Large Scale Data Management

Vector Databases

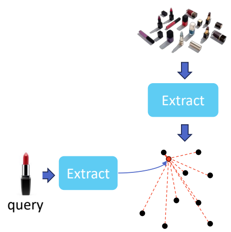
Silviu **Maniu**, LIG, Univ. Grenoble Alpes

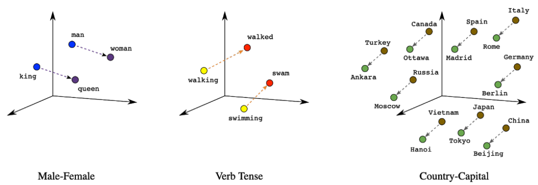
Vectors as Data

Semantic vs. Keyword Search

Application: take a photo -> search similar products

- Keyword search, SELECT FROM ... WHERE : need to define first the features (how?)
- Even if the feature are known, what we want is Semantic search ("close enough")vs. Keyword search (exact match)

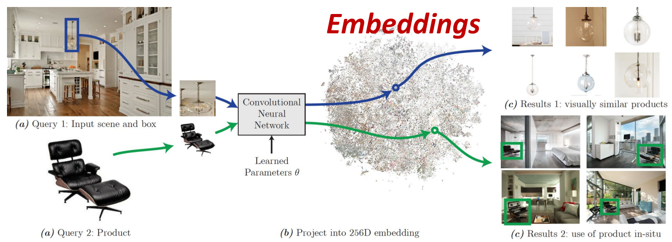




Vectors / Embeddings

Main idea: transform (embed) unstructured data (images, videos, etc.) into a metric space $\mathbb{R}^d$ so that similar data are close together into that space

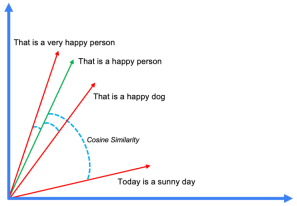
- Closeness can be measure by distances: Euclidean, Cosine



Queries in Vector Space

Query model: find most similar / closest vector

- Example query: "*That is a happy person*"



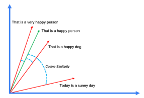
Cosine similarity

That is a happy dog	0.695
That is a very happy person	0.943
Today is a sunny day	0.257

Applications: Recommender Systems

Input: user, **Output**: recommend items

- Create vectors for products
- Create query vector *q* for user preference
- Find products close to *q*

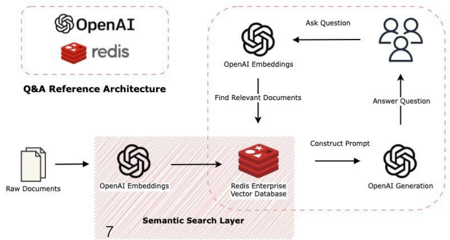


Requirements: high throughput, good accuracy

Applications: Retrieval Augmented Generation in LLMs

Input: user prompt, **Output**: generated answer

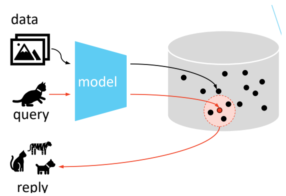
- Documents kept as vectors
- The prompt (transformed to a vector) is the query, search for enriched context (similar documents)
- Initial prompt + context : augmented prompt



Vector Databases: Basic Functionalities

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:

1. Embed query as vector *q*
2. Ask DB to find vectors similar to *q*

Querying and Storing Vectors

Storage problems:

- Ex: 100,000,000,000 documents stored as high dimensional vectors (d>1000) -> does not fit in RAM

Querying problems:

- Have to compute 100B floating point operations to get the exact answer

Vector DBs:

- Querying via approximate nearest neighbour (ANN) search
- Storage via indexing / sharding

Approximate Nearest Neighbour and Indexes

Approximate *k*-Nearest Neighbour Search

Problem: Given a query vector ***q*** find the ***k*** vectors that are approximately nearest to ***q*** by the distance ***d(q,v)***

Distances:

Euclidean $||v - q||_2$, Cosine $1 - \frac{\mathbf{q} \cdot \mathbf{v}}{||\mathbf{q}|| \ ||\mathbf{v}||}$, Manhattan, etc.

Maximise recall: $\frac{V \cap V^*}{V^*}$, where V^* the ground truth

Algorithms for AKNN

Tree based: KD-tree, R-tree

- Run slowly on high-dimensional data

Clustering-based: IVF_FLAT/SQ8/PQ

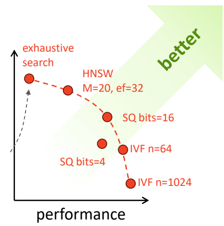
- High recall, clusters may be update-insensitive

Graph-based: HNSW, NSG, SSG

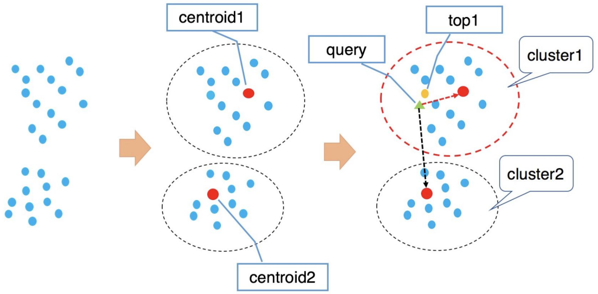
- High recall, graphs take time/space to maintain

Hash based: LSH (Locality Sensitive Hashing)

- Run slowly on high-dimensional data

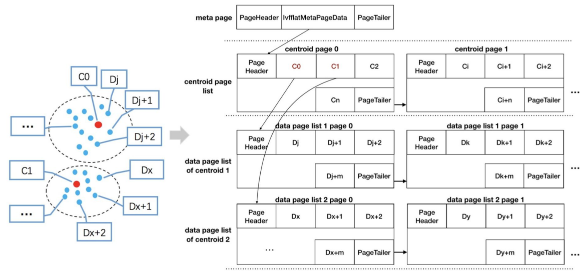


IVF_FLAT / SQ8 / PQ



Cluster data -> find closest cluster -> search in cluster: brute force (FLAT), compressed (SQ8), quantisation (PQ)

IVF_FLAT Index



Data is kept in codebooks per cluster, each has access to the pages containing vectors

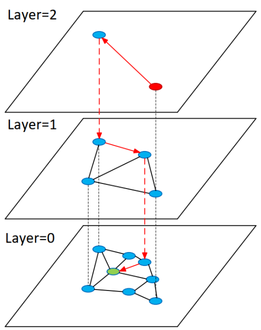
- **May miss nearest neighbours in close clusters!**

Hierarchical Navigable Small Worlds

State-of-the-art in indexes

Combines two ideas:

1. Traversal in a graph
2. Hierarchical Skips

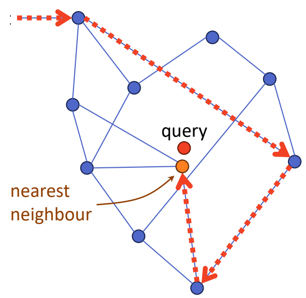


Navigable Small World

Construct a graph by adding short- and long-range edges, so that path length is $\mathcal{O}(\log N)$

Greedy search (DFS):

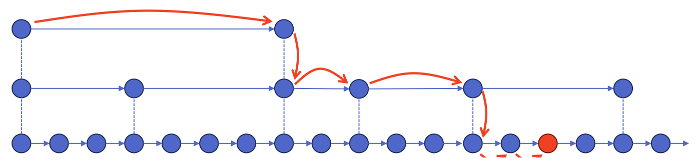
- Start at entry node
- Add neighbors to list
- Go to neighbour nearest to query
- Repeat



Problem: polylogarithmic search time $\mathcal{O}(\log^C N)$

Skip Lists

Hierarchical search: start at top layer -> search in layer -> move to lower if needed



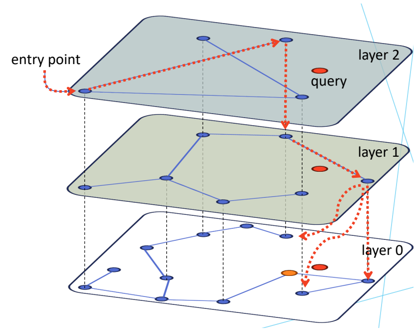
HNSW = NSW + Skip List

Hierarchical graphs: replicate nodes across layers, long edges at higher layers, short at lower layer, bounded out degree

Query:

- Enter at top layer
- Greedy search to nearer
- Move to lower layer

$\mathcal{O}(\log N)$ **search**!



ANN Implementations

Multiple **implementations**: FAISS (Meta AI), SPTAG (Microsoft), Annoy (Spotify)

Computation is optimised: usually low level implementations (C++, uses advance CPU instructions, uses GPU)

However:

- Only memory-based
- No dynamic data support, no attribute filtering

Vector Database Systems

Relational vs. Vector

	Traditional DB (RDBMS)	Vector DB (VDBMS)
Data	Records	Vectors
Queries	Relational algebra	Nearest neighbors + simple filtering
Advanced query features	Join, group, FK, cursors	None of those*
Updates	To part of record To multiple records	On whole vector Insert/delete/replace
Consistency	Strong + transactions	Eventual, tunable
Index updates	Fast	Slow
Storage	Row/column based, LSM	Vector is opaque blob
Hardware , scaling cost	Uniform , moderate	Diverse , expensive (GPUs)
Architecture	More monolithic	More disaggregated

Types of Vector Databases

Extended

- Extend existing DB (relational/NoSQL)
- Examples: **pgvector**, PASE, Redis, CosmosDB, Timescale
- Keep power of queries: ACID, transactions, SQL,...
- Slower (due to ACID), limited dimension
- Can store original documents also

Native

- Designed as vector DBs, specialised architecture
- Examples: **Chroma**, Pinecone, Azure AI Search, Search, ...
- Limited queries: similarity + filter usually
- High performance (ANN implementations)

pgvector

PostgreSQL plugin

Adds the `vector(dim)` type

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding vector(3));

INSERT INTO items (embedding) VALUES ('[1,2,3]'), ('[4,5,6]');
```

Can match using distances

```
SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

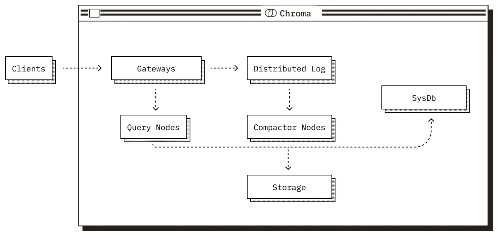
- L2 <->, L1 <+>, Cosine <=>, inner product <#>

By default exact embedding search, but can create IVF and HNSW indexes

```
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops);
```


ChromaDB

Native vector DB



Modes: embedded (py lib), single node (server), distributed

Log

- Write-ahead log (all writes recorded) for atomicity, replay

Query Executor

- All read operations: vector similarity, metadata search

Compactor

- Reads from Log and builds the indexes, writes them to storage
- Updates system metadata about index versions

ChromaDB
Data Model & Querying

Unit of storage: collections (equivalent to tables)

```
collection.add(ids=["id1"], documents=["cat"], metadatas=[{"color": "orange"}])
```

- Contains: unique id, embedding vector, optional metadata, original document
- Embeddings can be pre-trained (using Python libs such as SentenceTransformer), or can define own functions

Query collections

```
results = collection.query(query_texts=["Query document"], n_results=2)
```

- By default, L2 distance; HNSW index (only one supported by Chroma)
- Can provide embeddings directly as query, or even images (multimodal search)
- Can query metadata (used as filter in HNSW navigation)

To Read Further

Articles

- Jégou et al. *Product Quantization for Nearest Neighbour Search*. https://inria.hal.science/inria-00514462v2/document
- Malkov, Yashunin. *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs* https://arxiv.org/pdf/1603.09320

Specs and documentation

- ChromaDB: https://docs.trychroma.com/ (example vector DB)
- pgvector: https://github.com/pgvector/pgvector (example vectors on top of DBMS)
- FAISS: https://faiss.ai/index.html (example AKNN library)