

CS 7270: Advanced Database Systems Fall 2025

Lecture 24

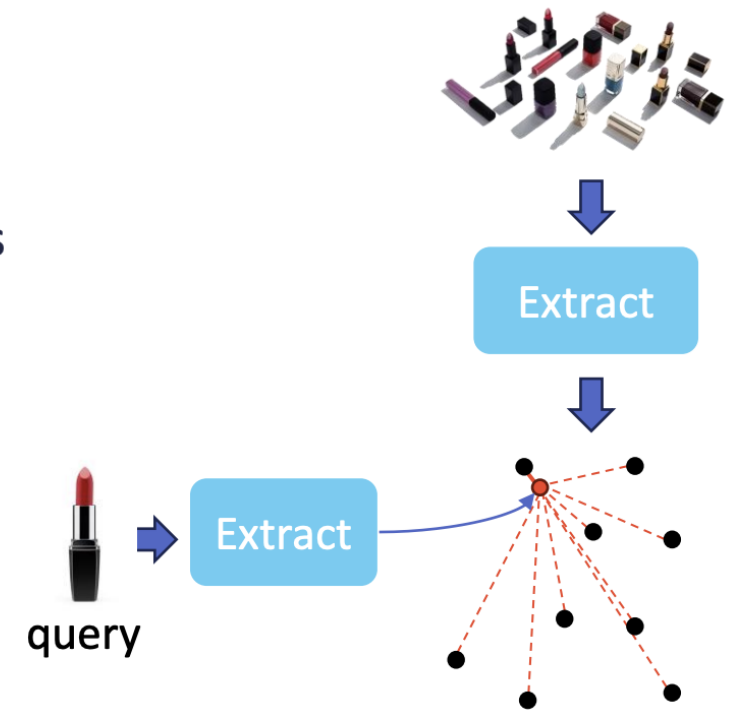
Vector Databases

Prashant Pandey

p.pandey@northeastern.edu

Visual product search

- Take photo → find matching product
- General idea:
 - Extract feature vectors (**embeddings**) from product images
 - Store in some DB
 - At runtime: extract image features
 - ... then find nearest neighbours
- Example: JD.com [Li, Middleware'18]
 - **100B** products, **1B** daily updates
 - Requirement: support fast update
 - Requirement: query fresh data

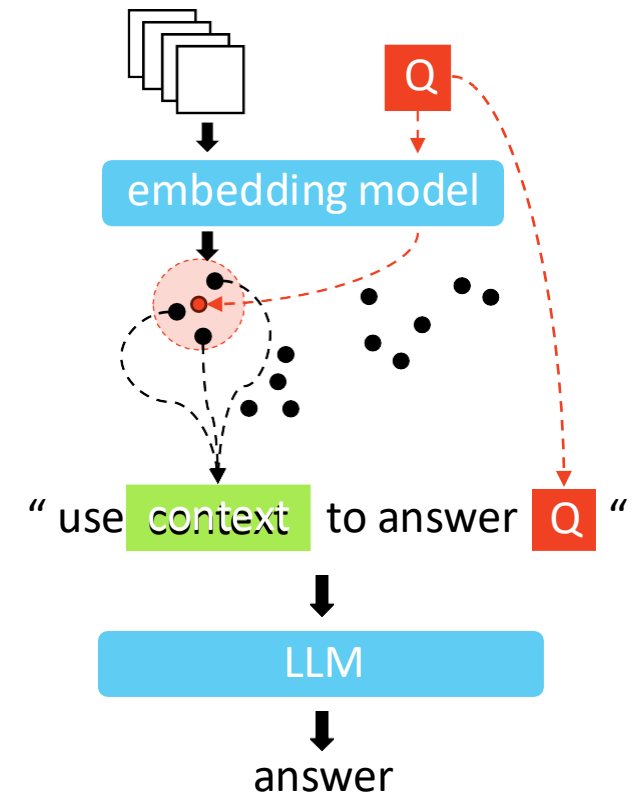


Problems

- $N = 100B = 100,000,000,000$ vectors
- Each vector is large: $D = \sim 1,000$ floats
- **Problem 1:** storing N vectors for fast access: **400 terabytes**
 - Too much for RAM
- **Problem 2:** finding nearest neighbour:
 - Distance to N vectors = $O(ND)$ multiply-adds $\rightarrow N * D = 100T$
 - Even at 20 TFLOPS, **5 second latency** per query (ignoring other costs)
- “Put it in a database and index?”
 - Index what? DB indices designed for individual attributes, not ANN search on vectors
 - Not clear how to shard vectors

RAG question answering

- Get question, use LLM to generate answer
- Retrieval-Augmented-Generation (RAG) adds context:
 1. Generate document embeddings Storing vectors
 2. To query: generate embedding of user query
 3. Get relevant documents Neighbor search + rerank
 4. Create prompt to incorporate documents as context
 5. Use LLM to generate response



Recommender systems

Given user, recommend products/videos/music:

1. Create vectors for products
2. Create vector q for user preference
 - From purchases, searches, user profile
3. Find products “close to” q
4. Re-rank based on recent activity



Storing vectors



Neighbor search

Requirements:

- **High throughput**
- **Good accuracy**
- **Elasticity** due daily demand fluctuations

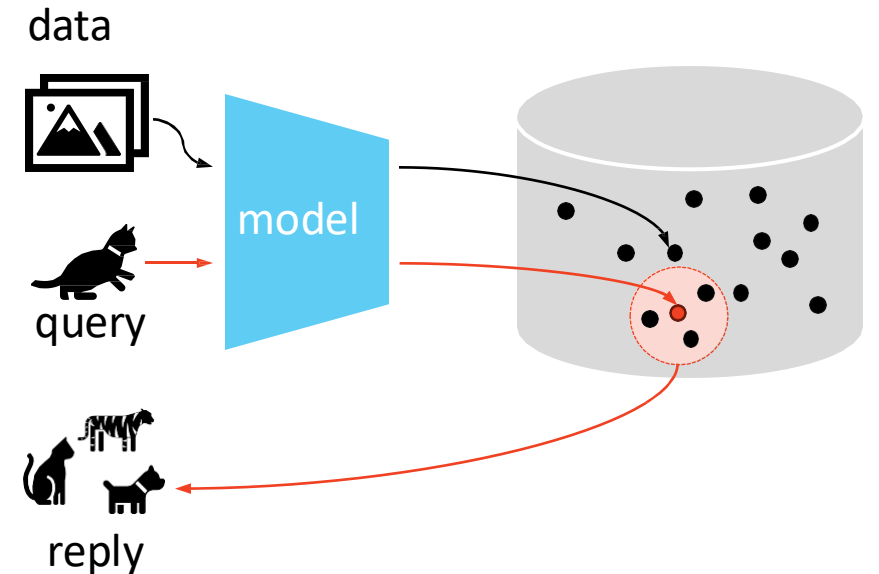
A common approach

Inserting data:

1. Convert data (text, image, sound, graph) into an *embedding vector*
 - Usually using an ML model
2. Store vectors (embeddings) in specialized DB

Querying Data:

3. Embed query as vector q
4. Ask DB to find vectors similar to q



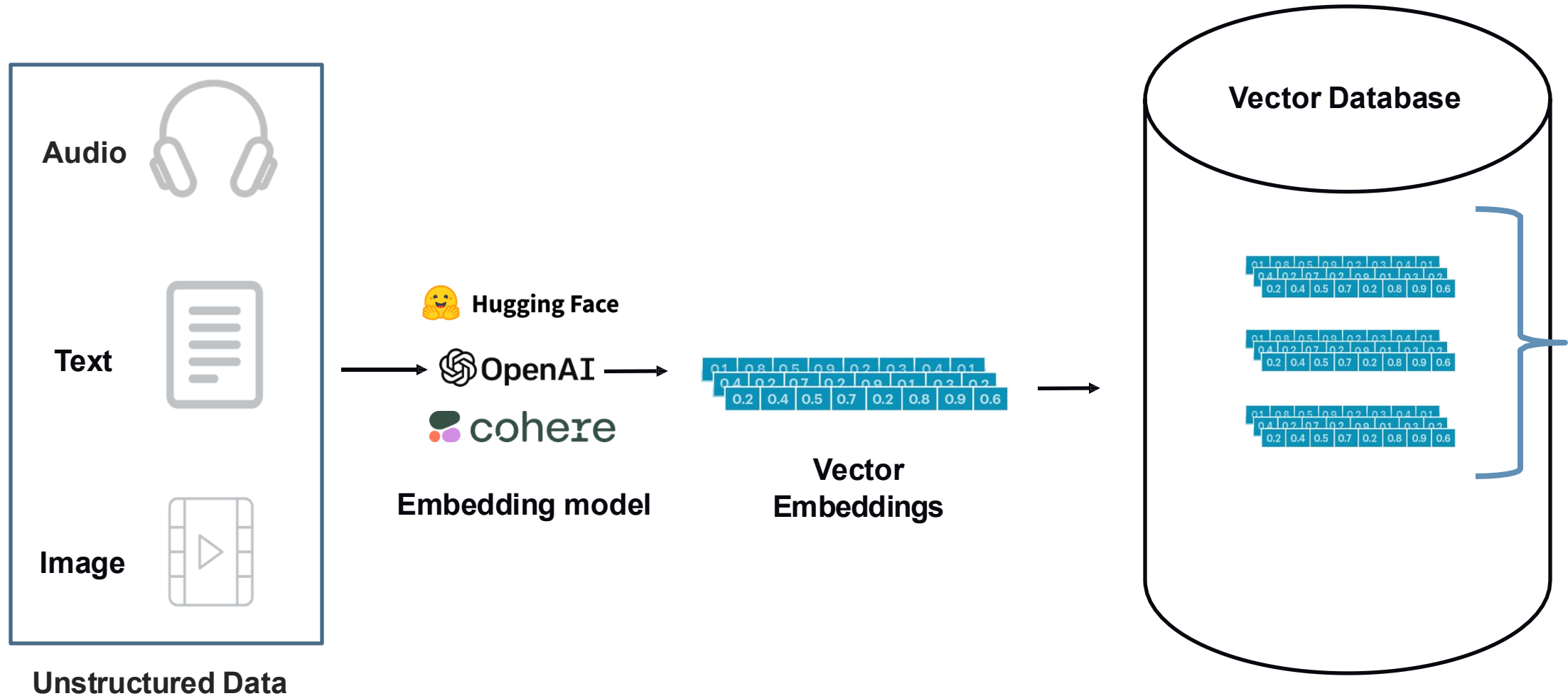
Visual product search

- Take photo → find matching product
- General idea:
 - Extract feature vectors (**embeddings**) from product images
 - Store in some **spatially-indexed** DB
 - At runtime: extract image features
 - ... then find nearest neighbours
- **Index** to accelerate search:
 - First cluster vectors, store in per-cluster list
 - Query = search only list of nearest cluster
 - Inserts = add to list of nearest cluster

Vector databases

- Specialized databases designed to store, index, and retrieve high-dimensional vectors efficiently
- Particularly useful for tasks like similarity search, recommendation systems, and AI model outputs

Vector databases



Metric-space vector databases

- These databases use distance metrics (e.g., Euclidean, cosine similarity) to organize and search vectors
- **Examples:**
 - Milvus
 - Weaviate
 - Pinecone

Graph-based vector databases

- Utilize graph structures (e.g., k-NN graphs, HNSW) for efficient similarity search
- These are well-suited for large-scale datasets where approximate nearest neighbor (ANN) searches are common
- **Examples:**
 - Elasticsearch (with ANN plugins)
 - Vespa
 - HNSWlib-based databases
 - DiskANN

Hash-based vector databases

- Use hashing techniques like Locality-Sensitive Hashing (LSH) for fast approximate searches
- Suitable for sparse or low-dimensional datasets.
- **Examples:**
 - FAISS (Flat and Hash-based indexing options)
 - Annoy (Approximate Nearest Neighbors)

Hybrid vector databases

- Combine vector indexing with traditional relational or document-based databases
- Ideal for applications needing structured data along with unstructured vector queries
- **Examples:**
 - Redis with vector similarity search
 - PostgreSQL with vector search extensions (e.g., pgvector)
 - MongoDB Atlas Search (supports vector fields)

Cloud-native vector databases

- Fully managed, scalable vector databases optimized for cloud platforms
- Simplify setup, scaling, and maintenance
- **Examples:**
 - Amazon Kendra
 - Google Vertex AI Matching Engine
 - Azure Cognitive Search

Specialized vector databases

- Tailored for specific use cases, such as video search, genomics, or geospatial data
- May incorporate domain-specific optimizations
- **Examples:**
 - Zilliz (AI and ML-focused)
 - Deep Lake (designed for AI datasets)
 - Mantis/Metagraph for genomics

k-NN search

- Find k nearest neighbours (kNN)

- Denote:

S = stored vectors

q = query

R = result set

- Find set of k vectors in S closest to q
- Generally: returned vectors sorted by distance from q
- Formally:

return $R = \{r_i\} \subseteq S$ such that $\forall v \in R: d(q, v) \leq \min_{x \in S \setminus R} d(q, x)$ and

$|R| = k$ and

$d(r_i, q) < d(r_{i+1}, q)$

In pgvector

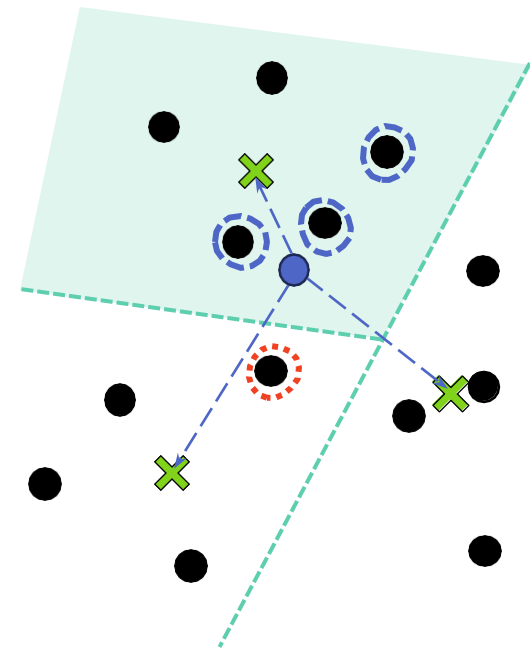
```
SELECT * FROM items
ORDER BY vec <-> '[1,6.4,-2.1]'
LIMIT 5;
```

Exact k-NN search is slow!

- Exhaustive search is **very slow**:
 - Compute $d(q, v)$ for all v , then sort / use priority queue / top-k
 - Supports exact kNN, range queries, predicated queries, everything
 - Time: $O(ND)$ + top k time
- Solution: **approximate nearest neighbour search (ANNS)**
 - Return k vectors, not guaranteed to be nearest
- Key enabler: **ANNS index**
 - Quick but potentially inaccurate queries
 - Slower inserts, potential memory, storage costs

Searching with ANNS index

- Given q , quickly find potential neighbours
- The index deal:
 - Faster search 😊
 - **Less accuracy**, more memory, slower updates 😞
- Example
 - **Building**: cluster vectors, associate with nearest centroid
 - **Querying**: find nearest centroid to q , search its list
 - **Errors**: q near edge → may miss nearer neighbours!
- **Let's talk about indexing!**



Nearest neighbor search

Recall: Given a set $X \in \mathbb{R}^d$ of size n .

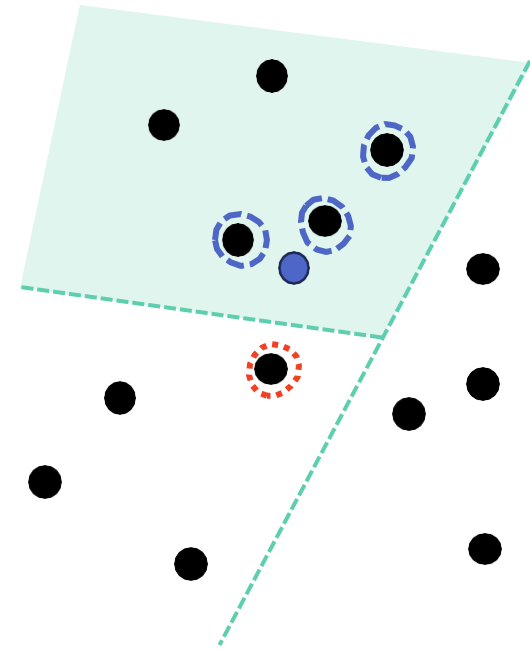
Goal: For query $q \in \mathbb{R}^d$, find nearest neighbor $x^* = NN_X(q) = \arg \min_{x \in X} \|x - q\|$.

Two phases:

1. Build data structure (hopefully not too much larger than $|X|$)
2. Answer queries for $q \in \mathbb{R}^d$.

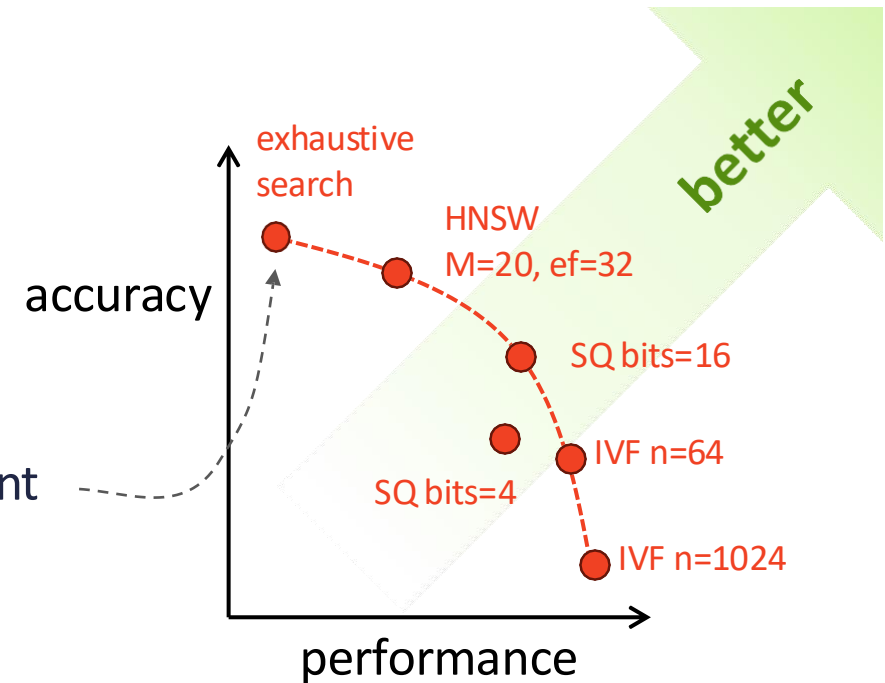
Why Index

- Flat (brute force) kNN search
 - Exact results
 - But N comparisons → Too slow!
- If we partition dataset
 - Fewer comparisons → faster search
 - Can introduce **errors**
- ANNS index is key to VDBMS performance!
- **But has costs!**
 - Can increase errors, memory, update cost



Performance tradeoffs

- ANNS indexes trade between:
 - Search speed
 - Accuracy
 - Memory
 - Build/update cost
- Crucial: search speed-accuracy tradeoff
 - **Index type + configuration** determine specific point
- How to choose index and configuration?
 - **Tune manually** on your data
 - **Automatically** using VDBMS optimizer (if exists)



(illustration only, not real data)

Metrics

- **Recall-K@K** (aka “recall@K” aka “recall”)

- Fraction of true k nearest neighbours returned by query
 - Alternative: out of k vectors returned, how many are truly kNN?
 - Alternative: overlap between true kNN and vectors returned by query.

- Example:

query q with $k = 5$ returns $x_1 \dots x_5$

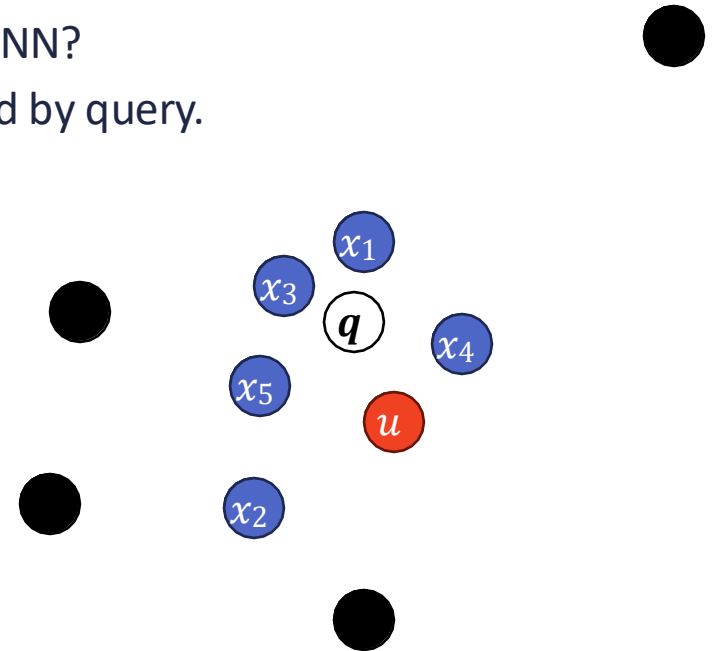
x_1, x_3, x_4, x_5 are true nearest neighbours

but x_2 is not (u is closer to q but is missed)

→ $\text{recall}_{5@5} = 4/k = 4/5 = 0.8$ or 80%

- **Latency**: milliseconds per query

- **Throughput**: queries per second



Index types

- **Cluster-based:** partition space to buckets of similar vectors
 - IVF, PQ
- **Graph-based:** connect similar vectors to make traversal graph
 - HNSW
 - DiskANN
- **LSH:** locality-sensitive hashing
 - Many variants
- **Tree:** partition space hierarchically
 - RP tree
 - ANNOY

ANNS index characteristics:

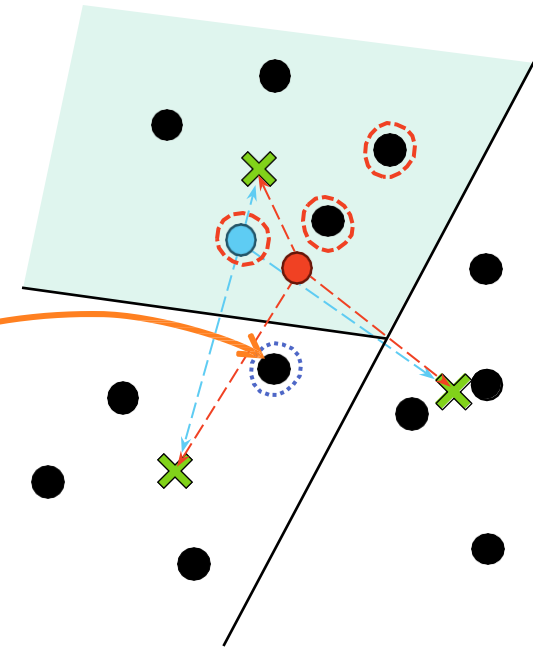
- Memory or disk resident:
 - Most are memory-resident
- Periodic rebuilding:
 - Some indexes require occasional rebuilds
 - E.g., classic HNSW, IVF
- Support incremental updates:
 - Yes, no, or partially supported
 - Real delete or tombstone?
- Error bounds:
 - Only LSH, RPTree

IVF: Inverted File Index

(also called clustering)

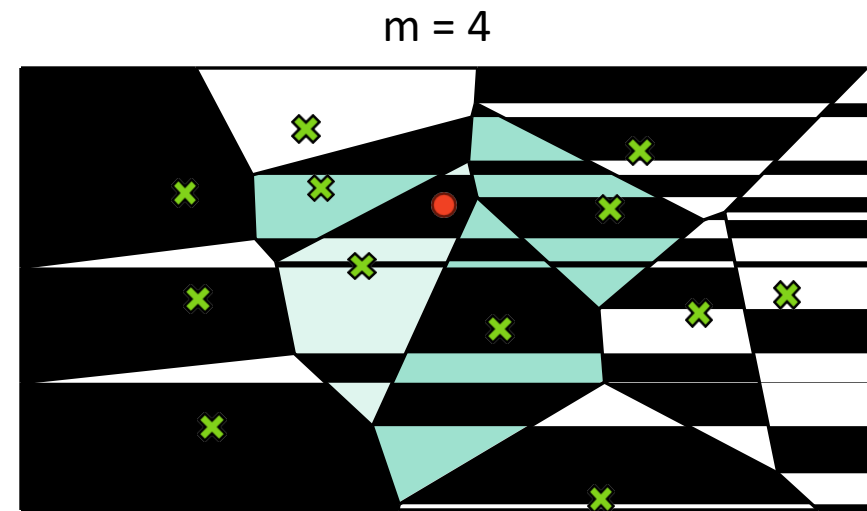
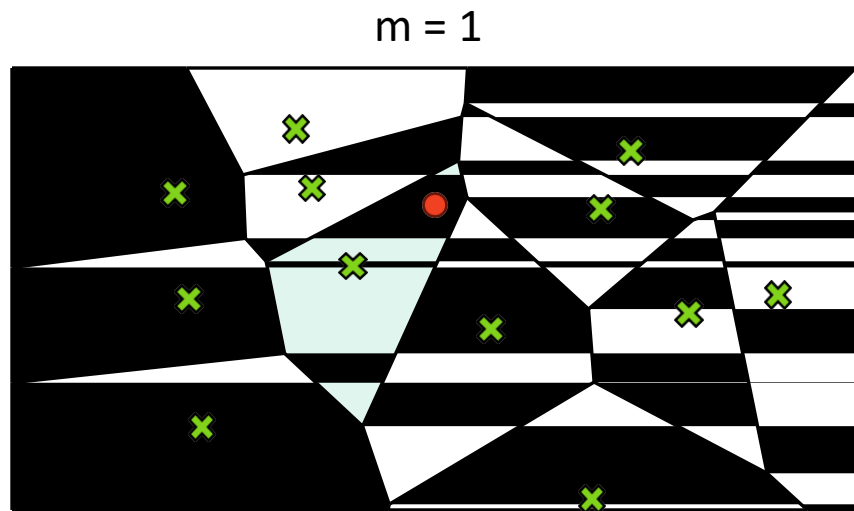
- Choose number of buckets (clusters) k
- Cluster vectors (e.g., k-means)
- To index v ●
 - Find nearest centroid
 - Add v to cell (cluster)
- To query q ●
 - Find nearest centroid
 - Compare q to vectors inside cell

Problem:
missed neighbour
in other cluster



Neighbor in other clusters

- List vectors near edges in two cells
- Increase search scope: probe $m > 1$ nearest cells



Nearest neighbor search

We introduce a new strategy today: Greedy Graph Search

1. Build a sparse graph $G = (X, E)$ on X , with $E(x)$ including at least its near neighbors
2. On query q , start with (any, random?) node $x \in X$, and see if any neighbors $x' \in E(x)$ are closer. If so, recurse on x' .
(More robust (and useful) to maintain k closest point.)

Terminate when no improvement is possible.

If each point $x \in X$ has at most m neighbors,
and each path is at most h hops,
then this approach has

- $O(nm)$ space
- $O(mh)$ query time.

Nearest neighbor search

Hierarchical Navigable Small World Graphs (HNSW)

Includes Neighborhood graph, undirected

Consider $L \leq \log n$ levels of edge length in graph

Each $x \in X$, for each level $\ell \in L$, choose \approx closest point

Approximate by building points $x_1, x_2, \dots \in X$, where $X_i = \{x_1, x_2, \dots, x_i\}$

When building $E(x_i)$ find closest K among points in X_{i-1}

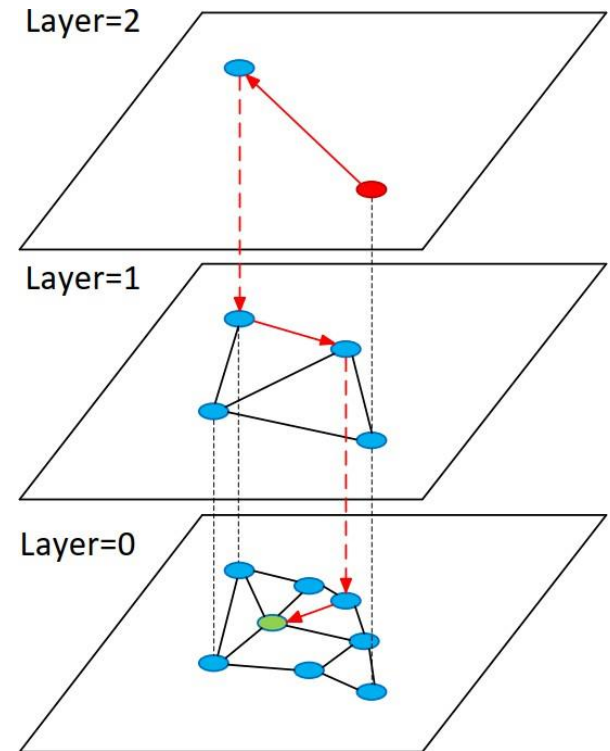
Add reverse nearest neighbor edges (undirected)

Make sure includes K' nearest neighbors.

Start search from one of first x_1, \dots, x_s for small s (with long edges).

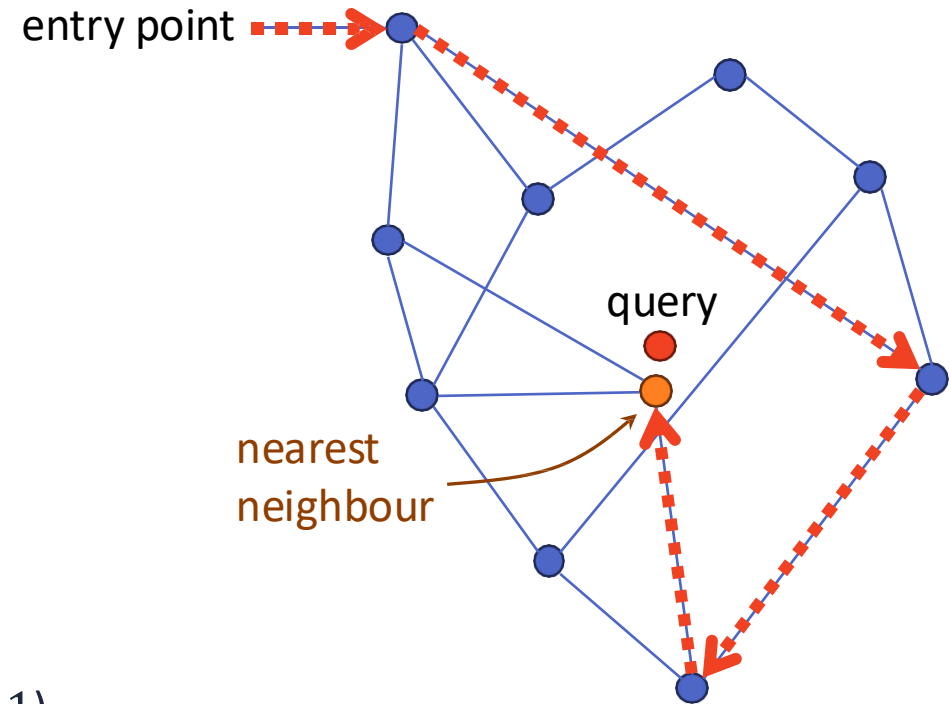
HNSW [Malkov, TPAMI'20]

- **H**ierarchically **N**avigable **S**mall **W**orlds
- Crown jewel of ANN indexes
 - Near SotA speed-accuracy tradeoff
 - Available quality implementations
 - Used in Qdrant, Weaviate (custom variants)
 - ... not quite SotA in academia
- Combines two ideas:
 1. Navigable Small World:
graph for traversal with greedy search
 2. Hierarchical skips:
higher layers allow fast skips



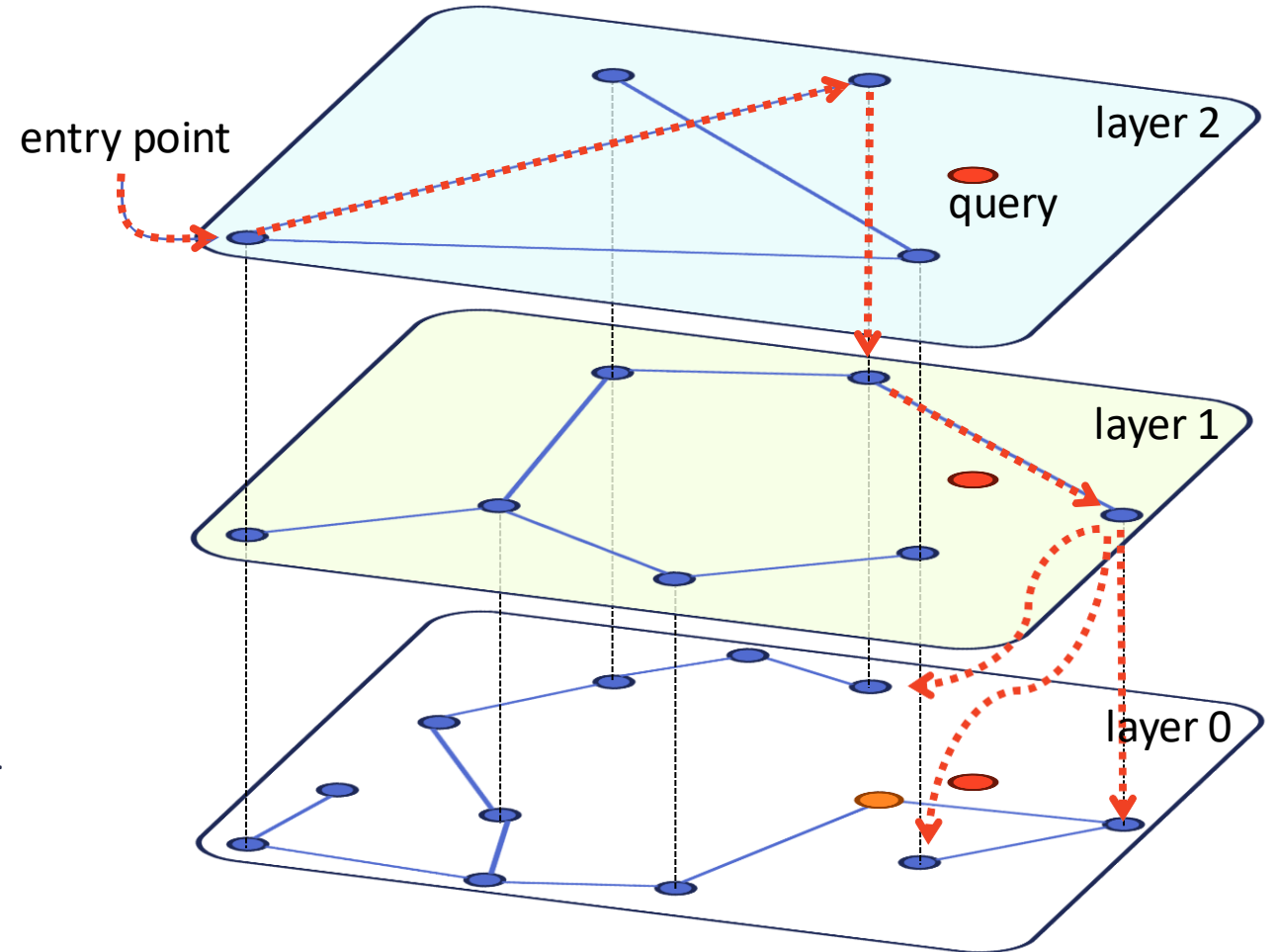
Navigable small world graphs

- Class of graphs:
 - Add long- and short-range edges
 - Characteristic path length $O(\log N)$
- Greedy search (DFS):
 - Start at entry point
 - Add its neighbours to candidate list
 - Go to candidate nearest to query
 - Repeat until no such candidate
- Search path is $O(\log N)$
- **Problem:** average out-degree $O(\log N)$
 - Polylogarithmic $O(\log^C N)$ search time ($C > 1$)



HNSW = NSW + Skip List

- To query:
 - Enter at top layer
 - Greedy search
 - Move to nearest connected neighbour
 - Done? Move to lower layer
- Result: $O(\log N)$ search
 - Because out-degree bounded
- Improve recall (incurs overhead):
 - At layer 0, expand search to $efSearch > 1$ neighbours



RAG implementation

FAISS

Facebook AI Similarity Search

mostly for $x^* = \arg \min_{x \in X} \|x - q\|$

Combination of 2 ideas

1. quantized index
2. GPU acceleration

Quantized index

database vector as $x \approx Q(x) = Q_1(x) + Q_2(x) + \dots + Q_m(x)$

- where $Q_j : \mathbb{R}^d \rightarrow C_j$
so C_j is a codebook of size k (k points in \mathbb{R}^d)
e.g., k centers of k -means clustering
- C_j has more detail than C_{j-1}
 C_j has more "weight" than C_{j+1} - measures larger distances
- m is small 2, 3, ... 6?

Roughly we want

- the same number of points $X_j \subset X$ which quantize to the $c_j \in C_1$
- the maximum distance $\max_{x \in X_j} \|c_j - x\|$ similar for each c_j .
(Should be feasible if doubling dimension bounded, and measure fairly uniform)

Quantized index

Then quantize each X_j with next another k codewords with that set recursively down to C_2, C_3, \dots, C_m .

each distance stored with limited precision (over limited range) --> saves space

On search $q \in \mathbb{R}^d$:

- find $c^* = \arg \min_{c \in C_1} \|q - c\|$
- recurse on X_{j^*} and its quantization C_{j^*}
- data adaptive, very-wide hierarchical index

More efficient and robust with maintaining top- K

GPU acceleration

GPU acceleration

The problem with very-wide architecture is that

... find $c^* = \arg \min_{c \in C_1} \|q - c\|$

requires a NN search!

GPUs are fast parallel processes

can min of k operations at once

solves N on size- k codebooks efficient

RAG and Pinecone

- Pinecone is a Billion-dollar company
- Based on graph-based similarity search
- Build to deliver RAG for companies

LLM + Vector DB Use Cases

Because large was not large enough

Vector database for LLMs



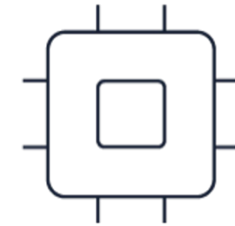
Context Retrieval

- Search for relevant sources of text from the “knowledge base”
- Provide as “context” to LLM



LLM “Memory”

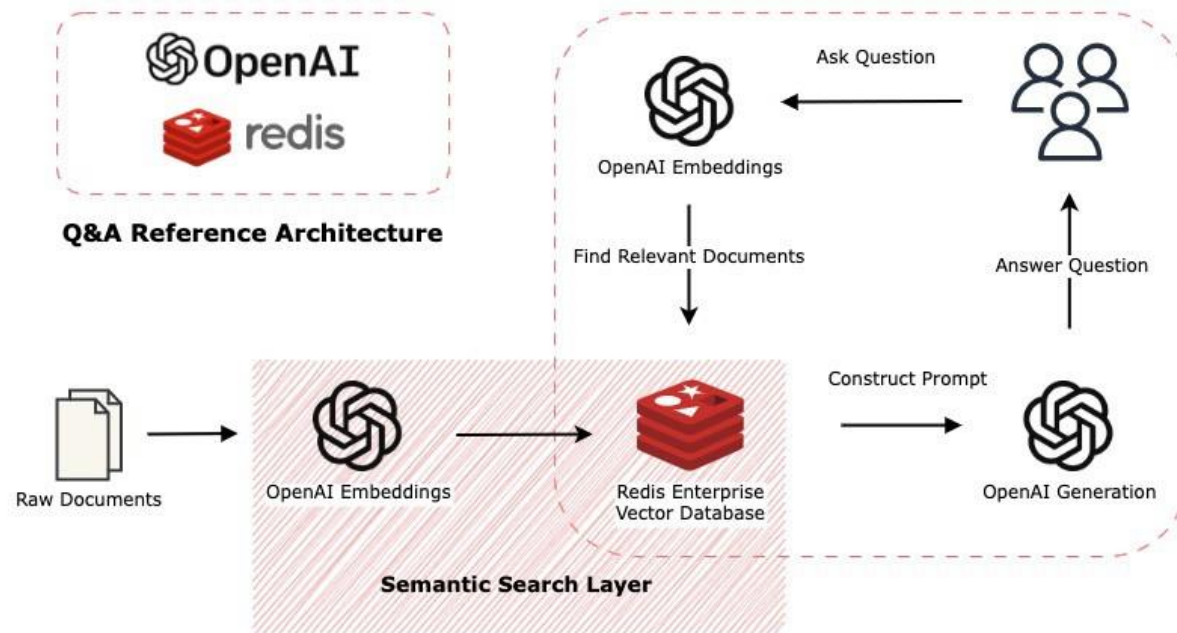
- Persist embedded conversation history
- Search for relevant conversation pieces as context for LLM



LLM Cache

- Search for semantically similar LLM prompts (inputs)
- Return cached responses

Context retrieval



- Description

- Vector database is used as an external knowledge base for the large language model.
- Queries are used to detect similar information (context) within the knowledge base

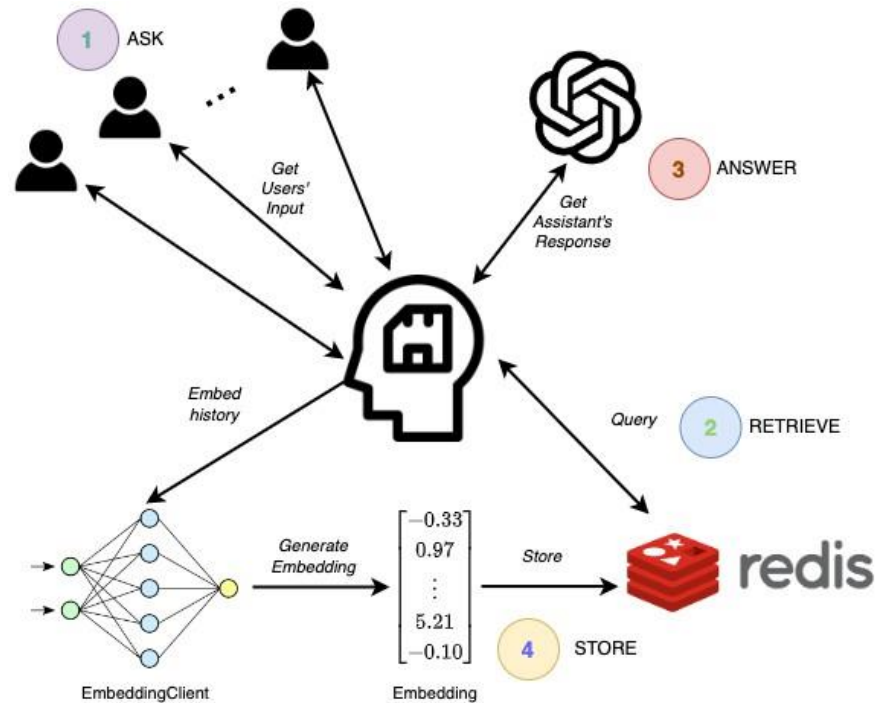
- Benefits

- **Cheaper and faster** than fine-tuning
- **Real-time updates** to knowledge base
- **Sensitive data** doesn't need to be used in model training or fine tuning

- Use Cases

- Document discovery and analysis
- Chatbots

Long term memory for LLMs



Description

- Theoretically infinite, contextual memory that encompasses multiple simultaneous sessions
- Retrieves only last K messages relevant to the current message in the entire history.

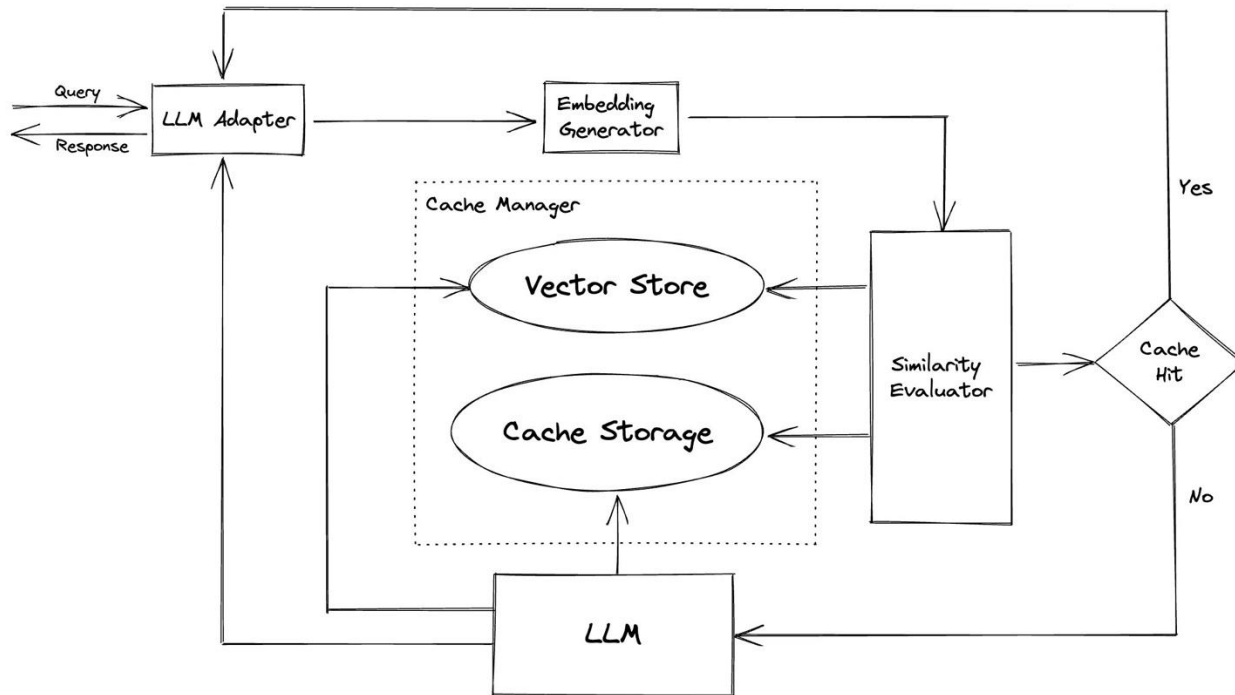
Benefits

- Provides **solution to context length limitations** of large language models
- Capable of **addressing topic changes** in conversation without context overflow

Use Cases

- Chatbots
- Information retrieval
- Continuous Knowledge Gathering

LLM query caching



- Description

- Vector database used to cache similar queries and answers
- Queries embedded and used as a cache lookup prior to LLM invocation

- Benefits

- **Saves on computational and monetary cost** of calling LLM models.
- Can **speed up applications** (LLMs are slow)

- Use Cases

- Every single use case we've talked about that uses an LLM.

Parting thoughts

- Vector databases are hot and underlie modern ML-based platforms
- There are still a bunch of open research questions regarding
 - Most efficient indexing technique for managing embeddings