



**POLITECNICO**  
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

# Vector Databases

Marco Brambilla

marco.brambilla@polimi.it

 @marcobrambi

# Vector Databases Overview

Vector databases are gaining popularity due to the rise of AI and ML applications.

In particular, applications that use LLM + RAG (Retrieval Augmented Generation)

Companies are actively developing vector search capabilities for their SQL or NoSQL databases.



# Why Are Vector Databases Important?

Vector databases allow for more efficient and semantically rich data retrieval.

They enable AI systems to understand **context and similarity** in large datasets, enhancing search and recommendation systems.

# Available Vector Databases

Several vector databases have recently emerged and are available.

Companies are investing heavily in the development of these technologies

# The big names of Vect DB ecosystem



Milvus



Weaviate



Qdrant



Elastic



Chroma



FAISS

pgvector

Pgvector



Zilliz



MongoDB Atlas



Pinecone

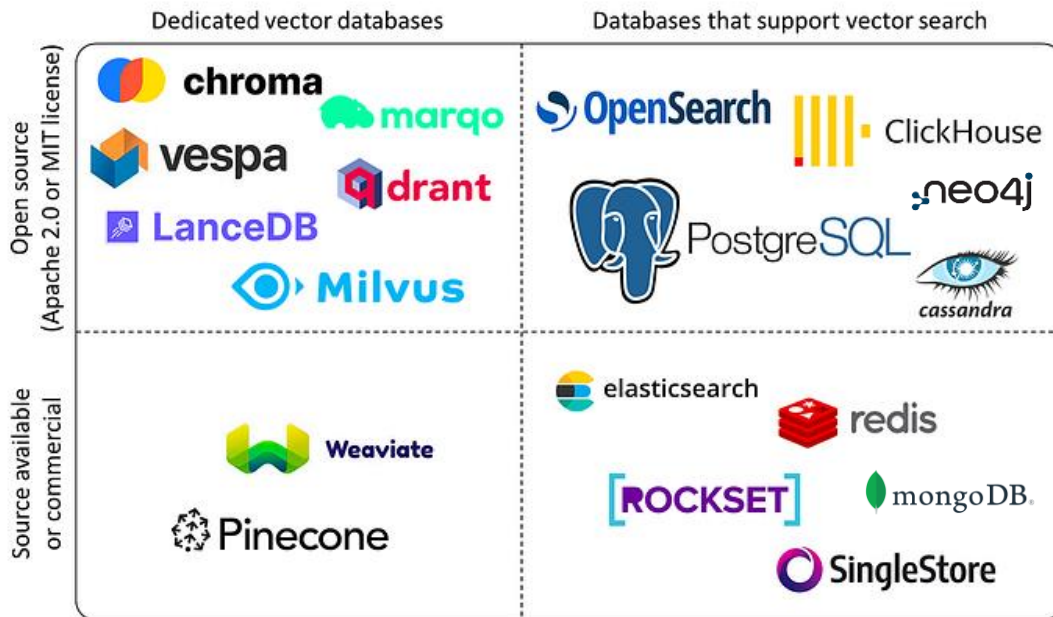


LanceDB



vespa

# Jumping on board



Many databases are considering incorporating vector search capabilities to meet the demands of their current user base. For databases that currently lack vector search functionality, it is only a matter of time before they implement these features.

# What is a Vector Database?

A vector database stores data as

**high-dimensional vector representations.**

It enables contextual search and discovery of data, going beyond keyword matching.

# How Vector Databases Work

Vector DBs store data as vectors.

They **organize the vectors by their semantic similarity**, allowing for efficient and meaningful data retrieval based on context.



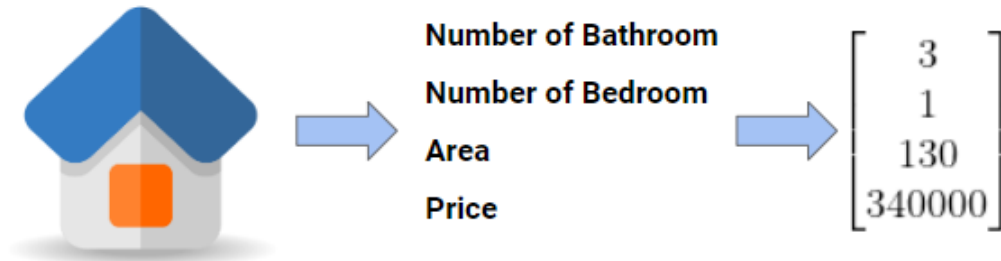
# Vector: A Basic Concept

In mathematics and physics, a vector is a quantity with both magnitude and direction. In vector databases, vectors represent various attributes of an object.

Vectors are the fundamental representations of information for all the AI and ML applications and technologies

# Vectors in Data Representation

For example, a house can be represented by a vector containing the number of rooms, area, and price. Vector DBs store and manage such data in vector form.



# What is Embedding?

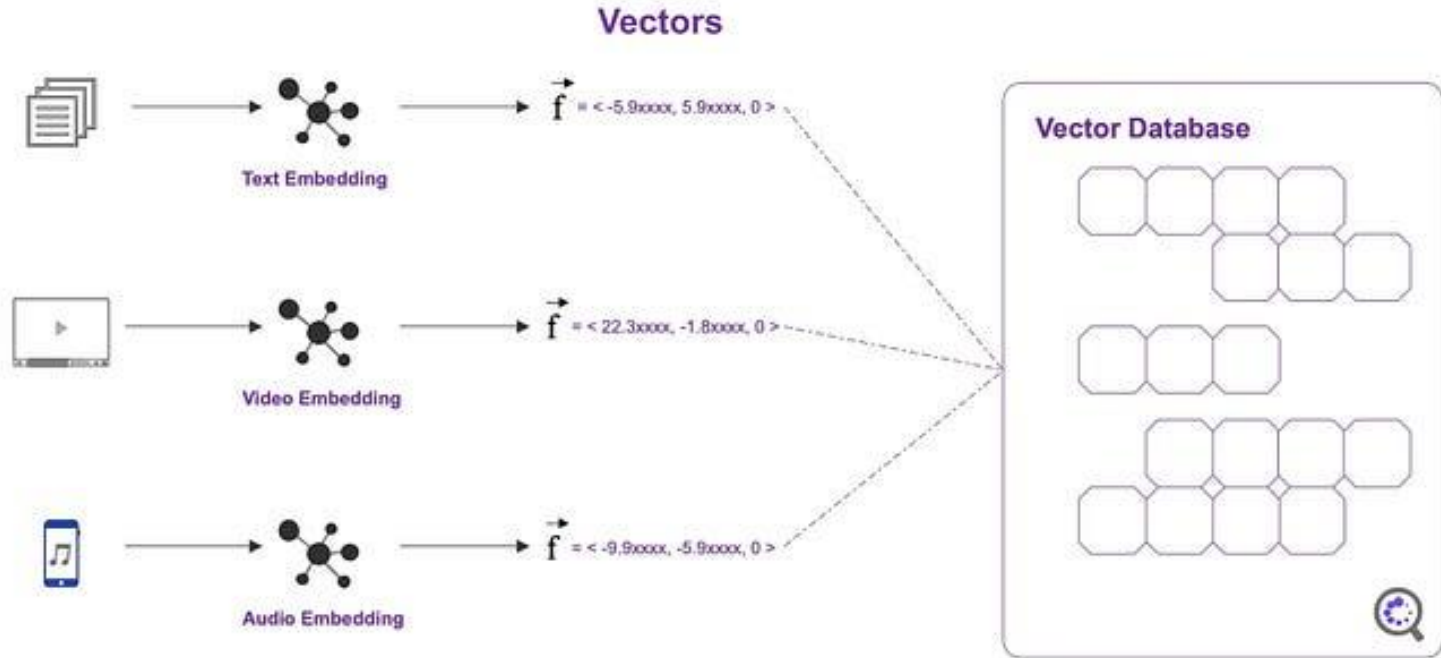
Embedding is a technique to transform text, images, or any other data into vectors.

Pre-trained models for textual data map words into a multi-dimensional space, capturing their meaning and context.

Examples:

OpenAI's Embedding Models	Sentence Transformer Embeddings	Other models and provider
e.g. <ul style="list-style-type: none"><li>• text-embedding-ada-002</li><li>• text-embedding-davinci-001</li><li>• text-embedding-curie-001</li><li>• text-embedding-babbage-001</li><li>• text-embedding-ada-001</li></ul>	e.g. <ul style="list-style-type: none"><li>• all-MiniLM-L6-v2</li><li>• all-MiniLM-L12-v1</li><li>• all-mpnet-base-v1</li><li>• all-roberta-large-v1</li></ul>	<ul style="list-style-type: none"><li>• Google Vertex AI</li><li>• Google PaLM</li><li>• Aleph Alpha</li><li>• Elasticsearch</li><li>• ...</li></ul>

# Multi-modal vectorization



# Example of Embedding

Example: Transforming sentences like 'Sunsets are breathtaking' into vectors using models like paraphrase-MiniLM. Vectors representing similar sentiments will be close to each other.

Process:

- Take a sentence

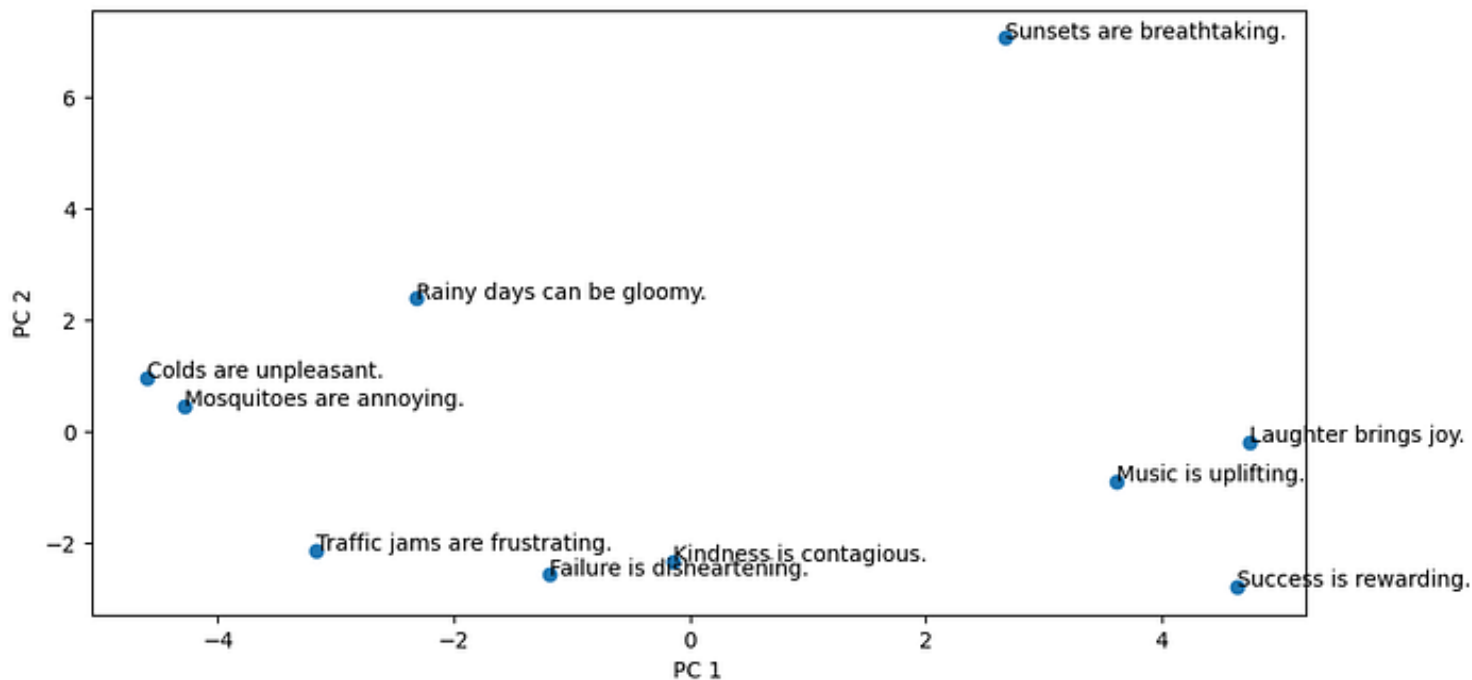
- Transform it into (numerical) vector representation

- Possibly reduce dimensionality if needed / wanted



# Text Vectorization

Example: sentences to 32 dimensions vectors, and then PCA to top 2 principal components

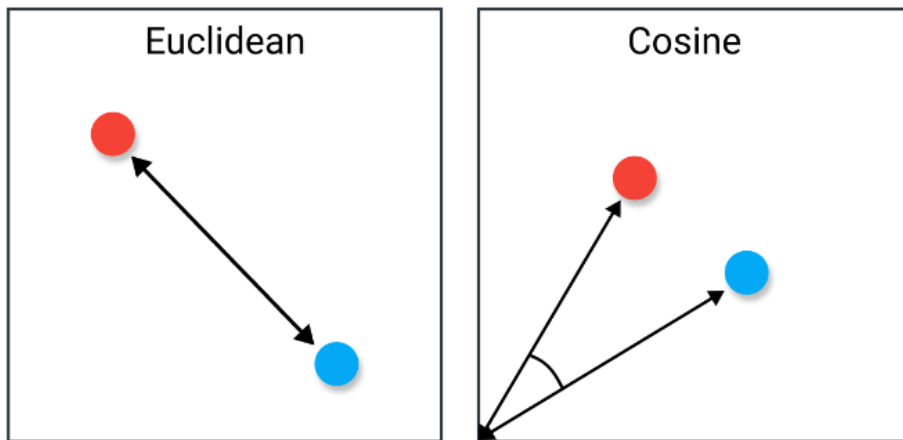


# Importance of Embedding

Embedding allows for meaningful representation of data, enabling vector databases to find relevant information through vector similarity, rather than exact text matches.

# Similarity Score

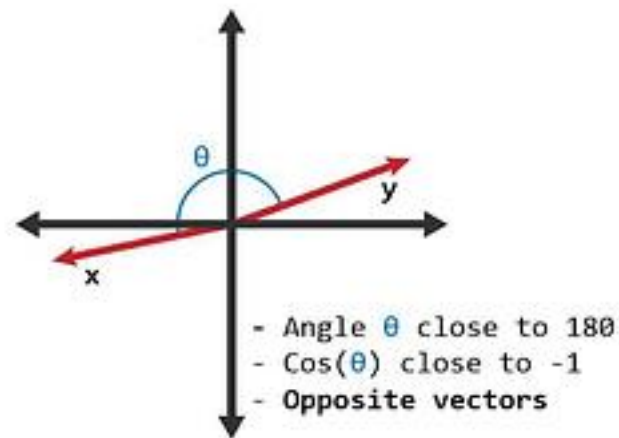
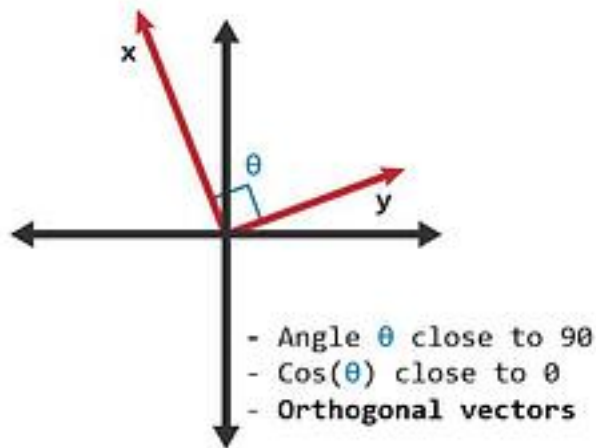
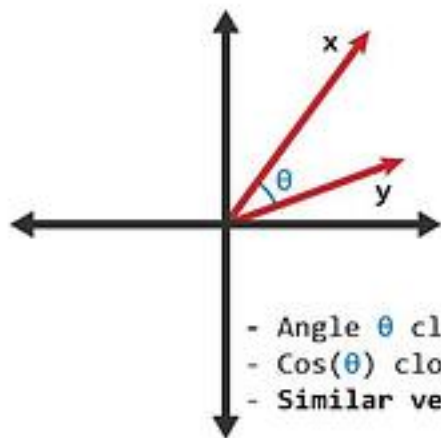
Metrics like Cosine and Euclidean distances are used to measure the similarity between two vectors. Cosine similarity is commonly used due to its efficiency.





# Similarity Calculation Example

Example: Calculate the similarity between 'Rainy days make me sad' and 'Rainy days can be gloomy'. The cosine score between these sentences shows high similarity.



# Distance Calculation

let's consider a new sentence: 'Rainy days make me sad'. What's the distance wrt the other sentences? Which is the best match?

	text_chunk	embeddings	cosine
0	Rainy days can be gloomy.	[0.118068404, 0.00039941736, 0.7828672, 0.7579...	0.764286
1	Colds are unpleasant.	[-0.6519384, 0.47603434, 0.3156743, 1.2116411,...	0.418937
2	Traffic jams are frustrating.	[0.06174645, -0.4527145, 0.46639186, -0.283001...	0.323204
3	Kindness is contagious.	[-0.44167116, -0.006431464, 0.20754915, 0.0754...	0.319614
4	Mosquitoes are annoying.	[0.095289715, 0.061180115, 0.31327882, 0.28330...	0.296069
5	Laughter brings joy.	[-0.10080257, 0.45041445, 0.4479737, -0.647958...	0.255170
6	Failure is disheartening.	[-0.3408494, -0.5745256, 0.75116944, 0.0891811...	0.239249
7	Music is uplifting.	[0.26783174, -0.005341845, 0.38134262, 0.09905...	0.189170
8	Sunsets are breathtaking.	[0.5539868, 0.4602928, 0.067907006, 0.46939665...	0.091070
9	Success is rewarding.	[-0.08308301, 0.31891197, -0.16729859, -0.5068...	0.033636



POLITECNICO  
MILANO 1863

# Vector Matching Efficiency and Scalability

Step 1 – Tree-based indexing

# Speeding Up Similarity Search

Vector database store vectors efficiently

But most importantly: **they let you find matches efficiently!**

Nearest neighbor search without an index involves computing the distance from the query to each point in the database, which for large datasets is computationally prohibitive

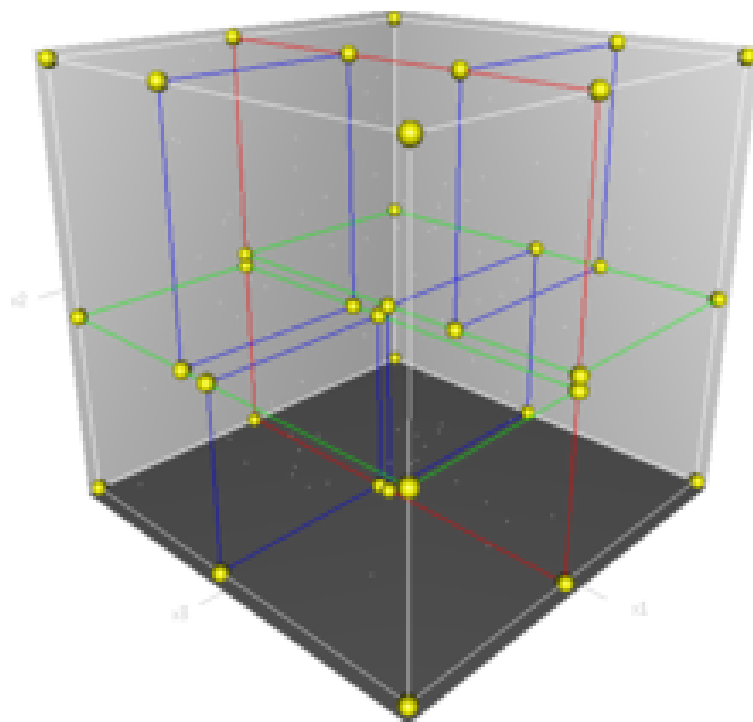
Let's create indexes with tree-based exact vector search techniques

# K-D Trees [1975]

k-d trees are a special case of binary space partitioning trees a binary tree in which every node is a k-dimensional point.

Every non-leaf node implicitly generates splitting hyperplane that divides the space into two parts

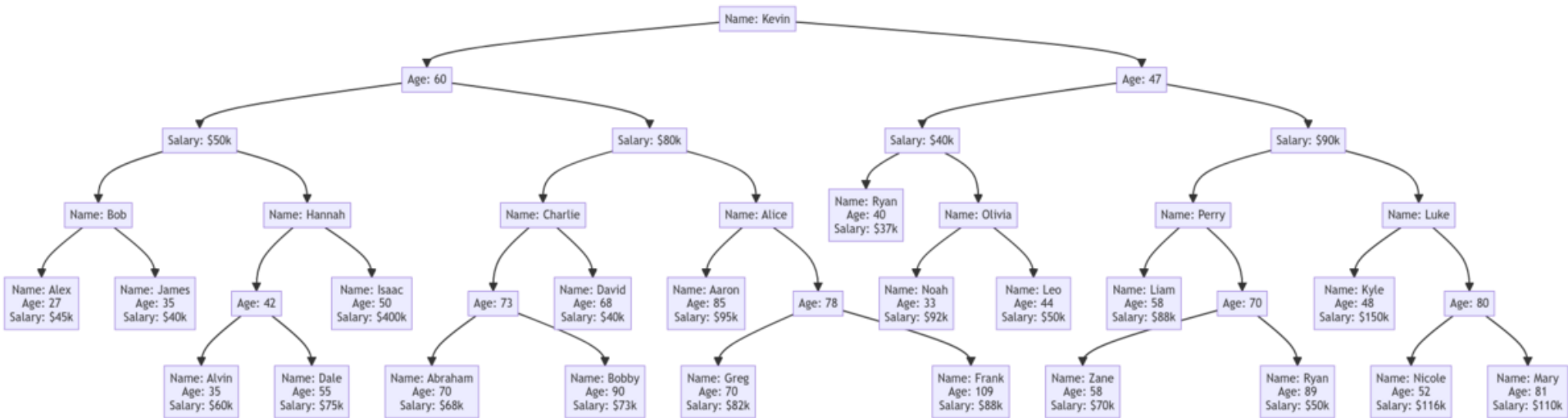
The split is on the median value



# Space splitting in K-D Trees



POLITECNICO  
MILANO 1863





# K-D Trees Complexity

$n$  = number of objects (aka. Points or vectors)

$k$  = Dimensionality of the vectors

Complexity of search (and other operations):  $O(\log n)$

Hyp.  $n \gg 2^k$

With high-dimensional data, most of the points in the tree will be evaluated and the efficiency is no better than exhaustive search

# R-Trees [1984]



POLITECNICO  
MILANO 1863

the "R" in R-tree stands for rectangle

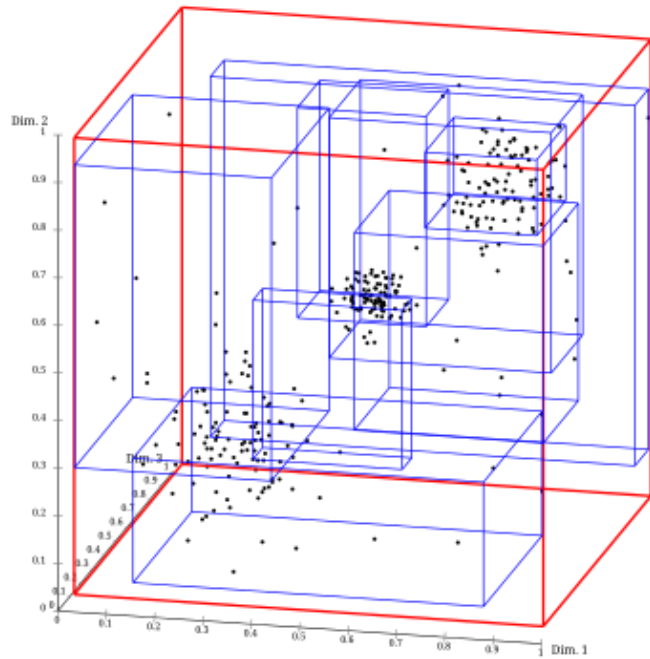
Idea: to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree

Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects

- At the leaf level, each rectangle describes a single object

- At higher levels the aggregation includes an increasing number of objects

It can be seen as an increasingly coarse approximation of the data set.





# R-Trees storage

It's a balanced tree

Organizes data in pages

Each page has a max and min filling storage limit

Best performance with min filling around 40%

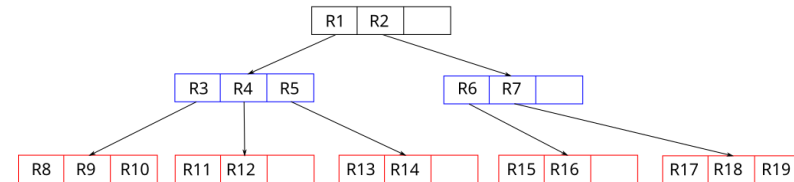
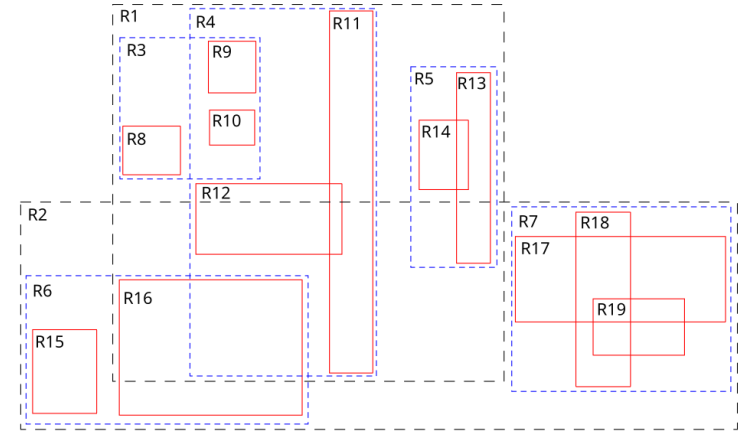
The key difficulty of the algorithm for R-tree is to build an efficient tree that:

- is balanced (so the leaf nodes are at the same height)
- the rectangles do not cover too much empty space
- the rectangles do not overlap too much (so that during search, fewer subtrees need to be processed)

Variants: M-Trees, using circles / spheres



POLITECNICO  
MILANO 1863





POLITECNICO  
MILANO 1863

# Vector Matching Efficiency and Scalability

Step 2 - Approximation

# Speeding Up Similarity Search – Episode 2

For high-dimensional data, also tree-based exact vector search techniques such as the k-d tree and R-tree do not perform well enough because of the curse of dimensionality

Vector databases use **Approximate Nearest Neighbor algorithms** for speeding up the match for large datasets of high-dimensionality data

# Examples of optimized (approximate) matching algorithms



POLITECNICO  
MILANO 1863

- **Locality-sensitive hashing (LSH)** is a fuzzy hashing technique that hashes similar input items into the same "buckets" with high probability
- **Hierarchical Navigable Small World (HNSW)**: constructs a hierarchical graph structure to index high-dimensional vectors, to quickly store and search high-dimensional vectors with minimal memory usage.  
Video: [HNSW for Vector Search Explained](#)
- **Inverted File with Approximate Distance Calculation (IVFADC)**: utilizes an inverted index structure to index high-dimensional vectors. It is known for its fast search speed and ability to handle large-scale datasets.
- **Inverted File with Product Quantization (IVFPQ)**: uses product quantization to compress high-dimensional vectors before indexing. This results in a high-accuracy search capability for massive datasets.

# LSH

**locality-sensitive hashing (LSH)** is a fuzzy hashing technique that hashes similar input items into the same "buckets" with high probability

The number of buckets is much smaller than the universe of possible input items

Similar items end up in the same buckets

LSH can be used for data clustering and nearest-neighbor search

It differs from conventional hashing techniques in that hash collisions are maximized, not minimized.

LSH can be seen as a way to reduce the dimensionality of high-dimensional data

# Hierarchical Navigable Small World (HNSW)

An efficient algorithm for approximate nearest neighbor (ANN) search in high-dimensional spaces, i.e., approximate k-nearest neighbor search.

**Navigable Small-World Graphs:** graphs where most nodes can be reached from any other in a small number of hops. In HNSW, data points (vectors) are treated as nodes in a graph.

Nodes close to each other in the vector space (i.e., similar vectors) are connected, forming a graph structure.

**Hierarchical Structure:** HNSW organizes the graph into multiple layers, each containing fewer nodes than the layer below.

The highest layer consists of the most representative points (like hubs), and as you go down the hierarchy, more detailed points are added, eventually representing all the data at the lowest level.

# Hierarchical Navigable Small World (HNSW)



POLITECNICO  
MILANO 1863

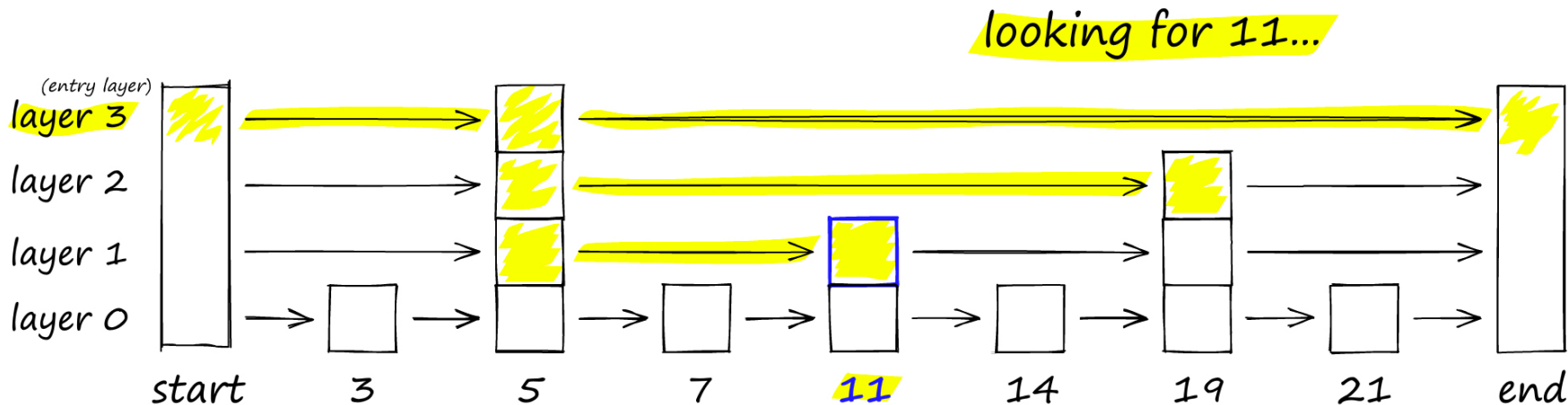
## Efficient Search Process:

When a query is made, the algorithm begins by searching for the nearest neighbor in the top layer, which contains only a few nodes.

Once a good candidate is found in this layer, the search moves down to the next layer and continues, refining the result by looking at more nodes.

Each layer contains more nodes than the previous one, allowing for a more precise search as you move down.

It can find the nearest neighbor much more efficiently than searching the entire dataset directly.



# Hierarchical Navigable Small World (HNSW)

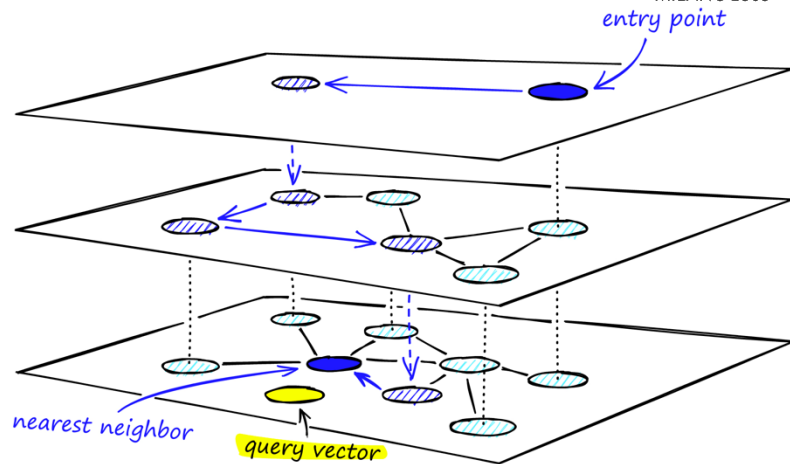


POLITECNICO  
MILANO 1863

## Approximate Nearest Neighbor:

HNSW does not guarantee an exact nearest neighbor search but instead focuses on approximate nearest neighbor (ANN) search.

It sacrifices a small amount of accuracy to achieve much faster search times



## Connections and Randomized Insertions:

New points (vectors) are inserted into the HNSW graph in a probabilistic manner.

When adding a new point to the dataset, the algorithm randomly selects a level in the hierarchy where the point should be placed.

It then connects the new point to its closest neighbors at that level.

These connections allow the graph to remain well-connected and navigable, even as new points are added.



# Advantages of HNSW

**Scalability:** The algorithm scales well to large datasets, handling millions or even billions of vectors while maintaining efficient search times.

•**Speed:** By using a hierarchical graph structure, HNSW significantly reduces the number of comparisons needed to find a similar vector, leading to fast retrieval.

•**Adaptability:** HNSW supports dynamic insertion and deletion of data points, making it suitable for real-time applications where the dataset is constantly evolving.

•**Memory Efficiency:** The algorithm is designed to use memory efficiently, allowing it to handle large-scale datasets without requiring excessive computational resources.

# Inverted File with Product Quantization (IVFPQ)

Approximate nearest neighbor (ANN), especially in high-dimensional vector spaces. It is designed to handle large datasets while keeping the search both fast and memory-efficient.

Combines **coarse quantization** (to partition the vector space) and **product quantization** (to compress vectors).

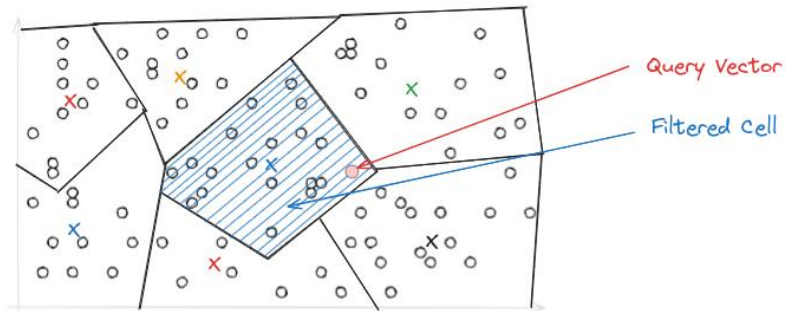
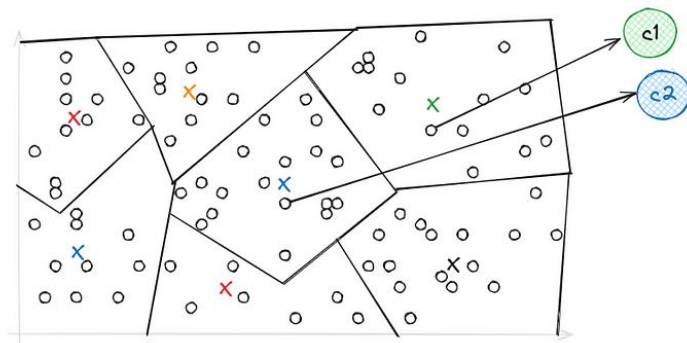
Used in vector databases for image retrieval, natural language processing, and recommendation systems.

# Inverted File with Product Quantization (IVFPQ)



POLITECNICO  
MILANO 1863

1. **Inverted Index:** inverted index, which is commonly used in search engines to map data points (or vectors) to a set of representative clusters.
2. **Coarse Quantization:** where the entire dataset is divided into clusters using a clustering algorithm like k-means. Each vector is assigned to its nearest cluster center (called a coarse centroid). These centroids represent the “bins” or regions in the space where similar vectors reside.
3. During a search, instead of searching the entire dataset, the algorithm only searches within the relevant clusters.



# Inverted File with Product Quantization (IVFPQ)



POLITECNICO  
MILANO 1863

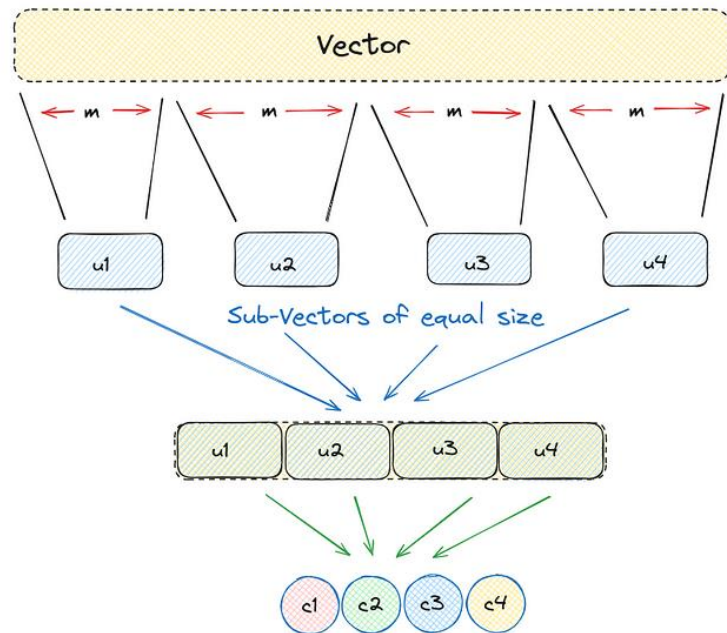
**Product Quantization (PQ):** After clustering, the algorithm applies **product quantization** to compress the vectors within each cluster.

Product quantization works by dividing each vector into smaller sub-vectors and then quantizing each sub-vector separately.

This technique reduces the size of the data, making it more memory-efficient, and speeds up the similarity search process.

1. Each sub-vector is encoded into a smaller codebook, representing it with fewer bits, which significantly reduces the storage requirements.
2. By comparing these compressed sub-vectors during the search, IVFPQ can approximate the distance between vectors with reduced computation.

**NOTE:** the reconstructed vector is not identical to the original vector. This discrepancy arises due to inherent losses during the compression and reconstruction process in all compression algorithms.



# Inverted File with Product Quantization (IVFPQ)



POLITECNICO  
MILANO 1863

**1. Search Process:** The search process in IVFPQ involves two main steps:

1. **Coarse Search:** When a query vector is presented, the algorithm first identifies the relevant clusters (coarse centroids) where similar vectors are likely to be found.
2. **Refined Search:** Within the identified clusters, product quantization is used to compare the query vector with the compressed representations of the vectors in those clusters. This allows the algorithm to approximate the nearest neighbors efficiently.

**2. Approximate Nearest Neighbor (ANN) Search:** IVFPQ provides an approximate solution to the nearest neighbor search problem. It may not find the exact nearest neighbors.

# Inverted File with Product Quantization (IVFPQ)

## Advantages of IVFPQ:

- **Memory Efficiency:** The use of product quantization allows IVFPQ to store high-dimensional vectors in a compressed format, reducing memory usage significantly.
- **Fast Search:** By organizing vectors into clusters and only searching within relevant clusters, IVFPQ speeds up the search process compared to brute-force methods.
- **Scalability:** IVFPQ scales well to very large datasets, making it useful in applications like search engines, image retrieval, and recommendation systems.
- **Customizable Accuracy:** IVFPQ allows a trade-off between speed and accuracy. Users can adjust parameters (like the number of clusters or the precision of quantization) to find the right balance for their specific application.



POLITECNICO  
MILANO 1863

# Systems

# Systems Categorization

There are a few alternatives to choose from.

- Stand-alone, proprietary, fully vectorized databases such as Pinecone.
- Open-source solutions such as Weaviate or Milvus, which provide built-in RESTful APIs and support for Python and Java programming languages.
- Platforms with vector database capabilities integrated, such as IBM watsonx.data, or similar by Microsoft and others
- Vector database and database search extensions such as PostgreSQL's open source pgvector extension, which provides vector similarity search capabilities. An SQL vector database can combine the advantages of a traditional SQL database with the power of a vector database
- Libraries for vector search and comparisons, like FAISS





POLITECNICO  
MILANO 1863

# FAISS

# FAISS

Faiss is a library — developed by Facebook AI — that enables efficient similarity search.

Given a set of vectors, we can index them using Faiss — then using another vector (the query vector), we search for the most similar vectors within the index.

FAISS is highly optimized for both accuracy and performance, making it one of the most popular tools for approximate nearest neighbor (ANN) search.

# FAISS Features

**Efficient Similarity Search:** FAISS is primarily used for performing fast and efficient similarity search between high-dimensional vectors. It can quickly find the closest (most similar) vectors to a given query vector, which is useful in applications like image search, document retrieval, and recommendation systems.

**Large-Scale Handling:** FAISS is specifically designed to handle massive datasets with millions or even billions of vectors. It is optimized to work with both in-memory and out-of-core (disk-based) datasets, ensuring scalability even with extremely large collections of vectors.

**Approximate Nearest Neighbor Search:** FAISS provides various algorithms for **approximate nearest neighbor (ANN)** search, allowing for fast search times with a small trade-off in accuracy. ANN search is essential when working with large datasets because exact nearest neighbor search can be too slow and computationally expensive.

**Product Quantization (PQ) and Inverted Index:** FAISS supports advanced techniques like **product quantization (PQ)** and **inverted indexing**, which allow for efficient storage and fast retrieval of vectors. Product quantization compresses high-dimensional vectors into smaller representations, reducing memory usage while preserving similarity information. This technique is particularly useful for handling large datasets.

# FAISS Features

**Support for Various Similarity Metrics:** FAISS supports different similarity metrics, such as **Euclidean distance** and **cosine similarity**, making it flexible for different types of data and use cases. Users can choose the metric that best suits their specific application.

**Highly Optimized for Modern Hardware:** FAISS is optimized for both **CPU** and **GPU** processing. It can leverage multi-threading and SIMD (Single Instruction, Multiple Data) instructions on CPUs to speed up search processes. It also supports **GPU acceleration** using CUDA, which can drastically reduce search times for very large datasets.

**Indexing Structures:** FAISS provides a range of indexing structures to optimize the search process, depending on the size and type of dataset. Some of the popular indexing structures include:

- **Flat Index:** A simple structure where all vectors are stored and searched directly, used for exact nearest neighbor search.
- **IVF (Inverted File Index):** A structure that organizes vectors into clusters to narrow down the search space, often used in conjunction with product quantization.
- **HNSW (Hierarchical Navigable Small World):** A graph-based structure that allows fast ANN search through a hierarchical graph of vectors.

**Clustering:** FAISS can also perform efficient **clustering** of high-dimensional data, making it useful for tasks like vector quantization, document clustering, and other unsupervised learning tasks.

# FAISS Processes

**1.Indexing:** The first step in FAISS is to build an index of the dataset. Depending on the size and complexity of the dataset, different indexing structures (like Flat, IVF, or HNSW) can be used. The choice of index depends on the trade-off between speed and accuracy that the user desires.

**2.Quantization (Optional):** For large datasets, FAISS can apply product quantization (PQ) or other compression techniques to reduce the size of the vectors. This step reduces memory usage and speeds up the search process.

**3.Search:** Once the index is built, a query vector is submitted, and FAISS searches for the nearest neighbors in the index. If an approximate search is used, FAISS will return vectors that are close to the query vector, but not necessarily the exact nearest neighbors. The search process is optimized for both CPU and GPU, depending on the system's hardware.

# FAISS Features

- 1.Scalability:** FAISS can handle datasets with millions or billions of vectors, making it suitable for large-scale applications.
- 2.Speed:** With its support for GPU acceleration and advanced indexing structures, FAISS can perform searches in milliseconds, even with large datasets.
- 3.Flexibility:** FAISS offers a wide variety of indexing structures, quantization techniques, and search algorithms, allowing users to customize the solution to their needs.
- 4.Open-Source:** FAISS is free and open-source, making it accessible to developers and researchers.
- 5.Cross-Platform Support:** FAISS can be used on different platforms, including Linux, macOS, and Windows, and it can be integrated with popular programming languages like Python and C++.

# FAISS Implementation

Faiss handles collections of vectors of a fixed dimensionality  $d$ , typically a few 10s to 100s.

Collections can be stored in matrices.

We assume row-major storage, ie. the  $j$ 'th component of vector number  $i$  is stored in row  $i$ , column  $j$  of the matrix.

Faiss uses only 32-bit floating point matrices.

# Code snippets

```
import numpy as np
d = 64                                # dimension
nb = 100000                           # database size
nq = 10000                            # nb of queries
np.random.seed(1234)                 # make reproducible
xb = np.random.random((nb, d)).astype('float32')
xb[:, 0] += np.arange(nb) / 1000.
xq = np.random.random((nq, d)).astype('float32')
xq[:, 0] += np.arange(nq) / 1000.
```

We need two matrices:

- **xb** for the database, that contains all the vectors that must be indexed. Size  $nb \times d$
- **xq** for the query vectors, for which we need to find the nearest neighbors. Size  $nq \times d$



# Code snippets

```
import numpy as np
d = 64                                # dimension
nb = 100000                           # database size
nq = 10000                             # nb of queries
np.random.seed(1234)                  # make reproducible
xb = np.random.random((nb, d)).astype('float32')
xb[:, 0] += np.arange(nb) / 1000.
xq = np.random.random((nq, d)).astype('float32')
xq[:, 0] += np.arange(nq) / 1000.
```

We need two matrices:

- **xb** for the database, that contains all the vectors that must be indexed. Size  $nb \times d$
- **xq** for the query vectors, for which we need to find the nearest neighbors. Size  $nq \times d$

# Code snippets

```
import faiss                                # make faiss available
index = faiss.IndexFlatL2(d)                # build the index
print(index.is_trained)
index.add(xb)                               # add vectors to the index
print(index.ntotal)
```

There are many types of indexes, we use the simplest version that just performs brute-force L2 distance search on them: IndexFlatL2.

All indexes need to know when they are built which is the dimensionality of the vectors **d**.

Most of the indexes also require a training phase, to analyze the distribution of the vectors. For IndexFlatL2, we can skip this operation.

When the index is built and trained, two operations can be performed on the index: **add** and **search**.

# Code snippets

```
k = 4                                # we want to see 4 nearest neighbors
D, I = index.search(xb[:5], k) # sanity check
print(I)
print(D)
D, I = index.search(xq, k)           # actual search
print(I[:5])                         # neighbors of the 5 first queries
print(I[-5:])                        # neighbors of the 5 last queries
```

The basic search operation that can be performed on an index is the k-nearest-neighbor search

The result of this operation can be conveniently stored in an integer matrix of size  $n_q \times k$ , where row  $i$  contains the IDs of the neighbors of query vector  $i$ , sorted by increasing distance.

The search operation returns also a  $n_q \times k$  floating-point matrix with the corresponding squared distances.

# Code snippets

To speed up the search, it is possible to segment the dataset into pieces. We define Voronoi cells in the  $d$ -dimensional space.

At search time, only the database vectors  $y$  contained in the cell the query  $x$  falls in and a few neighboring ones are compared against the query vector.

This is done via the IndexIVFFlat index. This type of index requires a training stage, that can be performed on any collection of vectors that has the same distribution as the database vectors.

The IndexIVFFlat also requires another index, the quantizer, that assigns vectors to Voronoi cells. Each cell is defined by a centroid, and finding the Voronoi cell a vector falls in consists in finding the nearest neighbor of the vector in the set of centroids. This is the task of the other index, which is typically an IndexFlatL2.

# Code snippets

```
nlist = 100
k = 4
quantizer = faiss.IndexFlatL2(d) # the other index
index = faiss.IndexIVFFlat(quantizer, d, nlist)
assert not index.is_trained
index.train(xb)
assert index.is_trained

index.add(xb) # add may be a bit slower as well
D, I = index.search(xq, k) # actual search
print(I[-5:]) # neighbors of the 5 last queries
index.nprobe = 10 # default nprobe is 1, try a few more
D, I = index.search(xq, k)
print(I[-5:]) # neighbors of the 5 last queries
```

There are two parameters to the search method:

nlist, the number of cells

nprobe, the number of cells (out of nlist) that are visited to perform a search.

The search time roughly increases linearly with the number of probes plus some constant due to the quantization.



# Further Details

The index can be saved on disk using the `write_index()` function and can be loaded later using the `read_index()` function

## Seamless use of GPUs

```
res = faiss.StandardGpuResources() # use a single GPU
```

# References

<https://github.com/facebookresearch/faiss/wiki>

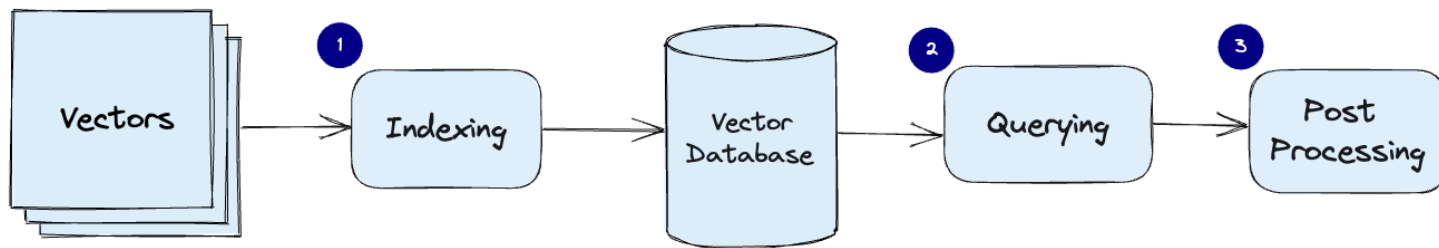


POLITECNICO  
MILANO 1863

# Pinecone



# Database architecture for Vector DB



**1. Indexing:** The vector database indexes vectors using an algorithm such as PQ, LSH, or HNSW to a data structure that will enable faster searching.

**2. Querying:** The vector database compares the indexed query vector to the indexed vectors in the dataset to find the nearest neighbors (applying a similarity metric used by that index)

**3. Post Processing:** The vector database post-processes the results to return the final results. This step can include re-ranking the nearest neighbors using a different similarity measure.

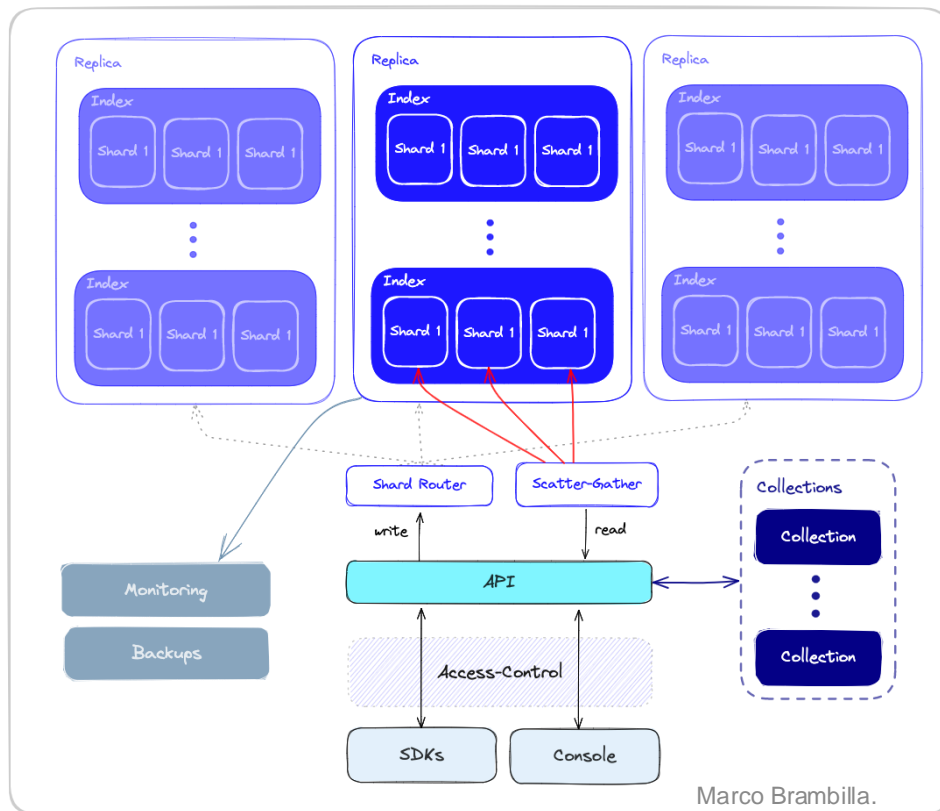
# Features

Unlike vector indexes, vector databases are equipped with a set of capabilities that makes them better qualified to be used in high-scale production settings:

- Distribution
- Scalability
- Access control
- Fault-tolerance
- Monitoring



POLITECNICO  
MILANO 1863



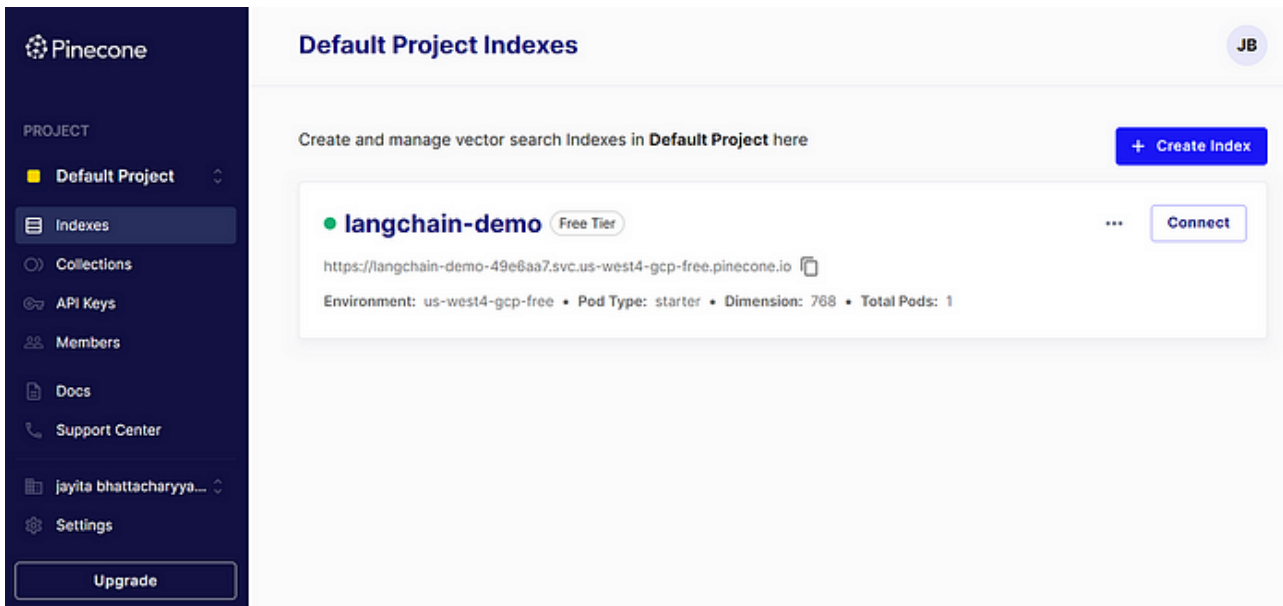
Marco Brambilla.

# Distribution and Scalability

**Sharding** - partitioning the data across multiple nodes. There are different methods for partitioning the data - for example, it can be partitioned by the similarity of different clusters of data so that similar vectors are stored in the same partition. When a query is made, it is sent to all the shards and the results are retrieved and combined. This is called the “scatter-gather” pattern.

**Replication** - creating multiple copies of the data across different nodes. This ensures that even if a particular node fails, other nodes will be able to replace it.

# Pinecone usage



**Pinecone**

PROJECT

- Default Project
- Indexes**
- Collections
- API Keys
- Members
- Docs
- Support Center
- jayita bhattacharyya...
- Settings

**Upgrade**

## Default Project Indexes

JB

Create and manage vector search Indexes in **Default Project** here

[+ Create Index](#)

- langchain-demo** Free Tier ... [Connect](#)

<https://langchain-demo-49e6aa7.svc.us-west4-gcp-free.pinecone.io>

Environment: us-west4-gcp-free • Pod Type: starter • Dimension: 768 • Total Pods: 1

# Pinecone usage

[← Back to Indexes](#)

## Create a new Index

Give it a name

Name

Configure your Index

The dimensions and metric depend on the model you select. Learn more in our [docs](#).

Dimensions \*  Metric \*




No monthly cost, included in:  
**Starter Plan**

[Cancel](#) [Create Index](#)

# Pinecone usage

**Default Project API Keys** JB

Create and manage API Keys in **Default Project** here + Create API Key

Name	Environment	Value	Actions
default	us-west4-gcp-free	*****-****-*****	  

# Pinecone coding

Import text

Initialize

Select index

Load data

```
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter
import pinecone
from langchain.vectorstores import Pinecone
from langchain.embeddings import HuggingFaceEmbeddings
```

```
# Initializing Pinecone Vector DB
pinecone.init(
    api_key=PINECONE_API_KEY,
    environment=PINECONE_ENV
)
```

```
# Pinecone Vector DB index name
index_name = 'langchain-demo'
index = pinecone.Index(index_name)
```

```
loader = TextLoader("PLACE FILE PATH HERE")
docs = loader.load()
```

# Pinecode coding

Split text in chunks

Vectorize

Similarity search

```
text_splitter = CharacterTextSplitter(  
    chunk_size=1000,      # Specify chunk size  
    chunk_overlap=200,    # Specify chunk overlap to prevent loss of informa  
)
```

```
docs_split = text_splitter.split_documents(docs)
```

```
embeddings = HuggingFaceEmbeddings()
```

```
# create new embedding to upsert in vector store  
doc_db = Pinecone.from_documents(  
    docs_split,  
    embeddings,  
    index_name=index_name  
)
```

```
query = "PLACE USER QUERY HERE"
```

```
# search for matched entities and return score  
search_docs = doc_db.similarity_search_with_score(query)
```





POLITECNICO  
MILANO 1863

# Zilliz – *Milvus*

# Zilliz

## Zilliz:

- A leading technology company focused on unstructured data management and AI-powered analytics.
- Mission: Revolutionize the way enterprises manage large, high-dimensional datasets (e.g., images, audio, video, text).
- Provides scalable, efficient solutions for data processing, aiding AI and machine learning-driven decision-making.

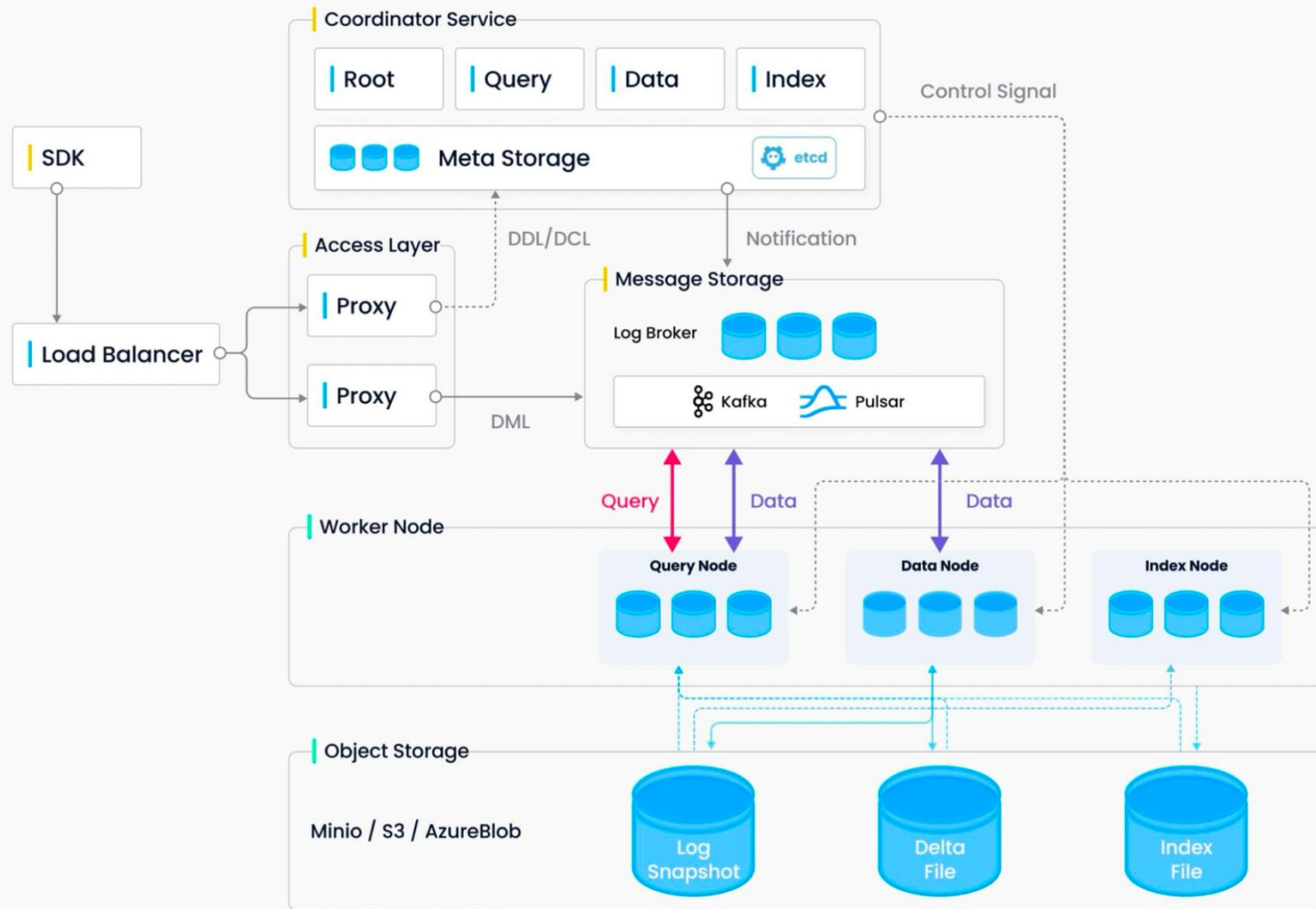
# Milvus

## Milvus:

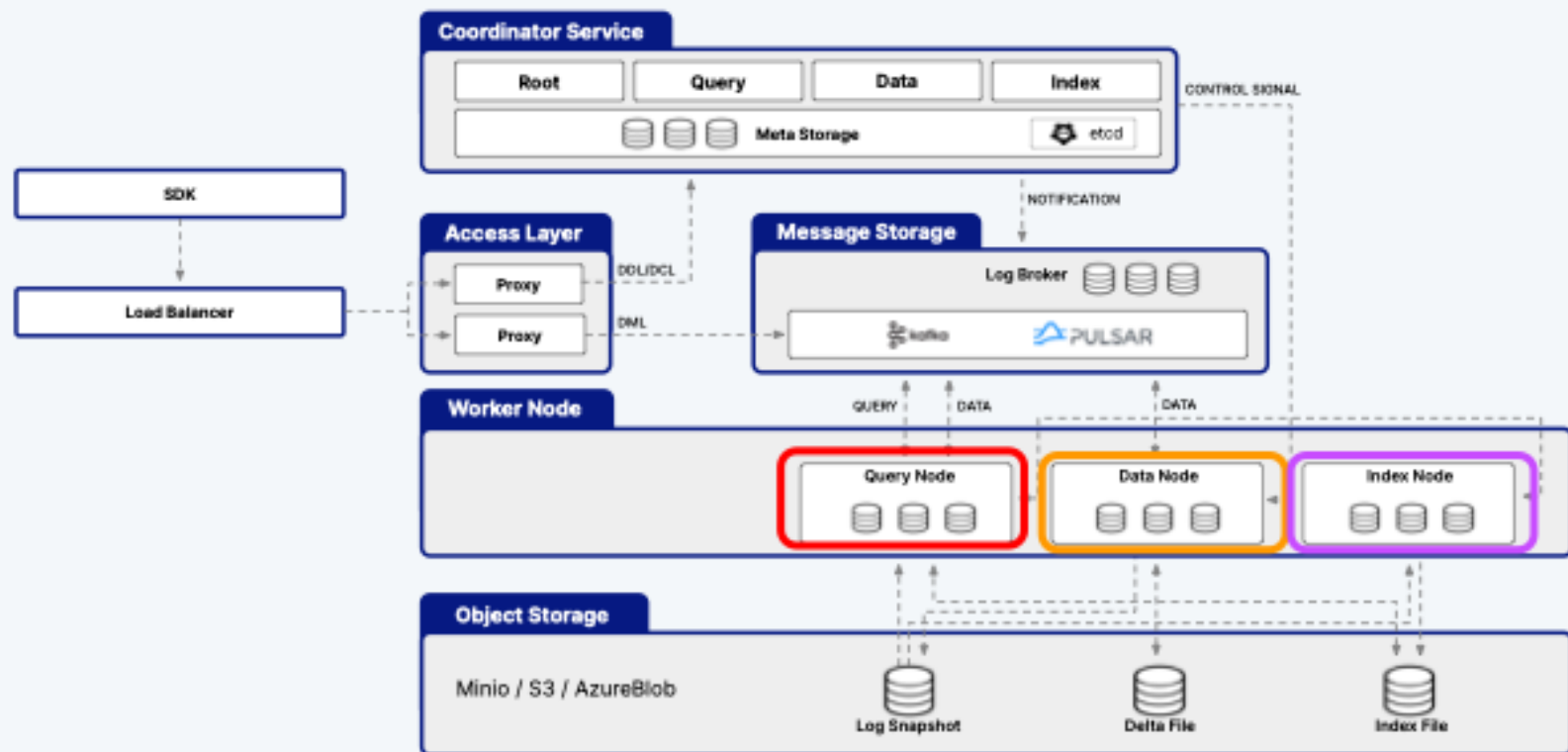
- An open-source vector database developed by Zilliz.
- Specialized in handling large-scale, high-dimensional data for similarity search and unstructured data retrieval.
- Seamlessly integrates with AI models and applications.
- Optimized for industries such as e-commerce, healthcare, and finance.
- Designed for scalability, offering performance from small projects to enterprise-level deployments.

# Design Principles

- Disaggregate storage and computation
- Fully depends on mature storage systems
- Micro Service – scale by functionality
- Separate Streaming and historical data
- Pluggable engine, storage and index
- Log As data



# Fully Distributed Architecture



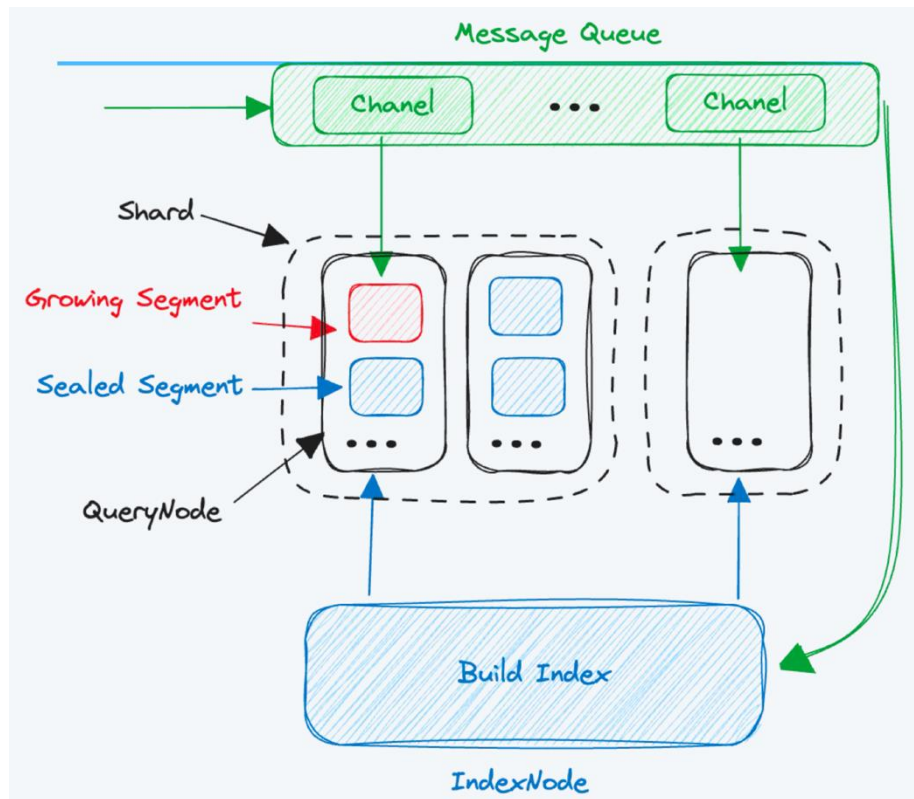
# Sharding architecture



POLITECNICO  
MILANO 1863

Each **shard** is managed by a supervisor (shard leader). This supervisor is responsible for:

- Adding new information to the shard.
- Regularly storing the data in a safe place (object storage).
- Serving the latest information for search requests.
- Forwarding historical data requests to other cabinets (query nodes) if needed.



# Sharding architecture



POLITECNICO  
MILANO 1863

## Growing Segment:

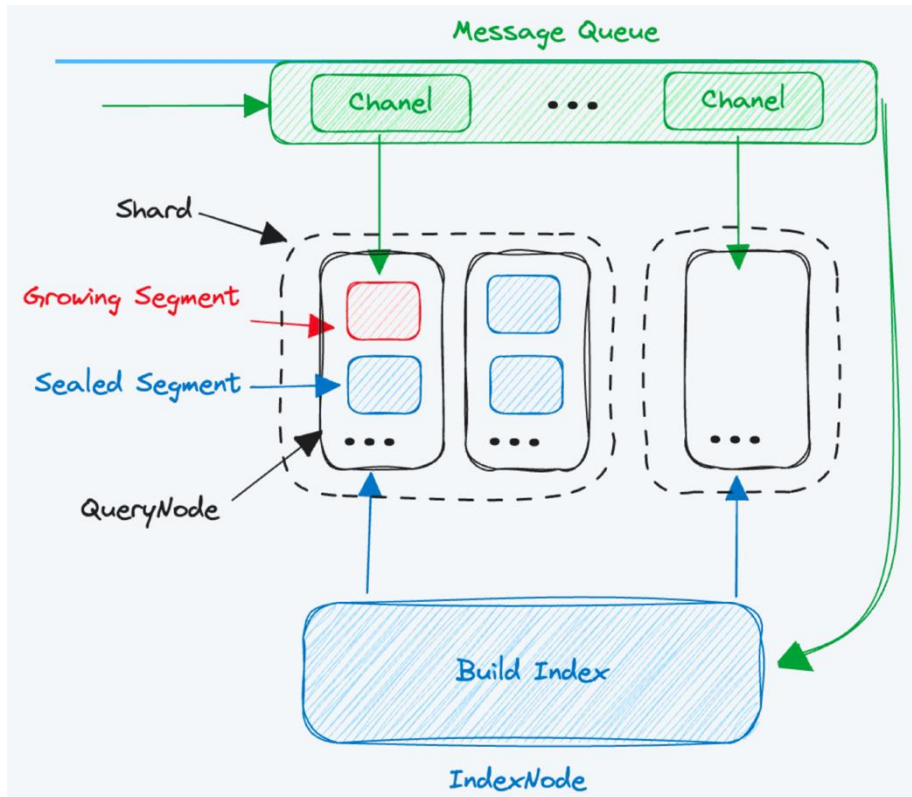
- In-memory segment replaying data from the Log Broker.
- Uses a FLAT index to ensure data is fresh and appendable.

## Sealed Segment:

Immutable segment using alternative indexing methods for efficiency.

## Write-Ahead Log (WAL):

When you add new data, a proxy service writes it to a temporary log called a WAL (e.g., Kafka, Pulsar). Think of it as a to-do list for the datanodes.







POLITECNICO  
MILANO 1863

# Neo4J

# Focus of Neo4J



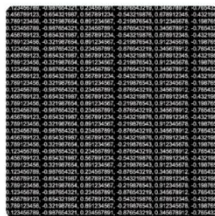
POLITECNICO  
MILANO 1863

Emphasis on integration of  
vector + semantic graph  
search

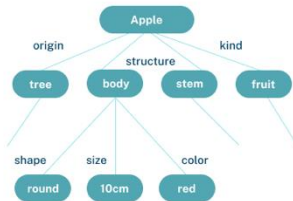
Human View of  
an Apple



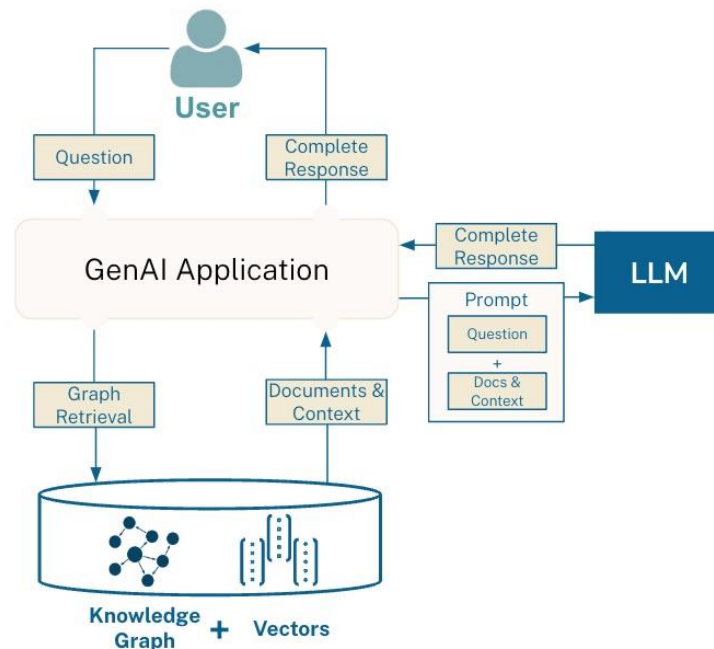
Vector View of  
an Apple



Knowledge Graph  
View of an Apple



Paves the way to  
neurosymbolic AI



# Neo4J

has integrated native vector search into its core database capabilities.

results achieve richer insights from semantic search and generative AI applications and serve as long-term memory for LLMs, all while reducing hallucinations.



POLITECNICO  
MILANO 1863

# Concluding

# Conclusion

Vector databases are essential for AI/ML-driven data retrieval and semantic search, like chatbots and RAG

They provide a foundation for advanced applications like recommendation engines and contextual search

They incorporate efficient algorithms for matching and retrieval (possibly approximated!)

They can be native or integrated in other DB technologies

# Ack

Pinecone

Zilliz

Facebook FAISS

Phaneendra Kumar Namala



**POLITECNICO**  
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

# Vector Databases

Marco Brambilla

marco.brambilla@polimi.it

 @marcobrambi