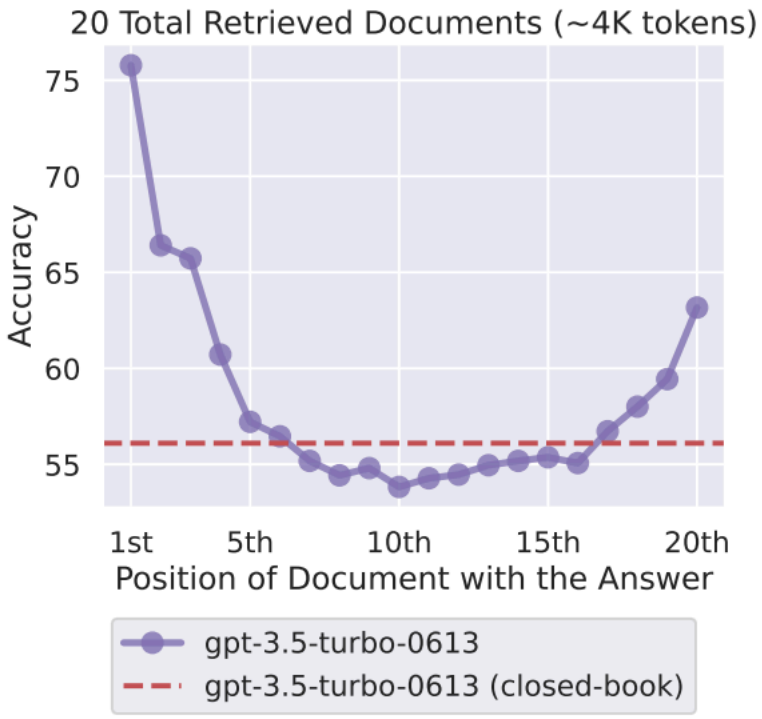
RAG의 흐름 중 아래 단계에 해당합니다:

```
[사용자 질문]
  ↓
[Retriever]  → 관련 문서 N개
  ↓
[Reordering] → 문서 순서 재조정 (중요도/논리순)
  ↓
[LLM] → 답변 생성
```

이 과정은 **"어떤 문서를 먼저 보여줄지"** 가 답변 품질에 큰 영향을 주기 때문에

RAG의 고급 튜닝 단계에서 매우 중요합니다.

⚙️ 2. 왜 필요한가?



논문의 실험에 따르면 20개의 문서를 검색했을때 정확도와 문서의 위치를 그래프를 나타냈더니 위와 같은 패턴이 나타났습니다.
중요한 정보가 문장의 맨 앞이나 맨 끝에 있을 때는 성능이 좋지만, 중간에 있을 때는 성능이 뚝 떨어집니다.
그래서 검색 결과에서 중요한 정보를 양 끝단에 배치하고, 관련성이 떨어지는 정보는 중간에 배치하자는 것이 이 논문의 핵심입니다.

🧩 3. 긴 문맥 재정렬의 주요 전략

긴 문맥 재정렬에는 여러 방식이 있습니다.

대표적인 4가지 전략을 정리하면 다음과 같습니다:

유형	개념	대표 알고리즘 / 구현체
① **유사도 기반 Re-ranking**	기존 retriever 결과를 LLM 또는 다른 모델이 다시 점수화	`Cross-Encoder`, `ColBERT`, `SentenceTransformers`
② **질문 중심 재정렬 (Query-aware ordering)**	질문 내 핵심 키워드와 문서 내용을 비교해 중요도 순 정렬	LangChain `ParentDocumentRetriever`, OpenAI ReRank API
③ **문맥 연결 기반 재정렬 (Contextual reordering)**	문서 간 연관성을 고려해 논리적 순서로 배치	RAG Fusion, Contextual Reorderer
④ **LLM 기반 동적 재정렬**	LLM에게 "이 문서들을 어떤 순서로 읽으면 좋을지" 판단시킴	`LLMReorderer`, ReAct 기반 LLM Judge

---

🔍 4. 대표적인 재정렬 기법들

1️⃣ Cross-Encoder Re-ranking

- 검색된 문서들을 다시 LLM (또는 작은 Transformer 모델)에 넣어
  "질문과의 연관성 점수"를 재계산합니다.
- 예: `ms-marco-MiniLM-L-6-v2`, `bge-reranker-large` 등.

```
from sentence_transformers import CrossEncoder

model = CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")
```

```
scores = model.predict([(query, doc) for doc in retrieved_docs])
sorted_docs = [x for _, x in sorted(zip(scores, retrieved_docs), reverse=True)]
```

> 장점: 정확한 순위 재조정
>
> 단점: 비용이 큼 (각 문서마다 모델 호출 필요)

2️⃣ LangChain 내 LLM 기반 Reorderer

LangChain은 간단하게 사용할 수 있는 재정렬기 `LongContextReorderer`를 제공합니다.

```
from langchain.retrievers.document_compressors import LongContextReorderer

reorderer = LongContextReorderer()
reordered_docs = reorderer.compress_documents(retrieved_docs, query)
```

🔷 동작 원리

- 여러 문서를 받아서, 질문과 문서 간의 연관성을 재평가하고
- LLM이 긴 문맥을 이해하기 좋은 순서(논리적 흐름, 시간 순 등)로 재배열합니다.

🔷 특징

- 단순한 임베딩 스코어 기반이 아닌, **문맥 기반 순서 조정**
- LLM 입력 길이가 길더라도 정보 손실 없이 자연스러운 답변 유도 가능

▼ 왜 LangChain이 `compress_documents()`로 옮겼나?

🧠 3. 왜 LangChain이 `compress_documents()`로 옮겼나?

이유는 세 가지입니다.

(1) RAG 파이프라인 통합

LangChain이 `Retriever → Compressor → LLM` 구조를 도입하면서,

"문서를 재정렬하거나 요약하는 행위"를 **압축(Compression)** 으로 통일했습니다.

즉,

```
과거: transform_documents()
현재: compress_documents()
```

이렇게 바뀐 이유는

"단순한 변환(transform)"보다 "입력 축소(compression)"가 RAG에서 더 본질적인 동작이기 때문입니다.

---

(2) ContextualCompressionRetriever 도입

`ContextualCompressionRetriever`가 추가되면서,

내부적으로 `compress_documents()`를 호출해야만 작동하도록 설계되었습니다.

```
compression_retriever = ContextualCompressionRetriever(
    base_retriever=retriever,
    base_compressor=LongContextReorderer()
)
```

즉, `transform_documents()` 기반의 객체는

이 구조 안에서 작동하지 않습니다.

(→ 유지보수 대상에서 제외됨)

(3) 명시적 역할 구분

LangChain 팀은 "transformer"라는 용어가

'데이터 포맷 변환'과 혼동된다는 점을 인식했습니다.

예를 들어,

- `HTML2TextTransformer` → 문서 포맷 변환
- `LLMChainExtractor` → 문서 요약

이 두 가지는 완전히 다른 목적이지만 이름은 둘 다 "transformer"로 불렸습니다.

그래서 문서 "요약·선택·정렬"과 관련된 로직은 전부 **"Compressor" 계열**로 이동한 것입니다.

📦 4. 실제 코드 비교

🔶 과거 방식 (비권장)

```
from langchain.document_transformers import LongContextReorder
docs = LongContextReorder().transform_documents(docs)
```

🔷 현재 표준 방식 (권장)

```
from langchain.retrievers.document_compressors import LongContextReorderer
docs = LongContextReorderer().compress_documents(docs, query="Kafka offset commit 관리")
```

→ 현재는 후자가 LangChain 팀이 공식적으로 유지보수·업데이트하는 API입니다.

전자는 **레거시(legacy)** 코드로 남아 있으며,

신규 버전에서는 경고가 출력되거나 완전히 제거될 예정입니다.

3️⃣ RAG-Fusion (Rank Fusion)

- 서로 다른 리트리버의 결과를 합친 뒤,

  각 문서의 순위를 평균내거나 합산하여 **통합 순위**를 산출합니다.

```
Final_Score = (Rank_in_RetrieverA + Rank_in_RetrieverB) / 2
```

> BM25와 DenseRetriever를 결합할 때 자주 사용됩니다.
>
> 이는 "앙상블 리트리버"와 유사하지만, **결합이 아니라 순위 재조정(rerank)** 에 더 가깝습니다.

4️⃣ LLM 기반 Judge Reorderer

- LLM에게 "이 문서들을 어떤 순서로 읽는 게 가장 이해하기 좋을까?"라고 직접 물어보는 방식입니다.

```
prompt = """
아래 문서들을 읽기 좋은 순서로 정렬해주세요.
사용자 질문: {query}
문서:
{docs}
"""
```

> 고급 RAG 시스템(예: OpenAI ReRank API, Anthropic Claude RAG 등)에서 활용되는 고급 방식입니다.

⚙️ 5. LangChain 구조 내 위치

LangChain에서는 "문서 압축기(Document Compressor)"나 "ContextualCompressionRetriever" 내부에

재정렬기(Reorderer) 를 넣어 사용할 수 있습니다.

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LongContextReorderer

reorderer = LongContextReorderer()
compression_retriever = ContextualCompressionRetriever(
    base_retriever=vectorstore.as_retriever(),
    base_compressor=reorderer
)
```

→ 이 구조에서는

1️⃣ Retriever가 문서를 찾고,

2️⃣ Reorderer가 순서를 재조정한 후,

3️⃣ 최종적으로 LLM에 전달됩니다.

🧾 6. 긴 문맥 재정렬의 장단점

항목	장점	단점
🎯 **정확도 향상**	핵심 문서를 앞에 배치하여 답변 품질 상승	LLM 호출 또는 추가 모델 필요
🧠 **논리 흐름 개선**	문서 간 시간·원인·결과 관계를 유지	순서 판단이 잘못되면 오히려 혼란 유발
🛏️ **토큰 효율 개선**	덜 중요한 문서를 뒤로 보내거나 생략 가능	추가 처리 시간 소요

부모 문서 리트리버

🧠 1. 개념 요약

Parent Document Retriever는

> 검색 효율성을 위해 문서를 "작은 청크 단위로 나눠서 저장"하지만,
>
> 실제로 LLM에게는 "더 큰 단위(부모 문서)"를 다시 복원해서 제공하는 리트리버입니다.

즉,

검색은 빠르게(짧은 청크 기준)

→ 응답 생성은 풍부하게(원래 긴 문맥 기준)

이 두 가지를 모두 달성하기 위한 구조입니다.

🔍 간단히 도식화하면

```
원본 문서 (부모)
 ├── Chunk 1
 ├── Chunk 2
 ├── Chunk 3
```

리트리버는 **Chunk 단위로 검색**하지만,

검색된 Chunk가 속한 **부모 문서 전체**를 다시 LLM에 전달합니다.

⚙️ 2. 왜 필요한가?

일반 RAG의 한계

기본 RAG 구조에서는 문서를 쪼개서(Chunking) 벡터로 저장합니다.

하지만 이렇게 하면 다음 문제가 생깁니다:

문제	설명
🔷 **문맥 손실**	검색된 chunk는 질문과 관련 있지만, 앞뒤 맥락이 사라짐
🔷 **정보 불충분**	중요한 세부 내용이 chunk 경계에 걸려서 일부만 포함됨
🔷 **LLM 혼란**	문서 조각이 중간만 포함되어 있어서 흐름이 부자연스러움

따라서 chunk 단위로만 검색하면 답변 품질이 떨어질 수 있습니다.

👉 그래서 **Parent Document Retriever**는

검색은 작은 단위로, 전달은 큰 단위로 함으로써

정확도 + 문맥 유지를 동시에 확보합니다.

🧩 3. 동작 구조

Parent Document Retriever는 두 단계로 동작합니다:

```
[1] Parent 문서를 여러 Chunk로 분할
    ↓
[2] 각 Chunk를 벡터 DB에 저장 (검색 효율 ↑)
    ↓
[3] 검색 시, 관련된 Chunk를 찾음
    ↓
[4] 해당 Chunk의 부모 문서를 찾아 LLM에 전달
```

⚙️ 4. LangChain 구조 예시

LangChain에서 Parent Document Retriever는

`ParentDocumentRetriever` 클래스로 구현되어 있습니다.

```
from langchain.retrievers import ParentDocumentRetriever
from langchain.storage import InMemoryStore
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS

# 1️⃣ 벡터 스토어와 부모 문서 저장소 설정
vectorstore = FAISS.from_texts([], embedding=OpenAIEmbeddings())
docstore = InMemoryStore()  # 부모 문서 저장소 (메모리 기반)

# 2️⃣ 부모 문서 리트리버 생성
retriever = ParentDocumentRetriever(
    vectorstore=vectorstore,
    docstore=docstore,
    child_splitter=RecursiveCharacterTextSplitter(chunk_size=300, chunk_overlap=50),
)

# 3️⃣ 문서 추가 (부모 단위)
retriever.add_texts([
    "Kafka의 오프셋은 메시지의 위치를 나타내며, 컨슈머가 메시지를 어디까지 읽었는지를 의미한다.",
    "Redis의 DECR 명령은 원자적 연산으로, 동시성 환경에서 안전하게 카운터를 감소시킨다."
])

# 4️⃣ 검색 실행
results = retriever.get_relevant_documents("Kafka offset commit 방법")
```

```
for r in results:
    print(r.page_content)
```

🔷 내부 동작

- `add_texts()` 시, 문서가 작은 chunk로 나뉘어 **vectorstore**에 저장됨
- 각 chunk는 "부모 ID"를 메타데이터로 가짐
- 검색 시 관련 chunk를 찾으면, **InMemoryStore**에서 부모 문서를 조회하여 전체 반환

🧱 5. ParentDocumentRetriever의 구성요소

구성요소	역할
vectorstore	chunk 단위로 임베딩이 저장되는 벡터 DB
docstore	부모 문서 전체가 저장되는 스토리지
child_splitter	부모 문서를 chunk로 쪼개는 분할기
search_kwargs	검색할 chunk 개수(k) 등 세부 설정
retriever.get_relevant_documents(query)	chunk 검색 → 부모 문서 반환

🧠 6. 핵심 아이디어

ParentDocumentRetriever는 "검색과 생성 단위"를 다르게 봅니다.

구분	ParentDocumentRetriever	일반 Retriever
검색 단위	작은 chunk	작은 chunk
LLM 입력 단위	부모 문서 전체	chunk 그대로
문맥 유지	✅ 좋음	❌ 단절됨
검색 속도	✅ 빠름	✅ 빠름
정확도	✅ 향상	⚠️ 낮음

즉, **retrieval granularity(검색 단위)** 와

generation granularity(생성 단위) 를 분리한 구조입니다.

⚡ 7. 장점과 단점

구분	장점	단점
✅ **장점**	- LLM이 문맥을 완전하게 이해- 중요한 세부정보 누락 방지- chunk 단위보다 자연스러운 답변	
⚠️ **단점**	- 부모 문서가 너무 길면 토큰 사용량이 증가- 동일 부모가 여러 번 검색될 수 있음- 메모리 저장소(docstore)가 추가로 필요	

다중 쿼리 생성 리트리버

사용자가 입력한 질문을 다양한 관점에 따라 여러 질문으로 변환하는 리트리버입니다.
사용자는 질문을 작성할 때 생각보다 자세하고 친절하게 내용을 써 주지 않습니다.
사용자의 불친절한 질문을 더욱 정확한 쿼리로 다듬어 주는 후처리 역할을 합니다.

🧠 1. 개념 요약

다중 쿼리 생성 리트리버(Multi-Query Retriever) 는

> 하나의 사용자 질문을 여러 형태의 "의미적으로 다른 쿼리"로 바꿔서
>
> 각각 검색한 결과를 합치는 리트리버입니다.

즉,

사용자 질문이 하나라도 → 내부적으로 **다양한 표현으로 변형된 여러 쿼리**를 생성해

리트리버가 놓칠 수 있는 문서를 더 많이 찾아냅니다.

🔍 간단한 예시

사용자 질문이 다음과 같다고 해보겠습니다.

> "Kafka consumer offset commit 방식은 어떻게 관리되나요?"

Multi-Query Retriever는 LLM을 이용해 이런 식으로 여러 쿼리를 자동 생성합니다 👇

1️⃣ Kafka의 오프셋 커밋 전략

2️⃣ Kafka 컨슈머의 오프셋 관리 방법

3️⃣ 오프셋 저장 위치와 커밋 주기 설정

4️⃣ Kafka auto.commit 기능 설명

→ 각 쿼리별로 벡터 검색을 수행한 뒤 결과를 **통합(merge)** 합니다.

⚙️ 2. 왜 필요한가?

기존의 단일 쿼리 리트리버는 한 문장으로만 검색하기 때문에 다음과 같은 문제가 있습니다.

문제점	설명
🔹 **표현 다양성 문제**	문서에 "같은 의미지만 다른 표현"이 들어 있으면 검색이 안 됨
🔹 **언어적 불일치**	사용자의 질문과 문서의 표현이 조금만 달라도 매칭이 약해짐
🔹 **리콜(Recall) 한계**	하나의 쿼리로는 충분히 많은 문서를 찾지 못함

Multi-Query Retriever는 **LLM이 쿼리를 확장(paraphrase)** 해서

이런 표현 차이를 보완합니다.

🧩 3. 동작 구조

다중 쿼리 리트리버는 내부적으로 다음 과정을 거칩니다.

```
[사용자 질문]
   ↓
[LLM] → 여러 쿼리 생성 (3~5개)
   ↓
각 쿼리별로 리트리버 실행
   ↓
결과 문서 집합 병합 (중복 제거)
   ↓
[최종 문서 리스트 반환]
```

LangChain에서 이 과정을 담당하는 클래스가 바로

`MultiQueryRetriever` 입니다.

💡 4. LangChain 구조 예시

```
from langchain.retrievers.multi_query import MultiQueryRetriever
from langchain.chat_models import ChatOpenAI
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

# 1️⃣ 벡터 DB
embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_texts(
   ["Kafka consumer offset commit", "Kafka auto commit config", "Redis atomic decrement command"],
```

```
    embeddings
)
retriever = vectorstore.as_retriever(search_kwargs={"k": 3})

# 2️⃣ Multi-Query Retriever 생성
llm = ChatOpenAI(model="gpt-4o-mini")
multi_query_retriever = MultiQueryRetriever.from_llm(
    retriever=retriever,
    llm=llm
)

# 3️⃣ 검색 수행
results = multi_query_retriever.get_relevant_documents("Kafka consumer offset commit 방식")
for i, doc in enumerate(results):
    print(f"--- 결과 {i+1} ---\n{doc.page_content}\n")
```

🧱 5. 내부 동작 세부 설명

LangChain의 `MultiQueryRetriever`는 다음 단계를 수행합니다:

1. **쿼리 확장 (Query Expansion)**

   → LLM이 사용자 질문을 다양한 의미적 변형으로 다시 작성.

   (예: “auto commit 설정 방법”, “offset commit이란?” 등)

2. **병렬 검색 (Parallel Retrieval)**

   → 각 쿼리에 대해 retriever(similarity search)를 실행.

3. **결과 통합 (Aggregation)**

   → 모든 결과를 합치고, 중복 문서는 제거.

4. **점수 기반 정렬 (Optional)**

   → 각 문서의 유사도 평균이나 최대값으로 최종 정렬.

📊 6. 성능 비교

구분	단일 쿼리 리트리버	다중 쿼리 리트리버
검색 정확도(Precision)	높음	높음
검색 포괄성(Recall)	낮음	✅ 높음
중복 문서 발생	적음	있음 (병합 단계 필요)
응답 품질	중간	✅ 향상 (더 다양한 문맥 확보)

즉,

> “놓치는 문서가 줄어든다” → LLM이 참고할 정보가 더 풍부해진다.

⚙️ 7. 주요 구성요소 및 파라미터

파라미터	설명
`retriever`	실제 검색을 수행할 리트리버 (예: FAISS, Chroma 등)
`llm`	다중 쿼리를 생성할 LLM
`include_original`	원본 질문도 함께 검색할지 여부 (기본값 True)
`num_queries`	생성할 쿼리 개수 (기본 3~5개)
`deduplicate`	결과 중복 제거 여부

🧠 8. 장점과 단점

구분	장점	단점
✅ **장점**	- LLM의 의미 확장을 통해 리콜 향상- 표현 다양성 보완- 단일 쿼리보다 더 풍부한 문맥 확보	
⚠️ **단점**	- LLM 호출이 추가되어 비용 증가- 여러 쿼리 검색으로 속도 느림- 중복 문서 처리 필요	

다중 벡터 스토어 리트리버

🧠 1. 개념 요약

Multi-Vector Retriever는

> 하나의 문서(Document)에 대해 여러 종류의 임베딩(embedding)을 생성해
> 각각을 별도의 벡터 스토어에 저장한 뒤,
> 검색 시 이 여러 스토어를 동시에 조회하여 더 풍부한 문서를 찾아내는 리트리버입니다.

즉,

하나의 문서를 "여러 시각으로 표현한 벡터"를 만들어

각기 다른 검색 관점에서 동시에 탐색하는 구조입니다.

📘 예시 상황

예를 들어, 다음처럼 한 문서가 있다고 가정해봅시다.

> "Kafka consumer offset commit은 메시지의 위치를 저장해 재처리를 방지한다."

이 문서를 다음처럼 다양한 방식으로 벡터화할 수 있습니다.

벡터 스토어	임베딩 대상	목적
스토어 A	원문 문장 전체	문맥 의미 중심 검색
스토어 B	문서 요약(summary)	주제 기반 검색
스토어 C	문서 제목(title)	키워드 중심 검색

→ Multi-Vector Retriever는 이 **세 스토어를 동시에 검색**하여

각각의 결과를 합쳐 최종적으로 LLM에 전달합니다.

⚙️ 2. 왜 필요한가?

단일 벡터 스토어(예: FAISS, Chroma)는

하나의 임베딩 모델만을 기준으로 검색하기 때문에 다음과 같은 한계가 있습니다.

한계	설명
🔷 단일 임베딩의 한계	문맥 의미는 잘 잡지만, 짧은 키워드 검색에 약함
🔷 멀티모달 데이터 대응 불가	텍스트 + 코드 + 요약 + 이미지 캡션 등 복합 데이터에 부적합
🔷 질의 다양성 한계	질문 형태에 따라 "요약 기반 검색이 유리한 경우"가 있음

이 문제를 해결하기 위해

여러 개의 벡터 스토어를 병렬로 사용하는 구조가 등장했습니다.

🧩 3. 동작 구조

Multi-Vector Retriever는 기본적으로 다음 순서로 작동합니다.

```
[사용자 질문]
    ↓
여러 개의 벡터 스토어 (A, B, C) 각각 검색
    ↓
```

```
각 스토어에서 상위 k개 결과 수집
    ↓
결과 병합 및 중복 제거
    ↓
[최종 문서 리스트 반환]
```

💡 4. LangChain에서의 구조

LangChain에서는 `MultiVectorRetriever` 클래스로 구현되어 있습니다.

이 클래스는 내부적으로 `VectorStoreRetriever` 여러 개를 묶어 사용합니다.

```
from langchain.retrievers.multi_vector import MultiVectorRetriever
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

# 1️⃣ 여러 벡터 스토어 생성
embeddings = OpenAIEmbeddings()
vectorstore_a = FAISS.from_texts(["Kafka offset commit"], embeddings)
vectorstore_b = FAISS.from_texts(["Kafka consumer summary of offset management"], embeddings)

# 2️⃣ MultiVectorRetriever 생성
retriever = MultiVectorRetriever(
    vectorstores=[vectorstore_a, vectorstore_b]
)

# 3️⃣ 검색 수행
results = retriever.get_relevant_documents("Kafka consumer offset 관리 방법")
for i, doc in enumerate(results):
    print(f"--- 결과 {i+1} ---\n{doc.page_content}\n")
```

⚙️ 5. 내부 동작 원리

1. **여러 벡터 스토어 병렬 검색**

   → 각 스토어에 동일한 쿼리를 던져서 `k`개의 문서를 가져옵니다.

2. **결과 통합 (Merge)**

   → 모든 스토어의 결과를 하나의 리스트로 합칩니다.

3. **중복 제거 (Deduplication)**

   → 같은 문서가 여러 스토어에서 반환된 경우 하나로 병합합니다.

4. **점수 기반 정렬 (Ranking)**

   → 각 결과의 유사도 점수를 평균 또는 가중 합산해 최종 순위를 계산합니다.

⚡ 6. 구성 요소 및 주요 파라미터

파라미터	설명
`vectorstores`	병렬 검색에 사용할 벡터 스토어 리스트
`weights`	각 스토어에 부여할 가중치 (기본값: 균등 분배)
`k`	각 스토어에서 가져올 문서 수
`deduplicate`	중복 문서 제거 여부
`search_kwargs`	각 벡터 스토어의 검색 세부 설정

🧠 7. 장점과 단점

구분	장점	단점
✅ **장점**	- 여러 임베딩 관점에서 검색 가능- 다양한 데이터(요약/원문/코드 등)에 유연- 리콜(Recall) 향상	
⚠️ **단점**	- 메모리/저장소 사용량 증가- 검색 속도 느림 (스토어 병렬 검색)- 중복 처리 및 점수 병합 추가 로직 필요	

🧱 8. MultiVectorRetriever vs EnsembleRetriever 차이

항목	Multi-Vector Retriever	Ensemble Retriever
목적	하나의 문서에 대해 여러 임베딩 시각을 활용	서로 다른 리트리버의 결과 결합
기반 구조	여러 **벡터 스토어**	여러 **리트리버** (BM25, FAISS 등)
검색 방식	동일 쿼리로 여러 벡터 공간 검색	서로 다른 방식의 리트리버를 병합
결과 병합	유사도 기반 벡터 스코어 결합	가중 평균으로 리트리버 결과 결합

👉 즉,

- **Multi-Vector Retriever**는 "한 문서를 여러 시각으로 임베딩한 경우" 유용하고,
- **Ensemble Retriever**는 "서로 다른 검색 전략(BM25, Dense 등)"을 결합할 때 적합합니다.

▼ 다중 벡터 스토어 리트리버 방법 3가지

🧩 1️⃣ 작은 청크 생성 방식 (Small Chunk Strategy)

💡 개념

문서를 여러 **작은 부분(청크, chunk)** 으로 나누어

각 청크마다 개별 임베딩을 생성하고 저장하는 방법입니다.

즉, **문서 하나 → N개의 벡터**로 분할해 저장합니다.

```
문서 1
 ├─ 청크 1 → Embedding ①
 ├─ 청크 2 → Embedding ②
 ├─ 청크 3 → Embedding ③
```

이 벡터들은 모두 **부모 문서(Parent Document)** 를 참조합니다.

⚙️ 동작 원리

1. 긴 문서를 여러 문단 단위로 분할합니다.
2. 각 문단을 임베딩하여 벡터 스토어에 저장합니다.
3. 검색 시, 쿼리 임베딩과 각 청크의 벡터를 비교해 가장 관련 있는 청크를 찾습니다.
4. 필요 시 해당 청크가 속한 **부모 문서 전체를 복원**할 수도 있습니다.

✅ 장점

항목	설명
세밀한 검색 가능	특정 문서 일부에 해당하는 내용도 검색됨
빠른 검색 속도	각 청크는 작아서 임베딩 계산 및 비교가 효율적
ParentDocumentRetriever와 궁합 좋음	작은 청크로 검색 후, 부모 문서로 확장 가능

⚠️ 단점

항목	설명
문맥 손실 가능성	청크 경계에 걸린 정보는 떨어져 저장됨
중복 정보 저장	겹치는 문맥이 여러 청크에 포함될 수 있음

📘 사용 예시

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
docs = splitter.split_documents(original_docs)
vectorstore.add_documents(docs)
```

이 방식은 가장 기본적이며, **대부분의 RAG 시스템의 기본 구조**입니다.

다만, 아래 두 방식은 이보다 더 "고도화된 변형"입니다.

🧩2️⃣ 요약 임베딩 방식 (Summary Embedding Strategy)

💡 개념

각 문서 또는 청크를 직접 임베딩하지 않고,

해당 내용을 LLM이 요약(summary) 한 뒤,

그 **요약문을 임베딩** 하는 방식입니다.

즉,

> "문서 내용을 직접 임베딩 → X"
>
> "문서의 요약(핵심 문장)을 임베딩 → ✅"

⚙️ 동작 원리

```
문서: "Kafka의 consumer offset commit은 메시지 위치를 저장한다..."
↓
LLM 요약: "Kafka는 consumer의 offset을 commit하여 중복 처리를 방지한다."
↓
요약문 임베딩 → 벡터 스토어 저장
```

검색 시에는 쿼리 임베딩과 **요약 임베딩**을 비교하여

문서의 핵심 주제와 의미가 비슷한 결과를 더 쉽게 찾습니다.

✅ 장점

항목	설명
의미 기반 검색 강화	LLM 요약 덕분에 핵심 개념 중심 검색
긴 문서에 적합	원문이 매우 길 때 효율적
불필요한 세부 내용 무시	핵심 주제 위주로 임베딩

⚠️ 단점

항목	설명
정보 손실 가능	세부 문맥은 요약 과정에서 사라질 수 있음
LLM 호출 비용 증가	모든 문서마다 요약 생성 필요
검색 정밀도 낮을 수 있음	주제는 맞지만 세부 내용이 부족할 수 있음

📘 사용 예시

```
from langchain.chat_models import ChatOpenAI
from langchain.chains.summarize import load_summarize_chain

llm = ChatOpenAI(model="gpt-4o-mini")
summary_chain = load_summarize_chain(llm, chain_type="map_reduce")

summaries = [summary_chain.run(doc.page_content) for doc in docs]
vectorstore.add_texts(summaries)
```

이 방법은 특히 **요약 문서 검색(Summary-based Retrieval)** 이나

정책/리포트/논문 요약형 QA 에 매우 효과적입니다.

🧩3️⃣ 가설 쿼리 활용 방식 (Hypothetical Query Embedding, HyDE)

💡 개념

사용자 질문(query) 을 바로 임베딩하지 않고,

LLM을 사용해 그 질문에 대한 **가상의 답변(hypothetical answer)** 을 먼저 생성한 후,

그 답변의 임베딩을 사용하여 검색하는 방식입니다.

즉,

> “질문을 벡터화” 대신
>
> “답변을 상상해서 벡터화” 하는 아이디어입니다.

⚙️ 동작 원리

```
질문: "Kafka offset commit 방법?"
↓
LLM이 가상의 답변 생성:
"Kafka의 consumer는 commitSync() 또는 commitAsync() 메서드로 오프셋을 커밋한다."
↓
이 가상 답변을 임베딩
↓
벡터 스토어 검색
```

이렇게 하면 단순히 “offset commit”이라는 단어 일치보다

“commitSync”, “consumer”, “메시지 재처리 방지” 등

의미적으로 관련된 문서를 더 정확히 찾을 수 있습니다.

✅ 장점

항목	설명
질문 의미 확장	질문의 의도를 구체적인 문장으로 변환
의미적 검색 강화	관련 문서를 더 정확히 탐색
짧은 질문에도 강함	짧은 질의에서 발생하는 정보 부족 해결

⚠️ 단점

항목	설명
LLM 호출 비용 발생	매 쿼리마다 가상 답변 생성 필요

항목	설명
잘못된 가설 위험	LLM이 틀린 가상 답변을 만들면 검색 왜곡 가능
추가 지연 발생	실시간 응답 속도 다소 느려짐

📘 사용 예시

LangChain에서는 `HypotheticalDocumentEmbedder (HyDE)` 로 구현됩니다.

```
from langchain.embeddings import HypotheticalDocumentEmbedder
from langchain.chat_models import ChatOpenAI
from langchain.embeddings import OpenAIEmbeddings

llm = ChatOpenAI(model="gpt-4o-mini")
base_embedder = OpenAIEmbeddings()

# HyDE 임베더 생성
hyde_embedder = HypotheticalDocumentEmbedder(
    llm=llm,
    base_embeddings=base_embedder
)

# 쿼리 임베딩 시 내부적으로 "가상의 답변 → 임베딩" 수행
vectorstore = FAISS.from_texts(docs, embedding=hyde_embedder)
```

⚡4️⃣ 세 가지 방법 비교

구분	작은 청크 방식	요약 임베딩	가설 쿼리(HyDE)
핵심 아이디어	문서를 여러 작은 조각으로 나눔	문서의 요약을 임베딩	질문에 대한 가상 답변을 임베딩
임베딩 대상	문서의 조각	문서 요약문	가상의 답변
검색 단위	세밀한 문장 단위	주제 중심 단위	의미 확장된 질문 단위
LLM 사용 시점	없음	임베딩 전	쿼리 시
장점	검색 세밀도 높음	의미 중심 검색	의미 확장, 질문 유연성
단점	문맥 손실 가능	세부정보 손실	비용·속도 저하
적합한 상황	일반 문서 검색	긴 문서·요약형 QA	짧은 질문·의미 확장형 검색

셀프 쿼리 리트리버

🧠1️⃣ 개념 요약

Self-Query Retriever란,

> LLM이 사용자의 질문을 스스로 해석(Self-querying)해서
> "어떤 필드를 검색해야 할지"와 "어떤 필터 조건을 걸어야 할지"를 자동으로 생성한 뒤
> 벡터 스토어(VectorStore)에 질의하는 리트리버입니다.

즉, **LLM이 검색 질의를 구성하는 '중간 브레인' 역할**을 합니다.

💡 비유로 설명하자면

보통의 RAG는 다음과 같이 작동합니다:

```
사용자: "2023년 이후에 작성된 Kafka 관련 문서 찾아줘"
↓
리트리버: "Kafka 2023 이후" 전체를 벡터로 변환해서 검색
```

```
↓
결과: 2023과 Kafka 단어가 들어간 문서 (정확하지 않음)
```

하지만 **Self-Query Retriever**는 이렇게 동작합니다 👇

```
1️⃣ LLM이 질의를 해석:
   - 검색 주제: Kafka
   - 필터: year > 2023
2️⃣ Vector DB에서 "Kafka" 관련 임베딩 검색
3️⃣ year 메타데이터가 2023 이후인 결과만 필터링
4️⃣ LLM에게 정확한 문맥 전달
```

→ 즉, **LLM이 스스로 쿼리를 '재작성'하여 더 정확한 검색을 수행**합니다.

⚙️2️⃣ 동작 구조

Self-Query Retriever는 다음 순서로 작동합니다:

```
[사용자 질문]
     ↓
[LLM이 질의 분석]
  ├─ 검색어(query) 추출
  └─ 필터 조건(filter) 추출 (메타데이터 기반)
     ↓
[벡터 스토어 검색 + 필터 적용]
     ↓
[관련 문서 반환]
```

🧩3️⃣ LangChain에서의 구조

LangChain은 이를 위해 `SelfQueryRetriever` 클래스를 제공합니다.

이 클래스는 `VectorStoreRetriever`를 감싸며, LLM을 통해 질의 해석을 수행합니다.

📘 예시 코드

```
from langchain.retrievers.self_query.base import SelfQueryRetriever
from langchain.chat_models import ChatOpenAI
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains.query_constructor.schema import AttributeInfo

# 1️⃣ Vector Store 구성
docs = [
    {"page_content": "Kafka offset commit guide", "metadata": {"year": 2024, "category": "streaming"}},
    {"page_content": "Redis atomic operation example", "metadata": {"year": 2022, "category": "database"}},
]
texts = [d["page_content"] for d in docs]
metadatas = [d["metadata"] for d in docs]

vectorstore = FAISS.from_texts(texts, embedding=OpenAIEmbeddings(), metadatas=metadatas)

# 2️⃣ 필드 정의 (LLM이 인식할 수 있도록)
metadata_field_info = [
```

```
    AttributeInfo(name="year", description="the year the document was written", type="integer"),
    AttributeInfo(name="category", description="the topic category of the document", type="string"),
]

# 3️⃣ Self-Query Retriever 생성
retriever = SelfQueryRetriever.from_llm(
    llm=ChatOpenAI(model="gpt-4o-mini"),
    vectorstore=vectorstore,
    document_content_description="technical documentation about backend systems",
    metadata_field_info=metadata_field_info,
    enable_limit=True,
)

# 4️⃣ 검색 수행
query = "2023년 이후 작성된 Kafka 관련 문서"
results = retriever.get_relevant_documents(query)

for doc in results:
    print(doc.page_content, doc.metadata)
```

🧠 4️⃣ 내부 동작 설명

위 예시의 실행 과정은 다음과 같습니다:

1️⃣ LLM이 질의 해석

→ "Kafka"는 검색 키워드로,

→ "2023년 이후"는 `year > 2023` 필터로 파악함.

2️⃣ 검색 요청 구성

```
{
  "query": "Kafka",
  "filter": {"year": {"$gt": 2023}}
}
```

3️⃣ Vector Store 검색 수행

- "Kafka"와 의미적으로 가까운 문서를 검색하고
- `year` 메타데이터가 2023보다 큰 문서만 남김.

4️⃣ 결과 반환

→ "Kafka offset commit guide (year=2024)" 문서 반환 ✅

⚡ 5️⃣ 장점과 단점

구분	장점	단점
✅ **장점**	- LLM이 필터 생성 → **정확한 조건 검색** 가능- 의미 검색 + 구조화 검색 결합- 자연어 질의를 그대로 사용할 수 있음	
⚠️ **단점**	- 매 쿼리마다 LLM 호출 필요 (비용·속도)- LLM이 필터를 잘못 생성하면 오검색- 메타데이터 설계가 필수적	

🧰 6️⃣ 주요 구성요소 정리

구성요소	설명
`vectorstore`	벡터 기반 검색 수행 (예: FAISS, Chroma 등)
`llm`	자연어 질의를 분석하여 검색어 + 필터 추출

구성요소	설명
`metadata_field_info`	필터 가능한 필드 목록 (이름, 설명, 타입)
`document_content_description`	문서 내용의 의미를 LLM이 이해하도록 돕는 설명
`enable_limit`	결과 제한 옵션 (예: 상위 3개만 반환)

시간 가중 벡터 스토어 리트리버

🧠 1️⃣ 개념 요약

시간 가중 벡터 스토어 리트리버(TimeWeightedVectorStoreRetriever) 는

> 문서의 "의미 유사도(score)"에 "시간 정보(time decay weight)"를 함께 고려해
> **최근 문서일수록 더 높은 점수를 주는 리트리버**입니다.

즉, 단순히 "내용이 비슷한 문서"만 찾는 것이 아니라

'얼마나 최근에 추가·수정되었는지' 도 검색 점수에 반영합니다.

💬 예시

사용자가 다음과 같이 질문한다고 가정해봅시다:

> "최근 Kafka offset commit 관련 업데이트가 뭐야?"

- 일반 벡터 리트리버 → 오래된 2021년 문서도 유사하면 상위에 옴 ❌
- 시간 가중 리트리버 → 2025년 문서가 훨씬 높은 점수로 상위에 옴 ✅

⚙️ 2️⃣ 동작 원리

`TimeWeightedVectorStoreRetriever` 는 내부적으로 다음과 같이 작동합니다.

```
[1] 사용자 쿼리 임베딩 생성
[2] 벡터 스토어에서 유사도 점수 계산
[3] 각 문서의 "last_access_time" 또는 "created_at" 메타데이터 확인
[4] 시간 감쇠(time decay) 함수 적용 → 최종 점수 계산
[5] 점수가 높은 순으로 문서 반환
```

📈 점수 계산 공식 (핵심)

```
final_score = similarity_score + (λ * recency_weight)
```

여기서

- `similarity_score` : 임베딩 기반 유사도 (기존 벡터 스토어 점수)
- `recency_weight` : 문서가 최근일수록 커짐 (예: e^(-Δt/τ))
- `λ` : "시간 가중치의 영향력"을 조정하는 계수

즉, 시간이 오래될수록 **감쇠(decay)** 되고,

새로운 문서일수록 더 큰 보너스를 받습니다.

🧩 3️⃣ LangChain에서의 구조

LangChain은 `TimeWeightedVectorStoreRetriever` 클래스로 제공합니다.

```
from langchain.retrievers import TimeWeightedVectorStoreRetriever
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from datetime import datetime, timedelta

# 1️⃣ 벡터 스토어 생성
embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_texts(
    ["Kafka offset commit guide", "Kafka consumer auto commit config"],
    embeddings
)

# 2️⃣ 시간 가중 리트리버 구성
retriever = TimeWeightedVectorStoreRetriever(
    vectorstore=vectorstore,
    decay_rate=0.01,  # 시간 감쇠 정도 (낮을수록 오래된 문서 영향 감소)
    k=3,              # 반환 문서 개수
)

# 3️⃣ 문서 삽입 시 시간 기록
retriever.add_documents([
    {"page_content": "Kafka consumer commit API (2024 update)", "metadata": {"last_accessed_at": datetime.now
()()},
])

# 4️⃣ 검색 실행
results = retriever.get_relevant_documents("Kafka offset commit 방법")
for doc in results:
    print(doc.page_content, doc.metadata)
```

🧠 4️⃣ 주요 파라미터

파라미터	설명
`decay_rate`	시간 감쇠 비율. 높을수록 시간이 빠르게 감쇠됨
`k`	반환할 문서 개수
`vectorstore`	사용할 벡터 스토어 객체 (예: FAISS, Chroma 등)
`score_threshold`	최소 점수 기준 (선택)
`search_kwargs`	내부 유사도 검색 시 세부 옵션
`last_accessed_at`	각 문서의 최근 접근/작성 시각 (datetime 객체)

📊 5️⃣ 시간 감쇠 모델 (Time Decay Model)

시간 가중 점수는 일반적으로 다음 중 한 가지 함수로 계산됩니다.

감쇠 모델	수식	특징
지수 감쇠(Exponential Decay)	`w = exp(-λ * Δt)`	오래된 문서 영향 급감
선형 감쇠(Linear Decay)	`w = max(0, 1 - α * Δt)`	일정 기간까지만 점차 감소
하이브리드	`score = sim + β * exp(-λ * Δt)`	유사도 + 시간 보정 혼합

LangChain 기본 구현은 **지수 감쇠**입니다.

⚡ 6️⃣ 장점과 단점

구분	장점	단점
✅ **장점**	- 최신 정보 우선 검색- 뉴스/로그 등 시계열 데이터에 적합- 시간 추세 반영 가능	
⚠️ **단점**	- 오래된 중요한 문서가 밀릴 수 있음- 시간 메타데이터가 없는 문서엔 적용 불가- decay_rate 튜닝 필요	