Deep Learning Application

Chapter 3. 벡터 스토어

RAG

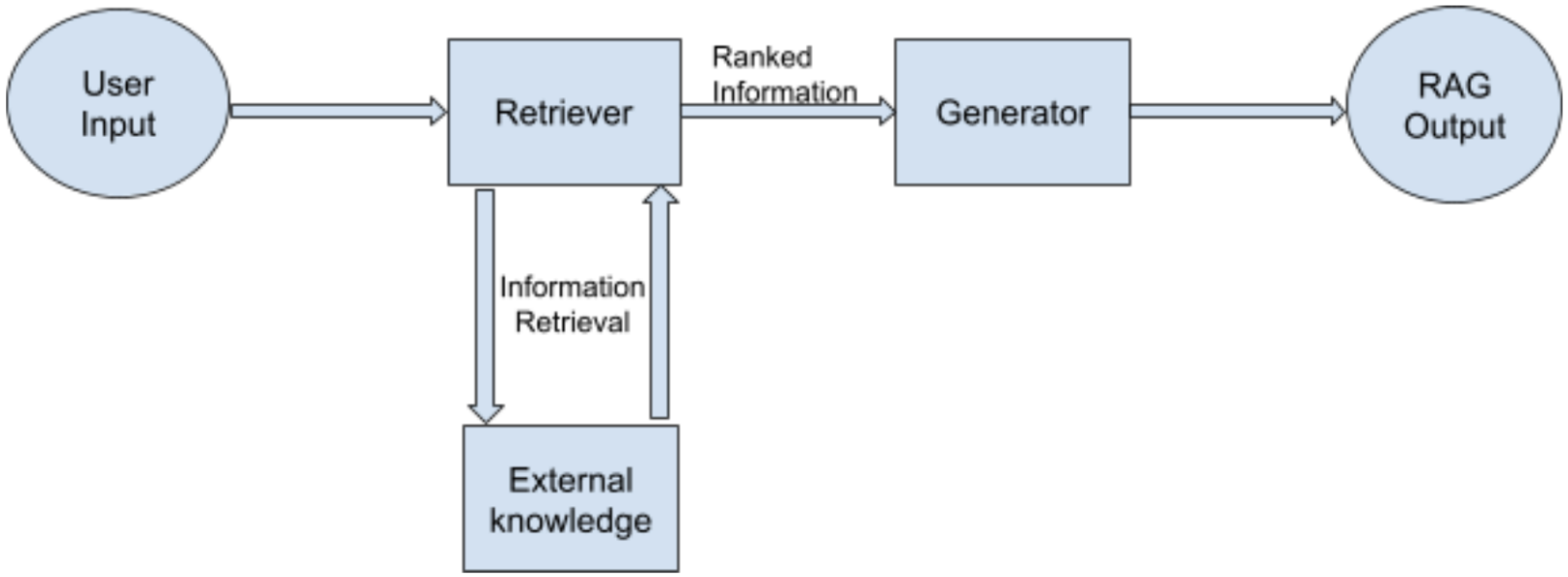


Figure 2: A basic flow of the RAG system along with its component

Shailja Gupta et. al, **"A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions"**

RAG

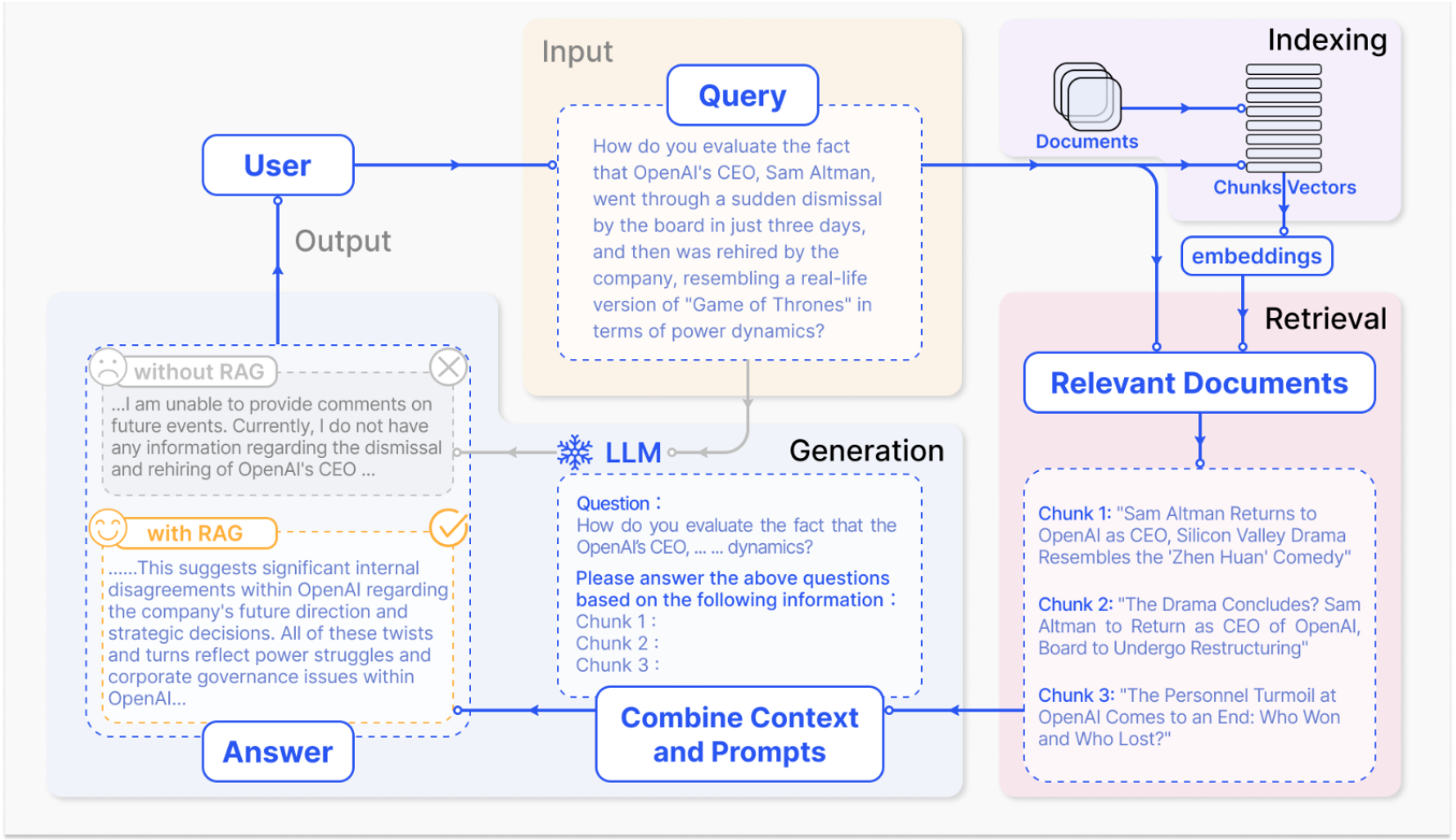


Fig. 2. A representative instance of the RAG process applied to question answering. It mainly consists of 3 steps. 1) Indexing. Documents are split into chunks, encoded into vectors, and stored in a vector database. 2) Retrieval. Retrieve the Top k chunks most relevant to the question based on semantic similarity. 3) Generation. Input the original question and the retrieved chunks together into LLM to generate the final answer.

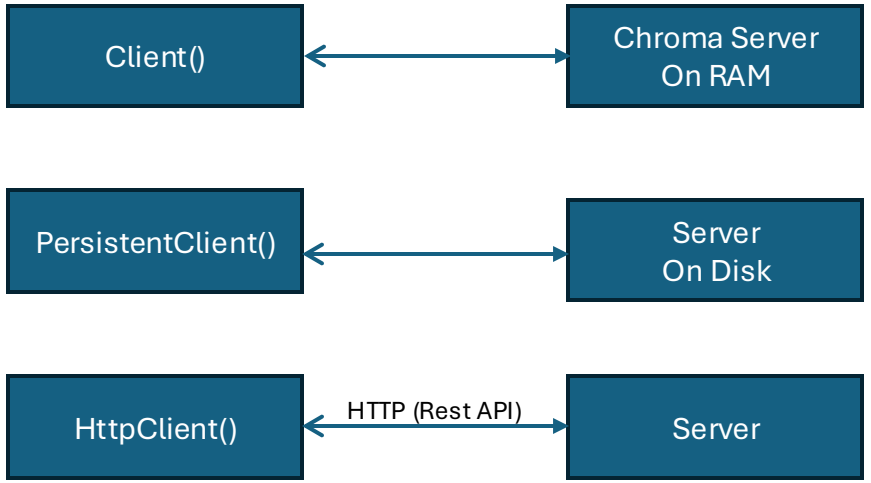
Yunfan Gao, et. al, “Retrieval-Augmented Generation for Large Language Models: A Survey”, 2024

VectorStore

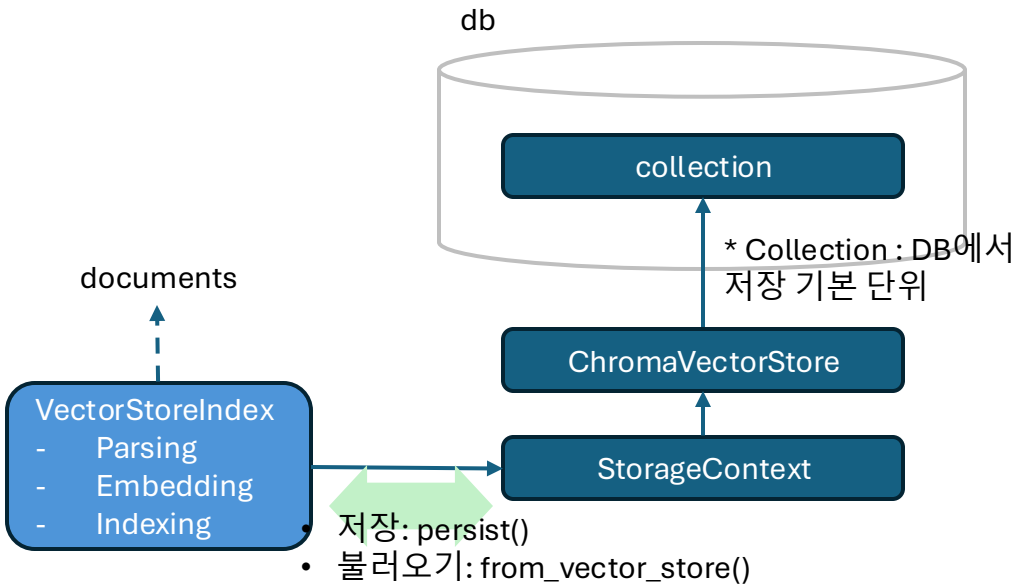
- 벡터 스토어(VectorStore):
  - 벡터화된 데이터를 효율적으로 저장하고 검색할 수 있도록 설계된 데이터 관리 시스템
  - 검색 시스템에서는 텍스트, 이미지 등 데이터를 벡터로 변환한 뒤, 이를 기반으로 유사도를 계산하여 검색하거나 분석하는 작업이 자주 필요
  - 별도의 벡터 저장소 없이 검색을 수행할 경우, 모든 데이터를 순차적으로 비교해야 하므로 데이터의 크기가 커질수록 검색 속도가 급격히 느려짐.
- 벡터: 데이터를 다차원 공간에서 수치화한 값
- 벡터 컬렉션:
  - 여러 벡터를 그룹화하여 관리하는 단위, 데이터를 그룹화하여 처리
  - 예) 고객 피드백 데이터를 긍정적인 리뷰와 부정적인 리뷰 컬렉션으로 분류
  - 예) 학술 데이터에서 생물학, 물리학, 수학 논문을 각각의 컬렉션으로 나누어 관리
- 라마인덱스는 다양한 벡터 저장소와 통합되어 있어 사용자가 작업 목적에 가장 적합한 벡터 스토어를 선택하여 활용
  - 대표적인 벡터 스토어인 크로마(Chroma), 파인콘 (Pinecone), 쿼드런트(Qdrant)를 중심으로 각각의 특성과 사용 사례를 실습

Chromadb 사전 지식

```
# 메모리 모드
client = chromadb.Client()
# 프로그램 종료 시 데이터 사라짐

# 디스크 모드
client = chromadb.PersistentClient(path="./chroma_db")
# ./chroma_db/ 안에 chroma.sqlite3, collections/ 등 파일 생성됨

# HTTP 클라이언트 생성 (서버 주소 지정)
client = chromadb.HttpClient(host="localhost", port=8000)
```



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

# --- 1. 크로마 클라이언트: 영구 저장 경로 지정 ---
db = chromadb.PersistentClient(path="./chroma_db")

# --- 2. 컬렉션 생성 (이미 있으면 재사용) ---
chroma_collection = db.get_or_create_collection("quickstart")

# --- 3. ChromaVectorStore를 스토리지 컨텍스트로 지정 ---
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# --- 4. LlamaIndex가 Chroma 내부에 바로 데이터 저장 ---
index = VectorStoreIndex.from_documents(
    documents,
    storage_context=storage_context,
)

# --- 5. 나중에 복원할 때 ---
db = chromadb.PersistentClient(path="./chroma_db")
chroma_collection = db.get_or_create_collection("quickstart")
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# Chroma에 저장된 데이터로 인덱스 복원
index = VectorStoreIndex.from_vector_store(
    vector_store=vector_store,
    storage_context=storage_context,
)

# --- 6. 쿼리 수행 ---
query_engine = index.as_query_engine()
response = query_engine.query("What is Chroma?")
print(response)
```



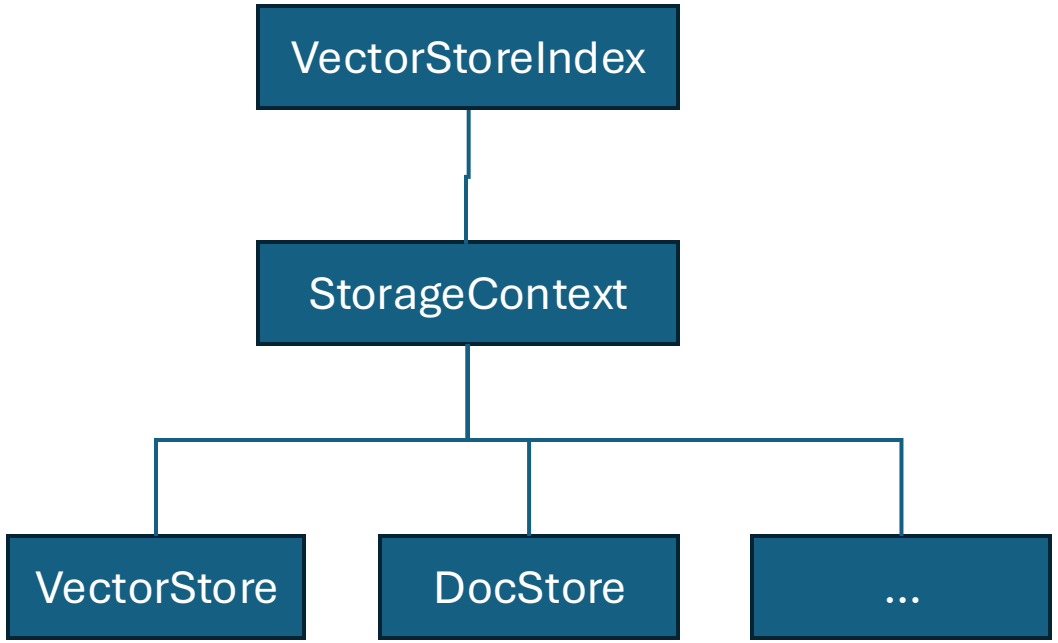
VectorStoreIndex.from_documents()
│
├─ ① 문서(Document) → Node로 분할 (chunking)
├─ ② Node 내용에 대해 임베딩 모델 호출 (벡터 생성)
├─ ③ 벡터 + 메타데이터를 vector_store에 add()
│ └→ (여기서 chroma_collection.add() 호출됨)
└─ ④ VectorStoreIndex 객체 생성

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

# --- 1. 크로마 클라이언트: 영구 저장 경로 지정 ---
db = chromadb.PersistentClient(path="./chroma_db")

# --- 2. 컬렉션 생성 (이미 있으면 재사용) ---
chroma_collection = db.get_or_create_collection("quickstart")

# --- 3. ChromaVectorStore를 스토리지 컨텍스트로 지정 ---
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# --- 4. LlamaIndex가 Chroma 내부에 바로 데이터 저장 ---
index = VectorStoreIndex.from_documents(
    documents,
    storage_context=storage_context,
)

# --- 5. 나중에 복원할 때 ---
db = chromadb.PersistentClient(path="./chroma_db")
chroma_collection = db.get_or_create_collection("quickstart")
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# Chroma에 저장된 데이터로 인덱스 복원
index = VectorStoreIndex.from_vector_store(
    vector_store=vector_store,
    storage_context=storage_context,
)

# --- 6. 쿼리 수행 ---
query_engine = index.as_query_engine()
response = query_engine.query("What is Chroma?")
print(response)
```

```python
import os, uuid
import chromadb
from chromadb.utils import embedding_functions

# 1) 영구 저장용 클라이언트
client = chromadb.PersistentClient(path="./chroma_db")

# 2) 임베딩 함수 지정(예: sentence-transformers)
#    - OpenAI 등 다른 임베딩도 가능 (keys 필요)
embedding_fn = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name="all-MiniLM-L6-v2"
)

# 3) 컬렉션 생성/재사용 (임베딩 함수 바인딩)
collection = client.get_or_create_collection(
    name="quickstart",
    embedding_function=embedding_fn,
)

vector_store = ChromaVectorStore(chroma_collection=collection)

# LlamaIndex의 VectorStoreIndex 생성
index = VectorStoreIndex.from_documents(
    nodes,
    vector_store=vector_store,
    embed_model= embed_model,
    llm=llm
)
```

두개의 embedding 모델?
혼용하면 항상 LlamaIndex의 embed_model이 우선

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

# --- 1. 크로마 클라이언트: 영구 저장 경로 지정 ---
db = chromadb.PersistentClient(path="./chroma_db")

# --- 2. 컬렉션 생성 (이미 있으면 재사용) ---
chroma_collection = db.get_or_create_collection("quickstart")

# --- 3. ChromaVectorStore를 스토리지 컨텍스트로 지정 ---
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# --- 4. LlamaIndex가 Chroma 내부에 바로 데이터 저장 ---
index = VectorStoreIndex.from_documents(
    documents,
    storage_context=storage_context,
)

# --- 5. 나중에 복원할 때 ---
db = chromadb.PersistentClient(path="./chroma_db")
chroma_collection = db.get_or_create_collection("quickstart")
vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# Chroma에 저장된 데이터로 인덱스 복원
index = VectorStoreIndex.from_vector_store(
    vector_store=vector_store,
    storage_context=storage_context,
)

# --- 6. 쿼리 수행 ---
query_engine = index.as_query_engine()
response = query_engine.query("What is Chroma?")
print(response)
```



ChromaDB

3.1 개발환경 구축

- /ch03
- Pipenv install chromadb

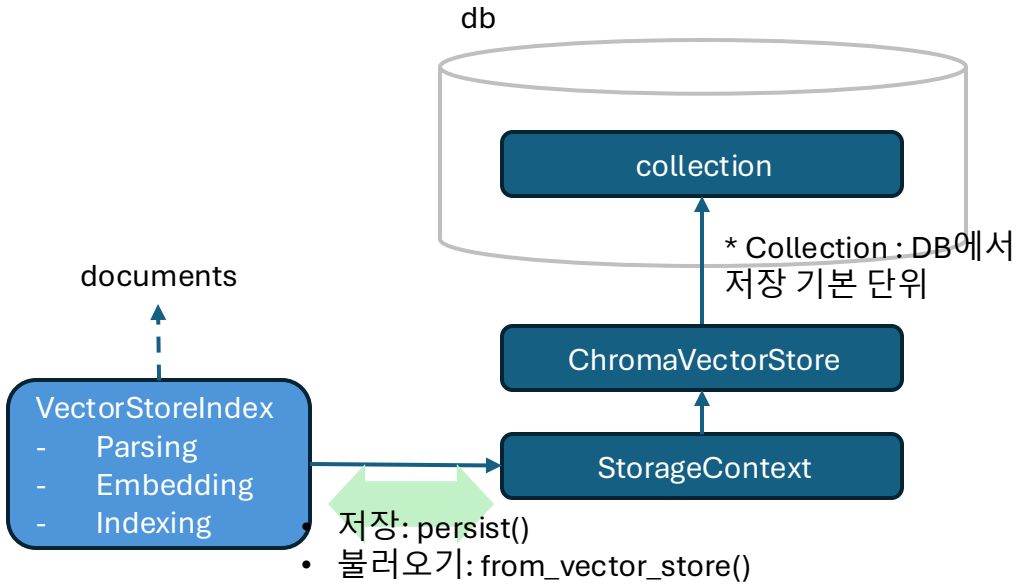
3.2 Chromadb
3.2.1-3.2.3 Client 생성, Collection 생성, 벡터데이터 추가

- 크로마(Chroma): 오픈 소스 벡터 저장소

```python
import chromadb

# 크로마 클라이언트 생성
client = chromadb.Client()
```

```python
# 벡터 컬렉션 생성
collection = client.create_collection("example_collection")
```

```python
# 임베딩된 벡터 데이터 (예시로 임의 벡터 사용)
vectors = [
[0.1, 0.2, 0.3], # 첫 번째 데이터의 벡터
[0.4, 0.5, 0.6], # 두 번째 데이터의 벡터
[0.7, 0.8, 0.9], # 세 번째 데이터의 벡터
]
# 벡터와 연결된 임의 고유 ID (각 벡터마다 고유한 ID 필요)
ids = ["doc1", "doc2", "doc3"]

# 벡터 데이터 추가
collection.add(ids=ids, embeddings=vectors)
print("벡터 데이터를 컬렉션에 추가했습니다.")
```



3.2 Chromadb
3.2.4 벡터 검색

```python
# 벡터 컬렉션 생성
collection = client.create_collection("example_collection")
```

```python
# 임베딩된 벡터 데이터 (예시로 임의 벡터 사용)
vectors = [
[0.1, 0.2, 0.3], # 첫 번째 데이터의 벡터
[0.4, 0.5, 0.6], # 두 번째 데이터의 벡터
[0.7, 0.8, 0.9], # 세 번째 데이터의 벡터
]
# 벡터와 연결된 임의의 고유 ID (각 벡터마다 고유한 ID 필요)
ids = ["doc1", "doc2", "doc3"]

# 벡터 데이터 추가
collection.add(ids=ids, embeddings=vectors)
print("벡터 데이터를 컬렉션에 추가했습니다.")
```

```python
import json

# 검색할 벡터 (예시로 임의 벡터 사용)
query_vector = [0.1, 0.2, 0.25]

# 벡터 컬렉션에서 유사한 벡터 검색
results = collection.query(query_embeddings=[query_vector], n_results=2)
formatted_results = {
    "검색된 문서 ID": results["ids"][0],
    "유사도 거리": results["distances"][0]
}
print("\n유사한 벡터 검색 결과:")
print(json.dumps(formatted_results, indent=4, ensure_ascii=False))
```

* The **distance metric** is a **collection setting**. By default, **new collections use L2 (squared Euclidean)**

```python
# 벡터 컬렉션 생성
#collection = client.create_collection("example_collection")
collection = client.get_or_create_collection(
    "example_collection",
    metadata={"hnsw:space": "cosine"}
)
```

```
유사한 벡터 검색 결과:
{
    "검색된 문서 ID": [
        "doc1",
        "doc2"
    ],
    "유사도 거리": [
        0.002500001108273864,
        0.30250000953674316
    ]
}
```

3.2 Chromadb
3.2.5 메타데이터 필터링

- 크로마의 메타데이터 기반 필터링
  - 검색 시 특정 조건을 만족하는 메타데이터를 가진 벡터만 검색 대상에 포함

```
유사한 벡터 검색 결과:
{
    "검색된 문서 ID": [
        "doc1"
    ],
    "유사도 거리": [
        0.0
    ],
    "메타데이터": [
        {
            "category": "A",
            "name": "example"
        }
    ]
}
```

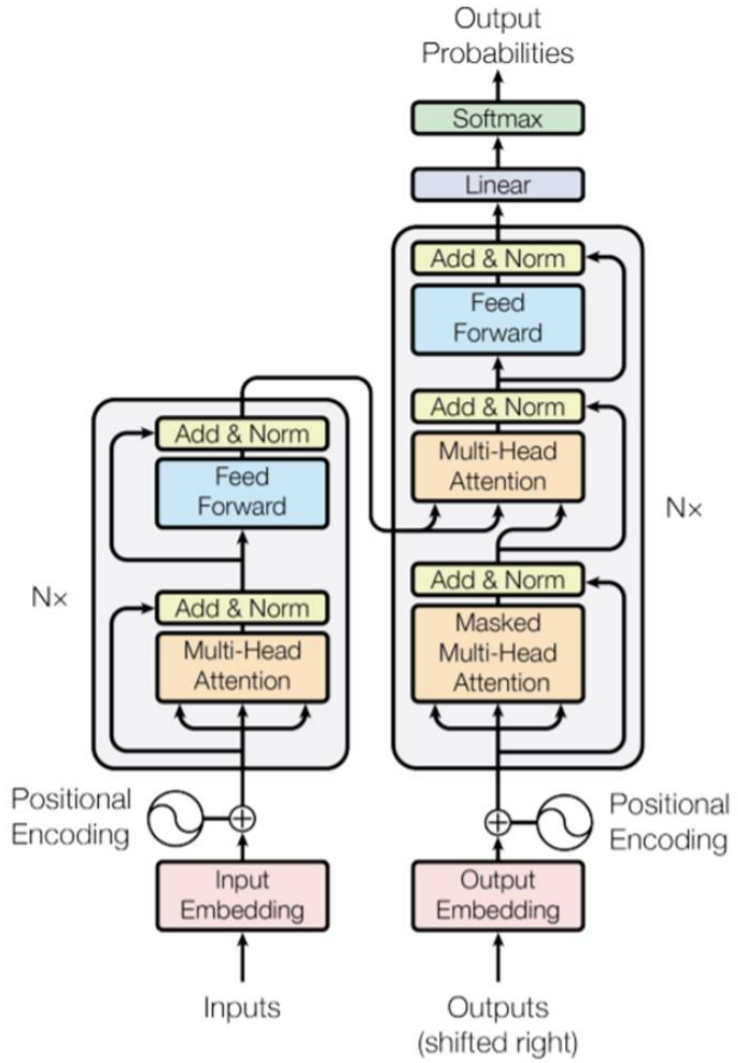
```python
collection = client.create_collection("metadata_example_collection")

# 임베딩된 벡터 데이터 및 메타데이터 추가
vectors = [
    [0.1, 0.2, 0.3], # 첫 번째 데이터 벡터
    [0.4, 0.5, 0.6], # 두 번째 데이터 벡터
    [0.7, 0.8, 0.9], # 세 번째 데이터 벡터
]
ids = ["doc1", "doc2", "doc3"]

metadatas = [
    {"name": "example", "category": "A"}, # 첫 번째 문서의 메타데이터
    {"name": "sample", "category": "B"}, # 두 번째 문서의 메타데이터
    {"name": "example", "category": "C"} # 세 번째 문서의 메타데이터
]

# 벡터 데이터 추가 (메타데이터 포함)
collection.add(ids=ids, embeddings=vectors, metadatas=metadatas)
results = collection.query(
    query_embeddings=[[0.1, 0.2, 0.3]],
    n_results=1,
    where={"name": "example"} # 메타데이터 필터 적용
)

# 검색 결과 정리
formatted_results = {
    "검색된 문서 ID": results["ids"][0] if results["ids"] else [],
    "유사도 거리": results["distances"][0] if results["distances"] else [],
    "메타데이터": results["metadatas"][0] if results["metadatas"] else []
}

print("\n유사한 벡터 검색 결과:")
print(json.dumps(formatted_results, indent=4, ensure_ascii=False))
```

3.2 Chromadb
3.2.6 임베딩 데이터 추가

- Sentence-transformer :
  - BERT 기반의 문장 임베딩 모델: 문장의 의미를 벡터로 변환
  - 설치: install sentence-transformers
  - 'all-MiniLM-L6-v2' 사용

```python
from sentence_transformers import SentenceTransformer

# 모델 로드
model = SentenceTransformer('all-MiniLM-L6-v2')

# 임베딩할 문장
sentence = "이것은 임베딩 예제입니다."

# 문장 임베딩 생성
embedding = model.encode(sentence)

# 임베딩 출력
print("임베딩 결과:", embedding)
print("임베딩 차원:", embedding.shape)
```

```
임베딩 차원: (384,)
```



3.2 Chromadb
3.2.6 임베딩 데이터 추가

```
client = chromadb.Client()
collection = client.create_collection("example-collection")
# 임베딩 모델 초기화
model = SentenceTransformer('all-MiniLM-L6-v2')
documents = [
    "고양이는 작은 육식동물로, 주로 애완동물로 기릅니다. 민첩하고 장난기 있는 행동으로 유명합니다.",
    "강아지는 충성심이 강하고 친절한 동물로, 흔히 인간의 최고의 친구로 불립니다. 주로 애완동물로 기르고, 동반자로서 유명합니다.",
    "고양이와 강아지는 전 세계적으로 인기 있는 애완동물로, 각각 독특한 특징을 가지고 있습니다."
]
ids = ["doc1", "doc2", "doc3"]
embeddings = model.encode(documents)

# 벡터 데이터 추가
collection.add(
    ids=ids,
    documents=documents,
    embeddings=embeddings,
)
print("벡터 데이터를 컬렉션에 추가했습니다.")
print("임베딩 차원:", embeddings.shape)
```

```
벡터 데이터를 컬렉션에 추가했습니다.
임베딩 차원: (3, 384)
```

3.2 Chromadb
3.2.7 임베딩 데이터 검색

- 저장된 벡터 데이터를 검색하려면 쿼리로 사용할 텍스트 역시 임베딩해야 함

```
query_text = "고양이"
query_embedding = model.encode([query_text])

# 벡터 컬렉션에서 유사한 벡터 검색
results = collection.query(query_embeddings=query_embedding, n_results=2)

formatted_results = {
    "검색된 문서 ID": results["ids"][0],
    "유사도 거리": results["distances"][0]
}
print("\n유사한 벡터 검색 결과:")
print(json.dumps(formatted_results, indent=4, ensure_ascii=False))
```

```
유사한 벡터 검색 결과:
{
    "검색된 문서 ID": [
        "doc1",
        "doc3"
    ],
    "유사도 거리": [
        1.4484705924987793,
        1.4485851526260376
    ]
}
```

3.2 Chromadb
3.2.8 크로마의 저장방식

- 크로마는
  - 기본적으로 메모리 기반으로 동작하지만,
  - 필요에 따라 디스크 기반으로 데이터 를 영구 저장할 수 있는 기능도 제공
    - 시스템을 재시작하거나 오랜 시간 동안 데이터를 유지해야 할 때 유용
    - 디스크에 저장된 벡터 데이터는 재시작 후에도 불러올 수 있어 영구적인 데이터 저장소로 활용

```python
from chromadb import PersistentClient

client = PersistentClient(path="chroma_storage") # 영구 저장 경로 설정
collection = client.get_or_create_collection("persistent_collection")

# 컬렉션 생성 및 데이터 추가
# 자동으로 디스크 저장됨
collection.add(
    embeddings=[[0.9, 0.8, 0.7]],
    metadatas=[{"name": "persistent_item"}],
    ids=["doc3"]
)
```

vs.

```python
import chromadb

# 크로마 클라이언트 생성
client = chromadb.Client()
```

```python
# 벡터 컬렉션 생성
collection = client.create_collection("example_collection")
```

3.2 Chromadb
3.2.8 크로마의 저장방식

```python
from chromadb import PersistentClient

client = PersistentClient(path="chroma_storage") # 영구 저장 경로 설정
collection = client.get_or_create_collection("persistent_collection")

# 컬렉션 생성 및 데이터 추가
# 자동으로 디스크 저장됨
collection.add(
    embeddings=[[0.9, 0.8, 0.7]],
    metadatas=[{"name": "persistent_item"}],
    ids=["doc3"]
)
```

```python
# Chroma 클라이언트 재시작 후 데이터를 불러옴
client = PersistentClient(path="chroma_storage")
collection = client.get_collection("persistent_collection")

# 저장된 데이터 확인
results = collection.query(
    query_embeddings=[[0.9, 0.8, 0.7]],
    n_results=1
)

# 검색 결과 정리
formatted_results = {
    "검색된 문서 ID": results["ids"][0],
    "유사도 거리": results["distances"][0],
    "메타데이터": results["metadatas"][0]
}

print("\n저장된 데이터 검색 결과:")
print(json.dumps(formatted_results, indent=4, ensure_ascii=False))
```

```python
results = collection.query(
    query_embeddings=[[0.9, 0.8, 0.7]],
    n_results=1,
    include=["embeddings", "documents", "metadatas", "distances", "uris"]
)
```

In Chroma, the `query()` method has an **`include` argument**.
If you don't set it, Chroma uses its **default**:

include = ["metadatas", "documents", "distances"]

It means, it returns **metadata** (which you added: {'name': 'persistent_item'}), **documents**, and **distances**

3.2 Chromadb
3.2.9 임베딩 기반 라마인덱스 답변 생성

- 크로마는 벡터 저장소이므로 벡터 유사도를 기반으로 적절한 문서를 검색
- 검색된 문서를 바탕으로 자연어 형태의 답변을 생성하려면 추가적인 처리가 필요
- 라마인덱스와 크로마, 그리고 허깅페이스 임베딩 모델을 사용하기 위한 패키지를 설치
    - Llama-index, llama-index-vector-stores-chroma, llama-index-embeddings-huggingface

```python
import chromadb
from llama_index.core.schema import Document
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
import os
api_key = os.environ.get("OPENAI_API_KEY")

# ChromaDB 클라이언트 생성 및 컬렉션 준비
client = chromadb.PersistentClient(path="./chroma_db") # 데이터를 저장할 로컬 경로 지정
collection = client.get_or_create_collection("example-collection") # 컬렉션 생성 또는 불러오기

# Hugging Face 임베딩 모델 설정
embed_model = HuggingFaceEmbedding(model_name="all-MiniLM-L6-v2")

# 문서 데이터 준비
documents = [
    "고양이는 작은 육식동물로, 주로 애완동물로 기릅니다. 민첩하고 장난기 있는 행동으로 유명합니다.",
    "강아지는 충성심이 강하고 친절한 동물로, 흔히 인간의 최고의 친구로 불립니다. 주로 애완동물로 기르고, 동반자로서 유명합니다.",
    "고양이와 강아지는 전 세계적으로 인기 있는 애완동물로, 각각 독특한 특징을 가지고 있습니다."
]
ids = ["doc1", "doc2", "doc3"]

# 문서를 LlamaIndex의 Document 형식으로 변환
nodes = [Document(text=doc, id_=doc_id) for doc, doc_id in zip(documents, ids)]
```

* We are creating a list of **Document objects**, not real LlamaIndex **Node objects**.

3.2 Chromadb
3.2.9 임베딩 기반 라마인덱스 답변 생성

```python
import os
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.core import VectorStoreIndex
from llama_index.llms.openai import OpenAI

llm = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

# Chroma 벡터 스토어 생성
vector_store = ChromaVectorStore(chroma_collection=collection)

# LlamaIndex의 VectorStoreIndex 생성
index = VectorStoreIndex.from_documents(nodes, vector_store=vector_store,
embed_model= embed_model, llm=llm)
```

```python
# after building index
print(f"Total nodes in index: {len(index.docstore.docs)}")
```

```python
# 쿼리 엔진 생성
query_engine = index.as_query_engine()

# 질의 수행
query_text = "고양이에 대해 알려줘"
response = query_engine.query(query_text)

# 결과 출력
print("[질의 결과]")
print(response)
```

Then `VectorStoreIndex.from_documents` will internally call the default **NodeParser** (`SentenceSplitter`, chunk size=1024, overlap=20).

```python
index = VectorStoreIndex.from_documents(
    nodes,
    vector_store=vector_store,
    embed_model=embed_model
    # no llm
)
```

- **embed_model** → only used to turn text into vectors for storage/retrieval.
- **llm** → used for *generation* and *reasoning* tasks (summarization, query-time synthesis, tree summarization, etc.).
- If llm is not set explicitly, any query-time response synthesis (as_query_engine()) will just use the **global default LLM** configured in llama_index.core.Settings.llm.

3.2 Chromadb
3.2.10 라마인덱스 기반 답변 생성

- 만약 임베딩 모델을 별도로 지정하지 않는다면
  - ChromaVectorStore는 기본적으로 OpenAI의 text-embedding-ada-002 사용 (변경 가능, 라마인덱스의 공식 문서' 참고)
- Default 임베딩 모델 체크

```
from llama_index.core import Settings
print(f"현재 임베딩 모델: {Settings.embed_model}")
```

model_name='text-embedding-ada-002'

- Default LLM 체크

```
from llama_index.core import Settings
print(f"현재 LLM 모델: {Settings.llm}")
```

model='gpt-3.5-turbo'
temperature=0.1

3.2 Chromadb
3.2.10 라마인덱스 기반 답변 생성

```python
documents = [
    "고양이는 작은 육식동물로, 주로 애완동물로 기릅니다. 민첩하고 장난기 있는 행동으로 유명합니다.",
    "강아지는 충성심이 강하고 친절한 동물로, 흔히 인간의 최고의 친구로 불립니다. 주로 애완동물로 기르고, 동반자로서 유명합니다.",
    "고양이와 강아지는 전 세계적으로 인기 있는 애완동물로, 각각 독특한 특징을 가지고 있습니다."
]
ids = ["doc1", "doc2", "doc3"]

# 문서를 LlamaIndex의 Document 형식으로 변환
nodes = [Document(text=doc, id_=doc_id) for doc, doc_id in zip(documents, ids)]

# Chroma 벡터 스토어 생성
vector_store = ChromaVectorStore(chroma_collection=collection)

# LlamaIndex의 VectorStoreIndex 생성
index = VectorStoreIndex.from_documents(nodes, vector_store=vector_store)

# 쿼리 엔진 생성 (기본적인 검색 + 답변 생성 기능 활성화)
query_engine = index.as_query_engine()
```

```python
query_text = "고양이에 대해 알려줘"
response = query_engine.query(query_text)

# 최종 응답 출력
print("[질의 결과]")
print(response)

# 응답 생성에 사용된 문서 확인
print("\n[검색 문서]")

for i, node in enumerate(response.source_nodes, 1):
    print(f"{i}. {node.text}\n")
```

3.3 Pinecone
3.3.1 Pinecone API 초기화

- Chroma: 온프레미스(로컬) 환경에 최적화된 벡터 저장소
- 파인콘(Pinecone):
  - 클라 우드 기반의 고성능 벡터 데이터베이스
  - 클라우드를 통해 글로벌 분산 아키텍처를 지원하여 여러 지역에 데이터를 분산 저장하고 검색 성능을 최적화
  - 전 세계 사용자에게 일관되고 빠른 응답 속도를 제공
  - 파인콘의 주요 특징 중 하나는 유연한 확정성: 클라우드 기반이기 때문에 데이터의 크기가 증가하더라도 복잡한 인프라 설정 없이 손쉽게 인프라를 확장. 사용자는 인프라 운영에 대한 부담 없이 벡터 데이터를 안정적으로 저장하고 검색

- API Key 발급 : https://www.pinecone.io
  - 환경변수에 추가 : 예) .env 파일
  - Pinecone 설치: pipenv install pinecone

```
from pinecone import Pinecone
import os
# Pinecone API 키 설정
pc = Pinecone(api_key=os.environ["PINECONE_API_KEY"])
```

* 가격정책 : https://www.pinecone.io/pricing/

3.3 Pinecone
3.3.2 벡터데이터 추가

- Pinecone index 생성: ServerlessSpec

```python
from pinecone import ServerlessSpec
index_lists = [item.get("name") for item in pc.list_indexes().get("indexes")]

index_name = "my-index"

if index_name not in index_lists:
    pc.create_index(
        name=index_name,
        dimension=3, # 벡터의 차원 수
        metric="cosine", # 유사도 측정 방식 (cosine, euclidean, dotproduct 중 선택)
        spec=ServerlessSpec(
            cloud="aws", # 사용할 클라우드: aws 또는 gcp
            region="us-east-1" # 리전 설정 (free tier에서 지원되는 리전을 사용해야 함)
        )
    )
```

🔷 **What "serverless" means in general**

•**Traditional (provisioned) databases/indexes**: you create and pay for a cluster of servers/VMs that are always on. You manage capacity, replicas, scaling, etc.

•**Serverless model**: you don't manage servers or clusters. Instead:

- The provider allocates compute/storage automatically.
- You pay only for what you use (queries, data stored).
- Scaling (up and down) happens behind the scenes.
- You don't deal with uptime, node provisioning, or replica counts.

This is similar to AWS Lambda or Firestore — you just declare "I want a service in region X" and call it; the backend handles capacity.

In short,
"Serverless" means *you don't rent a server; you just pay per use.*

- pc.list_indexes() returns a dictionary with the "indexes" key mapped to a **list of such index description dictionaries**.
- .get("indexes") retrieves the value
- pc.create_index()
  - Dimension: 저장할 벡터의 차원수, 삽입할 벡터의 크기와 반드시 일치해야함
  - ServerLessSpec은 파인콘의 serverless 인덱스를 생성하기 위한 설정
    - 파인콘의 무료 요금제에서는 일부 리전만 사용 가능, 예) aws - us-east-1또는 gcp - us-central1 리전
    - 만약 지원되지 않는 리전을 사용할 경우 "Bad request: Your free plan does not support..."와 같은 오류 발생

3.3 Pinecone
3.3.2 벡터데이터 추가

- 파인콘은 크로마와 마찬가지로 벡터와 메타데이터를 함께 저장 가능

```
index = pc.Index(index_name)

# 벡터 데이터 삽입
vectors = [
    [0.1, 0.2, 0.3], # 첫 번째 데이터의 벡터
    [0.4, 0.5, 0.6], # 두 번째 데이터의 벡터
    [0.7, 0.8, 0.9], # 세 번째 데이터의 벡터
]

# 벡터를 인덱스에 추가
ids = ["doc1", "doc2", "doc3"]
index.upsert([(id, vector) for id, vector in zip(ids, vectors)])
print("아이템 임베딩을 인덱스에 추가했습니다.")
```

* upsert = **update** + **insert**

https://docs.pinecone.io/reference/api/2024-07/data-plane/upsert

3.3 Pinecone
3.3.3 벡터데이터 검색

- 파인콘의 실시간 검색 기능 : 수백만 개 이상의 벡터 데이터 가 저장돼 있어도, 파인콘은 매우 빠른 속도로 유사한 벡터를 검색
- * 파인콘은 벡터 업서트 후 인덱싱이 완료되기까지 약간의 시간이 필요.업서트 직후 바로 쿼리하면 결과가 없을 수 있음

```python
import json

# 유사 벡터 검색
results = index.query(
    vector=[[0.1, 0.2, 0.3]], # 검색할 쿼리 벡터
    top_k=2, # 가장 유사한 2개의 벡터 반환
    include_metadata=False
)

print(results)

if len(results.get("matches")) == 0:
    print("검색된 결과가 없습니다.")
else:
    formatted_results = {
        "검색된 문서 ID": results["matches"][0]["id"],
        "유사도 거리": results["matches"][0]["score"]
    }
    print("\n유사한 벡터 검색 결과:")
    print(json.dumps(formatted_results, indent=4, ensure_ascii=False))
```

*** Pinecone 콘솔 접속 확인**

•https://app.pinecone.io/ 에 접속.
•로그인 후, 좌측 메뉴에서 **Indexes**를 클릭.

3.3 Pinecone
3.3.4 메타데이터 필터링

- Pinecone은 (id, vector, metadata) 순서를 기대
- Dictionary가 오류 예방

```
index.upsert([
   {"id": "doc1", "values": [0.1, 0.2, 0.3], "metadata": {"category": "A", "year": 2020}},
   {"id": "doc2", "values": [0.4, 0.5, 0.6], "metadata": {"category": "B", "year": 2021}},
])
```

- 벡터데이터와 함께 저장된 메타데이터를 기준으로 검색 범위를 좁힐 수 있음

```python
# 메타데이터와 함께 벡터 삽입
vectors = [
    ([0.1, 0.2, 0.3], {"category": "A", "year": 2020}),
    ([0.4, 0.5, 0.6], {"category": "B", "year": 2021}),
    ([0.7, 0.8, 0.9], {"category": "A", "year": 2022}),
]

ids = ["doc1", "doc2", "doc3"]
index.upsert([(id, vector, metadata) for id, (vector, metadata) in zip(ids, vectors)])

# 검색 벡터
query_vector = [0.1, 0.2, 0.25]

# 메타데이터 필터링 조건
filter_condition = {
    "category": {"$eq": "A"},
    "year": {"$gt": 2020}
}

# 검색
query_result = index.query(
    vector=query_vector,
    top_k=2,
    filter=filter_condition,
    include_metadata=True
)

formatted_results = {
    "검색된 문서 ID": query_result["matches"][0]["id"],
    "메타데이터": query_result["matches"][0]["metadata"]
}
print("\n유사한 벡터 검색 결과:")
print(json.dumps(formatted_results, indent=4, ensure_ascii=False))
```

$eq	값이 **같은** 경우
$ne	값이 **다르다**
$gt	값이 **크다**
$gte	값이 **크거나 같다**
$lt	값이 **작다**
$lte	값이 **작거나 같다**
$in	값이 **리스트에 포함**
$nin	값이 **리스트에 미포함**

```
유사한 벡터 검색 결과:
{
    "검색된 문서 ID": "doc3",
    "메타데이터": {
        "category": "A",
        "year": 2022.0
    }
}
```

3.3 Pinecone
3.3.5 임베딩 기반 라마인덱스 답변 생성

- 라마인덱스를 활용하여 임베딩을 구성하고 답변 생성
- 패키지 설치

```
# in your project folder (where Pipfile lives)
!pipenv install llama-index llama-index-vector-stores-pinecone pinecone
```

```
api_key = os.environ.get("PINECONE_API_KEY")
pc = Pinecone(api_key=api_key)
index_name = "example"
spec = {
    "serverless": {
    "cloud": "aws",
    "region": "us-east-1"
    }
}
index_lists = [item.get("name") for item in pc.list_indexes().get("indexes")]
if index_name not in index_lists:
    pc.create_index(
        name=index_name,
        dimension=3,
        metric="cosine",
        spec=spec
    )
index = pc.Index(index_name)

# LlamaIndex에서 사용할 HuggingFace 임베딩 모델 설정
embed_model = HuggingFaceEmbedding(model_name="all-MiniLM-L6-v2")
```

3.3 Pinecone
3.3.5 임베딩 기반 라마인덱스 답변 생성

```python
# 문서 데이터
documents = [
    "고양이는 작은 육식동물로, 주로 애완동물로 기릅니다. 민첩하고 장난기 있는 행동으로 유명합니다.",
    "강아지는 충성심이 강하고 친절한 동물로, 흔히 인간의 최고의 친구로 불립니다. 주로 애완동물로 기르고, 동반자로서 유명합니다.",
    "고양이와 강아지는 전 세계적으로 인기 있는 애완동물로, 각각 독특한 특징을 가지고 있습니다."
]
ids = ["doc1", "doc2", "doc3"]

# 문서를 Document 형식으로 변환
nodes = [Document(text=doc, id_=doc_id) for doc, doc_id in zip(documents, ids)]

# Pinecone 벡터 스토어 생성
vector_store = PineconeVectorStore(pinecone_index=index)

# LlamaIndex의 VectorStoreIndex 생성
index = VectorStoreIndex.from_documents(nodes, vector_store=vector_store, embed_model= embed_model)
```

VS

```python
documents = [
    "고양이는 작은 육식동물로, 주로 애완동물로 기릅니다. 민첩하고 장난기 있는 행동으로 유명합니다.",
    "강아지는 충성심이 강하고 친절한 동물로, 흔히 인간의 최고의 친구로 불립니다. 주로 애완동물로 기르고, 동반자로서 유명합니다.",
    "고양이와 강아지는 전 세계적으로 인기 있는 애완동물로, 각각 독특한 특징을 가지고 있습니다."
]
ids = ["doc1", "doc2", "doc3"]

# 문서를 LlamaIndex의 Document 형식으로 변환
nodes = [Document(text=doc, id_=doc_id) for doc, doc_id in zip(documents, ids)]

# Chroma 벡터 스토어 생성
vector_store = ChromaVectorStore(chroma_collection=collection)

# LlamaIndex의 VectorStoreIndex 생성
index = VectorStoreIndex.from_documents(nodes, vector_store=vector_store)

# 쿼리 엔진 생성 (기본적인 검색 + 답변 생성 기능 활성화)
query_engine = index.as_query_engine()
```

* VectorStoreIndex.from_documents() upserts your documents into the Pinecone repository

```python
query_engine = index.as_query_engine()

# 질의 수행
query_text = "고양이에 대해 알려줘"
response = query_engine.query(query_text)

# 결과 출력
print("\n[질의 결과]")
print(response)
```

3.3 Pinecone
3.3.6 라마인덱스 기반 답변 생성 (임베딩 생략)

```python
from pinecone import Pinecone
from llama_index.core.schema import Document
from llama_index.vector_stores.pinecone import PineconeVectorStore
from llama_index.core import VectorStoreIndex
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core.schema import Document
import os

api_key = os.environ.get("PINECONE_API_KEY")
pc = Pinecone(api_key=api_key)
index_name = "example"
spec = {
    "serverless": {
    "cloud": "aws",
    "region": "us-east-1"
    }
}
index_lists = [item.get("name") for item in pc.list_indexes().get("indexes")]
if index_name not in index_lists:
    pc.create_index(
        name=index_name,
        dimension=3,
        metric="cosine",
        spec=spec
    )
pc_index = pc.Index(index_name)

vector_store = PineconeVectorStore(pinecone_index=pc_index)

# 문서 데이터
documents = [
    "고양이는 작은 육식동물로, 주로 애완동물로 기릅니다. 민첩하고 장난기 있는 행동으로 유명합니다.",
    "강아지는 충성심이 강하고 친절한 동물로, 흔히 인간의 최고의 친구로 불립니다. 주로 애완동물로 기르고, 동반자로서 유명합니다.",
    "고양이와 강아지는 전 세계적으로 인기 있는 애완동물로, 각각 독특한 특징을 가지고 있습니다."
]
```

```python
ids = ["doc1", "doc2", "doc3"]

# 문서를 Document 형식으로 변환
nodes = [Document(text=doc, id_=doc_id) for doc, doc_id in zip(documents, ids)]

# LlamaIndex의 VectorStoreIndex 생성
index = VectorStoreIndex.from_documents(nodes, vector_store=vector_store)

# 쿼리 엔진 생성
query_engine = index.as_query_engine()

# 질의 수행
query_text = "고양이에 대해 알려줘"
response = query_engine.query(query_text)

# 최종 응답 출력
print("[질의 결과]")
print(response)

# 응답 생성에 사용된 문서 확인
print("\n[응답에 사용된 문서]")
for i, node in enumerate(response.source_nodes, 1):
    print(f"{i}. {node.text}\n")
```

- Can you find an error?

3.3 Pinecone
3.3.6 라마인덱스 기반 답변 생성 (임베딩 생략)

```python
from pinecone import Pinecone
from llama_index.core.schema import Document
from llama_index.vector_stores.pinecone import PineconeVectorStore
from llama_index.core import VectorStoreIndex
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core.schema import Document
import os

api_key = os.environ.get("PINECONE_API_KEY")
pc = Pinecone(api_key=api_key)
index_name = "example"
spec = {
    "serverless": {
    "cloud": "aws",
    "region": "us-east-1"
    }
}
index_lists = [item.get("name") for item in pc.list_indexes().get("indexes")]
if index_name not in index_lists:
    pc.create_index(
        name=index_name,
        dimension=3,
        metric="cosine",
        spec=spec
    )
pc_index = pc.Index(index_name)

vector_store = PineconeVectorStore(pinecone_index=pc_index)

# 문서 데이터
documents = [
    "고양이는 작은 육식동물로, 주로 애완동물로 기릅니다. 민첩하고 장난기 있는 행동으로 유명합니다.",
    "강아지는 충성심이 강하고 친절한 동물로, 흔히 인간의 최고의 친구로 불립니다. 주로 애완동물로 기르고, 동반자로서 유명합니다.",
    "고양이와 강아지는 전 세계적으로 인기 있는 애완동물로, 각각 독특한 특징을 가지고 있습니다."
]
```

```python
ids = ["doc1", "doc2", "doc3"]

# 문서를 Document 형식으로 변환
nodes = [Document(text=doc, id_=doc_id) for doc, doc_id in zip(documents, ids)]

# LlamaIndex의 VectorStoreIndex 생성
index = VectorStoreIndex.from_documents(nodes, vector_store=vector_store)

# 쿼리 엔진 생성
query_engine = index.as_query_engine()

# 질의 수행
query_text = "고양이에 대해 알려줘"
response = query_engine.query(query_text)

# 최종 응답 출력
print("[질의 결과]")
print(response)

# 응답 생성에 사용된 문서 확인
print("\n[응답에 사용된 문서]")
for i, node in enumerate(response.source_nodes, 1):
    print(f"{i}. {node.text}\n")
```

```
from llama_index.core import Settings
print(Settings.embed_model.model_name)
vec = Settings.embed_model.get_text_embedding("probe text")
print("Embedding dim:", len(vec))
```

text-embedding-ada-002
Embedding dim: 1536

3.3 Pinecone
3.3.6 라마인덱스 기반 답변 생성 (임베딩 생략)

```python
# choose ONE embedding model and use it everywhere
Settings.embed_model = HuggingFaceEmbedding(
    model_name="sentence-transformers/all-MiniLM-L6-v2"
)

# compute its vector length to drive Pinecone's dimension
_embed_dim = len(Settings.embed_model.get_text_embedding("probe"))

index_lists = [item.get("name") for item in pc.list_indexes().get("indexes")]
if index_name not in index_lists:
    pc.create_index(
        name=index_name,
        dimension=_embed_dim,
        metric="cosine",
        spec=spec
    )
else:
    # if it exists, ensure dimension matches; if not, recreate
    pc_index_tmp = pc.Index(index_name)
    stats = pc_index_tmp.describe_index_stats()
    if stats["dimension"] != _embed_dim:
        pc.delete_index(index_name)
        pc.create_index(
            name=index_name,
            dimension=_embed_dim,
            metric="cosine",
            spec=spec
        )

pc_index = pc.Index(index_name)

# (optional but recommended) fail fast if an existing index has wrong dim
stats = pc_index.describe_index_stats()
assert stats["dimension"] == _embed_dim, \
    f"Pinecone dim {stats['dimension']} != embed dim {_embed_dim}. " \
    "Delete and recreate the index with the correct dimension."

vector_store = PineconeVectorStore(pinecone_index=pc_index)
```

- **“index”** is a general term in information retrieval and databases. It means a data structure (and sometimes the container around it) that makes lookups fast by trading extra storage/maintenance for speed

Aspect	Pinecone **index**	LlamaIndex **Index** (e.g., `VectorStoreIndex`)
What it is	A **hosted vector database container** that persists vectors + metadata and serves similarity queries	An **application-level retrieval structure** that orchestrates chunking → embedding → retrieval → LLM synthesis
Level	**Storage/serving layer** (infrastructure)	**Orchestration/logic layer** (your app)