

# Day 6: Reranker & Hybrid Search - 완전 가이드

GitHub Repository: [RAG-on-Local-CPU-minimal](#)

검색 품질 향상 - 2단계 검색

## 💡 한 줄 정의

"Vector Search는 후보를 넓게 고르고, Hybrid + Reranker는 질문에 맞게 정밀하게 고른다."

## 🎯 학습 목표

Day 6의 핵심은 다음과 같습니다:

"Vector Search + BM25 Reranker로 검색 품질을 향상시킨다"

이 Day를 마치면:

- 2-Stage Retrieval 이해
- BM25 Reranker 적용
- Vector + Keyword 결합
- Top-K와 Top-N 차이
- 왜 재정렬이 필요한지 이해
- 실무 RAG 패턴 체득

핵심: 넓게 검색 (Vector) → 정확하게 재정렬 (BM25)

## 📁 프로젝트 구조

```
Rag_minimal_day6/
├── main.py          # FastAPI 진입점
├── pipeline.py      # RAG 파이프라인 (Hybrid)
├── embedding.py     # 임베딩 생성
├── vectorstore.py   # Vector 검색
├── reranker.py      # BM25 재정렬 ← 신규
├── ingest.py        # 자산 생성 (확장)
├── loader.py        # 문서 로딩
├── config.py        # 설정
└── data/
    └── docs.txt       # 입력 문서
```

## 👁️ 왜 이런 구조인가?

STEP 5까지의 상태

## Embedding + Vector Search

질문

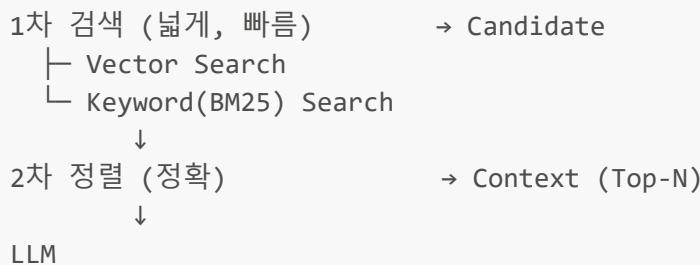
- 임베딩
- Vector Search (Top-K)
- 의미 유사 후보

**결과:** 의미적으로 가까운 후보

STEP 6에서 추가되는 것

Vector 검색 결과를 그대로 사용

Hybrid Search + Reranker로 질문 관점 정렬



☞ "답이 어색한데?"를 막는 최소 방어선

1. 왜 Vector Search만으로 부족한가?

**Vector Search의 구조적 한계:**

- 의미 유사 ≠ 질문 적합
- 키워드 정확도가 낮음
- Chunk가 짧을수록 노이즈 증가
- 문서 수 증가 시 Top-K 혼입 심화

실제 예시:

질문: "Embedding이 왜 필요해?"

Vector Search 결과:

1. "벡터 표현의 장점..." (의미는 유사)
2. "임베딩 기술 소개..." (단어는 있음!) ← 최적
3. "딥러닝 모델 구조..." (의미만 유사)

**문제점:**

- 의미만 보고 정렬
- 질문의 핵심 단어 무시

- 2번이 가장 정확한데 1번이 상위

## 2. 왜 Keyword Search(BM25)가 필요한가?

### BM25의 장점:

- 질문 키워드 정확 반영
- 숫자, 고유명사, 정의형 질문에 강함
- "Embedding" 같은 핵심 단어 인식

### BM25의 약점:

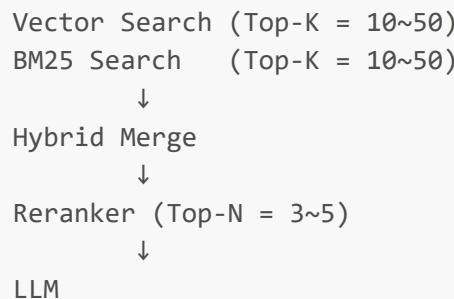
- 의미 표현에는 약함
- 동의어 인식 불가
- "강아지" vs "반려견" 구분 못함

## 3. 실무 RAG의 기본 패턴

### ❖ 현실:

Vector만 쓰는 RAG → 실험용 / 데모용  
Hybrid + Reranker → 실무용 / 프로덕션

### 실무 표준 구조:



## 4. 왜 BM25는 검색이 아니라 재정렬인가?

이것은 완전한 Hybrid Search가 아닙니다!

### 진짜 Hybrid Search:

```

Query
└─ Vector Search (Top-10)      병렬
└─ BM25 Search (Top-10)        병렬
└─ 결과 병합 → 재정렬
  
```

### Day 6 구조 (Light Hybrid / Pseudo Hybrid):



## 차이점:

- BM25가 새로 검색 ✗
- Vector 결과만 재정렬 ☑
- 키워드로만 검색되는 문서는 못 찾음

## 왜 이렇게?

- 구조 단순 (교육용 최적)
- 성능 빠름
- 실무에서도 많이 사용 (MVP)
- 완전한 Hybrid는 확장편에서

## 5. 왜 Top-K와 Top-N을 나눴나?

### 설정:

```

TOP_K = 10 # Vector Search (넓게)
TOP_N = 3   # BM25 Rerank (좁게)
  
```

### 전략:

K (넓게):

- 관련 있을 만한 문서 최대한 수집
- "혹시 이것도?"
- Recall 중시

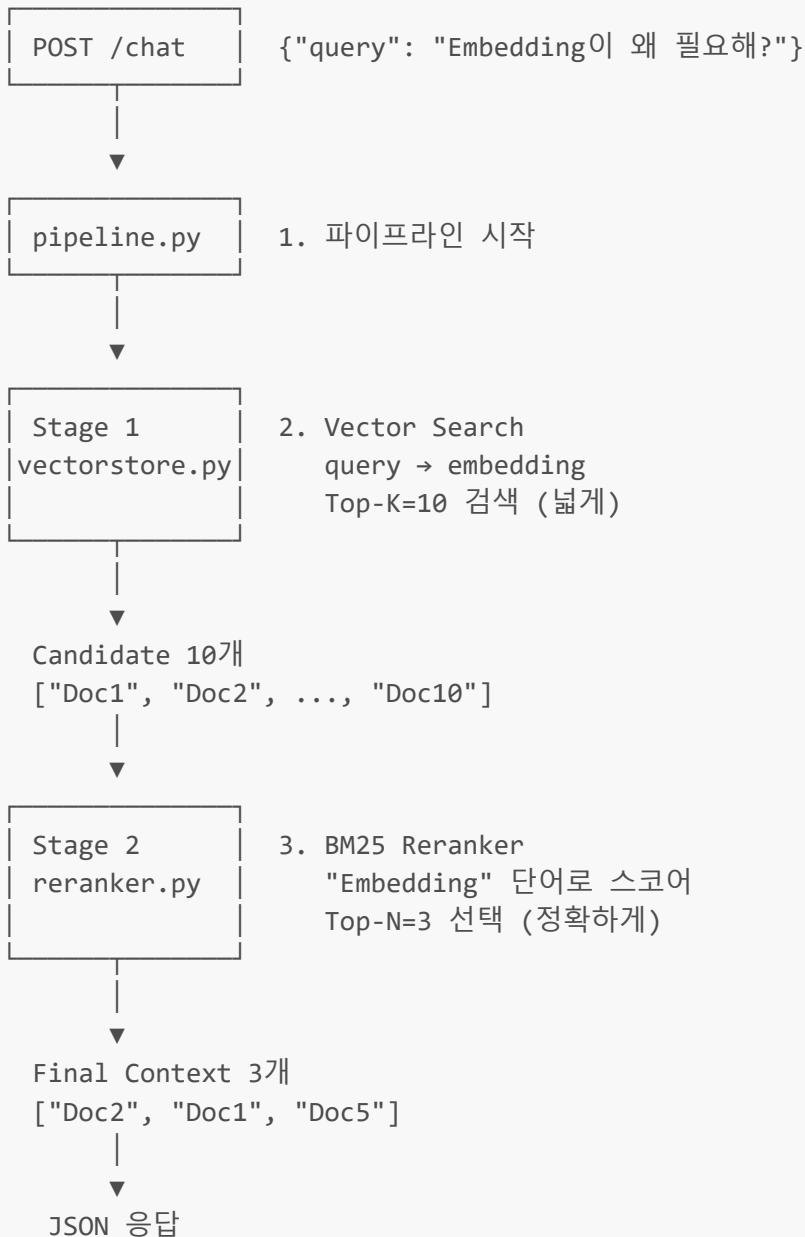
N (좁게):

- 정말 질문에 맞는 것만
- "확실한 것만!"
- Precision 중시

## 왜 K > N?

- 재정렬 풀(pool)이 커야 효과적
- N=K면 재정렬 의미 없음
- 일반적으로 K = 2~4 × N

## 실행 흐름도



## ⚙️ 설치 및 실행

### 1. 패키지 설치

```

cd Rag_minimal_day6
pip install fastapi uvicorn sentence-transformers faiss-cpu rank-bm25
  
```

### rank-bm25:

- BM25 알고리즘 구현
- 약 10KB (매우 가벼움)
- CPU 전용

### 2. 서버 실행

```
uvicorn main:app --reload
```

## 초기 로딩:

```
INFO: Loading embedding model...
INFO: Building vector store...
INFO: Building BM25 reranker...
INFO: 10 documents indexed
INFO: Server ready
```

## 3. API 테스트

### curl (PowerShell):

```
Invoke-RestMethod ` 
-Uri http://127.0.0.1:8000/chat ` 
-Method POST ` 
-ContentType "application/json; charset=utf-8" ` 
-Body '{"query":"RAG에서 Embedding이 왜 필요해?"}'
```

### curl (Bash):

```
curl -X POST "http://127.0.0.1:8000/chat" \
-H "Content-Type: application/json" \
-d '{"query":"RAG에서 Embedding이 왜 필요해?"}'
```

## ❷ 핵심 개념 Deep Dive

### 1. BM25 알고리즘

#### BM25 (Best Matching 25):

$$\text{Score}(D, Q) = \sum \text{IDF}(q_i) \times (\text{TF}(q_i, D) \times (k_1 + 1)) / (\text{TF}(q_i, D) + k_1 \times (1 - b + b \times |D| / \text{avgdl}))$$

용어:

- D: 문서
- Q: 질문
- q<sub>i</sub>: 질문의 단어
- TF: 단어 빈도 (Term Frequency)
- IDF: 역문서 빈도 (Inverse Document Frequency)

- k1, b: 파라미터
- avgdl: 평균 문서 길이

### 간단히:

- 질문 단어가 문서에 많을수록 점수 ↑
- 흔한 단어는 점수 ↓ (IDF)
- 문서 길이 정규화

## 2. 2-Stage Retrieval 상세

### 단계별 변화:

질문: "Embedding" 왜 필요해?"

[Stage 1: Vector Search Top-10]

1. Doc A (score: 0.85) - "벡터 표현..."
2. Doc B (score: 0.82) - "임베딩 기술..." ← "Embedding" 포함
3. Doc C (score: 0.78) - "딥러닝 모델..."
4. Doc D (score: 0.75) - "표현 학습..."
5. Doc E (score: 0.72) - "Feature 추출..."
6. Doc F (score: 0.70) - "임베딩 활용..." ← "Embedding" 포함
7. Doc G (score: 0.68) - "신경망 구조..."
8. Doc H (score: 0.65) - "차원 축소..."
9. Doc I (score: 0.62) - "벡터 공간..."
10. Doc J (score: 0.60) - "유사도 계산..."

[Stage 2: BM25 Rerank Top-3]

1. Doc B (bm25: 5.2) - "임베딩 기술..." ← 1등으로!
2. Doc F (bm25: 4.8) - "임베딩 활용..." ← 6등→2등
3. Doc A (bm25: 2.1) - "벡터 표현..." ← 1등→3등

### 효과:

- "Embedding" 단어가 있는 문서 우선
- 의미는 비슷하지만 질문 단어 없는 문서 하락

## 3. Top-K vs Top-N 최적값

### 일반적인 설정:

작은 문서셋 (< 100개):

K=5, N=2

중간 문서셋 (100~1000개):

K=10, N=3 ← Day 6 기본

큰 문서셋 (> 1000개):

K=20, N=5

## 비율:

K:N = 2:1 ~ 4:1

예시:

- K=10, N=3 (3.3:1)
- K=20, N=5 (4:1)
- K=10, N=5 (2:1)

## 코드 Deep Dive

### config.py

```
# Retrieval 설정
MODEL_NAME = "all-MiniLM-L6-v2"
EMBEDDING_DIM = 384
TOP_K = 10          # Vector Search (넓게)
RERANK_TOP_N = 3    # BM25 Rerank (좁게)
DATA_PATH = "data/docs.txt"
```

### reranker.py (신규 파일)

```
from rank_bm25 import BM25Okapi
from typing import List

def tokenize(text: str):
    """텍스트를 토큰화 (소문자 + 공백 분리)"""
    return text.lower().split()

class BM25Reranker:
    def __init__(self, documents: List[str]):
        """
        BM25 Reranker 초기화

        Args:
            documents: 전체 문서 리스트
        """
        self.documents = documents

        # 모든 문서를 토큰화
        self.corpus = [tokenize(doc) for doc in documents]

        # BM25 인덱스 생성
        self.bm25 = BM25Okapi(self.corpus)
```

```

def score(self, query: str):
    """
    질문에 대한 모든 문서의 BM25 스코어 계산

    Args:
        query: 질문 문자열

    Returns:
        scores: 각 문서의 BM25 점수 배열
    """
    query_tokens = tokenize(query)
    return self.bm25.get_scores(query_tokens)

def rerank(self, query: str, candidates: List[str], top_n: int = 3):
    """
    Candidate 문서들을 BM25로 재정렬

    Args:
        query: 질문
        candidates: Vector Search 결과 (후보)
        top_n: 반환할 최종 개수

    Returns:
        reranked: 재정렬된 상위 N개 문서
    """
    # 모든 문서의 BM25 스코어 계산
    scores = self.score(query)

    # Candidate 문서의 인덱스와 스코어 추출
    scored = [
        (candidate, float(scores[self.documents.index(candidate)]))
        for candidate in candidates
    ]

    # 스코어 기준 내림차순 정렬
    scored.sort(key=lambda x: x[1], reverse=True)

    # Top-N 반환
    return [doc for doc, score in scored[:top_n]]

```

### 핵심 포인트:

- `BM25Okapi`: 실전 검증된 BM25 구현
- `tokenize()`: 한글도 공백으로 분리 (간단)
- `score()`: 전체 문서 스코어 계산
- `rerank()`: Candidate만 필터링 후 정렬

### ingest.py 변경 사항

```

from loader import load_documents
from embedding import embed

```

```

from vectorstore import VectorStore
from reranker import BM25Reranker

def build_assets():
    """
    Retrieval 자산 생성

    Returns:
        store: Vector Store
        reranker: BM25 Reranker
        texts: 원본 문서 리스트
    """

    # 1. 문서 로딩
    docs = load_documents()
    texts = [d.text for d in docs]

    # 2. Vector Store 생성
    vectors = embed(texts)
    store = VectorStore(dim=len(vectors[0]))
    store.add(vectors, texts)

    # 3. Reranker 생성 ← 신규
    reranker = BM25Reranker(texts)

    return store, reranker, texts

```

## 변경점:

- Day 5: VectorStore만 반환
- Day 6: VectorStore + Reranker + texts 반환

## pipeline.py (핵심 로직)

```

from embedding import embed
from ingest import build_assets
from config import TOP_K, RERANK_TOP_N

# 전역 캐싱
_store = None
_reranker = None
_docs = None

def get_assets():
    """자산 로딩 (1회만 실행)"""
    global _store, _reranker, _docs
    if _store is None:
        _store, _reranker, _docs = build_assets()
    return _store, _reranker, _docs

def pipeline(query: str):
    """
    """

```

## RAG 파이프라인 (2-Stage Retrieval)

Args:

query: 사용자 질문

Returns:

```

        result: {query, candidates, contexts}
    """
    store, reranker, docs = get_assets()

    # Query 임베딩
    query_vector = embed([query])[0]

    # Stage 1: Vector Search (넓게)
    candidates = store.search(query_vector, top_k=TOP_K)

    # Stage 2: BM25 Rerank (좁게)
    contexts = reranker.rerank(query, candidates, top_n=RERANK_TOP_N)

    # 디버그 출력 (교육용)
    print("=" * 50)
    print("CANDIDATES (Vector Top-K):")
    for i, c in enumerate(candidates, 1):
        print(f" {i}. {c[:50]}...")
    print("\nCONTEXTS (Reranked Top-N):")
    for i, c in enumerate(contexts, 1):
        print(f" {i}. {c[:50]}...")
    print("=" * 50)

    return {
        "query": query,
        "candidates": candidates, # Vector 결과 (10개)
        "contexts": contexts      # 최종 결과 (3개)
    }

```

### 핵심 변경:

```

# Day 5
results = vector_store.search(query_vector, k=3)
return results

# Day 6
candidates = vector_store.search(query_vector, k=10) # 넓게
contexts = reranker.rerank(query, candidates, n=3) # 좁게
return {"candidates": candidates, "contexts": contexts}

```

## 💡 학습 효율을 높이는 팁

### 1. 터미널 로그로 효과 확인

**실행:**

```
uvicorn main:app --reload
```

**API 호출:**

```
Invoke-RestMethod ` 
-Uri http://127.0.0.1:8000/chat `
-Method POST `
-ContentType "application/json" `
-Body '{"query":"RAG에서 Embedding이 왜 필요해?"}'
```

**예상 터미널 출력:**

```
=====
CANDIDATES (Vector Top-K):
1. 벡터 표현의 장점과 활용...
2. 임베딩 기술 상세 소개...
3. 딥러닝 모델 구조 설명...
4. 표현 학습의 기초...
5. Feature 추출 방법...
6. 임베딩 활용 사례...
7. 신경망 구조 이해...
8. 차원 축소 기법...
9. 벡터 공간 모델...
10. 유사도 계산 방식...

CONTEXTS (Reranked Top-N):
1. 임베딩 기술 상세 소개...
2. 임베딩 활용 사례...
3. 벡터 표현의 장점과 활용...
=====
```

**☞ 이 출력 하나로 STEP 6의 목적이 설명됨****2. BM25 스코어 확인****reranker.py 디버그 추가:**

```
def rerank(self, query, candidates, top_n):
    scores = self.score(query)

    scored = [
        (candidate, float(scores[self.documents.index(candidate)]))
        for candidate in candidates
    ]
```

```

scored.sort(key=lambda x: x[1], reverse=True)

# 스코어 출력 (디버그)
print("\nBM25 Scores:")
for doc, score in scored[:top_n]:
    print(f" {score:.2f} - {doc[:50]}...")

return [doc for doc, score in scored[:top_n]]

```

### 3. K와 N 실험

#### config.py 설정:

```

# 실험 1: 좁게
TOP_K = 5
RERANK_TOP_N = 2

# 실험 2: 넓게
TOP_K = 20
RERANK_TOP_N = 5

# 실험 3: K=N (재정렬 무의미)
TOP_K = 3
RERANK_TOP_N = 3

```

#### 비교 항목:

- 검색 품질 (정확도)
- 응답 시간
- 최적 조합 찾기

### 🔑 직접 해보기: 실습

#### 실습 1: TOP\_K, TOP\_N 변경

#### config.py:

```

TOP_K = 15      # 10 → 15
RERANK_TOP_N = 5 # 3 → 5

```

#### 재시작 후 비교:

- 더 많은 후보
- 더 많은 최종 결과
- 품질 차이 확인

## 실습 2: BM25 vs Vector 순위 변화 추적

### test\_comparison.py:

```
from pipeline import pipeline

queries = [
    "Embedding이 왜 필요해?",          # 키워드 명확
    "벡터 검색의 장점은?",            # 의미 기반
    "RAG 시스템 구축 방법"           # 복합
]

for q in queries:
    result = pipeline(q)
    print(f"\n질문: {q}")
    print(f"\nVector 1위: {result['candidates'][0][:50]}")
    print(f"Rerank 1위: {result['contexts'][0][:50]}\n")

    # 순위 변화 계산
    original_rank = result['candidates'].index(result['contexts'][0]) + 1
    print(f"\n순위 변화: {original_rank} → 1")
```

## 실습 3: 한글 토큰화 개선 (선택)

### 현재:

```
def tokenize(text):
    return text.lower().split() # 공백 기준
```

### 개선 (형태소 분석):

```
# konlpy 설치
pip install konlpy

# reranker.py
from konlpy.tag import Okt
okt = Okt()

def tokenize(text):
    return okt.morphs(text) # 형태소 분석
```

### 효과:

- "임베딩이" → ["임베딩", "이"]
- "필요해?" → ["필요하다", "?"]
- 더 정확한 키워드 매칭

## Day 6 완료 기준 (명확)

아래를 설명할 수 있으면 통과입니다:

5가지 필수 질문

### 1. 왜 Vector Search만으로 부족한가?

- 의미 유사 ≠ 질문 적합
- 키워드 정확도 낮음
- 노이즈 혼입

### 2. Hybrid Search가 무엇을 보완하는가?

- Vector의 의미 유사도 보완
- Keyword의 정확도 보완
- 두 가지 장점 결합

### 3. Reranker의 역할은 무엇인가?

- 재정렬 (검색 아님)
- 질문 키워드 기반 스코어링
- Top-N 정밀 선택

### 4. 왜 LLM 전에 이 작업을 해야 하는가?

- LLM 컨텍스트 품질 향상
- "답이 이상하다" 문제 해결
- 토큰 비용 절감

### 5. Top-K와 Top-N의 차이는?

- K: Vector 후보 개수 (넓게)
- N: 최종 선택 개수 (좁게)
- $K > N$ , 일반적으로  $K = 2\sim4 \times N$

---

## → 다음 단계 (Day 7)

Day 7에서는:

### 3-Stage Orchestration:

```
def pipeline(query):  
    # Stage 1: Retrieve  
    candidates = step_retrieve(query)  
  
    # Stage 2: Rerank  
    contexts = step_rerank(query, candidates)  
  
    # Stage 3: Generate  
    response = step_generate(query, contexts)
```

```
# Trace 객체로 추적
return {
    "trace": trace,
    "response": response
}
```

## 추가 기능:

- 각 단계 명확히 분리
- Trace 객체로 디버깅
- 단계별 실행 시간 측정
- 확장 가능한 구조

---

## ○ 학습 후 자가 점검

모든 질문에 답할 수 있다면 Day 6 완료! 🎉

1. **Vector Search의 한계는?**  의미만 보고 질문 단어 무시, 정확도 낮음
  2. **BM25 Reranker의 역할은?**  재정렬 (검색 아님), 키워드 기반 스코어링
  3. **TOP\_K와 TOP\_N의 차이는?**  K: Vector 후보 개수 / N: 최종 선택 개수
  4. **왜 K > N인가?**  재정렬 풀이 커야 효과적, 일반적으로  $K = 2 \sim 4 \times N$
  5. **이것이 진짜 Hybrid Search인가?**  아니오, Light Hybrid (Vector만 검색, BM25는 재정렬)
  6. **실무에서는 어떻게 쓰나?**  Hybrid + Reranker가 표준, Vector만은 실험용
- 

## 문서 라이선스

Copyright © 2022 정상혁 (Sanghyuk Jung)

본 저작물은 [크리에이티브 커먼즈 저작자표시-비영리-변경금지 4.0 국제 라이선스](#)에 따라 이용할 수 있습니다.

### 허용

- 개인 학습 목적 사용
- 출처 표시 후 비영리 공유

### 금지

- 상업적 이용
- 내용 수정 및 2차 저작
- 저작자 허락 없는 재배포

상업적 이용 문의: j4angguiop@gmail.com

---

★ 이 프로젝트가 도움이 되었다면 Star를 눌러주세요!

Copyright © 2022-2026 정상혁 (Sanghyuk Jung). All Rights Reserved.