TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

# Vector Databases

**Corso di Sistemi e Architetture per Big Data**

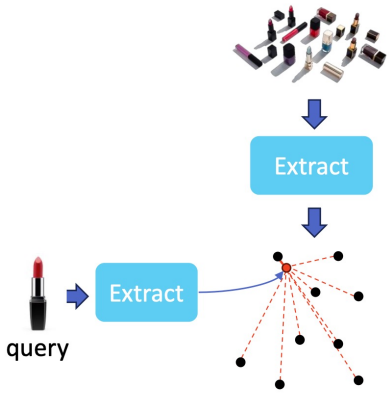
A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

## Example: visual product search

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




Li et al., The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform, Middleware '18

# Again, scale problem

- N = 100B = 100,000,000,000 vectors
- Each vector is large: D = ~1,000 floats
- Problem 1: storing N vectors for fast access requires 400 TB
  - Too much for RAM
- Problem 2: finding nearest neighbor:
  - Distance to N vectors = O(ND) multiply-adds → N*D = 100T
  - Even at 20 TFLOPS, 5 sec latency per query (ignoring other costs)
- "Put it in a database and index?"
  - Index what? DB indices designed for individual attributes, not Nearest Neighbors (NN) search on vectors
  - Not clear how to shard vectors



# Visual product search: find products fast

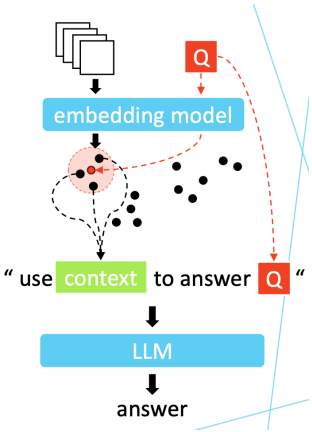
- Take photo → find matching products
- What do we need?
- Index to accelerate cluster-based search:
  - Group similar products into clusters, store products in per-cluster list
  - Query = find the nearest cluster(s), search only products in the list of nearest cluster(s)
  - Insert = add product vector to list of nearest cluster
- Let's consider other case studies
  - RAG question answering
  - Recommender systems

# RAG question answering

- Get question, use LLM to generate answer
- Retrieval-Augmented Generation (RAG) pipeline adds context to prompt:
  1. Generate document embeddings

  → store vectors
  2. To query: generate embedding of user query
  3. Get relevant documents with respect to user query; rerank documents using a scoring mode

  → neighbor search + rerank
  4. Create prompt to incorporate the reranked documents as context
  5. Use LLM to generate final response



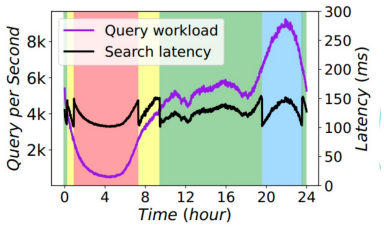



# Recommender systems

- Given user, recommend products, videos or music:
  1. Create vectors for products

  → store vectors
  2. Create vector q for user preference
  – Combined representation of user profile (historical) + real-time session activity (searches, recent clicks)
  3. Find candidate products "close to" q

  → neighbor search + rerank
  4. Re-rank candidate products based on recent activity
- Requirements:
  – High throughput
  – Good accuracy
  – Elasticity due daily demand fluctuations



# Vector database

- A Vector DB is a DB specifically designed to store, index, and search high-dimensional vectors that represent complex multimodal data such as:
  - Text
  - Images
  - Audio
  - Video
  - Documents
- Traditional DBs perform exact-match searches, while Vector DBs enable semantic search
  - Find items based on meaning rather than keywords

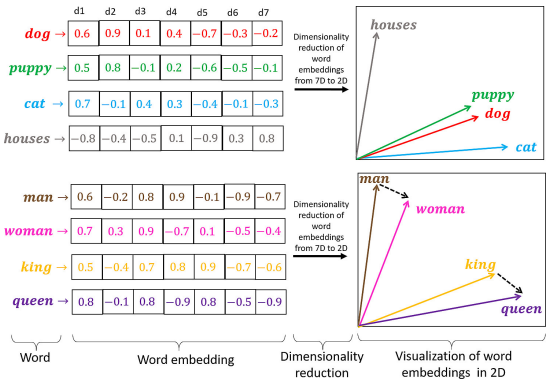


# Required functionalities

- Insert data:
  - Convert data (text, image, sound, graph) into an embedding vector
    - Usually using an ML model
  - Store vectors (embeddings) in vector DB
- Query data:
  - Embed query as vector q
  - Ask vector DB to find vectors similar to q

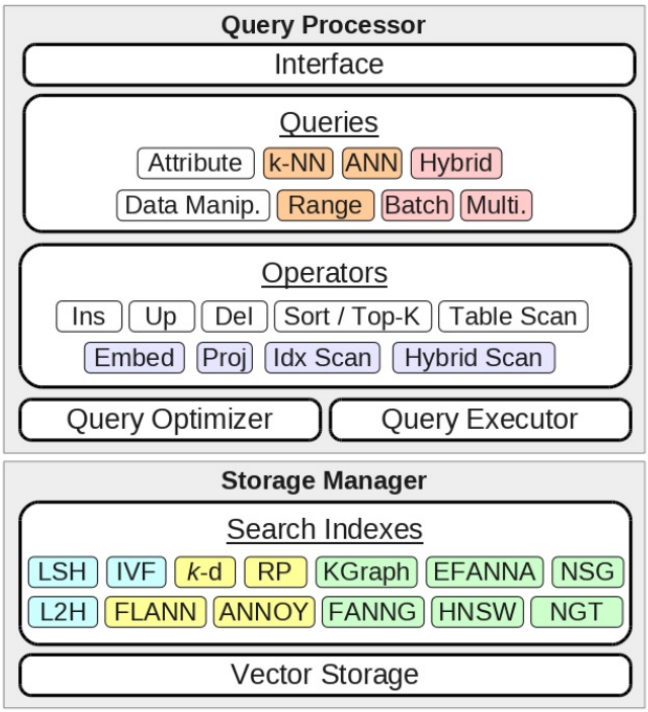
# From data to embeddings

- Embeddings
  - Before data can be stored in a Vector DB, it is converted into a numerical representation called an embedding using ML models
- Example: "The cat is sleeping on the couch"
  - Embedding model: [0.12, -0.45, 0.88, ..., 0.21]
- Key idea
  - Similar content → similar vectors
  - Different content → distant vectors
- The embedding captures the semantic meaning of the data



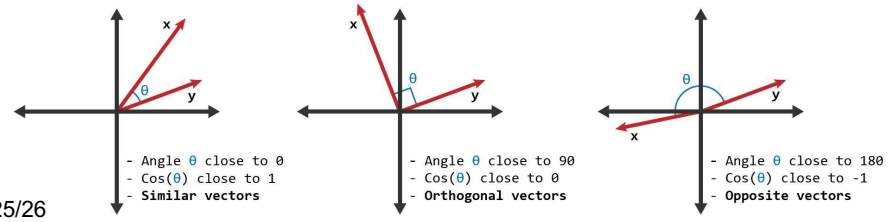



# Overview of Vector DB



# Similarity search

- How retrieval works
  - Convert the user query into an embedding
  - Compare it with vectors stored in the database
  - Retrieve the nearest vectors
- Similarity scores: map 2 vectors onto a scalar
  - Euclidean distance
  - Dot product
  - Cosine similarity
    - Cosine of the angle between 2 nonzero vectors
    - Take direction into account rather than length






# Similarity search

- Example
  - Query: How can I reduce energy consumption?
  - Retrieved documents:
    - Energy optimization techniques
    - Building efficiency strategies
    - Electricity-saving recommendations
  - Even if they do not contain the exact same keywords

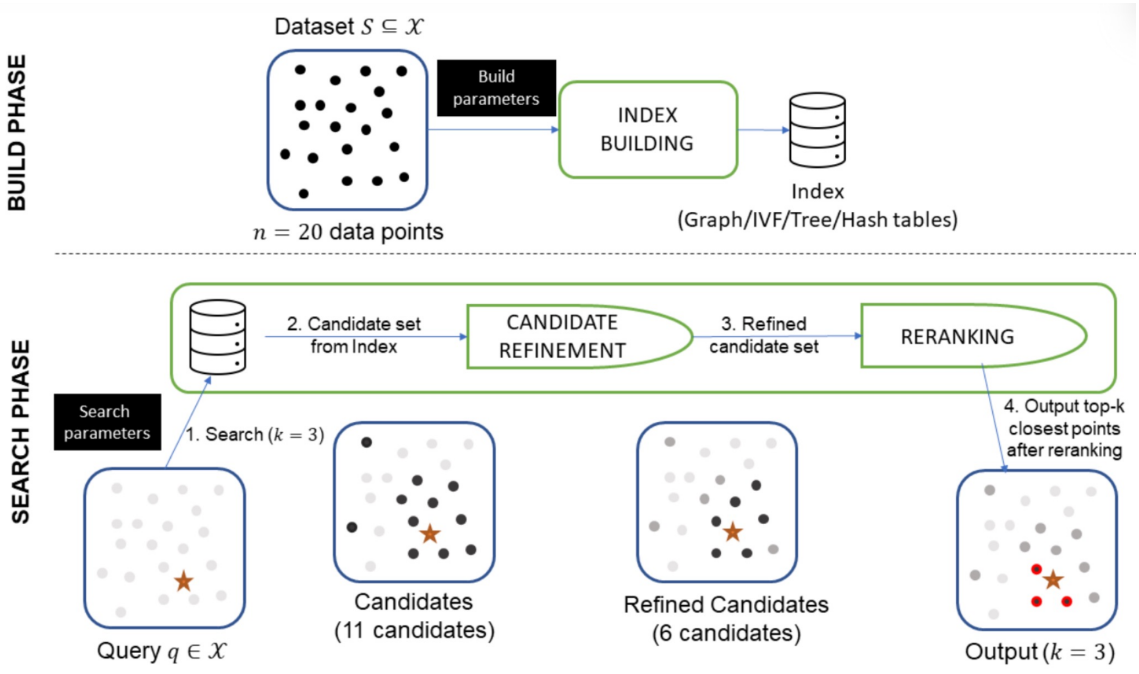
# Indexing and scalability

- Challenge: comparing a query against millions of high-dimensional vectors is computationally expensive
- Goals:
  - Fast retrieval
  - Scalability to billions of vectors
  - Reduced memory and compute requirements
- Solution: use **Approximate Nearest Neighbor (ANN)** algorithms to accelerate search by finding nearest neighbors approximately
  - Approximate: allow some of the reported nearest neighbors to be wrong



# ANN search architecture

- 2 main stages: build phase (offline) and search phase (online)



# ANN search architecture

- **Build** phase: transforms raw data into a searchable structure
  - Index building: process of structuring unstructured vectors into organized mathematical spaces
- Approaches for high-dimensional indexing
  - Clustering-based (e.g., Inverted File Index)
  - Graph-based (e.g., HNSW)
  - Tree-based (e.g., KD-Trees)
  - Hashing-based (e.g., Locality Ssensitive Hashing)
- **Search** phase: handles user queries in real-time, executing a multi-step filter-and-refine pipeline to find the top-k closest matches

  Pipeline: User query → 1. Index lookup → 2. Candidate set → 3. Refinement → 4. Reranking → Top-k output



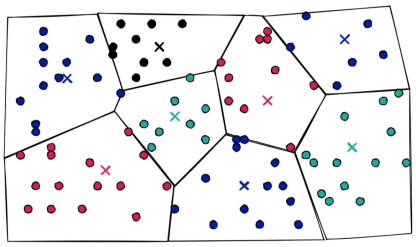
# Product quantization

- Idea: compress vectors by mapping them to a limited set of reference points
- Benefits
  - Reduced memory and storage usage
  - Faster search operations
  - Suitable for very large datasets
- Trade-off
  - Higher speed and efficiency
  - Slight loss of search accuracy
- Best for: large-scale applications where speed is more important than high precision

Product Quantization for Nearest Neighbor Search, https://inria.hal.science/inria-00514462/document

# Inverted File Index (IVF)

- How it works
  - Divides vectors into clusters (e.g., using K-means)
  - Searches only the most relevant cluster(s)
- Inverted file index: the data structure that stores the centroids and the associated lists

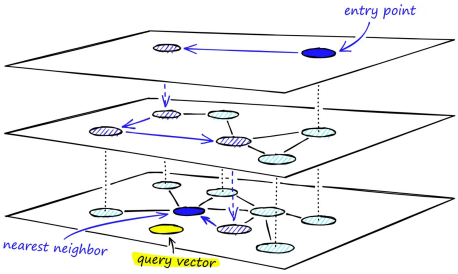


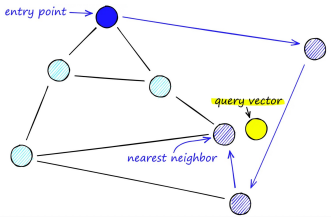
- Benefits
  - Reduces search space
  - Improves query speed
  - Scales efficiently with large datasets
  - Enhancement
- Often combined with quantization (e.g., IVF-PQ) for better performance
- Best for: large, relatively static datasets



# Hierarchical Navigable Small World (HNSW)

- How it works: creates a multi-layer graph structure
  - Upper layers provide coarse navigation
  - Lower layers provide detailed search




- Pros and cons
  - ✓ Balance between speed and accuracy
  - ✓ Near real-time retrieval
  - ✓ Highly scalable
  - X High memory consumption for very large datasets
- Best for: recommendation systems, semantic search, and real-time AI applications

## Search: candidate refinement

- Goal: quickly find the most promising candidates out of filtered points
- Idea: discard candidate points that are unlikely to be among the k nearest neighbors
- How: avoid computing the distance in the original space
  - Using a quantization (e.g., product quantization) or sketching technique that stores small summaries of each point



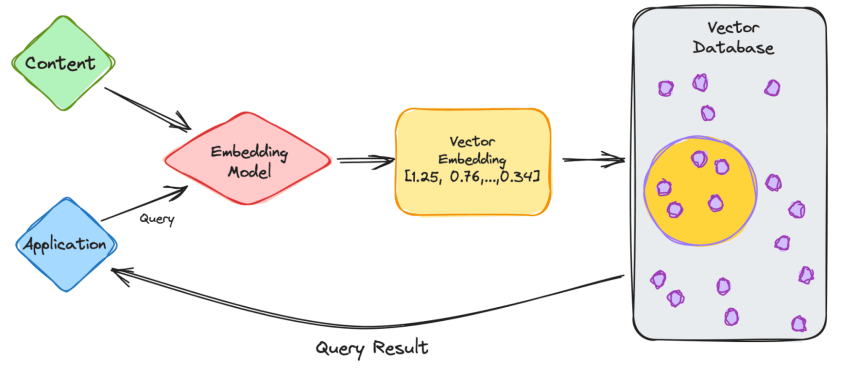
## Additional optimization techniques

- Dimensionality reduction
  - Techniques: PCA, autoencoders
  - Reduce vector size while preserving important information
  - Faster indexing and retrieval
- Parallel processing
  - Multi-core CPUs and GPUs
  - Multiple queries processed simultaneously
  - Higher throughput and lower latency
- Dynamic indexing
  - Efficient insertion and deletion of vectors
  - Maintains performance in frequently updated DBs

# Vector DB in RAG systems

- Typical pipeline

Content (e.g., documents)
↓
Generate embeddings
↓
Vector DB
↓
Similarity search
↓
Top-K relevant content using ANN search
↓
Relevant content passed as context to LLM
↓
Generated response






# Vector DB in RAG systems

- Benefits
  - Access to up-to-date information
  - Reduced hallucinations
  - Context-aware responses
- Use case
  - Enterprise chatbot answering questions using internal documentation

# Products

- Open-source
  - Milvus https://milvus.io
  - Qdrant https://qdrant.tech
  - Weaviate https://weaviate.io
  - Chroma https://www.trychroma.com
  - FAISS (library) https://github.com/facebookresearch/faiss
- Integrated
  - pgvector (PostgreSQL)
  - MongoDB Atlas Vector Search
  - Redis
- Managed Cloud services
  - Pinecone https://www.pinecone.io
  - Azure AI Search
  - Amazon OpenSearch
- Selection criteria
  - Performance
  - Scalability
  - Cost
  - Cloud integration
  - Ease of deployment



# Pros and cons

✓ Semantic search capabilities
✓ Efficient retrieval from unstructured data
✓ Essential for AI and LLM applications
✓ Foundation of RAG

X Retrieval quality depends on embedding quality
X Similarity search may occasionally retrieve irrelevant results
X Additional infrastructure complexity
X Computational cost at large scale

# Takeway

- Vector DBs have become a core component of modern AI systems, enabling fast and accurate semantic retrieval across large collections of unstructured data



# References


- Pan et al., Survey of vector database management systems, *The VLDB Journal*, 2024 https://doi.org/10.1007/s00778-024-00864-x
- Aumüller and Ceccarello, Recent Approaches and Trends in Approximate Nearest Neighbor Search, *Bulletin of the Technical Committee on Data Engineering*, 2023 http://sites.computer.org/debull/A23sept/p89.pdf