OpenSearchCon EUROPE

Unlocking next-gen semantic search with OpenSearch 3.x

About us

Pietro Mele

Software Engineer @ **Adelean**
OpenSearch Ambassador

Lucian Precup

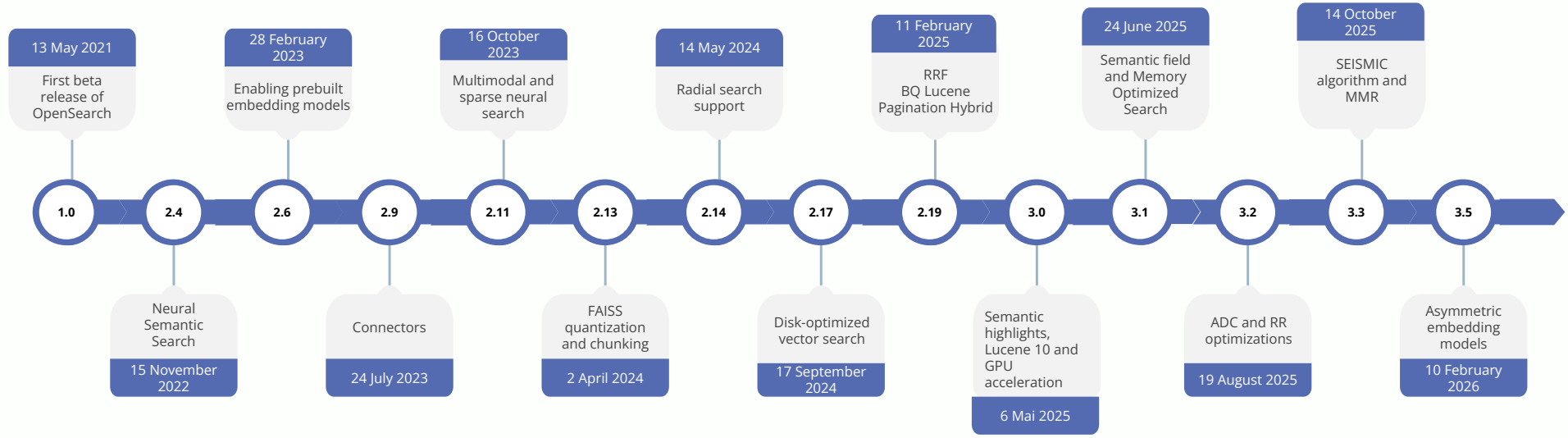
CTO @ **Adelean**

Some words about Adelean

Adelean

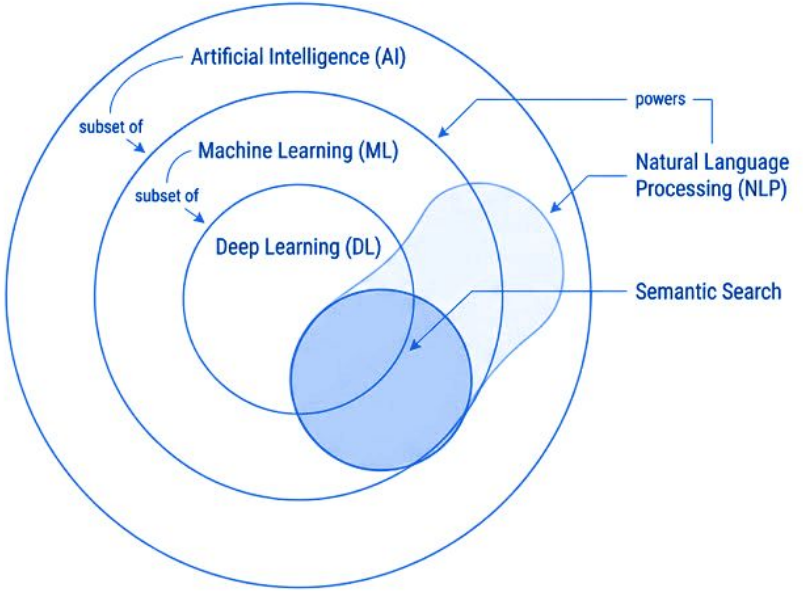
- Experts in **search technologies**
- Integrators of **Elasticsearch**, **OpenSearch** and **Solr**
- **Consulting** and **Training** providers
- Developers of **a2** - an e-commerce and Enterprise Search solution
- Developers of **all.site** - your Collaborative Search Engine

The OpenSearch AI Semantic Journey

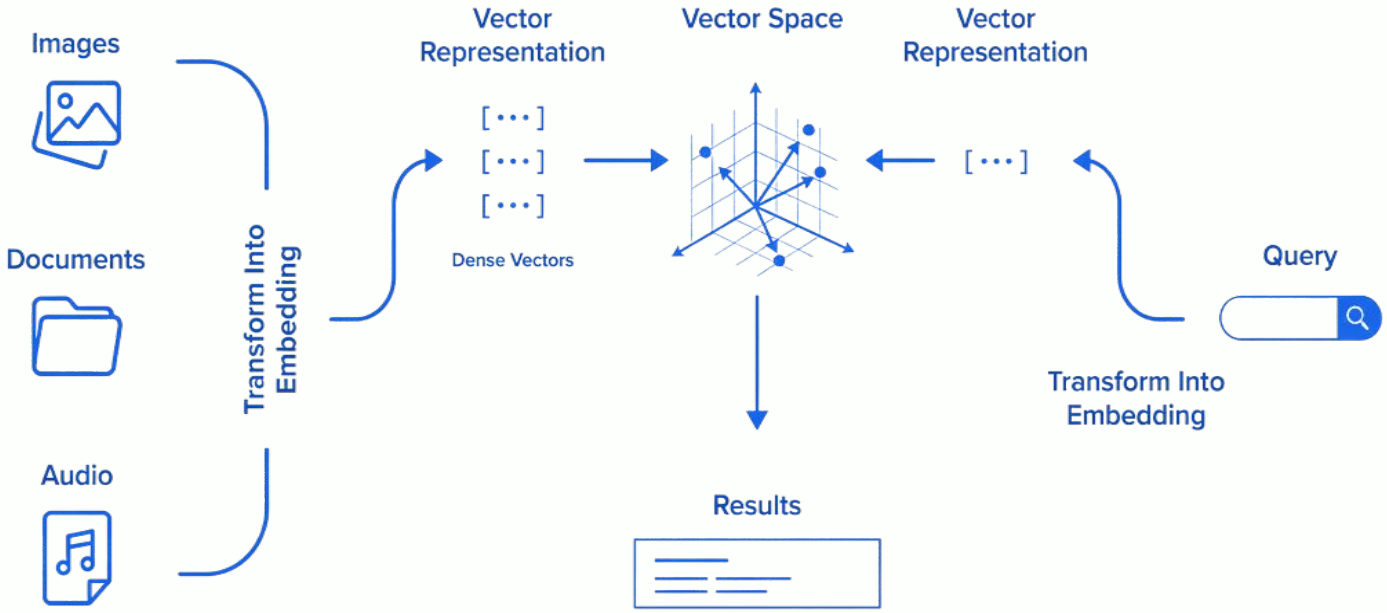


Semantic search

- Understands meaning and context, **not just keywords**
- Powered by transformer-based models
- Best suited for:
  - Verbose queries
  - Multilingual search
  - Synonyms or paraphrasing
- Represents text as vectors:
  - Sparse
  - Dense



Semantic search workflow



How to semantic search?

Deploy your model

```
POST
/_plugins/_ml/models/_register?deploy=true
{
    "name": "opensearch_neural",
    "version": "1.0.0",
    "model_group_id": "dw8EIP2",
    "model_format": "TORCH_SCRIPT"
}
```

In-cluster

```
POST /_plugins/_ml/connectors/_create
{
    "name": "embedding_ada",
    "description": "ada model example",
    "version": "1.0",
    "protocol": "http",
    "parameters": {
        "model": "text-embedding-ada-002"
    }...
}
```

Outside cluster, through connector

In cluster deployment: some adaptations

- Model need to be converted to ONNX
- Model has to be saved as zip:
  - Tokenizer
  - Weights
  - For +2GB model, weights are in model.onnx_data!
  - Check model configuration in config.json for registration

https://www.adelean.com/en/blog/20260316_jinav5_to_opensearch/

Overview on semantic search field's type

Field types	Vector type	Features	
knn_vector	**DENSE**	Fully configurable	Need pipeline, or ingest vector directy
rank_features	**SPARSE**		
sparse_vector	**SPARSE**		
semantic	**DENSE/SPARSE**	Auto-adaptive	No need for pipeline

knn_vector

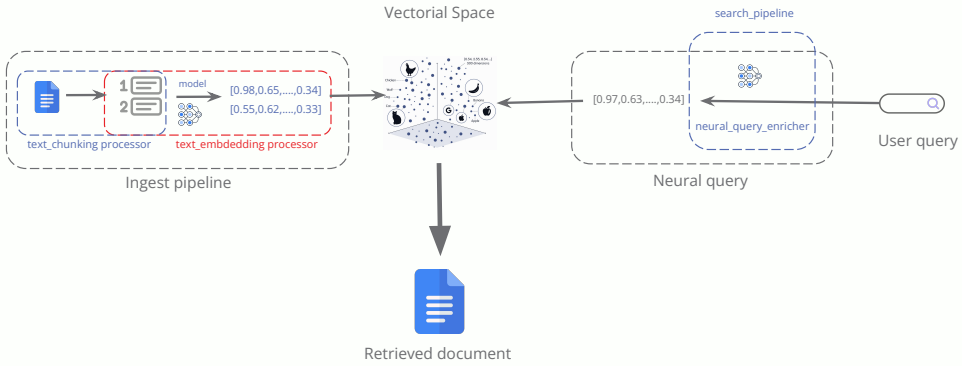
- k-NN vector offers great flexibility and a total control.

  You can specify:
  - **space_type**
  - **engine**

```
PUT test-index
{
  "settings": {
    "index": {
      "knn": true
    }
  },
  "mappings": {
    "properties": {
      "passage_embedding": {
        "type": "knn_vector",
        "dimension": 1024,
        "method": {
          "name": "hnsw",
          "space_type": "l2",
          "engine": "faiss"
        }}}}}
```

knn_vector

To use with *ingest_pipeline* at indexation time and *search pipeline* (optional)




```
PUT _ingest/pipeline/nlp-ingest-pipeline
{
  "description": "Embedding pipeline",
  "processors": [
    {
      "text_embedding": {
        "model_id": "bQ1J8ooBpHVUsb",
        "field_map": {
          "text": "passage_embedding"
        }
      }
    }
  ]
}
```

Space types

Euclidean: l2

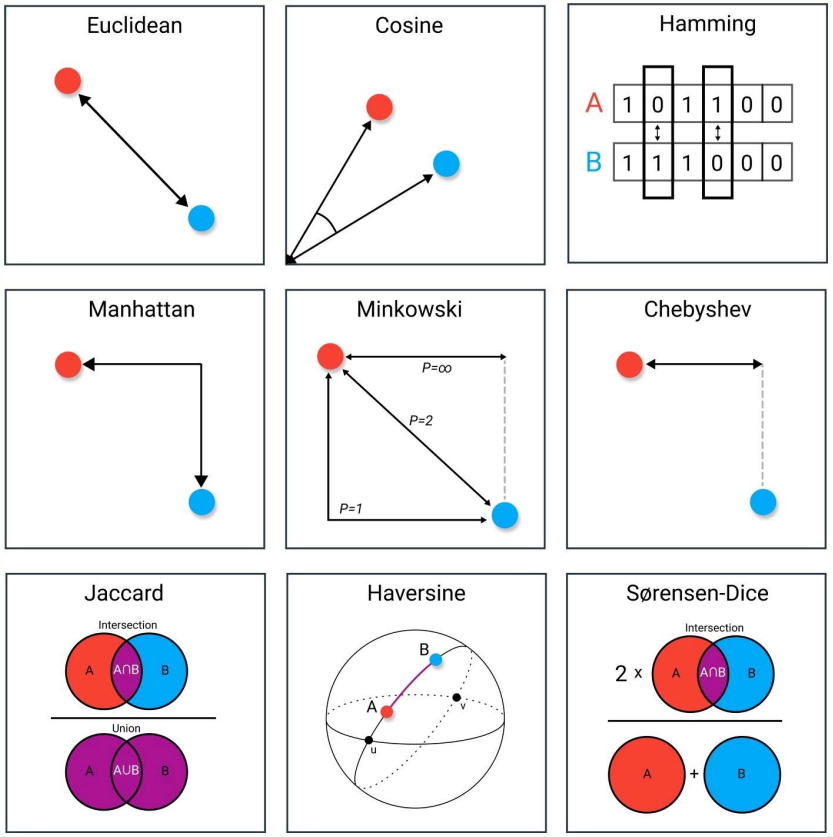
Manhattan: l1

Chebyshev: linf

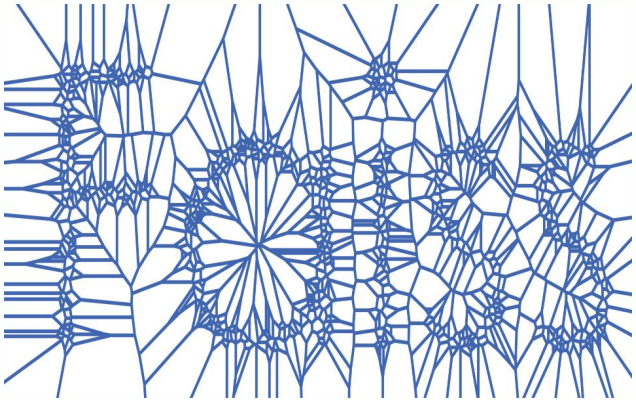
Cosine: cosinesimil

innerproduct

hammingbit, hamming



Implemented engines



NMSLIB

Deprecated

jvector

Plugin

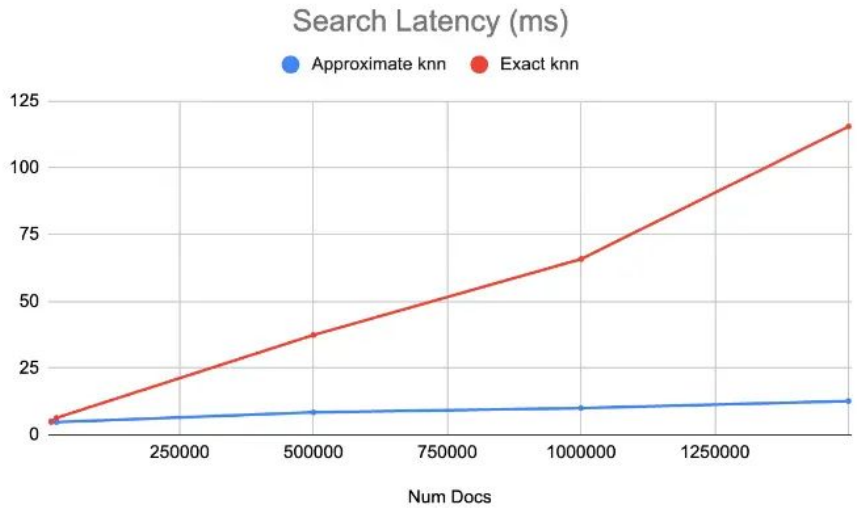
APACHE LUCENE™

Libraries and ANN algorithms availables

- **Lucene:**
  - HNSW
- **FAISS:**
  - IVF
  - HNSW
- **NMSLIB (legacy and deprecated) :**
  - HNSW
- **JVECTOR**
  - DISK_ANN (optimized HNSW)

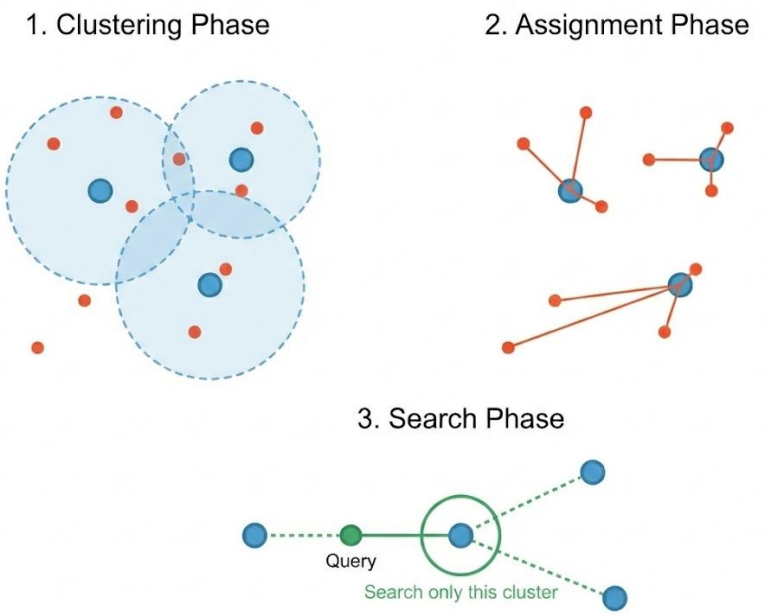
K-NN: the naive approach

- **Flat Indexation:** no preprocessing or clustering is performed at indexation time
- **Brute-force**: query vector is compared with the entire collection
- **Exact search** is possible through scripting



Approximate nearest neighbor: IVF

- **ANN** allows to reduce search query time
- IVF (Inverted File Indexing) is an ANN algorithm
- Indexation <-> Search



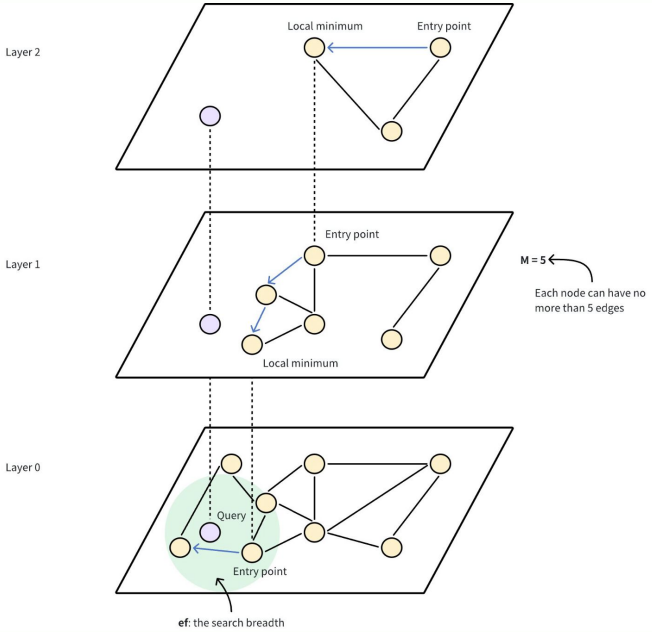
IVF Fine-Tuning

nprobes	Optimization Strategy
8	Optimize for memory and search latency
16	
32	Balance between latency, memory, and recall
64	
128	Optimize for recall

https://aws.amazon.com/blogs/big-data/choose-the-k-nn-algorithm-for-your-billion-scale-use-case-with-opensearch/

Approximate nearest neighbor: HNSW

- **HNSW** is the most popular used algorithm for ANN
- Based on the mechanics of probability skip lists and Navigable Small World graphs.
- Latest version of Lucene integrate ACORN algorithm for faster filtering



HNSW Fine-Tuning

m	ef_construction	ef_search	Optimization Strategy
8	32	32	Optimize for memory and search latency
16	32	32	
16	128	128	Balance between latency, memory, and recall
32	256	256	
32	512	512	Optimize for recall, more memory consumption
64	512	512	

https://aws.amazon.com/blogs/big-data/choose-the-k-nn-algorithm-for-your-billion-scale-use-case-with-opensearch/

Other ANN algorithm

- NSG
- ParaGraph
- CAGRA (we are going to talk about it ;)
- RoarGraph
- Vamana
- SPANN

Feature	Faiss/HNSW	Faiss/IVF	Lucene/HNSW	JVector/Diskan
Max dimensions	16,000	16,000	16,000	16,000
Filter	Post-filter	Post-filter	Filter during search	Filter during search
Training required	No (Yes for PQ)	Yes	No	No
Similarity metrics	l2, innerproduct, cosinesimil	l2, innerproduct, cosinesimil	l2, cosinesimil	l2, innerproduct, cosinesimil
Number of vectors	Tens of billions	Tens of billions	Less than 10 million	Hundreds of billions
Indexing latency	Low	Lowest	Low	Medium/Low
Query latency and quality	Low latency and high quality	Low latency and low quality	High latency and high quality	Low latency and high quality
Vector compression	Flat, PQ, SQ	Flat, PQ, SQ	Flat, SQ	PQ, BQ
Memory consumption	High, Low with PQ	Medium, Low with PQ	High	Very Low

Engine recommendations

Overview on semantic search field's type

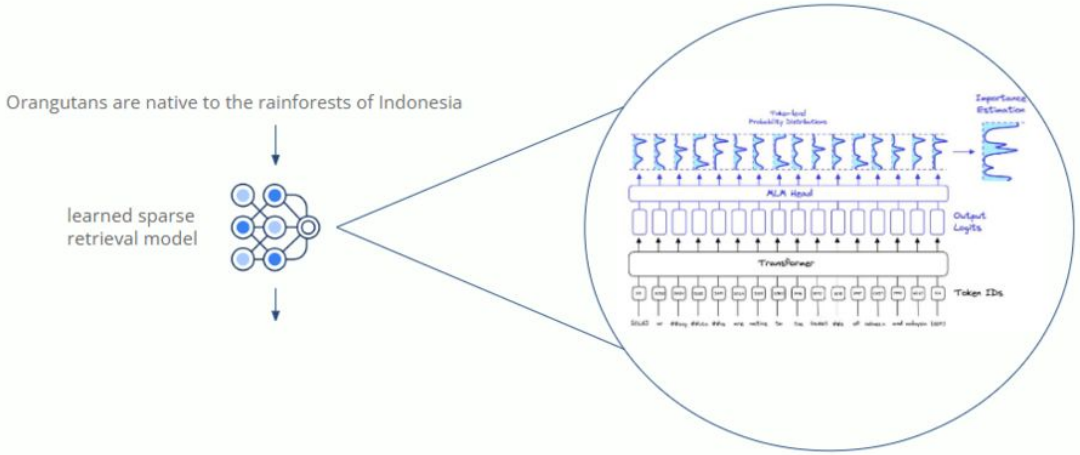
Field types	Vector type	Features	
knn_vector	**DENSE**	Fully configurable	Need pipeline, or ingest vector directy
rank_features	**SPARSE**		
sparse_vector	**SPARSE**		
semantic	**DENSE/SPARSE**	Auto-adaptive	No need for pipeline

Rank_features

ML text expansion to capture deep semantic context

Rank_features type

Possibility to index raw vector or use ingest pipeline



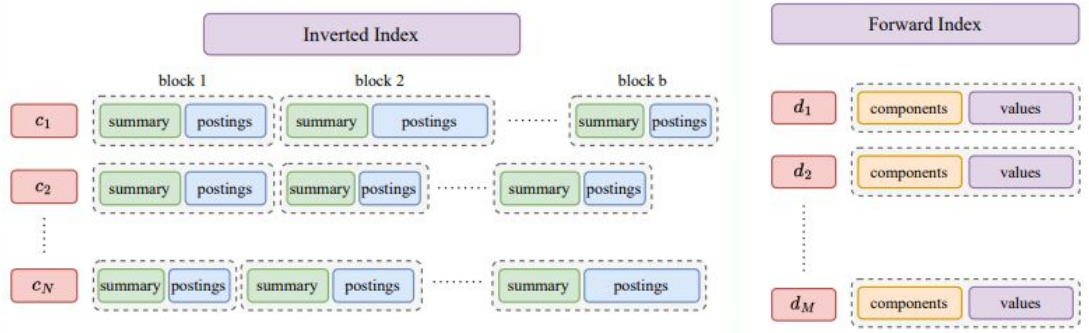
sparse_vector

3.3

New type: *sparse_vector*

Using **SEISMIC** algorithm to optimize search performance

Ideal for large-scale applications



Efficient Inverted Indexes for Approximate Retrieval over Learned Sparse Representations

Semantic type

New type: *semantic*

Simplified:

- automatic index configuration
- automatic pipelines

Works for internal inference and inference by connectors

```
PUT my_wonderful_index
{
  "settings": {
    "index": {
      "knn": true
    }
  },
  "mappings": {
    "properties": {
      "passage": {
        "type": "semantic",
        "model_id": "jkd433sjldh_560d"
      }
    }
  }
}
```

Semantic type

Good flexibility:

possibility to override generic behavior with specific configuration

```
PUT my_wonderful_index
{
  "mappings": {
    "properties": {
      "passage": {
        "type": "semantic",
        "model_id": "nF7yX5cBsaYnPfyOq2SG",
        "dense_embedding_config": {
          "method": {
            "name": "hnsw",
            "engine": "lucene",
            "parameters": {
              "ef_construction": 128,
              "m": 32
            }}}}}}}
```

Semantic type

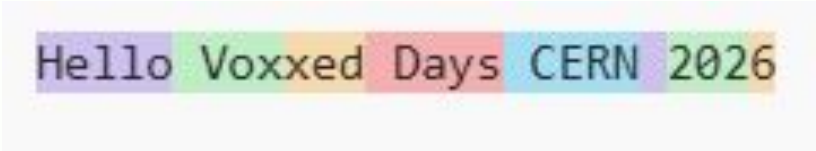
Easy chunking

Possibility to define specific chunking configuration

```
PUT my_wonderful_index
{
  "mappings": {
    "properties": {
      "passage": {
        "type": "semantic",
        "model_id": "nF7yX5cBsaYnPfyOq2SG",
        "chunking": true
      }
    }
  }
}
```

Chunking

- Models can’t fit everything – limited by number of tokens.
- Tokens ≠ words
- 1 token ≈ 4 characters of text ≈ ¾ of a word
- 100 tokens ≈ 75 words

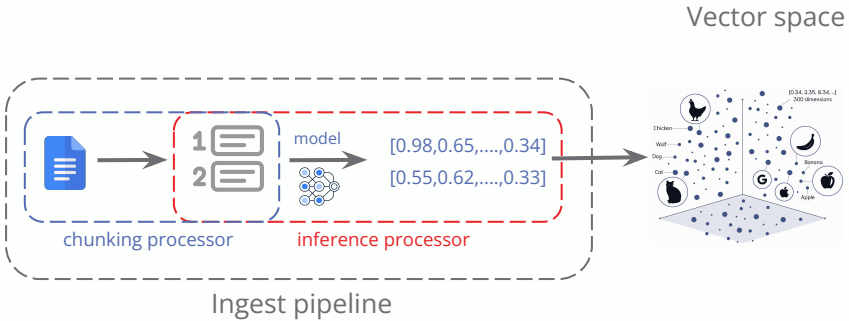


https://platform.openai.com/tokenizer

Chunking in OpenSearch

- Manual chunking method trough text_chunking processor.
- Chunking through semantic text chunking object.

- 3 algorithms available:
  - fixed token length
  - fixed chars length
  - delimiter

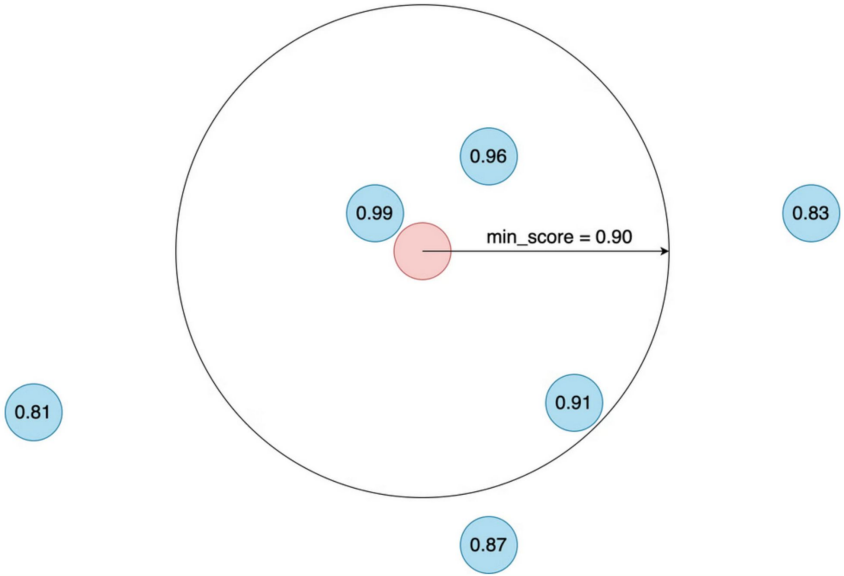


Search your vectors

<table>
<tr><th>Search mode</th><th>Field type</th><th>Query type</th><th>Main features</th></tr>
<tr><td rowspan="2">Dense</td><td rowspan="2">knn_vector</td><td>knn query</td><td>• vector query
• rescoring
• radial search</td></tr>
<tr><td rowspan="3">neural</td><td rowspan="3">• text,image and raw tokens query
• rescoring
• radial search
• works with dense and sparse (only semantic type)</td></tr>
<tr><td rowspan="3">Sparse</td><td>semantic</td></tr>
<tr><td>rank_features</td><td rowspan="2">neural sparse</td><td rowspan="2">• text and raw tokens query
• allows sparse ANN</td></tr>
<tr><td>sparse_vector</td></tr>
</table>

Radial search vs k-nn

- **Radial search** works on ANN algorithms, but changes the point of view:
  - doesn't select the closest k documents
  - it finds every document that falls inside a chosen radius around the query point.
  - max_distance or min_score



Rescoring

- **Rescoring** can be activated to increase recall
- Rescore uses full precision vectors
- **Oversample_factor** multiplies the number of retrieved candidate before rescoring

```
GET /my-vector-index/_search
{
  "size": 2,
  "query": {
    "knn": {
      "my_vector_field": {
        "vector": [1.5, 5.5, 1.5, 5.5, 1.5,
5.5, 1.5, 5.5],
        "k": 10,
        "method_parameters": {
            "ef_search": 10
        },
        "rescore": {
            "oversample_factor": 10.0
 }}}}}
```

Maximal Marginal Relevance

Pure semantic search often returns highly similar, redundant documents.

MRR results to balance relevance with diversity, avoiding duplicates.

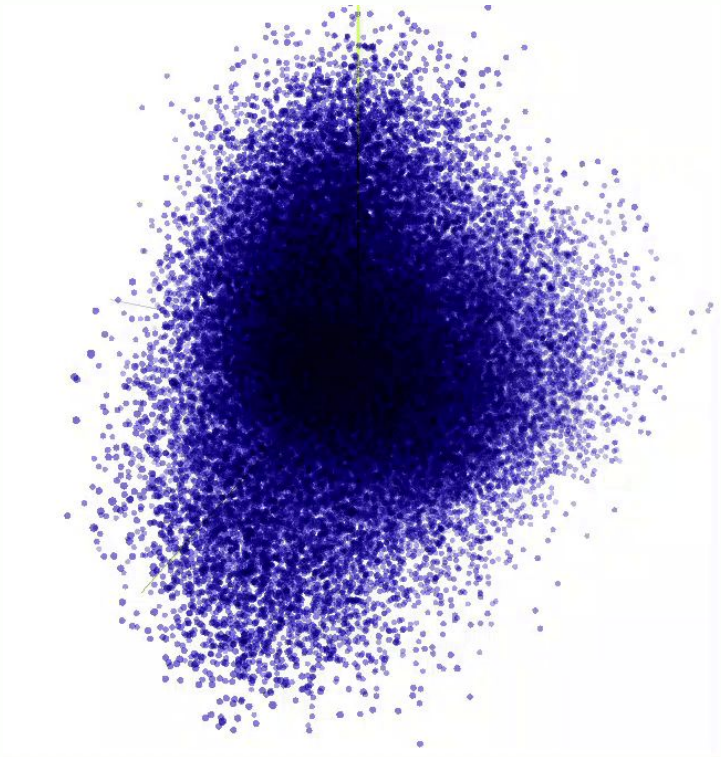
Still not compatible with nested and boolean !

```
POST /my-index/_search
{
  "query": {
    "knn": {
      "my_vector_field": {
        "vector": [0.12, 0.54, 0.91],
        "k": 10
      }
    }
  },
  "ext": {
    "mmr": {
      "diversity": 0.7
    }}}
```

Optimize your semantic search

A billion vector database and beyond

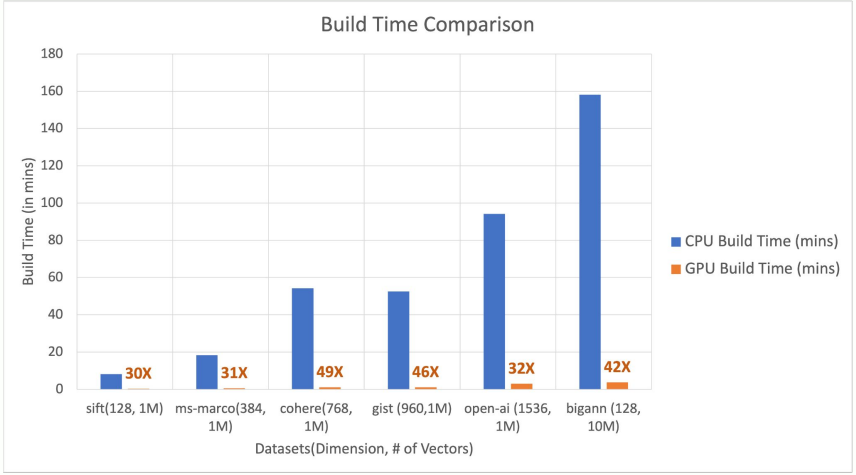
- Vector databases now scale to billions of vectors
- **This requires optimization:**
  - Indexing
    - Faster
    - Lower storage footprint
  - Search
    - Lower latency
    - Improved recall



GPU acceleration

Leveraging GPU:

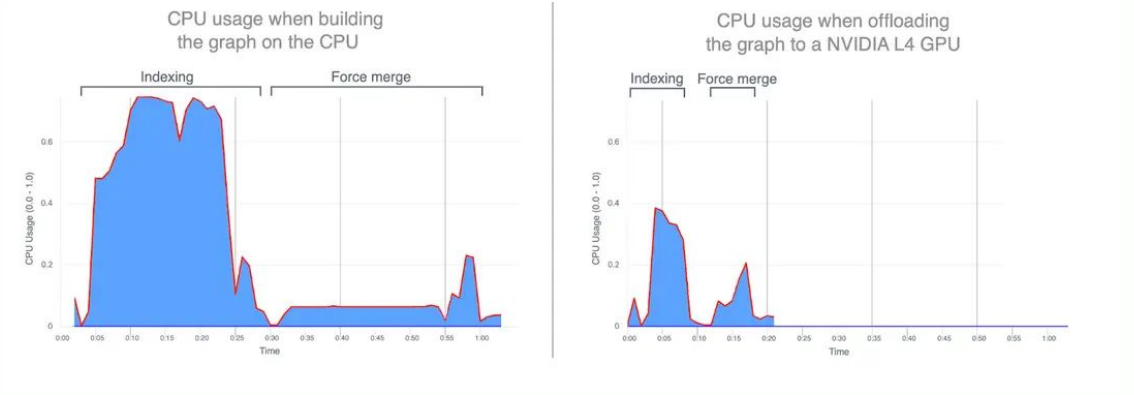
- indexation up to x50 faster
- leveraging CAGRA to build index on GPU, then converted to be used on CPU
- in cluster indexation with gpu is available in OpenSearch 2.5



CAGRA in action

1. Vectors buffered in Java heap
2. Sent to GPU to build CAGRA graph
3. Retrieved from GPU and converted to HNSW
4. Persisted to disk

GPU is faster also in merging.



https://www.elastic.co/search-labs/blog/elasticsearch-gpu-accelerated-vector-indexing-nvidia

Disk Memory Requirements

Disk Memory = Number of documents *

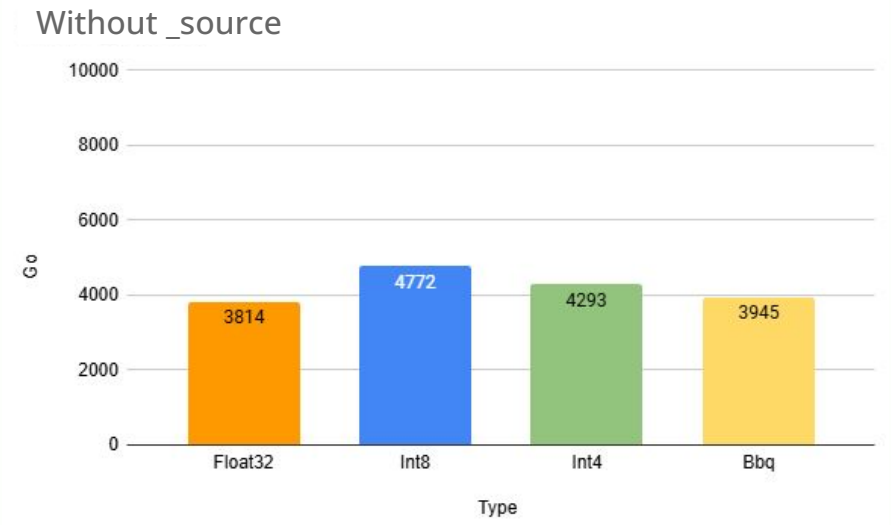
Number of words x document / (¾ context length) *

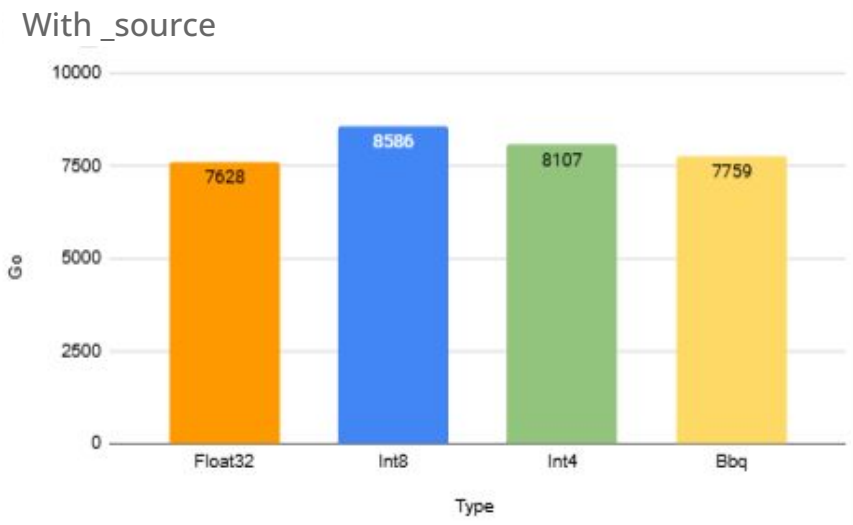
Dimensions * Data type

Disk Memory = 100.000.000 * 3840 / ¾ 512 * 1024 * 4 = 3.8TB

Disk Memory Requirements in real life

With 1 billion vectors and 1024 dimensions





Disk optimization: disable source

Vectors are stored multiple times:

- KnnVectorField at Lucene level
- _source level

Disabling source, limits certains operations:

- Reindex
- Update and update_by_query

```
"mappings": {
      "_source": {
        "excludes": [
          "passage_embeddings"
        ]
      }
```

Disk optimization: derived source

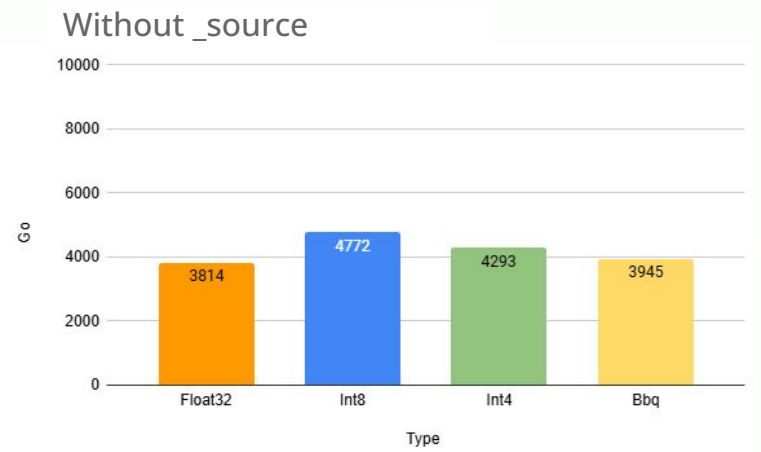
derived_source is available

- to not store vectors in _source
- to have dynamic retrieval from internal format when needed
- **it is the default option**

Disk memory optimization: KnnVectorField

KnnVectorField contains:

- The original vector
- The quantized vector (quantification int8 is done by default)



Disk memory optimization: KnnVectorField

Another way to reduce disk memory requirements:

1. Change your index data_type
2. Index quantized vector

1. Change your index data type

With *knn_vector* type, change index data_type to the desired level of quantization:

a. binary
b. byte
c. float

2. Index quantized vector

```python
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",
)
```

External Quantization

```
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;
        """
      }
    }
  ]
}
```

Quantization in pipeline

RAM Requirements with HNSW

To work properly, vectors have to be loaded into RAM.

Note: HNSW and IVF have similar requirements, IVF slightly cheaper

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

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

Quantization

Help to reduce the cost, compressing into lower-precision formats:

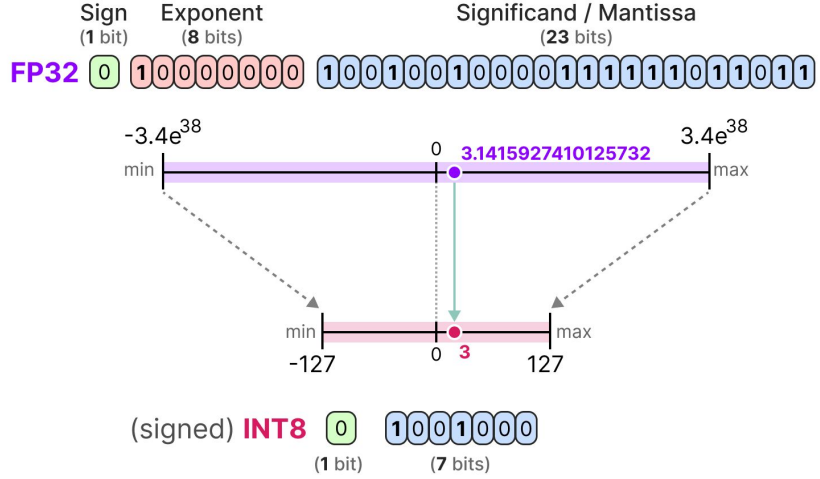
- Scalar Quantization
- Product Quantization
- Binary Quantization
- many others: OPQ, DPQ, ...

Scalar quantization

Use normalization, like min-max to compress the space type

Great accuracy, decent compression

Required memory (int8) = num_vectors × (vector_size + 4) + *(num_vectors * 4 * HNSW.m)*

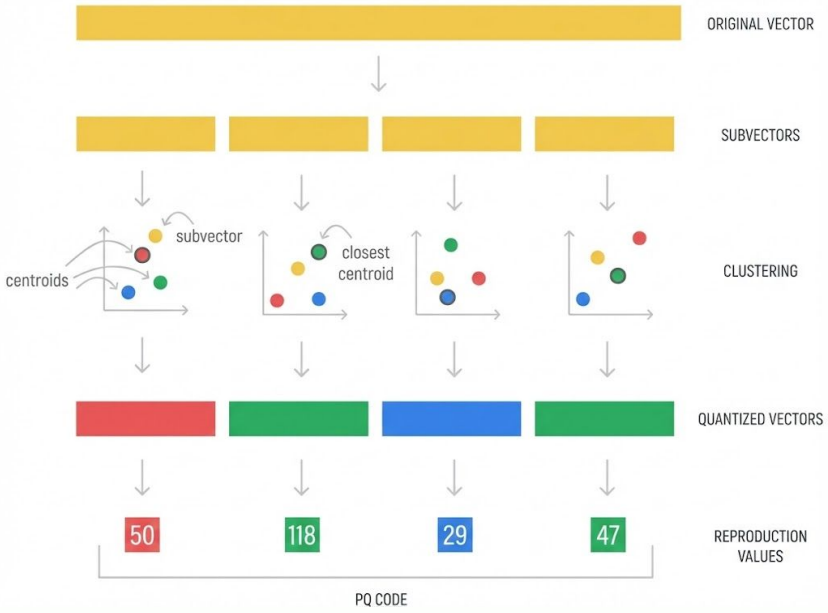


https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization

Product quantization

Compress vectors by clustering subvectors into shared codebooks.

Good accuracy, good compression

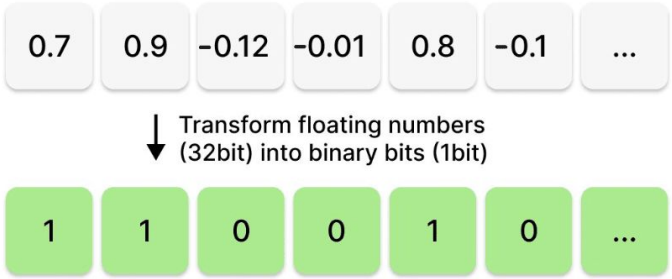


Required memory = 1.1(((pq_code_size / 8) * pq_m + 24 + 8 * HNSW.m) * num_vectors + num_segments * (2^pq_code_size * 4 * d))*

Binary quantization

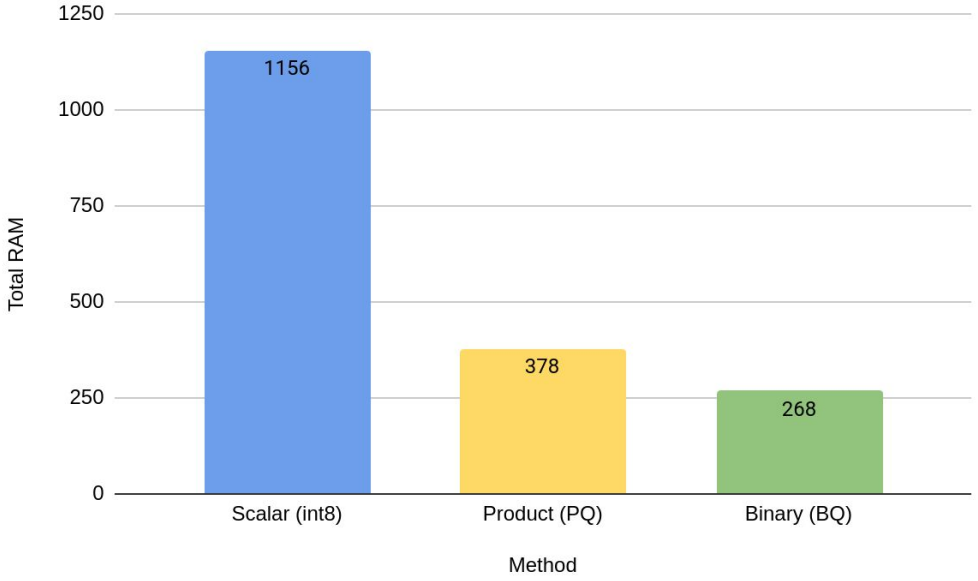
Apply a threshold to convert each element into a single bit (e.g., values > 0 become 1, otherwise 0).

Reduced accuracy, huge compression.



*Required memory = num_vectors × (vector_size/8 + 12) + (num_vectors * 4 * HNSW.m)*

RAM Requirements with quantization (HNSW)



With 1 billion vectors

1024 dimensions

HNSW ->

- hsnw.m = 32

PQ ->

- pq_code_size: 8 bits
- pq_m: 16

Quantization techniques

Available:

- Lucene scalar quantization
- Faiss scalar quantization
- Faiss product quantization
- Binary quantization

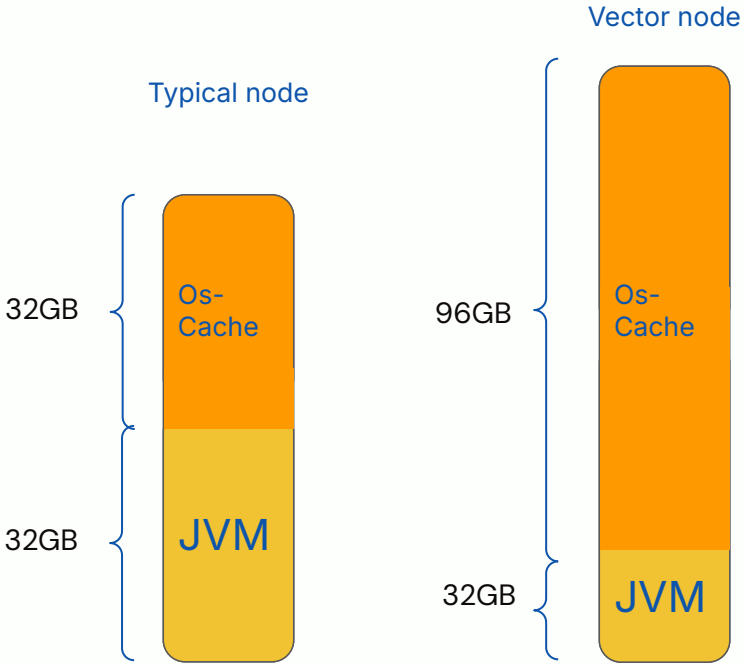
Easy to configure through encoding parameters

Faiss 1-bit quantization ->

```
PUT my_wonderful_index
{
  "mappings": {
    "properties": {
      "my_vector1": {
        "type": "knn_vector",
        "dimension": 3,
        "space_type": "l2",
        "method": {
          "name": "hnsw",
          "engine": "faiss",
          "parameters": {
            "encoder": {
              "name": "sq",
              "parameters": {
                "bits": 1
              }}}}}}}}
```

3.6

Node configuration



For dedicated vector search node, the standard 50/50 RAM split between the JVM and **the file system cache** isn't optimal.

A better strategy is to allocate a smaller, sufficient heap for operations (<32 GB) and dedicate the majority of the server's RAM to the OS page cache.

This approach has been validated in large-scale tests.

Vectors on disk

Different strategies:

- Disk Memory Search
- Memory-Optimized search

Disk-based vector search

1. Binary vector compression (default x32)
2. Disk-based search execution
3. Reranking using full-precision vectors

```
PUT my_wonderful_index
{
  "settings" : {
    "index": {
      "knn": true
    }
  },
  "mappings": {
    "properties": {
      "my_vector_field": {
        "type": "knn_vector",
        "dimension": 8,
        "space_type": "l2",
        "data_type": "float",
        "mode": "on_disk",
       "compression_level": "32x"
   }}}}
```

Disk-based vector search

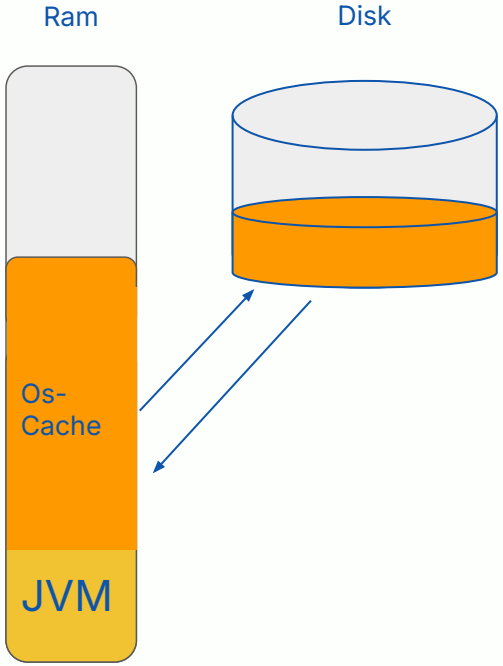
1. Scalar 1-bit ~~Binary vector~~ compression (default x32)
2. Disk-based search execution
3. Reranking using full-precision vectors


```
PUT my_wonderful_index
{
  "settings" : {
    "index": {
      "knn": true
    }
  },
  "mappings": {
    "properties": {
      "my_vector_field": {
        "type": "knn_vector",
        "dimension": 8,
        "space_type": "l2",
        "data_type": "float",
        "mode": "on_disk",
       "compression_level": "32x"
   }}}}
```

Memory optimized search

3.3

- Works only with FAISS and HNSW
- mMap charged on os-cache, actual vectors on disk
- Portions of HNSW are charged to memory at search time



Better recall

Compression and On-Disk Vectors offer more flexibility:

- Require less RAM
- Lower costs

But what about Recall?

- Asymmetric Distance Calculation (ADC)
- Random Rotation (RR)

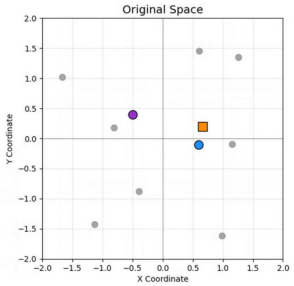
```
PUT my_wonderful_index
{
 ..............................
 "mappings": {
    "properties": {
      "vector_field": {
        "type": "knn_vector",
        "method": {
            "name": "hnsw",
            "engine": "faiss",
            "parameters": {
              "encoder": {
                "name": "binary",
                "parameters": {
                  "bits": 1,
                "random_rotation": true,
                  "enable_adc": true
                }}}}}}}}
```

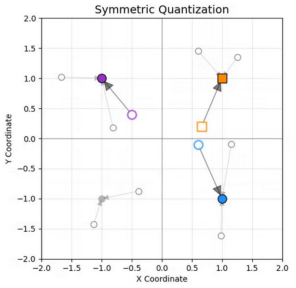
Asymmetric Distance Calculation

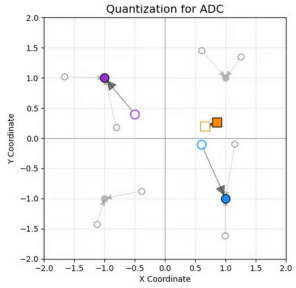
3.2

Maintains high recall even with quantized vectors.

Calculates distance using the uncompressed query vector against the compressed database vectors.





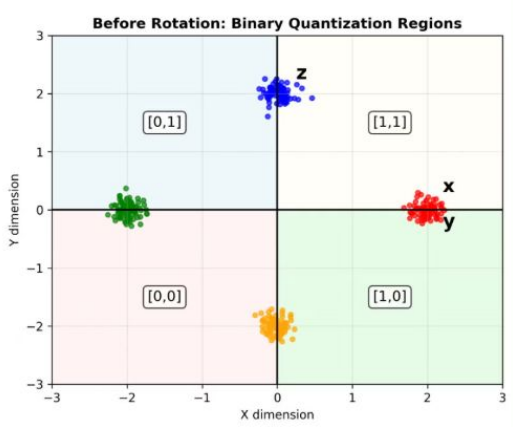


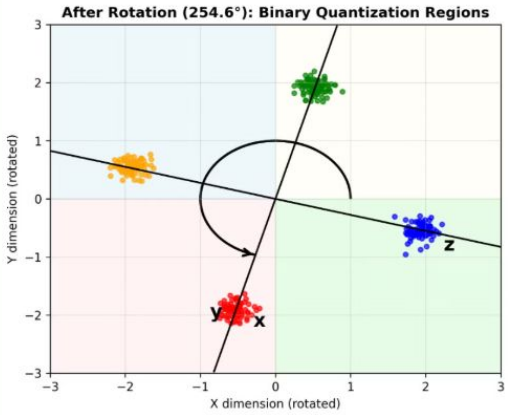
Random Rotation

3.2

Binary Quantization on unbalanced data causes a sharp drop in accuracy.

Random Rotation evenly distributes the variance across all dimensions.



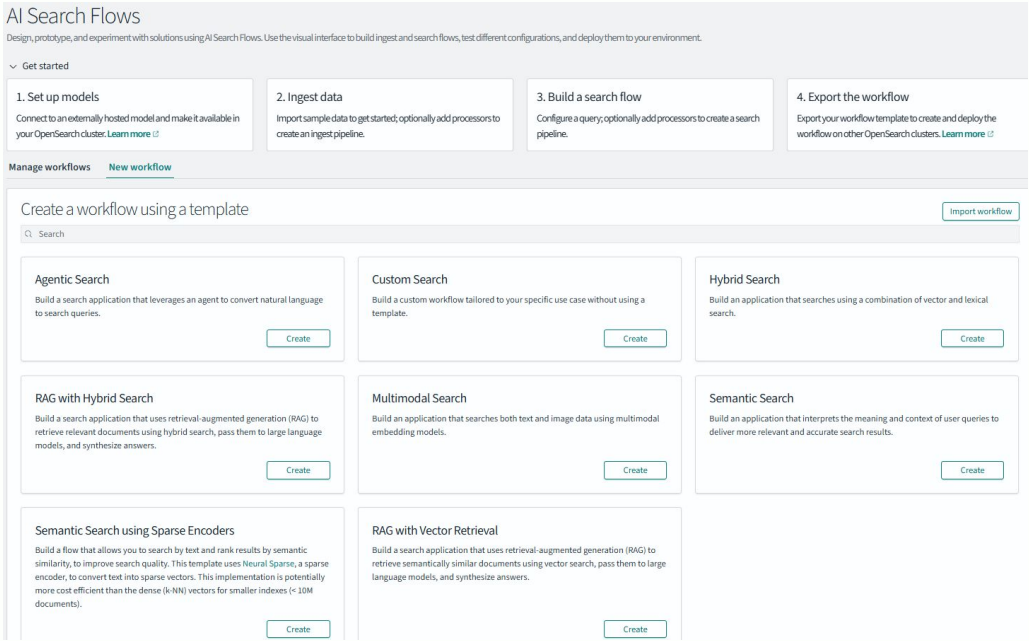


OpenSearch Workflow Builder

UI to easily create a workflow.

Different use cases:

- Agentic
- Hybrid
- Multimodal
- Semantic Search



Conclusion

OpenSearch is the best choice for lexical, semantic and hybrid search at scale:

- IVF, HNSW, DiskANN
- Quantization and memory optimizations
- Search recall optimizations
- Automatizations: from new semantic type to UI for building workflows


OpenSearchCon
EUROPE
adelean
Thank you!