Billion vector baby!

Amine Gani

Roudy Khoury

2025-04-23

#HaystackConf

@a2lean

Who are we?

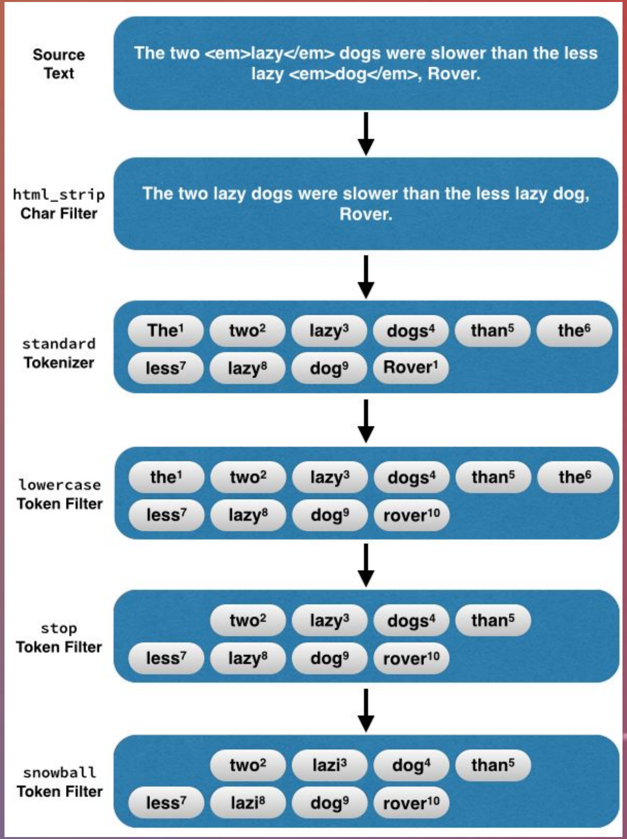
Adelean

- Experts in **search** technologies
- Integrators of **Elasticsearch**, **OpenSearch** and **Solr**
- **Consulting** and **Training** providers
- Developers of **a2** E-Commerce and Enterprise Search solution
- Developers of **all.site** - your **Collaborative** Search Engine

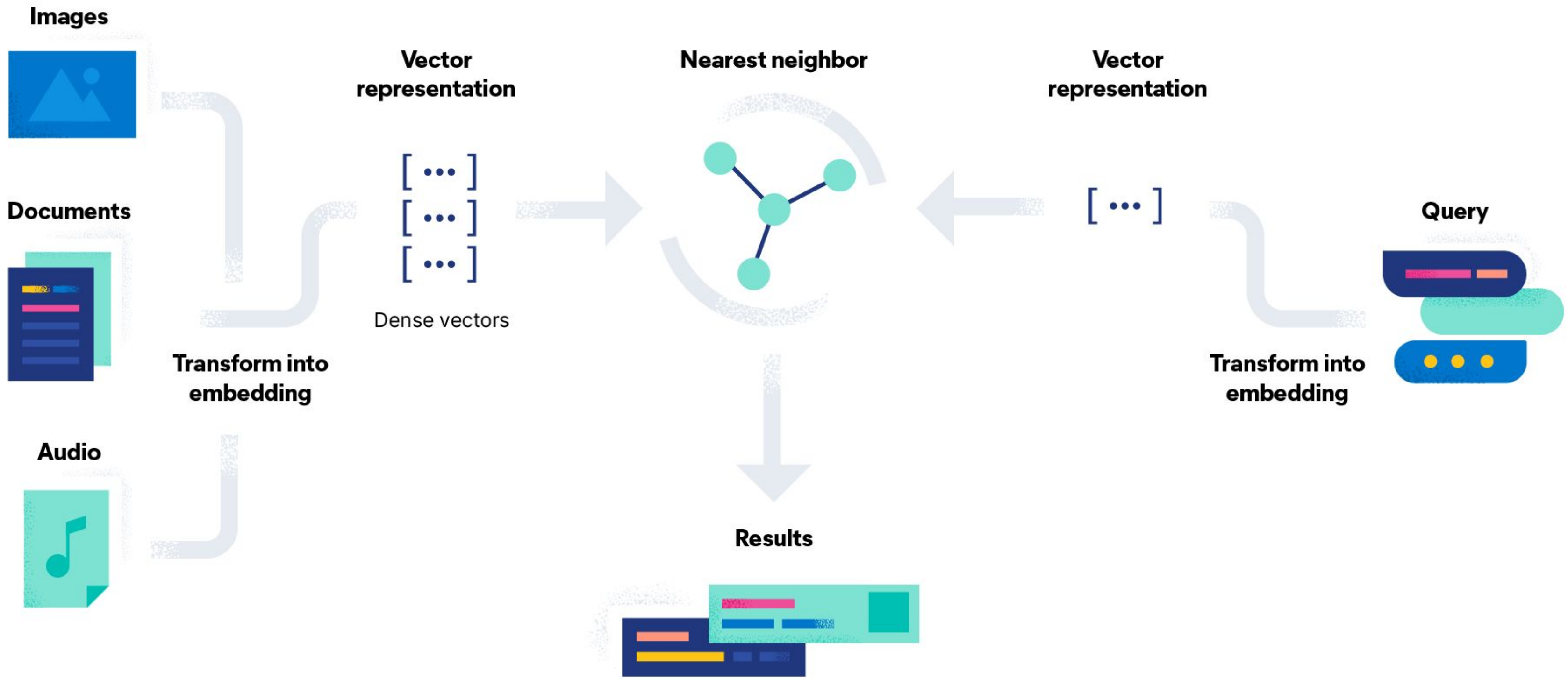
A Bit of Context:
Lexical Search vs Semantic Search

Lexical Search

- Keyword based
- Limited context
- Requires advanced configuration:
  - Stemming
  - Synonyms
  - Lemmatization
- Low cost



Semantic search



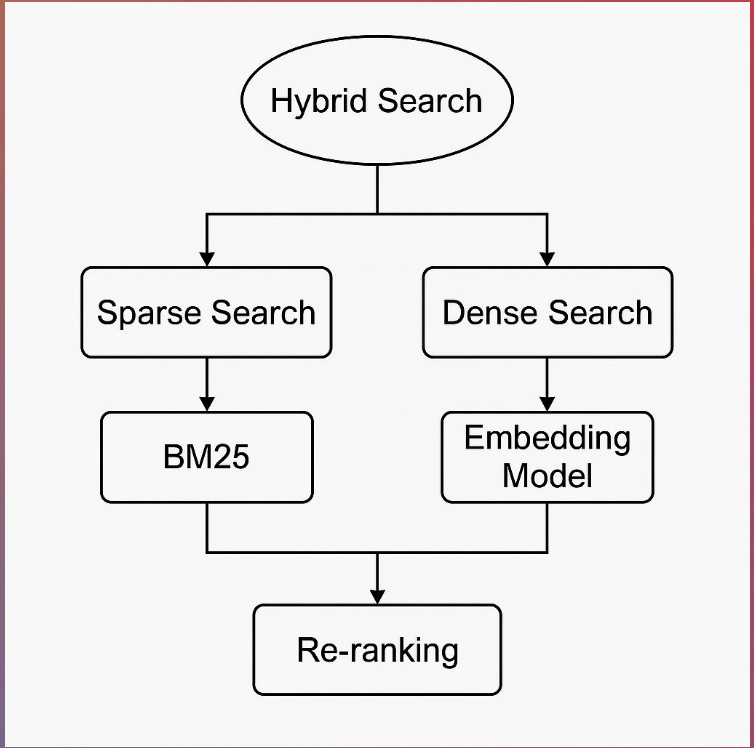
https://www.elastic.co/fr/what-is/vector-search

Hybrid search

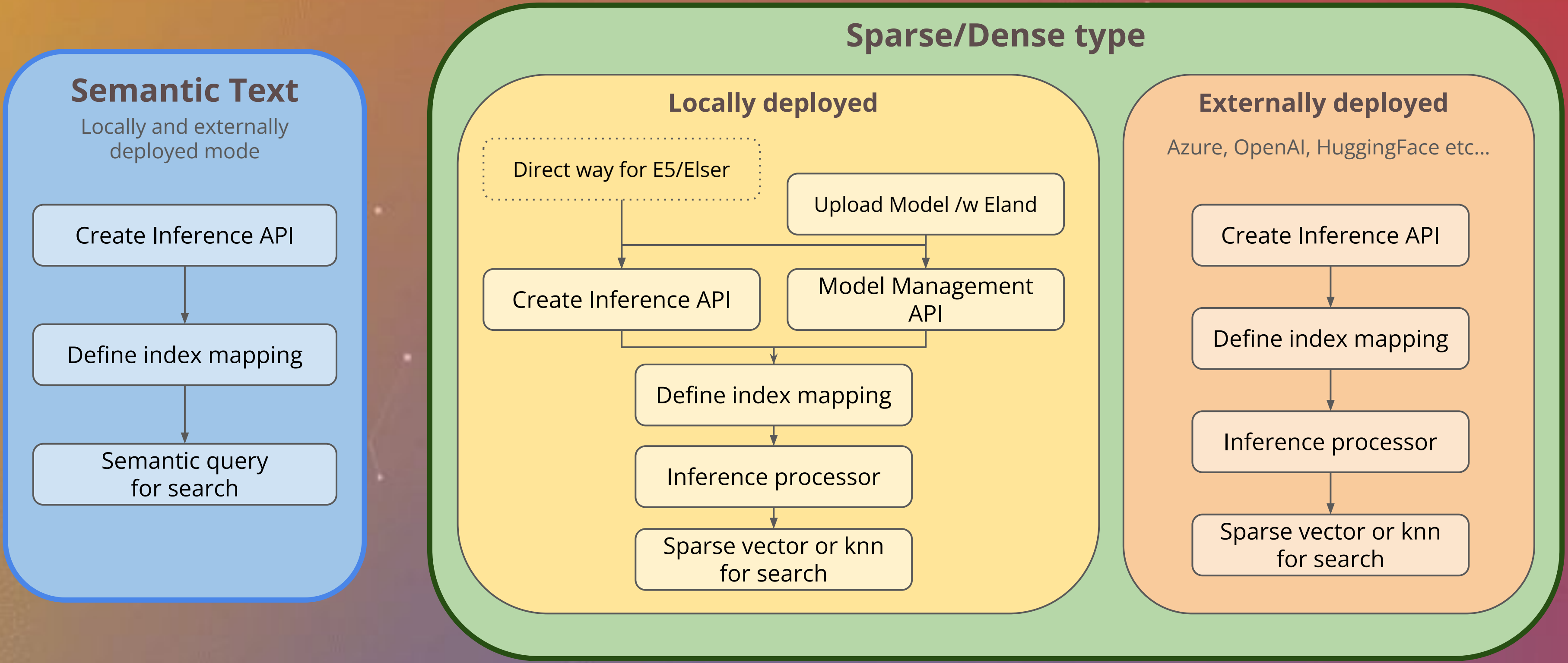
Best of both worlds

- Sparse vector for fast recall
- Then rerank using dense similarity

1. Get top 100 docs with TF-IDF, BM25…
2. Compute similarity with dense vectors (cosine, dot product…)
3. Rerank results



Semantic search: in practice



In this presentation, we'll mainly focus on dense vectors

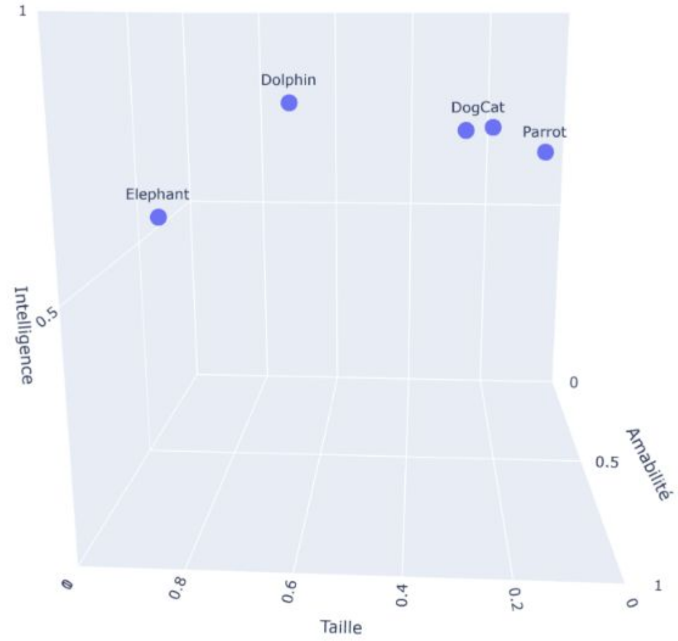
Vectorization in a three-dimensional vector

Animal	Size	Friendliness	Intelligence
Cat	0.25	0.85	0.80
Dog	0.30	0.90	0.80
Elephant	0.90	0.70	0.60
Dolphin	0.60	0.95	0.85
Parrot	0.15	0.80	0.75

https://www.adelean.com/en/blog/20240131_vectors_sparse_and_dense/

Vectorization in a three-dimensional vector

Animal	Size	Friendliness	Intelligence
Cat	0.25	0.85	0.80
Dog	0.30	0.90	0.80
Elephant	0.90	0.70	0.60
Dolphin	0.60	0.95	0.85
Parrot	0.15	0.80	0.75

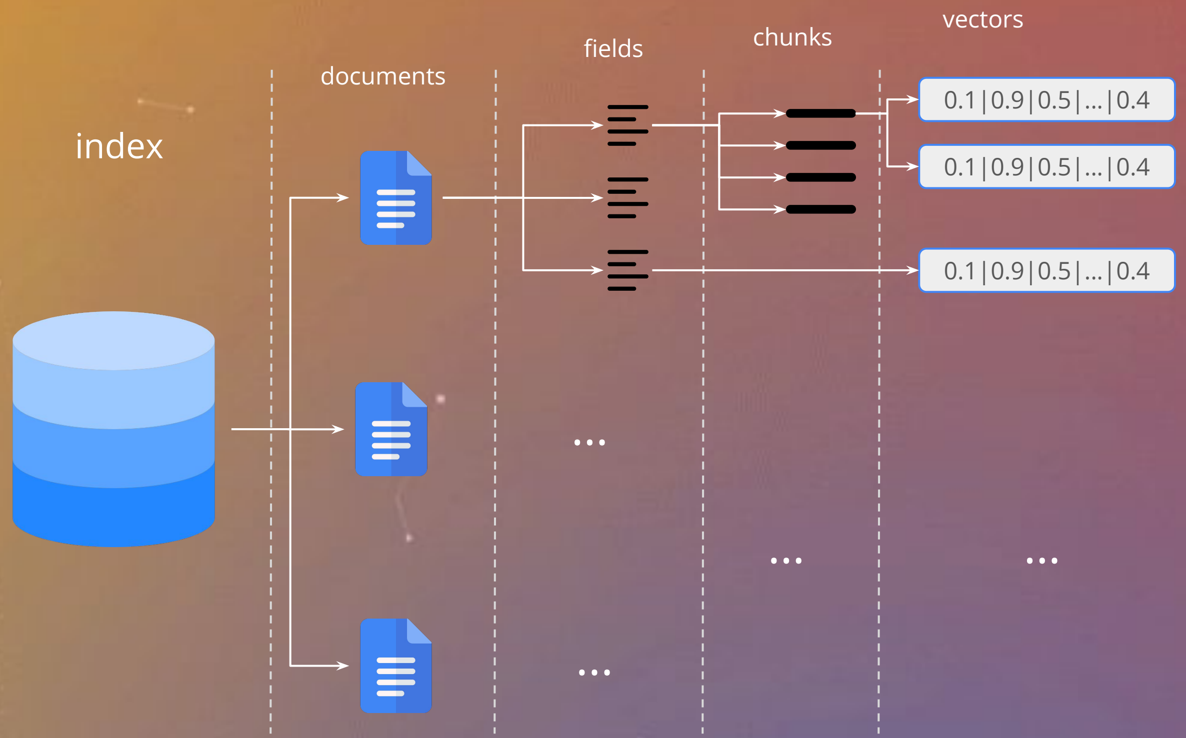


https://www.adelean.com/en/blog/20240131_vectors_sparse_and_dense/

Much more than 3 dimensions

Rank	Model	Model Size (Million Parameters)	Memory Usage (GB, fp32)	Embedding Dimensions	Max Tokens
44	e5-mistral-7b-instruct	7111	26.49	4096	32768
65	e5-mistral-7b-instruct	7111	26.49	4096	32768
78	SGPT-5.8B-weightedmean-nli-bi	5874	21.88	4096	2048
81	sgpt-bloom-7b1-msmarco	7068	26.33	4096	2048
1	bge-multilingual-gemma2	9242	34.43	3584	8192
2	gte-Qwen2-7B-instruct	7613	28.36	3584	131072
21	sentence_croissant_alpha_v0.4	1280	4.77	2048	2048
22	sentence_croissant_alpha_v0.3	1280	4.77	2048	2048
24	sentence_croissant_alpha_v0.2	1280	4.77	2048	2048

Number of vectors



The number of vectors can grow rapidly:

- Chunking strategy
- Vectorizing multiple fields
- Using multiple models

What if you need to handle 1 billion vectors?

Element type

Defined at index creation time

This choice has a huge impact on memory and storage

The available options are:

- float: single-precision floating point numbers - high precision, use more space
- byte: 8-bit integers
- bit: binary vectors

The default value is float.

```
  "vector": {
    "type": "dense_vector",
    "element_type": "byte",
    "dims": 1024,
    "index": true,
    "similarity": "cosine",
    "index_options": {
      "type": "hnsw",
      "m": 16,
      "ef_construction": 100
    }
  }
}
```

Index options type

- The type of algorithm to use

Some of the available options are:

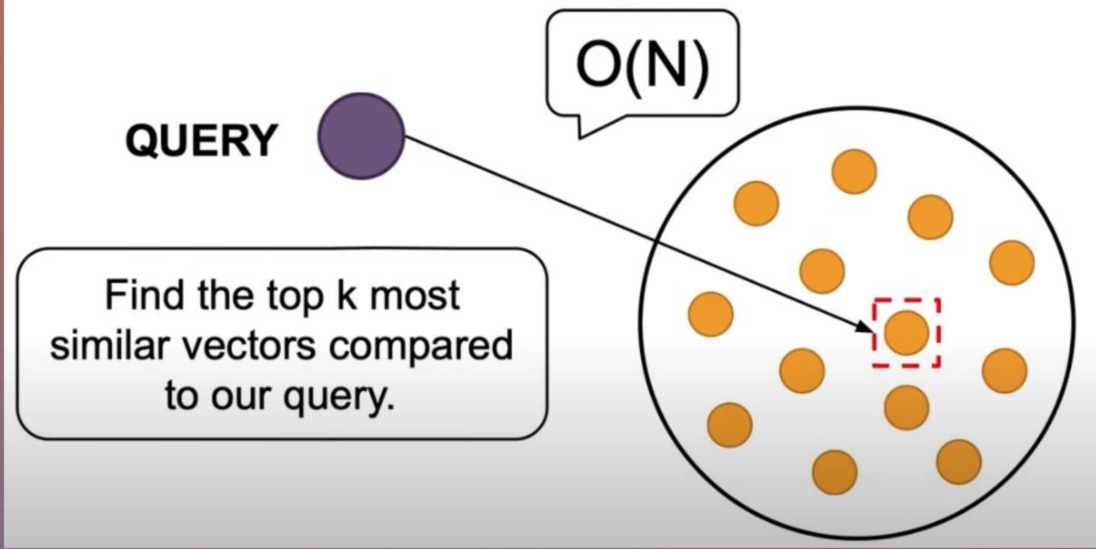
- hnsw: Hierarchical Navigable Small World — approximate nearest neighbor (aNN)
- flat: brute-force kNN search over all vectors -> not scalable at billion vector level

```
"vector": {
  "type": "dense_vector",
  "element_type": "byte",
  "dims": 1024,
  "index": true,
  "similarity": "cosine",
  "index_options": {
    "type": "hnsw",
    "m": 16,
    "ef_construction": 100
  }
}
}
```

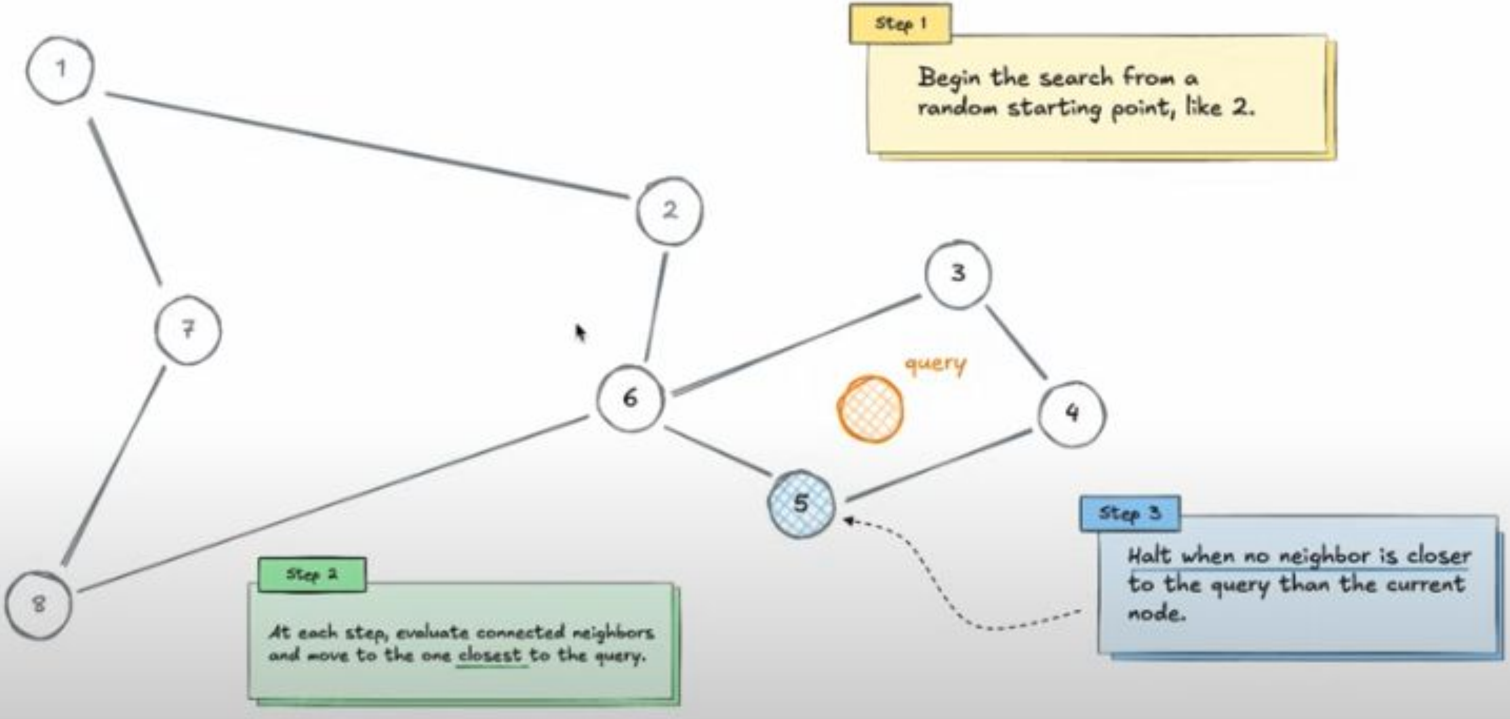
Flat indexing - KNN

- Simplest form of indexing
- Brute-force method: all vectors must be scanned to compute similarity.
- It does not scale well with large datasets.

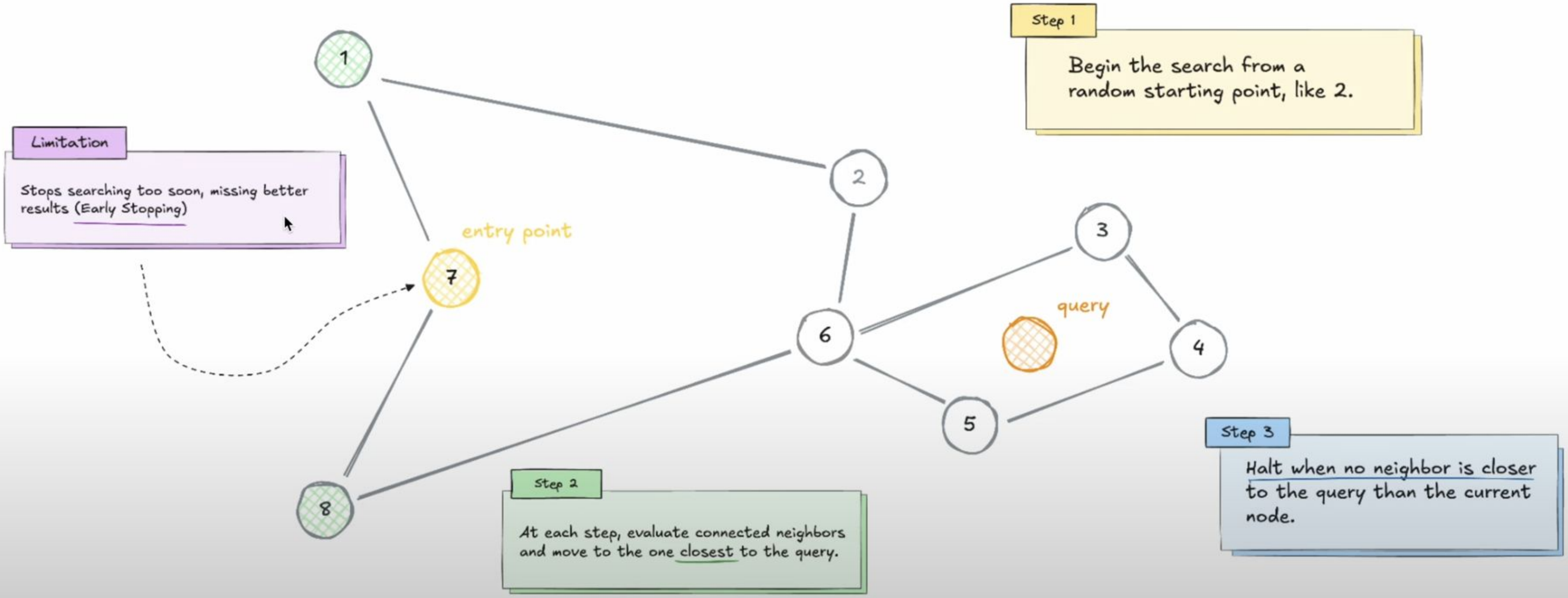
ANN methods like HNSW are often preferred for production.



Navigable Small World



Navigable Small World



Skip List

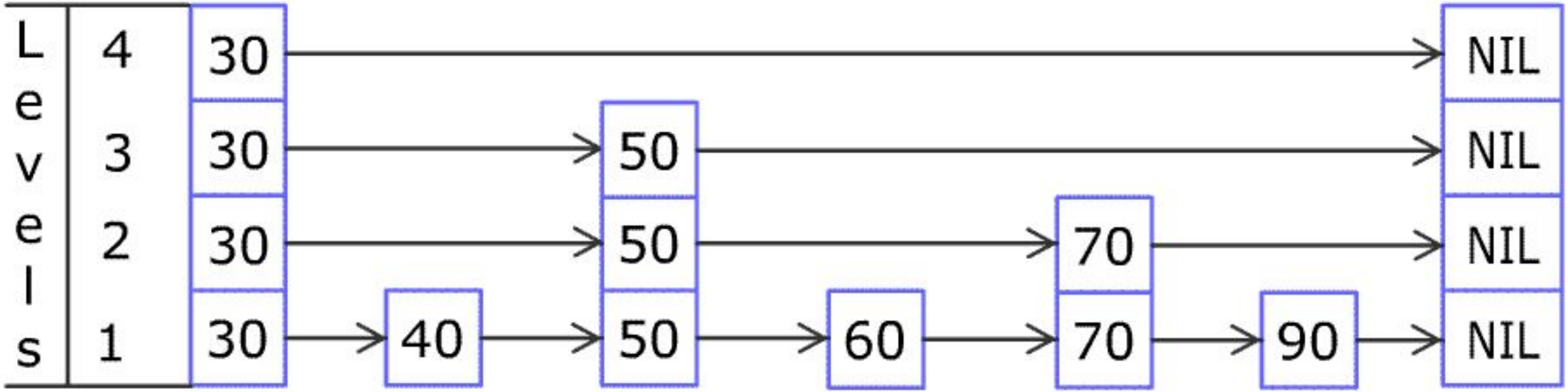
- A skip list is a data structure that allows fast search, insertion, and deletion
- Like a balanced tree, but built on top of linked lists.
- It uses multiple levels of linked lists to "skip over" elements, speeding up operations.

O(LOG n)

```
Level 3:      A --------> G
Level 2:      A ----> C ----> G
Level 1: A -> B -> C -> D -> E -> F -> G
```

- Bottom layer = normal sorted linked list.
- Each higher level skips over more elements.
- Top level has very few nodes, just enough to make fast jumps.

Skip List



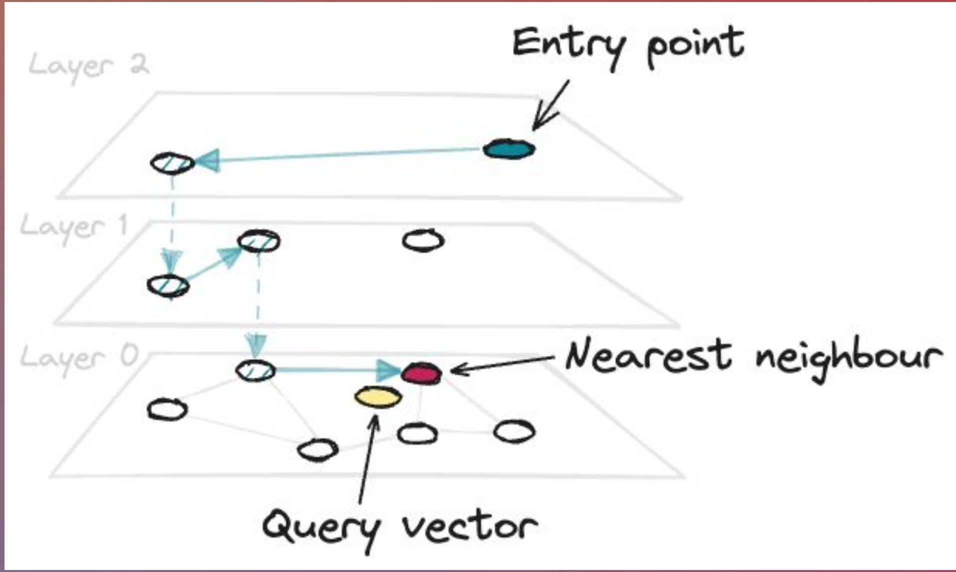
HNSW - Hierarchical Navigable Small World

Based on the mechanics of probability skip lists and Navigable Small World (NSW) graphs.

Approximate search is faster but less accurate.

A few key parameters:

- m: the number of connections between each node in the graph at a given layer
- ef_construction: the size of the candidate list during graph construction

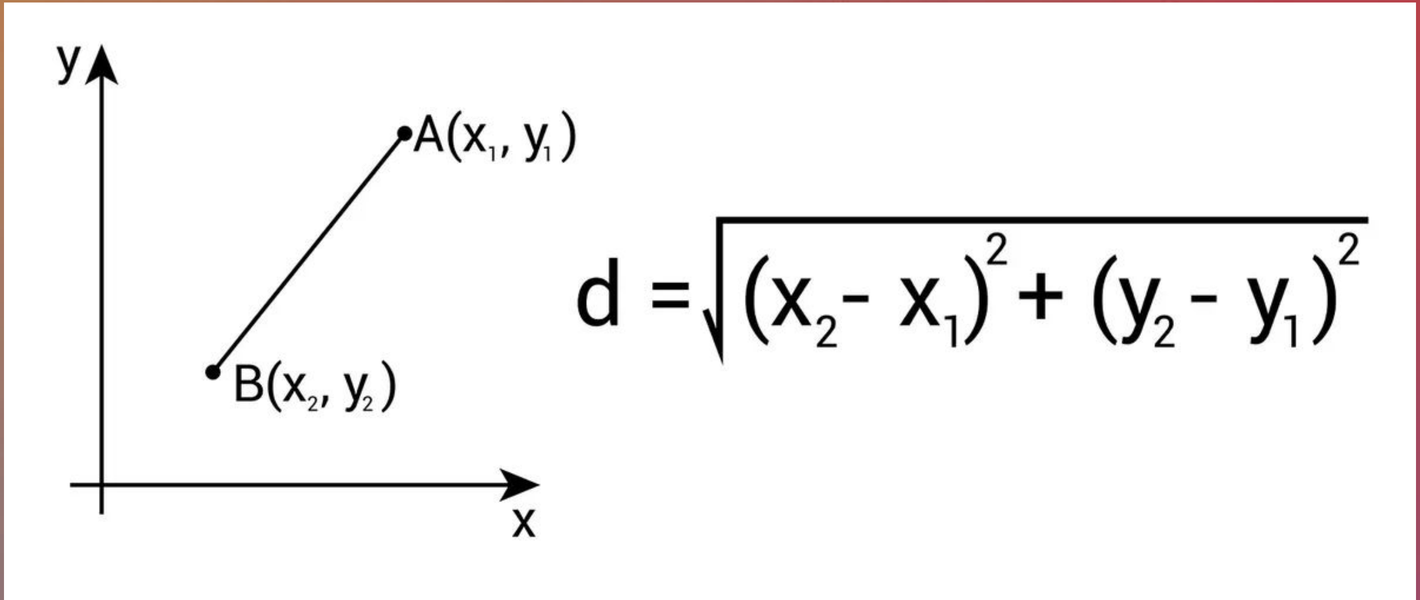


Measuring distances

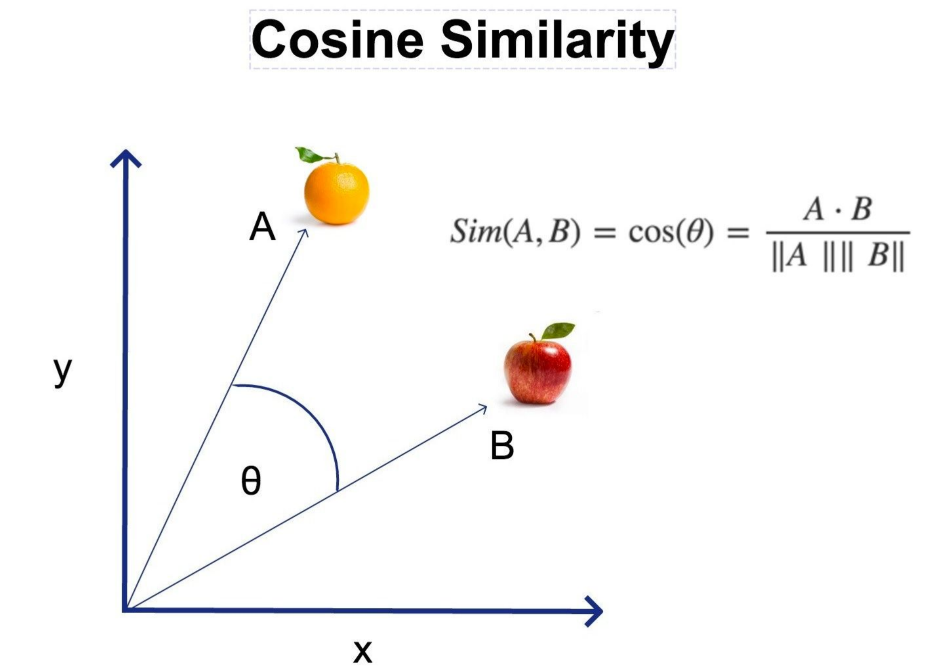
- Cosine Similarity is widely used and preferred for semantic search (texts, queries…)
- Euclidean is common in feature rich vectors
- Dot Product is also used when vectors are not normalized and we want to take into account the length of the vectors

Euclidean distance

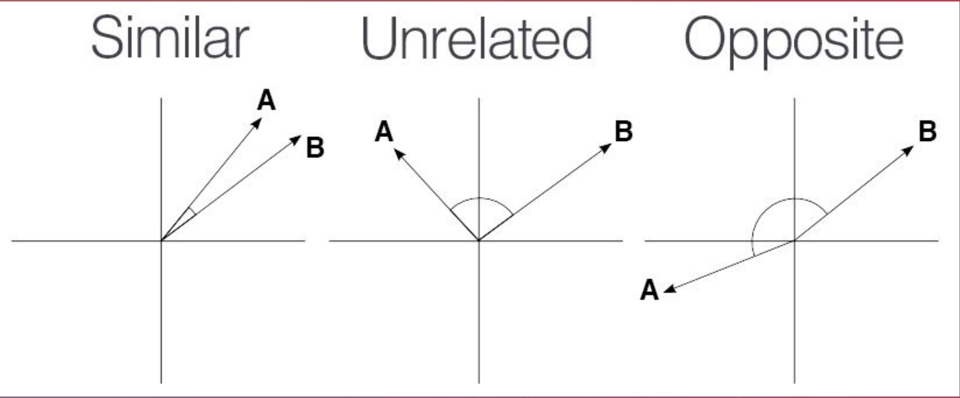
$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$



Cosine similarity



$$Sim(A,B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$



Cosine Similarity vs Euclidean Distance

Use case	Algo
Search engines, NLP, embeddings	Cosine
Feature-rich numeric datasets (images, etc.)	Euclidean
Mixed types or hybrid models	Sometimes a combination
You don't know?	Normalize & try both!

Quantization

- **Binary Quantization**
  - Fastest and most memory-efficient method
  - Up to 40x faster search speed and 32x smaller memory footprint
- **Scalar Quantization**
  - Minimal loss in precision
  - Memory footprint reduced by up to 4x
- **Product Quantization**
  - Highest compression ratio
  - Memory footprint reduced by up to 64x

Disk Memory Requirements

In the case of **Float** and **hnsw:**

Required memory = (Number of vectors × Vector size × Size of Type) + (Number of vectors * 4 * HNSW.m)

In our case :

1 billion × 1024 × 4 + 1 billion × 4 × 16 = 3.8 TB of RAM

Disk Memory Requirements

In the case of **Float** and **hnws:**

```
Required memory =  (Number of vectors × Vector size × Type)
+ (Number of vectors * 4 * HNSW.m )
```

In our case :

1 billion × 1024 × 4 + 1 billion × 4 × 16 = 3.8 TB of RAM

Better with quantization

In the case of **Float** and **hnws_int8:**

Required memory = Number of vectors × (Vector size + 4)

In our case :

1 billion × 1024 × 4 = 610 Go

Better with quantization

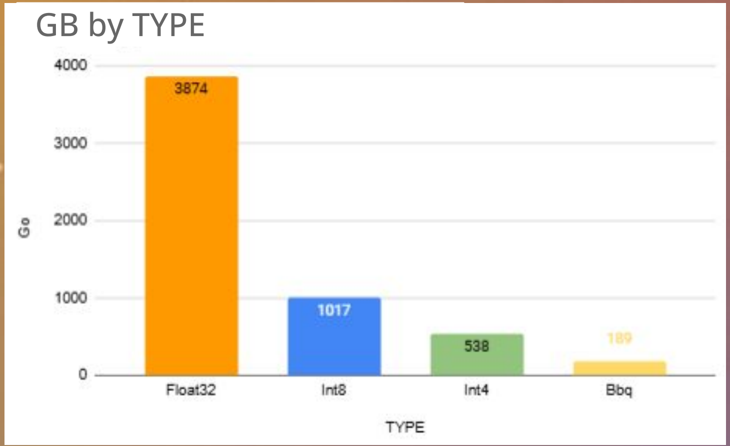
In the case of **Float** and **hnws_int8:**

Required memory = Number of vectors × (Vector size + 4)

In our case :

1 billion × 1024 × 4 = 610 Go

Better with quantization



Float32 = (Number of vectors × Vector size × Size of *Type) + (Number of vectors * 4 * HNSW.m)*

*int8 = Number of vectors × (Vector size + 4) + (Number de vectors * 4 * HNSW.m)*

*int4 = Number of vectors × (Vector size/2 + 4) + (Number de vectors * 4 * HNSW.m)*

*bbq = Number of vectors × (Vector size/8 + 12) + (Number of vectors * 4 * HNSW.m)*

Quantization methods

Method	Type	Available in Free Tier	Description
`element_type: byte`	8-bit	Yes	Lightweight, fast search; lowest memory usage but may reduce accuracy.
`element_type: bfloat16`	16-bit	Yes (from 8.12)	Balanced approach; lower memory than float32 with better accuracy than byte.
External PQ / OPQ	Preprocessing	Yes (store + search)	Quantize vectors externally; Elasticsearch stores and searches the result.
BBQ (Blockwise Quantization)	Blockwise	No (experimental only)	Prototype stage; aims for high compression with minimal loss in quality.

Quantization methods

Lucene scalar quantization

Use built-in scalar quantization for the Lucene engine

Faiss 16-bit scalar quantization

Use built-in scalar quantization for the Faiss engine

Faiss product quantization

Use built-in product quantization for the Faiss engine

Binary quantization

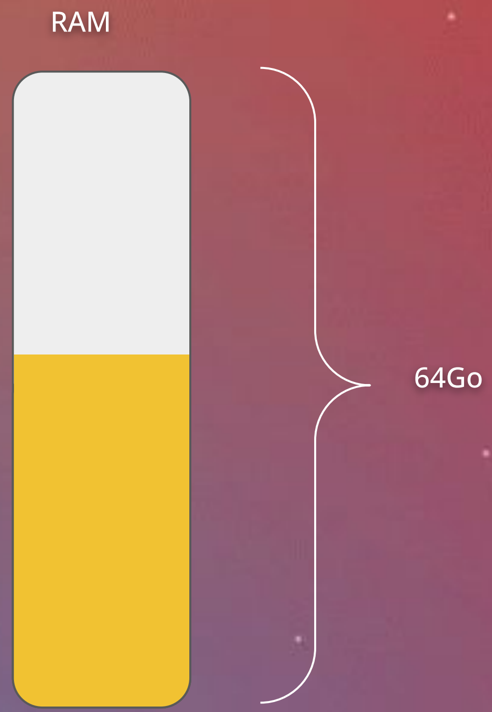
Use built-in binary quantization for the Faiss engine

Cluster configuration

- 64 Go of RAM for each node

Cluster configuration

- 64 Go of RAM for each node
- 32 Go dedicated to the JVM :
  - allows to benefit from compressed object pointers and Garbage collection issues



Cluster configuration

- 64 Go of RAM for each node
- 32 Go dedicated to the JVM :
  - allows to benefit from compressed object pointers and Garbage collection issues
- Vectors are stored off-heap, in the filesystem cache

Cluster configuration

If we simplify, we could say that the entire filesystem cache is available for our vectors.
But that's not entirely true — benchmarks are essential to understand real-world behavior!

In our case, with 610 GB of quantized int8 vectors, we need around:

- 20 data nodes
- dedicated master nodes
- coordinator nodes

and possibly ML nodes (depending on your use case).
This setup ensures enough memory and compute to support efficient search, ingestion, and model-based operations across the cluster.

Preloading vectors into the cache

This can be very useful to speed up operations after a cluster restart.

However, don't overuse it, or it might actually slow down search performance due to memory pressure.

There are different extensions depending on the type of vector being loaded:

- `vex` for HNSW graphs
- `veq` for quantized vectors
- `vec` for all non-quantized vectors

```
{
  "settings": {
    "index.store.preload":
                  ["veq", "vex"]
  }
}
```

Disk Memory Requirements with quantization

Disk memory required = Number of vectors × Size of vector × Size of Element Type + Number of vectors × Size of vector × Size of Type (quantization)

When using Lucene quantization (which is the default when element_type is set to float), both quantized and non-quantized vectors are stored within the `knn_vectors` object.

To analyze how disk space is being used, you can run `index/_disk_usage?run_expensive_tasks=true`

_source and knn

Additionally, non-quantized vectors are stored twice:

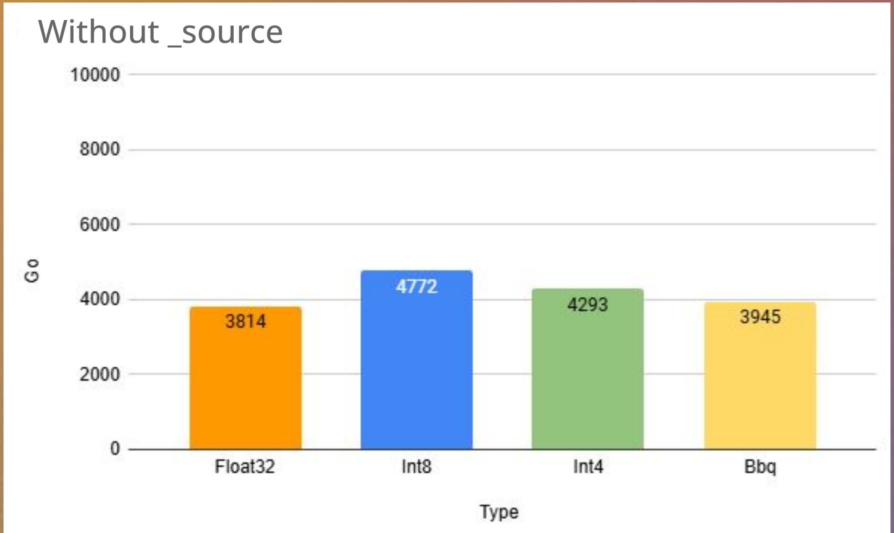
- In the knn_vector field
- In the _source field

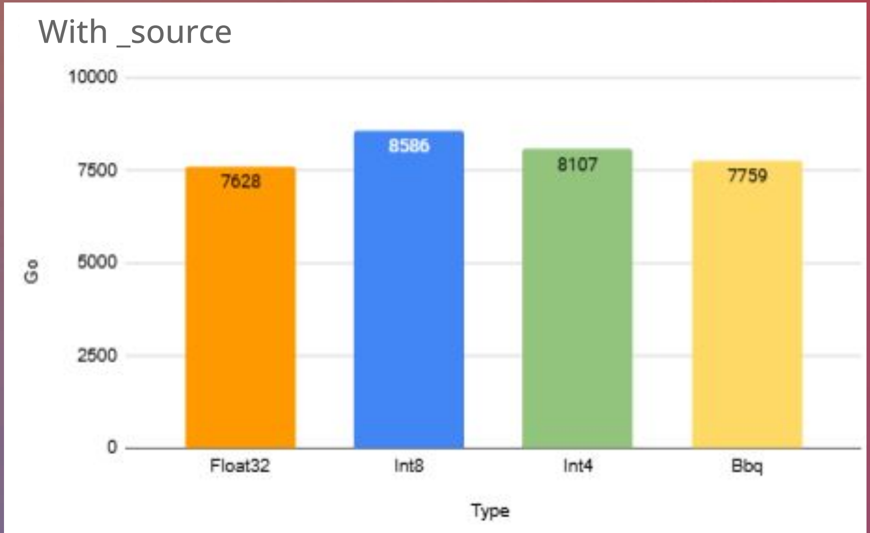
```
"mappings": {
      "_source": {
        "excludes": [
          "vectors fields"
        ]
      }
```

You can disable storing vectors in _source to save space, but this removes the ability to perform a reindex later on—so it's a trade-off between storage optimization and operational flexibility.

Disk Memory Requirements with quantization

With 1 billion vectors of 1024 dimensions





Disk Memory Requirements with qu

I can reduce it
to less than
1TB

With 1 billion vectors of 102

Memory saving with quantization

How to maximize memory savings?

- External Quantization (binary or scalar)

```
from sentence_transformers import SentenceTransformer
from sentence_transformers.quantization import quantize_embeddings

# 1. Load an embedding model
model = SentenceTransformer('Lajavaness/bilingual-embedding-large', trust_remote_code=True)

# 2a. Encode some text using "binary" quantization
binary_embeddings = model.encode(
    ["I am driving to the lake.", "It is a beautiful day."],
    precision="binary",
)
```

Element_type quantization

How to maximize memory savings?

- External Quantization (binary or scalar)
- Quantization with pipeline

```
PUT _ingest/pipeline/scalar_quantization_pipeline
{
  "description": "Pipeline to quantize to int8",
  "processors": [
    {
      "script": {
        "source": """
          def min_val = 100;
          def max_val = 0;

          for(value in ctx.vector){
            if(value < min_val) min_val = value;
            if(value > max_val) max_val = value;
          }

          def range = max_val - min_val;

          def quant_min = -128;
          def quant_max = 127;

          def quantized_vector = [];
          for (v in ctx.vector) {
            def normalized = (v - min_val) / range;
            def scaled = normalized * (quant_max - quant_min) + quant_min;
            quantized_vector.add(Math.round(scaled));
          }
          ctx.quantized_vector = quantized_vector;
        """
      }
    }
  ]
}
```

Demo

Conclusion

What have we seen ?

- Vector search can be extremely resource-intensive, but we can adopt several strategies to reduce the cost:
  - Quantization
  - Better chunking strategies
  - Excluding _source

What's next ?

- We'll explore how performance changes when RAM is insufficient.
- We'll learn how to optimize vector search using different types of modeling.

Thank you!

www.adelean.com

info@adelean.com

@a2lean

linkedin.com/company/adelean

adelean
search with all.site