생성형 AI 기반 RAG 개발자 과정

Retriever 구성
검색 모드 · 메타데이터 필터

Session 15 / 33 — Day 2

similarity · with_score · MMR · 메타데이터 필터 · Reranker

박수현 · (사)한국정보공학기술사회 · 2026

LLM 이론 · LangChain · ChromaDB CRUD · Agent — 4일 핸즈온

이번 시간을 마치면

기본기 (본선)

- ✅ `as_retriever()` — 벡터 스토어를 검색 부품으로
- ✅ **검색 모드 3종** — similarity / **with_score(거리→유사도%)** / MMR
- ✅ **메타데이터 필터** — `where`, `$gte` · `$and`, 권한 필터
- ✅ **MMR** — 관련성과 다양성의 균형

심화 (선택)

- ✅ **BM25 + Vector 하이브리드** (EnsembleRetriever)
- ✅ **Self-Query** — LLM 이 메타데이터 필터 자동 생성
- ✅ **Reranker** — Bi-Encoder(1차) → Cross-Encoder(2차)
- ✅ Reranker 실전 결합 (Cohere 등 3가지 옵션)

📂 소스: `2.langchain/7.RAG/3.vectorstore/`, `7.RAG/4.rag_chain/`

PART A

PART A
검색 모드 · 메타데이터 필터

실무 RAG 검색의 기본기 — 약 25분

`as_retriever()` — 질문 → 문서 표준 인터페이스

벡터 스토어 → Retriever

```
# Session 14 에서 만든 Chroma 스토어를 재사용
retriever = store.as_retriever(
    search_kwargs={"k": 3},
)

# Retriever = 질문 → 관련 Document 리스트
# 단일 책임 → LCEL 파이프에 그대로 연결
docs = retriever.invoke("NVMe 의 속도와 인터페이스")
```

- 벡터 스토어는 검색 메서드를 직접 노출하나, RAG 체인에 연결하려면 **표준화된 검색 인터페이스**가 필요하다
- `as_retriever()` 가 이를 표준 부품(Runnable)으로 감싼다

`k` — 반환 문서 개수

`k`	결과
너무 작음	답에 필요한 근거가 빠짐
적정 (3~5)	핵심 근거만 압축
너무 큼	무관 청크 섞임 → 프롬프트 길어지고 품질 ↓

> 💡 `search_kwargs` 에는 `k` 외에 `search_type` (검색 모드), `filter` (메타데이터 필터)를 함께 전달한다. 다음 슬라이드부터 순서대로 다룬다.

같은 질문, 세 가지 검색 방식

모드 ① · ② — similarity / with_score

```
# 2.langchain/7.RAG/3.vectorstore/3.4_search_modes.py
query = "NVMe 의 속도와 인터페이스"

# ① similarity_search — 가장 가까운 N개 (기본)
for d in store.similarity_search(query, k=3):
    print(f"  → {d.page_content[:60]}...")

# ② similarity_search_with_score — 점수 포함
#    Chroma 의 score 는 '거리' → 작을수록 유사
for d, score in store.similarity_search_with_score(
        query, k=3):
    sim_pct = round((1 - score) * 100, 1)   # 거리 → 유사도%
    print(f"  거리 {score:.3f} "
          f"(유사도 ≈ {sim_pct}%)  "
          f"{d.page_content[:40]}...")
```

🎯 **점수를 환산해 임계값으로 거른다**: 거리가 일정 이상이면 무관한 결과로 간주해 제거하는 후처리가 가능하다. RAG 가 부적합한 근거를 끌어오는 것을 막는 1차 방어선이다.

모드 ③ — MMR (다양성)

```
# 3) MMR — 다양성 보장
# 비슷한 청크가 top-k 를 다 차지하지 않게
for d in store.max_marginal_relevance_search(
        query, k=3, fetch_k=10, lambda_mult=0.5):
    print(f"  → {d.page_content[:60]}...")
```

세 모드 한눈에

모드	특징	적합
`similarity_search`	가장 가까운 N개	일반 기본 검색
`with_score`	거리 점수 포함	**임계값 필터링**
MMR	관련성 + 다양성	중복 청크 회피

💡 Chroma 점수 = **거리**(작을수록 유사), FAISS IP 점수 = **내적**(클수록 유사). 스토어마다 점수의 방향이 다르므로 환산 식을 맞춰야 한다.

벡터 유사도에 정형 조건을 더한다

풍부한 metadata 를 붙여 저장

```
# 2.langchain/7.RAG/3.vectorstore/3.11_metadata_filter.py
from langchain_core.documents import Document

# page_content(벡터화 대상) + metadata(필터 대상)
docs = [
    Document(page_content="NVMe SSD 는 PCIe 로 약 7GB/s …",
        metadata={"category": "nvme", "year": 2022}),
    Document(page_content="SATA SSD 는 약 0.5GB/s …",
        metadata={"category": "sata", "year": 2018}),
    Document(page_content="최신 PCIe 5.0 NVMe 는 14GB/s …",
        metadata={"category": "nvme", "year": 2024}),
]
store = Chroma.from_documents(
    docs, embeddings, collection_name="meta_demo")
```

⚠️ Chroma metadata 값은 **문자열·정수·실수·불리언만** 가능 (리스트·딕셔너리 불가).

`filter` 로 후보를 먼저 좁힌다

```
query = "저장장치 속도"

# 동등 — category 가 nvme 인 것만
store.similarity_search(query, k=4,
    filter={"category": "nvme"})

# 숫자 비교 — 2020년 이후 문서만
store.similarity_search(query, k=4,
    filter={"year": {"$gte": 2020}})

# 복합 AND — nvme 이면서 2023년 이후 (→ 1건만)
store.similarity_search(query, k=4,
    filter={"$and": [
        {"category": "nvme"},
        {"year": {"$gte": 2023}},
    ]})
```

연산자	의미
`{"k": v}`	동등
`$gt` · `$gte` · `$lt` · `$lte` · `$ne`	숫자 비교
`$in` · `$nin`	목록 포함/제외
`$and` · `$or`	복합 조건

동일 질문에서 결과 집합을 정밀하게 좁힌다

정확도 — 후보를 먼저 거른다

```
질문: "저장장치 속도"  (동일)

필터 없음          → nvme·sata·hdd 전부 후보
year ⩾ 2020       → 2022·2024 문서만
nvme AND y⩾2023 → 2024년 한 건만
```

- 벡터 유사도만으로는 "최신 문서만", "특정 출처만" 같은 **정형 조건**을 걸 수 없다
- 의미 유사도(벡터) + 정형 조건(메타데이터) = **실무 RAG 기본기**

보안 — 권한 필터

```
# 사용자가 권한 있는 문서만 검색하도록 제한
retriever = store.as_retriever(
    search_kwargs={
        "k": 4,
        "filter": {"$and": [
            {"dept": current_user_dept},   # 부서 격리
            {"status": {"$ne": "draft"}},  # 미공개 제외
        ]},
    }
)
```

- **권한 있는 문서만 / 최신 버전만 / 특정 부서만** 검색하도록 제한
- 멀티테넌트 RAG 에서 사용자 간 문서 격리의 1차 수단

> 🎯 메타데이터 필터는 **정확도 향상을 넘어 보안의 핵심**. 답변 근거의 허용 범위를 검색 단계에서 고정한다.

"가장 가까운 K 개" 의 한계

시나리오 — 5개 chunk 모두 유사한 내용

```
Q: "Python 의 decorator 가 뭐예요?"

검색 결과 (Top-5, similarity score):
1. "decorator 는 함수를 감싸는 함수다…"            0.92
2. "데코레이터는 메타프로그래밍 도구로…"           0.91   ← 거의 같은 말
3. "@property 데코레이터는 getter 를…"             0.89   ← 거의 같은 말
4. "데코레이터 패턴은 GoF 디자인 패턴 중…"         0.88   ← 거의 같은 말
5. "Python 의 데코레이터 문법은 @ 기호로…"         0.87   ← 거의 같은 말
```

문제

- 5개 chunk 가 **거의 같은 정보** → LLM 에게 중복 정보 5번 줌
- 정작 "decorator 예시 코드" 나 "decorator 활용 사례" 는 누락
- **다양성 (diversity) 부재** → RAG 답변이 한쪽으로 치우침
- score threshold 로 멀리 떨어진 chunk 를 버려도 **중복 쏠림 자체는 못 막음**

중복 쏠림 해소 전략 — 본선과 심화

해결책 — 본선과 심화

전략	해결 방법	위상
MMR (Maximal Marginal Relevance)	관련성 + **이미 뽑힌 것과의 차이** 점수	본선
메타데이터 필터	`where` 로 후보를 정형 조건으로 좁힘	본선
BM25 hybrid	키워드 검색과 결합해 약어·고유명사 강화	심화
Self-query	LLM 이 질의에서 메타데이터 필터를 자동 추출	심화
Reranker	1차 retrieval 결과를 2차 정렬 (Cross-Encoder)	심화

> 🎯 앞서 다룬 **메타데이터 필터**가 가장 직접적인 해결책이며, MMR 은 중복 쏠림 자체를 줄이는 또 하나의 본선 전략이다.

중복 검색 vs 다양성 — MMR 의 필요성

순수 Similarity 의 함정

- Top-K 를 점수순으로만 추출하면 **거의 동일한 chunk 가 몰려** 들어온다
- 같은 정보의 K 회 반복 → LLM 이 받는 실질 정보량은 1개분
- 정작 필요한 예시·반례·다른 관점은 **점수가 근소하게 낮다는 이유로 탈락**

MMR 이 바꾸는 것

- 이미 뽑은 chunk 와의 **유사도를 페널티**로 더한다
- 결과적으로 벡터 공간 전체에 **골고루 퍼진 K 개**를 선택
- 관련성은 유지하면서 **중복은 줄이고 다양성은 확보** → 다음 슬라이드에서 수식·도식으로 확인

관련성 · 다양성의 균형

수식

$$ \text{MMR} = \arg\max_{d_i \in R \setminus S} \left[\lambda \cdot \text{sim}(d_i, q) - (1-\lambda) \cdot \max_{d_j \in S} \text{sim}(d_i, d_j) \right] $$

- `λ = 1` → 순수 similarity
- `λ = 0` → 최대 다양성
- `λ = 0.5` → **균형 (본 코스 권장)**

비교 — 같은 질의 결과

전략	결과 특성
Similarity (k=5)	모두 "decorator 정의" 만 5개
MMR (k=5, λ=0.5)	정의·예시·활용·@property·디자인패턴 — **다양**

> 🎯 **광범위 질문일수록 MMR**. 정확한 사실 질문이면 similarity 가 더 나음.

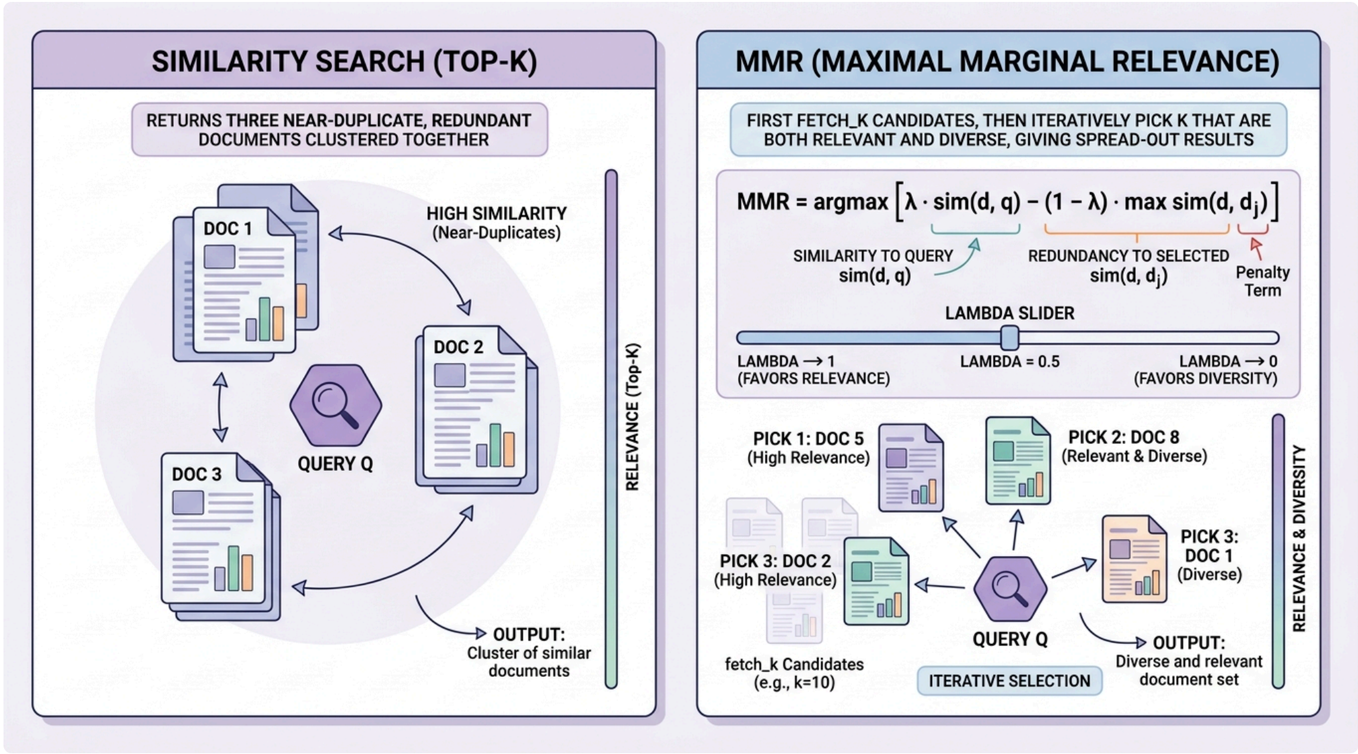
LangChain 에서 사용

```
# 단순 similarity
retriever = vectorstore.as_retriever(
    search_type="similarity",       # 기본
    search_kwargs={"k": 5},
)

# MMR — fetch_k 후보 중 k 개 선택
retriever = vectorstore.as_retriever(
    search_type="mmr",
    search_kwargs={
        "k": 5,
        "fetch_k": 20,         # 1차 후보 풀
        "lambda_mult": 0.5,    # diversity vs relevance
    },
)
```

> 💡 `as_retriever()` 는 벡터 스토어를 질문 → 문서 표준 부품으로 감싼다. `search_type` 만 변경하면 동일 스토어에서 similarity ↔ MMR 을 전환할 수 있다.

MMR — 관련성과 다양성의 균형 (도식)



MMR 검색 · argmax[λ·sim(d,q) − (1−λ)·max sim(d,dⱼ)] · λ로 관련성↔다양성 균형, fetch_k 후보에서 중복 적은 k개 선택

PART A-심화

PART A 심화
하이브리드 · Self-Query

본선(검색 모드·필터·MMR) 위의 선택지 — 시간 여유 시 — 약 10분

키워드 매칭과 의미 검색의 결합

하이브리드의 근거

- **Vector**: 의미·동의어·다국어 강함, 약어·고유명사 약함
- **BM25** (키워드): 정확한 단어 매칭 강함, 의미 약함
- **결합**: 각자의 장점만 사용

BM25 수식 (Best Matching 25)

$$ \text{BM25}(D, Q) = \sum_{q \in Q} \text{IDF}(q) \cdot \frac{f(q, D) \cdot (k_1 + 1)}{f(q, D) + k_1 (1 - b + b \frac{|D|}{\text{avgdl}})} $$

- `f(q, D)`: 문서 D 에서 단어 q 의 빈도
- `IDF`: 단어의 희귀도 (드물수록 ↑)
- `k1, b`: 튜닝 파라미터 (보통 k1=1.5, b=0.75)

LangChain EnsembleRetriever

```
from langchain_community.retrievers import BM25Retriever
from langchain.retrievers import EnsembleRetriever

# BM25 retriever (별도 색인 필요)
bm25 = BM25Retriever.from_documents(all_chunks)
bm25.k = 5

# Vector retriever
vector = vectorstore.as_retriever(
    search_kwargs={"k": 5})

# 결합 — Reciprocal Rank Fusion (RRF)
hybrid = EnsembleRetriever(
    retrievers=[bm25, vector],
    weights=[0.4, 0.6],    # BM25:Vector 가중치
)

results = hybrid.invoke(
    "FAISS HNSW ef_construction 파라미터")
# → BM25 가 "FAISS"·"HNSW" 정확 매칭
#   Vector 가 의미 유사 chunk 도 포함
```

"2024년 이후 OpenAI 논문" 같은 질의

메타데이터 스키마 + SelfQueryRetriever

```
from langchain.retrievers.self_query.base \
    import SelfQueryRetriever
from langchain.chains.query_constructor.schema \
    import AttributeInfo
from langchain_openai import ChatOpenAI

metadata_field_info = [
    AttributeInfo(
        name="source",
        description="문서 출처 (논문 제목)",
        type="string"),
    AttributeInfo(
        name="year",
        description="발행 연도",
        type="integer"),
    AttributeInfo(
        name="venue",
        description="컨퍼런스 (NeurIPS, ICML 등)",
        type="string"),
]

retriever = SelfQueryRetriever.from_llm(
    llm=ChatOpenAI(model="gpt-4o-mini", temperature=0),
    vectorstore=vectorstore,
    document_contents="ML/AI 연구 논문 본문",
    metadata_field_info=metadata_field_info,
)
```

자연어 질의 → 필터 자동 추출

```
results = retriever.invoke(
    "2024년 이후 NeurIPS 에 발표된 RLHF 관련 논문")

# 내부적으로 변환됨:
#   filter = {"$and": [
#     {"year": {"$gte": 2024}},
#     {"venue": "NeurIPS"}
#   ]}
#   query  = "RLHF"
```

Self-Query 가 빛나는 경우

- 메타데이터가 풍부한 경우 (논문 DB, 제품 카탈로그)
- 사용자가 **시간·카테고리·범위** 를 자연어로 표현

> ⚠️ LLM 호출 1번 추가 = latency + 비용. 단순 검색에는 과함.

PART B-심화

PART B 심화
Reranker — 2단계 검색의 끝

> Cross-Encoder 가 만드는 정확도 점프 — 심화 — 약 15분

임베딩 모델 (Bi) vs Reranker (Cross)

Bi-Encoder (지금까지의 임베딩)

```
질의  →  [임베딩 모델]  →  q_vec (1536d)
문서  →  [임베딩 모델]  →  d_vec (1536d)
유사도 = cos(q_vec, d_vec)
```

- **장점**: 문서 임베딩을 **사전 계산** → 검색 빠름 (1ms)
- **단점**: 질의와 문서를 **독립적으로** 처리 → 미세한 상호작용 못 잡음

사용

- 1차 검색 (검색 범위 줄이기)
- N → top-K 후보 추출

Cross-Encoder (Reranker)

```
질의 + 문서  →  [모델 하나에 함께 입력]  →  점수 (0~1)
```

- **장점**: 질의·문서를 **함께 보면서 score** → 정밀
- **단점**: 매 질의마다 K 개 문서 각각 모델 호출 → 느림 (100ms × K)

사용

- 2차 정렬 (1차 검색 결과 K=50 을 K=5 로)
- LLM 호출 직전 마지막 필터

정확도 vs 속도

	Bi-Encoder	Cross-Encoder
정확도	★★★	★★★★★
속도	< 1ms	100ms × K
비용	색인 시 1회	**매 검색** K번

🎯 **표준 패턴**: Bi-Encoder 로 N=10000 → top 50 후보 → Cross-Encoder 로 top 5 정렬.

Cohere · sentence-transformers · BGE

옵션 1 — Cohere Rerank API (가장 정확, 유료)

```
# pip install langchain-cohere cohere
from langchain_cohere import CohereRerank
from langchain.retrievers.contextual_compression \
    import ContextualCompressionRetriever

base_retriever = vectorstore.as_retriever(
    search_kwargs={"k": 50})

reranker = CohereRerank(
    cohere_api_key=os.environ["COHERE_API_KEY"],
    model="rerank-multilingual-v3.0",
    top_n=5,    # 1차 50 → 2차 5
)

compression_retriever = ContextualCompressionRetriever(
    base_compressor=reranker,
    base_retriever=base_retriever,
)
results = compression_retriever.invoke("질의")
```

옵션 2 — sentence-transformers Cross-Encoder (무료, 로컬)

```
from sentence_transformers import CrossEncoder
model = CrossEncoder("BAAI/bge-reranker-v2-m3")
# 다국어, 한국어 우수

pairs = [[query, doc.page_content] for doc in candidates]
scores = model.predict(pairs)   # (K,) 점수
ranked = sorted(zip(candidates, scores),
                key=lambda x: -x[1])
top_k = [d for d, _ in ranked[:5]]
```

옵션 3 (LLM) · 정확도 비교 · 선택 가이드

옵션 3 — LLM as Reranker (가장 유연, 비쌈)

```
# LLM 이 직접 문서 압축·정렬
from langchain.retrievers.document_compressors \
    import LLMChainExtractor

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

정확도 향상 — 실측 (개략치)

단계	Recall@5
Vector only	65%
Vector → Cross-Encoder	**78%**
Vector → Cohere Rerank	82%

선택 가이드

- **학습·실험**: 옵션 2 (무료, 한국어 OK)
- **운영 정확도 최우선**: 옵션 1 (Cohere)
- **복잡한 압축 필요**: 옵션 3 (LLM)

핵심 7가지

본선 — 실무 RAG 기본기

1. ✅ `as_retriever()` — 벡터 스토어를 "질문 → 문서" 표준 부품으로
2. ✅ **검색 모드 3종** — similarity / `with_score`(거리→유사도% 환산·임계값 필터) / MMR
3. ✅ **메타데이터 필터** — `where`, `$gte`·`$and`, 권한 필터 = 정확도 + 보안
4. ✅ **MMR** — 관련성과 다양성을 동시에 고려한 검색

심화 — 시간 여유 시

5. ✅ **BM25 + Vector 하이브리드** — 키워드와 의미를 함께
6. ✅ **Self-Query** — LLM 이 메타데이터 필터를 자동 생성
7. ✅ **Reranker** — Bi-Encoder(빠른 1차) → Cross-Encoder(정밀 2차)