Department of Computer Science
University of Cyprus

EPL646 – Advanced Topics in Databases

Lecture 5b

Vector Databases

Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl646

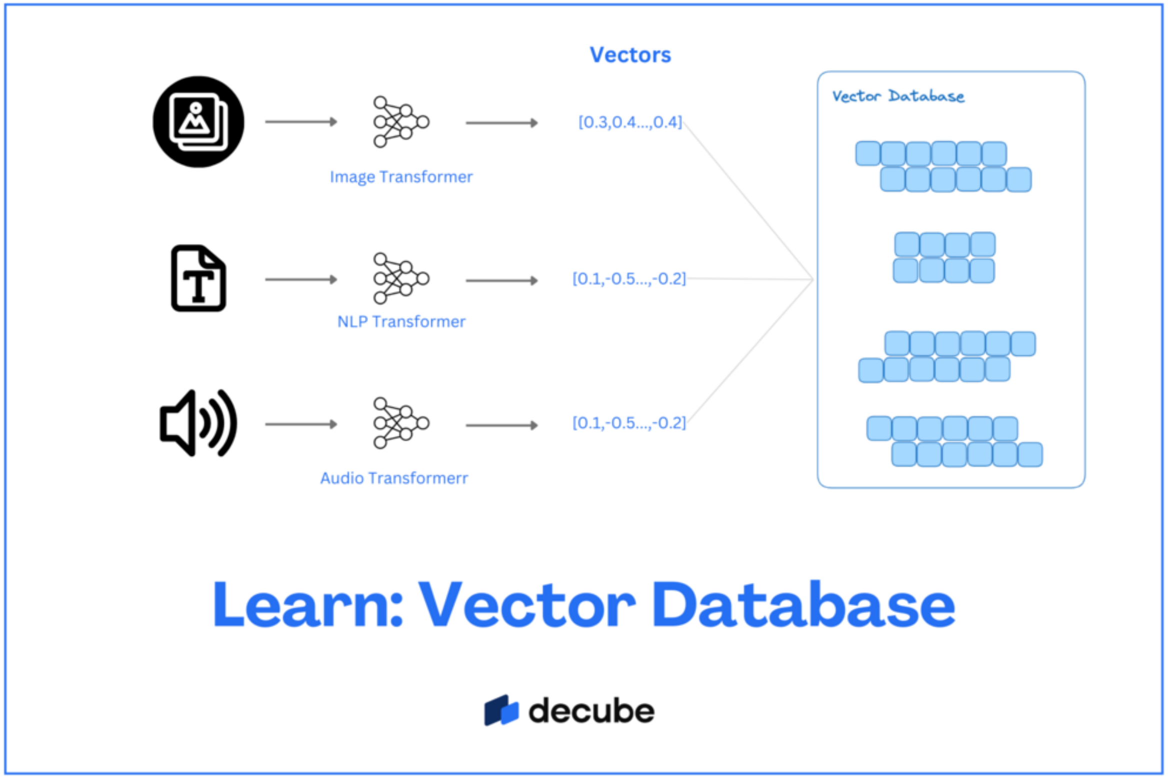
Credits: https://mlops.community/vector-similarity-search-from-basics-to-production/
ChatGPT

Overview

- Vector Databases Concepts
  - Embeddings (Text vs Sentence)
  - Similarity Search & Approximate Similarity Search (Lp-Norms), Libraries
- Chroma DB
  - Internals (Main-Memory vs. Persistency with DuckDB)
  - Storage: Parquet (lecture 3) | DuckDB
  - Indexing: Hierarchical navigable small world (HNSW)
  - Other Vector Databases Products

Vector Databases

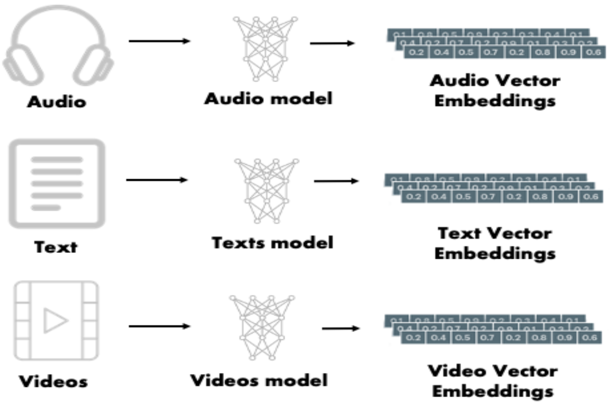
- A **vector database** is a specialized type of database designed to **store, index, and search high-dimensional vector embeddings efficiently**.



Some background first, then we come back to vector databases!

Vector Embeddings

- **Numerical representations of data (e.g., text, images, audio, video) generated using machine learning models like word2vec, OpenAI's CLIP, or BERT.**
  - Simply put, vector embeddings are lists of numbers that can represent many types of data.
  - Instead of traditional structured data (like rows and columns in a relational database), vector databases manage numerical representations of data points in a multi-dimensional space.



Text vs. Word Embeddings

- **Word Embedding:** Creates **separate vector for each word.** It does **not** consider the entire sentence or its context beyond neighboring words. (e.g., word2vec)
- **Sentence Embedding:** Captures **context, syntax, and overall meaning of entire sentences, paragraphs, or documents** as dense vectors. (ChatGPT text-embedding-ada-00)

```
import openai

def get_embedding(text, model="text-embedding-ada-002"):
  response = openai.Embedding.create(
    input=text,
    model=model
  )
  return response['data'][0]['embedding']

# Example usage
text = "Hello, this is an example of text embedding."
embedding = get_embedding(text)
print("Embedding vector:", embedding[:5], "...")  # Print first 5 elements for brevity
```

Using OpenAI API ☹
Not open! Let’s see some alternatives

Example: Word Embeedings

- pip install -U genism
- pip3 install gensim // mac Mx silicon

```
from gensim.models import Word2Vec

# Example training data
sentences = [["hello", "world"], ["machine", "learning", "is", "fun"]]

# Train Word2Vec model
model = Word2Vec(sentences, vector_size=100, min_count=1)

# Get word embedding for "hello"
embedding = model.wv["hello"]

print("Word Embedding for 'hello':", embedding[:5])  # First 5 values
```

(Open) Sentence Embeedings

SentenceTransformers (SBERT)
📌 **Best for:** General-purpose text embeddings with high performance.
🔥 **Why?** Efficient, high-quality embeddings with transformer-based models (e.g., all-MiniLM-L6-v2).
📦 Install: pip install sentence-transformers

Hugging Face Transformers
📌 **Best for:** Custom embeddings using any transformer model.
🔥 **Why?** Supports a wide range of models like BERT, RoBERTa, and GPT.
📦 Install: pip install transformers torch

Our Focus for the next slides

FastText (Facebook)
📌 **Best for:** Word-level embeddings, especially for low-resource languages.
🔥 **Why?** Works well with subword information and OOV (out-of-vocabulary) words.
📦 Install: pip install fasttext

Gensim (Word2Vec, Doc2Vec)
📌 **Best for:** Classic word and document embeddings.
🔥 **Why?** Lightweight and easy to use for traditional NLP tasks.
📦 Install: pip install genism

- **Which One Should You Choose?**
  - **For general text embeddings:** ✅ SentenceTransformers (SBERT)
  - **For transformer-based models:** ✅ Hugging Face Transformers
  - **For word embeddings:** ✅ FastText or Word2Vec
  - **For unsupervised large-scale embeddings:** ✅ FastText

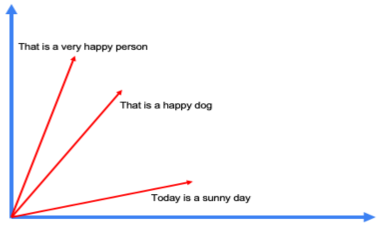
Example: Sentence Embeeding with Sentence-Transformers

- pip install sentence-transformers

```
# Import the necessary library
from sentence_transformers import SentenceTransformer


# Initialize the model
model = SentenceTransformer('paraphrase-MiniLM-L6-v2')


# Example sentence
sentence = "This is an example sentence for embedding."


# Generate the embedding
embedding = model.encode(sentence)


# Print the embedding
print(embedding)
```

Semantic Similarity Search

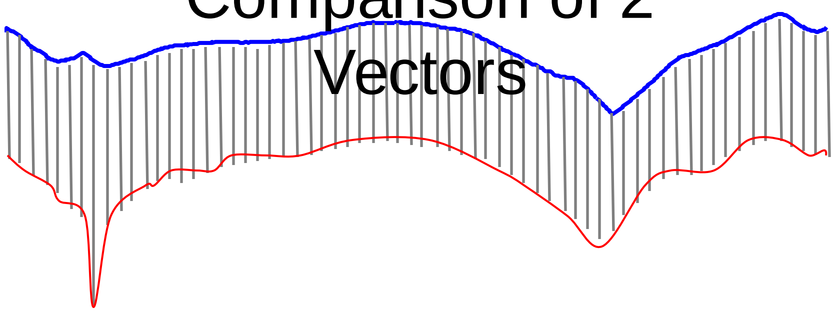
- **Semantic Similarity Search** is the process by which pieces of text are compared in order to find which contain the most similar meaning.
- Example:
  - “That is a happy dog”
  - “That is a very happy person”
  - “Today is a sunny day”

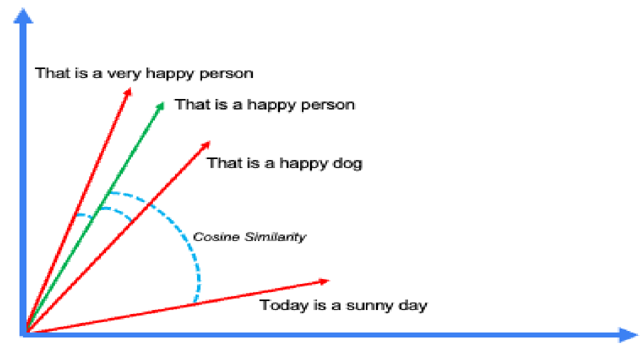


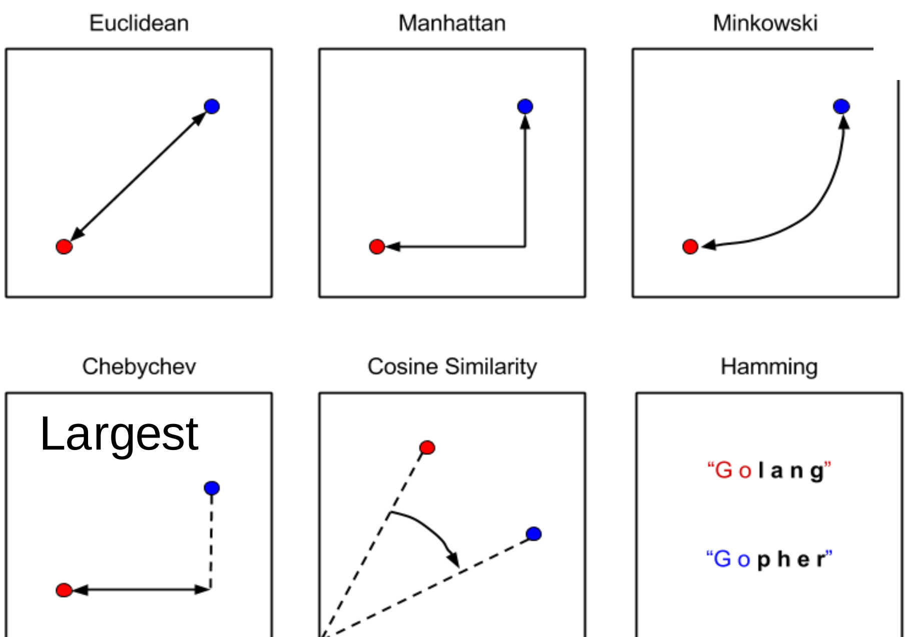
- **You guessed it, our aim is to compare the vectors not the string sentences!**

Vector Comparison Underpinnings!

Lp-norm
Comparison of 2
Vectors







$$\cos(\mathbf{t},\mathbf{e}) = \frac{\mathbf{t}\mathbf{e}}{\|\mathbf{t}\|\|\mathbf{e}\|} = \frac{\sum_{i=1}^{n} \mathbf{t}_i \mathbf{e}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{t}_i)^2}\sqrt{\sum_{i=1}^{n} (\mathbf{e}_i)^2}}$$

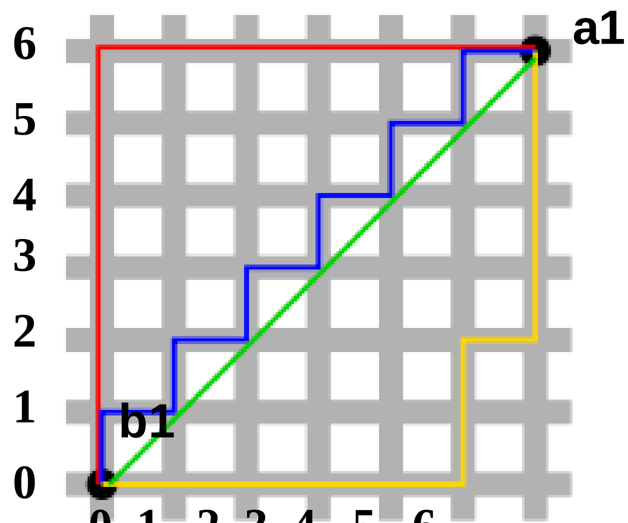
```
def cosine_similarity(a, b):
   return np.dot(a, b)/
          (norm(a)*norm(b))
```

Euclidean vs. Manhattan Distance

- ***Euclidean vs. Manhattan distance:***

  *- Euclidean Distance (using Pythagoras theorem) is 6 x √2 = **8.48 points**): Diagonal **Green** line*

  *- Manhattan (city-block) Distance (**12 points**): Red, Blue, and Yellow lines*



2-Dimensional Scenario

Comparing 2 Vectors with Cosine Similarity

```
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity

# Initialize the model
model = SentenceTransformer('paraphrase-MiniLM-L6-v2')

# Example sentences
sentence1 = "This is a sentence about machine learning."
sentence2 = "Machine learning is a fascinating topic."

# Generate embeddings for both sentences
embedding1 = model.encode(sentence1)
embedding2 = model.encode(sentence2)

# Compute cosine similarity between the two embeddings
similarity_score = cosine_similarity([embedding1], [embedding2])

# Print the similarity score
print(f"Cosine Similarity: {similarity_score[0][0]}")
```

$\cos 0^\circ$ = 1 // Perfect Match
$\cos 90$ = 0 // No Match

Cosine similarity is a metric used to measure how similar two vectors (or documents, in the context of text) are, based on the cosine of the angle between them. It is widely used in information retrieval, text mining, and machine learning to compare the similarity between two objects.

Problem: If I have a database with N objects comparing all vectors requires N comparisons which is slow! We need some DB/index to speed up the computation

Chroma: A Simple Vector Database

- **Example of How It Works Internally**

1. Embeddings and metadata are (tentatively) stored in DuckDB (can be main memory too)
2. Chroma builds an **Hierarchical navigable small world (HNSW)** index to speed up vector similarity search.
   – HNSW does the so called Approximate Nearest Neighbor Search (will see this next)
3. DuckDB handles filtering on metadata.
4. Chroma queries the HNSW index for nearest neighbors and refines results using metadata filters.

Example: Chroma Hello World (with persistency)

- By default, Chroma uses **DuckDB** as its embedded database for efficient data storage and retrieval, but you can configure it to use alternative backends.
  - **DuckDB – An in-process SQL OLAP database management** ... a column-oriented sqlite that supports parquet

```
import chromadb

# Create a persistent ChromaDB instance using DuckDB as storage
chroma_client = chromadb.PersistentClient(path="./chroma_db")

# Create a collection (automatically stored in DuckDB)
collection = chroma_client.get_or_create_collection(name="my_collection")

# Add some example data
collection.add(
    ids=["id1"],
    embeddings=[[0.1, 0.2, 0.3]],
    metadatas=[{"category": "example"}]
)

# Query the collection
results = collection.query(
    query_embeddings=[[0.1, 0.2, 0.3]],
    n_results=1
)

print(results)
```

DuckDB

Chroma

Chroma / In-Memory vs DuckDB

- Chroma stores its data in different ways depending on the **storage mode** you choose:
- **1. In-Memory Mode (Default)**
  - If you initialize Chroma without specifying persistence, it keeps everything **in RAM**.
  - **Data is lost** when the process is stopped.
  - Example:

```
import chromadb
chroma_client = chromadb.Client() # In-memory storage
```

- 2. **Persistent Mode (Using DuckDB)**
  - Chroma **stores data on disk** using **DuckDB** as the underlying database.
  - Metadata and vector embeddings are saved in a **DuckDB file** at the specified path.
  - **chroma_client = chromadb.PersistentClient(path="./chroma_db") # Stores data in ./chroma_db**

DuckDB: **Columnar Embedded OLAP Database**

- DuckDB is a columnar database, making it highly efficient for running analytical queries.
- It is embeeded (like SQLlite), so doesn't run a a service but part of the caller memory space
- It supports two main storage formats: its native ***.duckdb*** format or open-standard file formats like Parquet, which DuckDB reads and writes with impressive efficiency

```
import duckdb
import pandas as pd
import polars as pl

pd_df = pd.DataFrame({ "name": ["tea", "coffee", "apple","orange"],
                       "category_id": ["1", "1", "2", "2" ] })
pl_df = pl.DataFrame({ "id": ["1", "2", "3"],
                       "category": ["food", "fruit", "drink"] })
duckdb.sql("SELECT * FROM pd_df INNER JOIN pl_df ON pd_df.category_id=pl_df.id WHERE category = 'fruit'")
```

name varchar	category_id varchar	id varchar	category varchar
apple	2	2	fruit
orange	2	2	fruit

https://www.pracdata.io/p/duckdb-beyond-the-hype

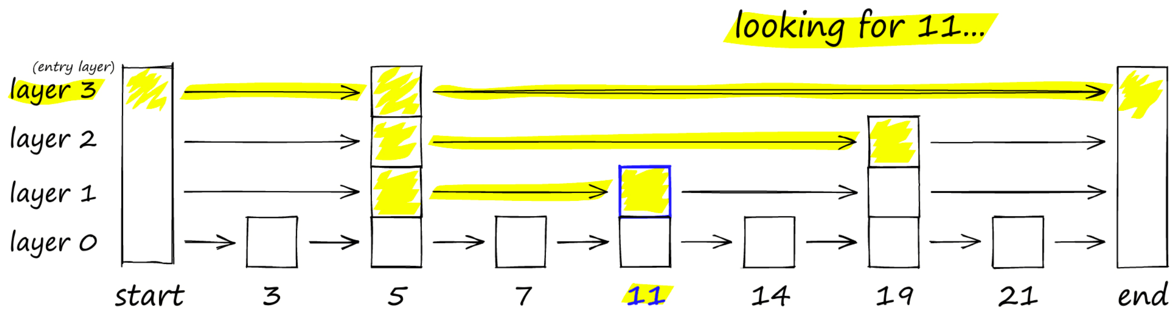
https://duckdb.org/pdf/SIGMOD2019-demo-duckdb.pdf

Chroma Similarity Search Example / DuckDB+Parquet

```
import chromadb
from chromadb.config import Settings

# Step 1: Initialize Chroma client with a local storage path
client = chromadb.Client(Settings(chroma_db_impl="duckdb+parquet", persist_directory="./chroma_db"))

# Step 2: Create or access an existing collection
collection = client.create_collection("example_collection")  # You can also use .get_collection() if it already exists

# Step 3: Define your data (for this example, we'll use some simple sentences)
texts = [
    "Chroma is an open-source vector database.",
    "Chroma supports fast similarity search for machine learning applications.",
    "You can store vectors and perform searches using Chroma."
]

# Step 4: Insert the data into Chroma
# You can use embeddings for real-world cases, here we'll just insert the raw data for simplicity
# Normally, you'd embed the text before inserting, but for the sake of the example, we'll skip that.
metadata = [{"text": text} for text in texts]

# Insert into Chroma collection
collection.add(
    documents=texts,  # The raw text data
    metadatas=metadata, # Optional metadata
    ids=[str(i) for i in range(len(texts))]  # Unique IDs for each document
)

# Step 5: Verify the insertion by querying the collection
results = collection.query(query_texts=["What is Chroma?"], n_results=3)

print("Query Results:")
for result in results["documents"]:
    print(result)
```

Hierarchical navigable small world (HNSW)

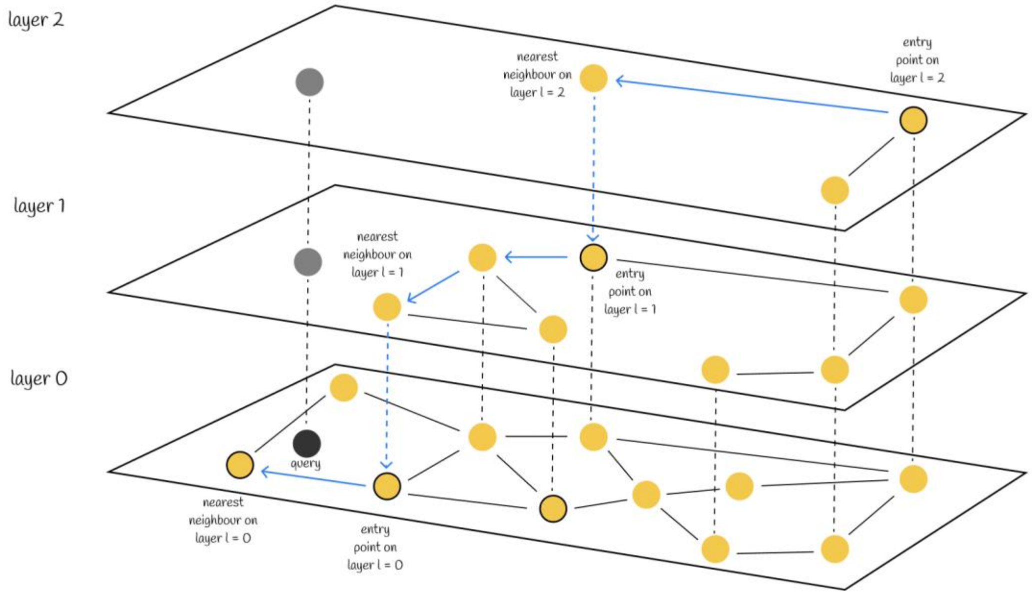
- Approximate Nearest Neighbor Search
  - Instead of exact matches, vector databases use **Approximate Nearest Neighbor (ANN) search** to find vectors that are most similar to a given query.
- HNSW resembles skip list that have a O(logn) search



- HNSW howewer is a graph search method with polylogarithmic $T = O(\log^k n)$ search complexity which uses greedy routing.

HNSW Query Routing Principles

- The search starts from the highest layer and proceeds to one level below every time the local nearest neighbour is greedily found among the layer nodes. Ultimately, the found nearest neighbour on the lowest layer is the answer to the query.



Libraries Implementing HNSW

- **FAISS (Facebook AI Similarity Search) - Facebook**
  - Language: C++, Python
  - Features: Optimized for both CPU and GPU-based searches, scalable for large datasets.

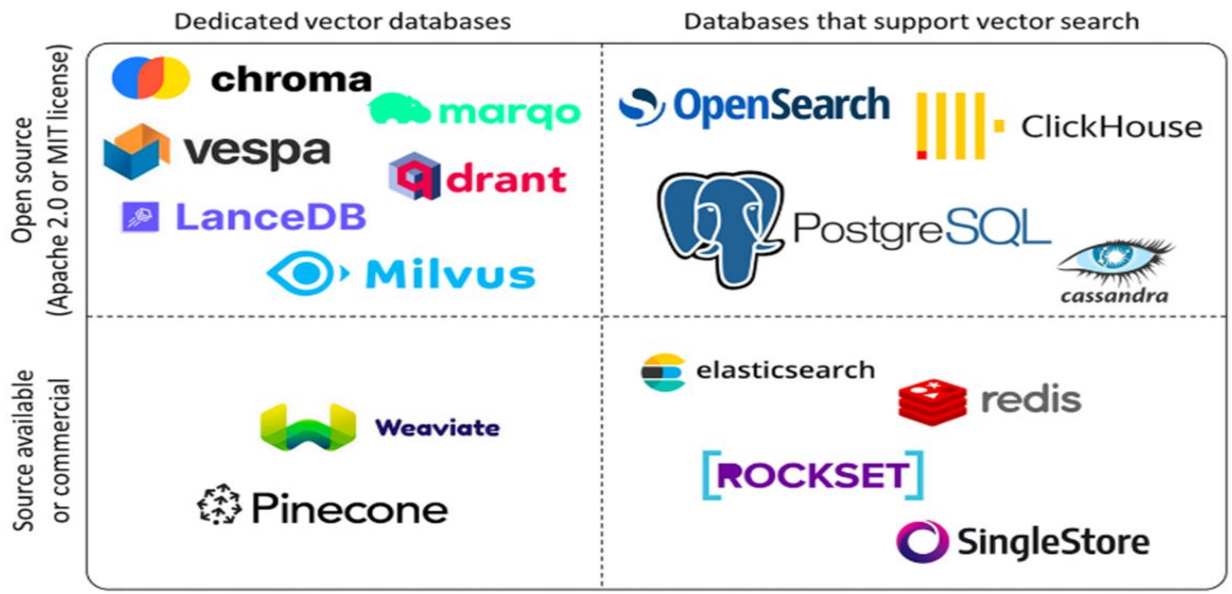
- **ScaNN (Scalable Nearest Neighbors) – Google / AlloyDB**
  - Language: Python
  - Can handle large-scale data with high efficiency, supports multiple distance metrics.

- **Hnswlib**
  - Language: Python, C++
  - simplicity and speed.

- **Annoy (Approximate Nearest Neighbors Oh Yeah)**
  - extensively in recommendation systems.
  - It supports HNSW for indexing and is designed for large-scale applications.
  - extensively in recommendation systems.

Vector Databases in the Wild

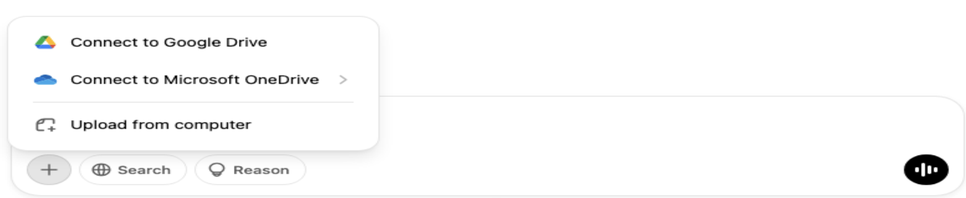
- HNSW Implementing by many vendors: Milvus, Pinecone, Weaviate, Qdrant, Vespa, Chroma .. even postgres with pgvector !



https://github.com/pgvector/pgvector

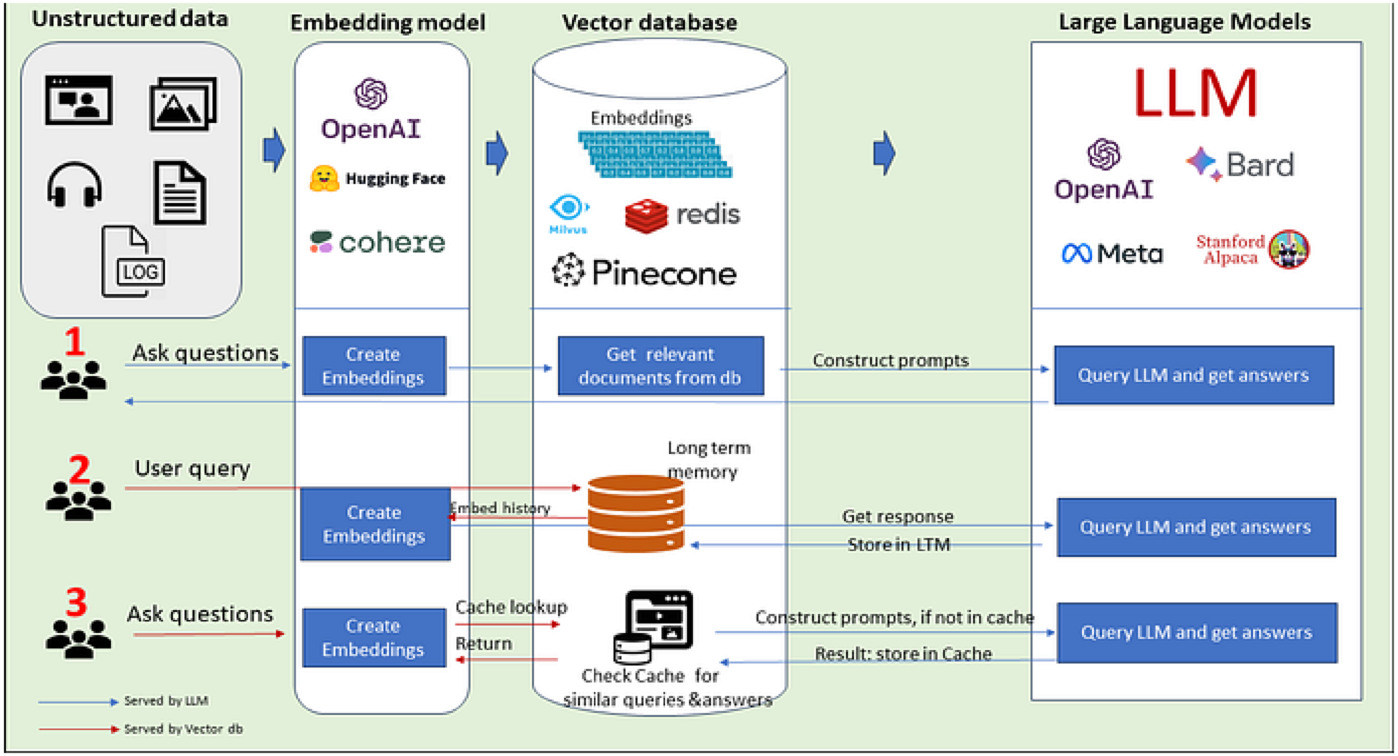
LLMs, RAG and Vector Databases

- **LLMs (Large Language Models)** are advanced AI models trained on vast amounts of text data to understand and generate human-like language.
  - These models use **deep learning**, specifically **transformers** (like GPT, BERT, and LLaMA), to process and generate text based on input prompts.
- LLMs require re-training to incorporate new content. This is expensive.
- ChatGPT and other systems allow uploading a variety of files that undergo **Retrieval-Augmented Generation (RAG).**



LLMs, RAG and Vector Databases

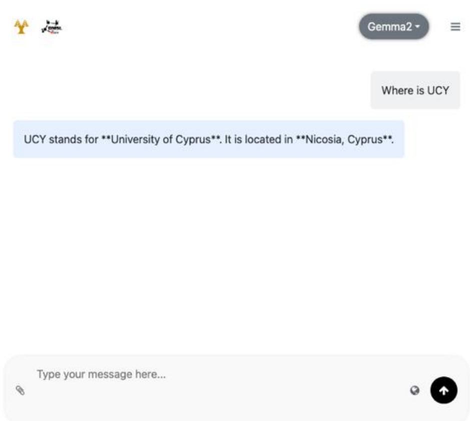
- **Retrieval** – The model searches a knowledge base (e.g., documents, databases, or vector stores) for relevant information. **Augmentation** – The retrieved data is added to the model's input context. **Generation** – A language model (like GPT) generates a response using both the external data and its internal knowledge.



https://medium.com/@akriti.upadhyay/unveiling-the-dynamics-of-vector-databases-in-ai-landscape-0784ea0f2f50

Open Source LLMs

- **Ollama is an open-source framework** that enables users to run and **interact** with **large language models** (LLMs) **locally on their machines**.
  - It simplifies **downloading**, **managing**, and **running** AI models without requiring cloud services.
  - Example Models: Mistral (French), Llama (Facebook), Falcon (UAE), Qwen (Alibaba), Palm (Google), Grok (xAI), Deekseek (Chinese)
  - Integration with AI Marketplaces: Huggingface



https://medium.com/@yugank.aman/top-10-open-source-llm-models-and-their-uses-6f4a9aced6af

https://chatucy.cs.ucy.ac.cy/

Hugging Face