Vector Search, Semantic Search & Hybrid Search

Day 4 — 4-Day Elasticsearch Course

Elasticsearch 8.18 | dense_vector · ELSER · semantic_text · RRF

Agenda

1. Vector Search Fundamentals
2. **Practice 4A** — Vector Search
3. ELSER & Semantic Text
4. **Practice 4B** — Semantic Search
5. Hybrid Search with RRF
6. **Practice 4C** — Hybrid Search
7. Advanced Topics
8. **Practice 4D** — Advanced (Bonus)
9. Course Wrap-up

Stack Check

Today requires the **elk-ml** stack (8GB+ RAM):

```
# Should be running already
docker compose -f docker/elk-ml/docker-compose.yml --env-file .env ps

# If not, start it
docker compose -f docker/elk-ml/docker-compose.yml --env-file .env up -d
```

Endpoints:

- Elasticsearch: `https://localhost:9200` (auth: `elastic` / `elastic`)
- Kibana: `http://localhost:5601`

> HTTPS for this stack — Kibana handles the certificate internally.

Vector Search Fundamentals

From keywords to meaning

The Evolution of Search

```
Keyword Search (BM25)          Vector Search              Hybrid Search
"space adventure"              [0.2, 0.8, ...]           BM25 + kNN + RRF
      ↓                             ↓                         ↓
Exact term matching            Semantic similarity       Best of both worlds
"space" AND "adventure"        "cosmic journey" ≈ match  Exact + semantic
```

Generation	Technology	Weakness
Keyword	BM25 (inverted index)	Misses synonyms, paraphrases
Semantic	Embeddings + kNN	May miss exact terms, proper nouns
Hybrid	BM25 + kNN + RRF	More complex, but most accurate

What Are Embeddings?

Embeddings are **numerical representations** of text (or images) in a high-dimensional space.

```
"king"   → [0.2, 0.8, 0.1, 0.9, ...]    384 dimensions
"queen"  → [0.2, 0.7, 0.1, 0.8, ...]    ← close to "king"
"banana" → [0.9, 0.1, 0.7, 0.2, ...]    ← far from "king"
```

Key insight: Similar meanings → similar vectors → small distance

Embeddings are generated by ML models:

- ELSER (Elastic) — sparse embeddings, English
- all-MiniLM-L6-v2 (HuggingFace) — 384-dim dense
- text-embedding-3-small (OpenAI) — 1536-dim dense

dense_vector Field Type

```
PUT /vector-demo
{
  "mappings": {
    "properties": {
      "title": { "type": "text" },
      "overview": { "type": "text" },
      "embedding": {
        "type": "dense_vector",
        "dims": 3,
        "index": true,
        "similarity": "cosine"
      }
    }
  }
}
```

Parameter	Purpose
`dims`	Vector dimensions (3 for demo, 384-1536 in production)

Similarity Metrics

$$\text{Cosine Similarity} \quad \frac{A \cdot B}{|A| \times |B|}$$

$$\text{L2 Norm (Euclidean)} \quad \sqrt{\Sigma(Ai - Bi)^2}$$

$$\text{Dot Product} \quad \Sigma(Ai \times Bi)$$

Metric	Best For	Note
cosine	Text embeddings	Direction matters, not magnitude
l2_norm	Image/spatial	Actual distance between points
dot_product	Pre-normalized vectors	Fastest computation

Default choice: Use `cosine` for text search.

HNSW: How kNN Search is Fast

HNSW (Hierarchical Navigable Small World) — an approximate nearest neighbor algorithm:

```
Layer 2:  [A] ---- [D]              ← Long-range connections
           |        |
Layer 1:  [A] - [B] - [D] - [F]     ← Medium connections
           |   |   |   |   |
Layer 0:  [A]-[B]-[C]-[D]-[E]-[F]   ← All nodes, local connections
```

- Multi-layer graph structure
- Search starts at top layer (coarse), drills down (fine)
- Trade-off: accuracy vs speed (controlled by `num_candidates`)
- Not exact — "approximate" nearest neighbors (ANN)

kNN Query

```
GET /movies-embeddings/_search
{
  "knn": {
    "field": "overview_embedding",
    "query_vector": [0.02, -0.05, 0.08, ...],
    "k": 5,
    "num_candidates": 50
  },
  "_source": ["title", "overview", "genres"]
}
```

Parameter	Purpose
`k`	Number of nearest neighbors to return
`num_candidates`	Size of initial candidate pool (higher = more accurate, slower)
`query_vector`	The vector to find neighbors for

kNN with Filtering (Pre-filtering)

```
GET /movies-embeddings/_search
{
  "knn": {
    "field": "overview_embedding",
    "query_vector": [0.02, -0.05, 0.08, ...],
    "k": 5,
    "num_candidates": 50,
    "filter": {
      "term": { "genres": "Drama" }
    }
  }
}
```

- Filter is applied **before** kNN search (pre-filtering)
- Only matching documents are considered as candidates
- Filters use the same Query DSL as regular searches

Loading the Embeddings Dataset

Our movies dataset includes pre-computed 384-dimensional embeddings:

```
python data/load_data.py --dataset movies --with-embeddings
```

This creates `movies-embeddings` with 500 movies, each containing:

Field	Type	Description
`title`	text	Movie title
`overview`	text	Plot summary
`genres`	keyword	Genre array
`vote_average`	float	Rating 0-10
`release_date`	date	Release date
`overview_embedding`	dense_vector (384)	Pre-computed embedding

Practice 4A

Vector Search Basics

`day4-exercises.md` — Part A (5 tasks) | ~20 min

ELSER & Semantic Text

Zero-config semantic search

What is ELSER?

Elastic Learned Sparse EncodeR — Elastic's built-in NLP model.

Feature	Detail
Type	Sparse embedding model
Language	English
Output	Sparse vectors (weighted token expansions)
Runs on	ML nodes (in-cluster, no external API)
Cost	Free (included with Elastic license)

Sparse vs Dense:

- **Dense:** `[0.2, 0.8, 0.1, ...]` — 384 floats, every dimension used
- **Sparse:** `{"prison": 2.1, "escape": 1.8, "freedom": 1.5, ...}` — weighted tokens

Deploying ELSER

Via Kibana (Recommended)

1. **Machine Learning** → **Trained Models**
2. Find `.elser_model_2_linux-x86_64`
3. Click **Download** → **Deploy**
4. Wait for status: **started**

Via API

```
PUT /_inference/sparse_embedding/my-elser-endpoint
{
  "service": "elser",
  "service_settings": {
    "num_allocations": 1,
    "num_threads": 1
  }
}
```

Inference Endpoints

The gateway between your data and ML models:

```
PUT /_inference/sparse_embedding/my-elser-endpoint
{
  "service": "elser",
  "service_settings": {
    "num_allocations": 1,
    "num_threads": 1
  }
}
```

Testing Inference Endpoints

```
POST /_inference/sparse_embedding/my-elser-endpoint
{
  "input": "Two men escape from prison"
}
```

Returns weighted token expansion:

```
{
  "sparse_embedding": [{
    "prison": 2.1, "escape": 1.8, "flee": 1.2, "convict": 0.9, ...
  }]
}
```

> ELSER expands the input into **semantically related tokens** with relevance weights.

semantic_text Field Type

The simplest way to add semantic search — embeddings generated automatically:

```
PUT /movies-semantic
{
  "mappings": {
    "properties": {
      "title": { "type": "text" },
      "overview": { "type": "text" },
      "overview_semantic": {
        "type": "semantic_text",
        "inference_id": "my-elser-endpoint"
      },
      "genres": { "type": "keyword" },
      "vote_average": { "type": "float" }
    }
  }
}
```

semantic_text: How It Works

- Just index text → ELSER generates embeddings **automatically**
- No pre-computation pipeline needed
- Embeddings update when documents are updated

```
// Copy text to both fields when indexing
{
  "overview": "Two imprisoned men bond over years...",
  "overview_semantic": "Two imprisoned men bond over years..."
}
```

`overview` for keyword search, `overview_semantic` for semantic search.

Indexing with semantic_text

```
POST /movies-semantic/_bulk
{"index": {"_id": "1"}}
{"title": "The Shawshank Redemption", "overview": "Two imprisoned men bond over a number of years, finding solace and eventual redemption through acts of common decency.", "overview_semantic": "Two imprisoned men bond over a number of years, finding solace and eventual redemption through acts of common decency.", "genres": ["Drama"], "vote_average": 8.7}
{"index": {"_id": "2"}}
{"title": "The Godfather", "overview": "The aging patriarch of an organized crime dynasty transfers control of his clandestine empire to his reluctant son.", "overview_semantic": "The aging patriarch of an organized crime dynasty transfers control of his clandestine empire to his reluctant son.", "genres": ["Crime", "Drama"], "vote_average": 8.7}
```

> Copy the same text to both `overview` (for keyword search) and `overview_semantic` (for semantic search).

Semantic Query

```
GET /movies-semantic/_search
{
  "query": {
    "semantic": {
      "field": "overview_semantic",
      "query": "movies about escaping from prison"
    }
  }
}
```

ELSER understands meaning:

- `"escaping from prison"` → finds "The Shawshank Redemption"
- Even though the overview says "imprisoned men" and "redemption" (not "escaping" or "prison")
- A keyword search for "escaping prison" would **miss** this document!

Keyword vs Semantic: Side by Side

```
# Keyword search — matches exact tokens
GET /movies-semantic/_search
{
  "query": {
    "match": { "overview": "escaping prison" }
  }
}
// Might miss Shawshank (no exact "escaping" or "prison" in overview)

# Semantic search — matches meaning
GET /movies-semantic/_search
{
  "query": {
    "semantic": {
      "field": "overview_semantic",
      "query": "escaping prison"
    }
  }
}
// Finds Shawshank! "imprisoned" ≈ "prison", "redemption" ≈ "escaping"
```

When to Use semantic_text

Use semantic_text	Use dense_vector
Want simplicity	Need custom embedding model
English text	Multi-language support
ML nodes available	No ML infrastructure
Standard use cases	Sharing embeddings across systems
Auto-updated embeddings	Maximum indexing speed

Practice 4B

ELSER & Semantic Search

`day4-exercises.md` — Part B (4 tasks) | ~20 min

Hybrid Search with RRF

The best of both worlds

Why Hybrid?

Query	BM25 finds	kNN finds	Hybrid finds
"Shawshank"	Exact title match	Nothing (rare word)	Title match
"prison escape movie"	Docs with those words	Semantically similar	Both
"films about hope"	"hope" keyword matches	Thematic matches	Comprehensive

Neither method alone is perfect. Hybrid combines their strengths.

Reciprocal Rank Fusion (RRF)

Combines rankings from multiple retrievers without needing score normalization:

```
RRF(d) = Σ  1 / (k + rank_i(d))
```

Example:

Document	BM25 Rank	kNN Rank	RRF Score (k=60)
Movie A	1	5	1/61 + 1/65 = **0.0318**
Movie B	3	1	1/63 + 1/61 = **0.0323**
Movie C	2	10	1/62 + 1/70 = **0.0304**

Movie B wins — ranked highly by **both** retrievers.

The Retriever API

Elasticsearch 8.14+ provides a clean API for hybrid search:

```
GET /movies-embeddings/_search
{
  "retriever": {
    "rrf": {
      "retrievers": [
        { "standard": { "query": {
            "match": { "overview": "space adventure" }
        }}},
        { "knn": {
            "field": "overview_embedding",
            "query_vector": [/* 384 dims */],
            "k": 10, "num_candidates": 50
        }}
      ],
      "rank_window_size": 100,
      "rank_constant": 60
    }
  }
}
```

RRF Parameters

Parameter	Default	Effect
`rank_constant`	60	Higher = more equal weighting across ranks
`rank_window_size`	100	How many results from each retriever to consider

rank_constant tuning

```
k=1:    Top result gets ~50% of score weight (steep dropoff)
k=60:   Balanced — good default for most cases
k=1000: All ranks get similar weight (nearly flat)
```

rank_window_size tuning

```
Small (50):  Fast, but may miss relevant docs ranked 50–100
Large (200): Better recall, considers more candidates
```

Three-way Hybrid: BM25 + kNN + Semantic

```
GET /movies-semantic/_search
{
  "retriever": {
    "rrf": {
      "retrievers": [
        { "standard": { "query": {
            "multi_match": {
              "query": "prison escape redemption",
              "fields": ["title^2", "overview"]
        }}}},
        { "knn": {
            "field": "overview_embedding",
            "query_vector": [/* embedding */],
            "k": 10, "num_candidates": 50
        }},
        { "standard": { "query": {
            "semantic": {
              "field": "overview_semantic",
              "query": "escaping from prison"
        }}}}
      ]
    }
  }
}
```

Adding Filters to Hybrid Search

Filters must be applied to **each retriever** separately:

```
{
  "retriever": { "rrf": { "retrievers": [
    { "standard": { "query": { "bool": {
        "must": { "match": { "overview": "war" } },
        "filter": { "range": { "vote_average": { "gte": 8 } } } }
    }}},
    { "knn": {
        "field": "overview_embedding",
        "query_vector": [/* ... */],
        "k": 10, "num_candidates": 50,
        "filter": { "range": { "vote_average": { "gte": 8 } } } }
    }}
  ]}}
}
```

The same filter appears in **both** retrievers — there's no global filter in RRF.

Practice 4C

Hybrid Search with RRF

`day4-exercises.md` — Part C (5 tasks) | ~25 min

Advanced Topics

Production-ready semantic search

Quantization: Reducing Memory

int8 scalar quantization reduces vector memory by ~4x:

```
PUT /movies-quantized
{
  "mappings": {
    "properties": {
      "overview_embedding": {
        "type": "dense_vector",
        "dims": 384,
        "index": true,
        "similarity": "cosine",
        "index_options": {
          "type": "int8_hnsw"
        }
      }
    }
  }
}
```

Quantization: Trade-offs

	Float32 (default)	Int8 (quantized)
Memory per vector	384 × 4 = 1,536 bytes	384 × 1 = 384 bytes
Accuracy	Exact	~95-99% of original
Use when	< 1M documents	> 1M documents

4x memory reduction with minimal accuracy loss — essential for large-scale vector search.

Chunking Long Documents

Long documents need to be split into chunks for effective embedding:

```
Original: 5000-word article
     ↓
Chunk 1: words 1-500     → embedding_1
Chunk 2: words 450-950   → embedding_2  (50 word overlap)
Chunk 3: words 900-1400  → embedding_3
...
```

- Embedding models have a **token limit** (typically 256-512 tokens)
- Use **overlapping windows** to preserve context at boundaries

Chunking: Index Schema

```
PUT /articles-chunked
{
  "mappings": {
    "properties": {
      "article_id": { "type": "integer" },
      "chunk_id": { "type": "integer" },
      "chunk_text": { "type": "text" },
      "chunk_embedding": {
        "type": "dense_vector",
        "dims": 384,
        "index": true,
        "similarity": "cosine"
      }
    }
  }
}
```

Use `collapse` on `article_id` to deduplicate results.

Pre-filtering vs Post-filtering

Pre-filtering (recommended)

```
"knn": {
  "field": "embedding",
  "query_vector": [...],
  "k": 10,
  "num_candidates": 100,
  "filter": { "term": { "genre": "Drama" } }
}
```

Filter applied **BEFORE** kNN → always returns k results matching filter.

Post-filtering

```
"knn": { "field": "embedding", "query_vector": [...], "k": 10 },
"post_filter": { "term": { "genre": "Drama" } }
```

Filter applied **AFTER** kNN → may return **fewer than k** results.

	Pre-filter	**Post-filter**
Result count	Always k	≤ k
Use when	Default choice	Need facet counts from unfiltered results

Custom Embedding Models: OpenAI

```
PUT /_inference/text_embedding/openai-embeddings
{
  "service": "openai",
  "service_settings": {
    "api_key": "sk-...",
    "model_id": "text-embedding-3-small"
  }
}
```

- 1536 dimensions, multi-language
- Requires API key + external network call

Custom Embedding Models: Hugging Face

```
PUT /_inference/text_embedding/hf-embeddings
{
  "service": "hugging_face",
  "service_settings": {
    "api_key": "hf_...",
    "url": "https://api-inference.huggingface.co/...all-MiniLM-L6-v2"
  }
}
```

External models: better quality, multi-language, but add **latency + cost** and data leaves your cluster.

Approximate kNN (default — HNSW)

```
GET /movies-embeddings/_search
{
  "knn": {
    "field": "overview_embedding",
    "query_vector": [...],
    "k": 10,
    "num_candidates": 100
  }
}
```

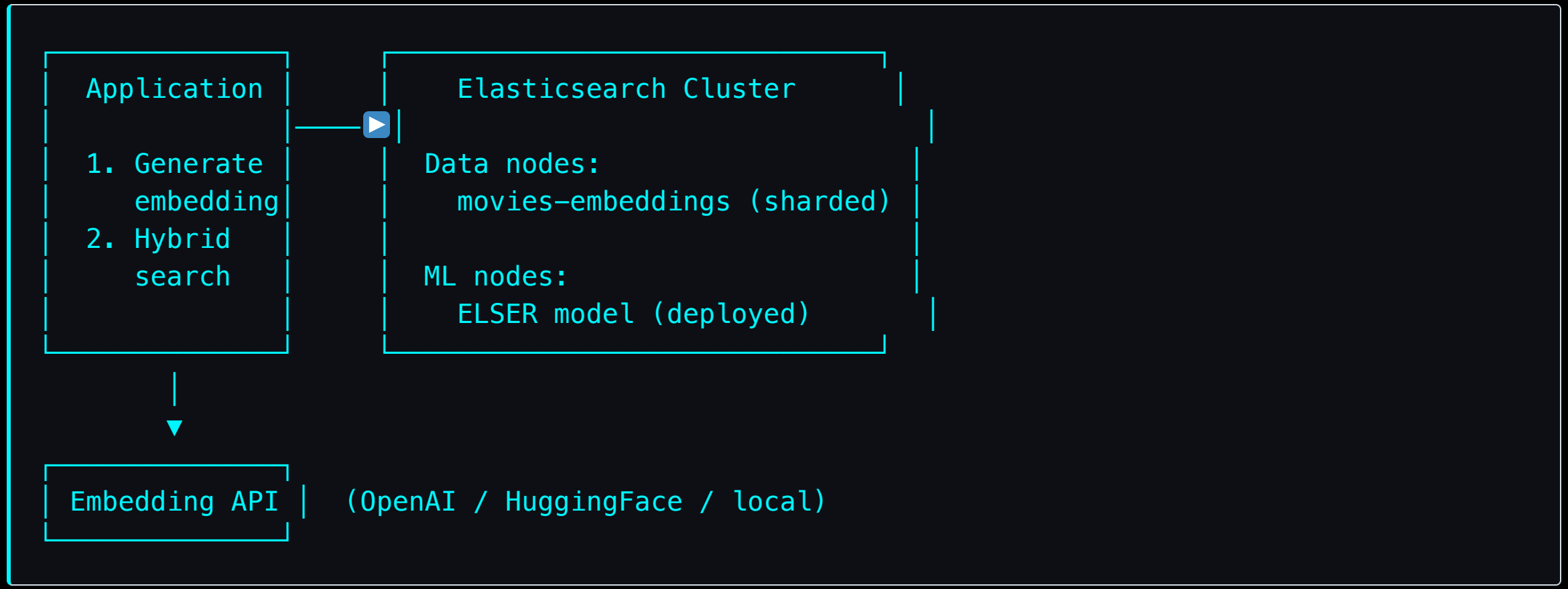
- Uses HNSW graph for fast approximate search
- Trade-off: accuracy vs speed (controlled by `num_candidates`)

Exact kNN (brute-force — script_score)

```
GET /movies-embeddings/_search
{
  "query": {
    "script_score": {
      "query": { "match_all": {} },
      "script": {
        "source": "cosineSimilarity(params.qv, 'overview_embedding') + 1.0",
        "params": { "qv": [/* 384 dims */] }
      }
    }
  }
}
```

Exact is **O(n)** — only viable for small datasets (<10K docs) or evaluation.

Production Architecture



Key decisions:

- ELSER (in-cluster) vs external embeddings
- Approximate (HNSW) vs exact kNN

Practice 4D

Advanced Techniques (Bonus)

`day4-exercises.md` — Part D (4 tasks) | ~20 min

Course Wrap-up

4-Day Recap

Day	Topic	Key Skills
Day 1	Fundamentals	Cluster concepts, CRUD, inverted index
Day 2	Query DSL & ES|QL	match, bool, highlighting, pipe syntax
Day 3	Indexing & Analysis	Bulk ops, analyzers, mappings, aggregations
Day 4	Semantic Search	Vectors, ELSER, hybrid RRF, production tips

What to Explore Next

- **Elastic Observability** — Logs, metrics, APM
- **Elastic Security** — SIEM, endpoint protection
- **Index Lifecycle Management** — Automated rollover/delete
- **Cross-cluster Search** — Federated search across clusters
- **Inference Pipelines** — Enrich documents on ingest
- **ES|QL** — Continues to gain features each release

Resources

- **This repository** — keep it for reference and continued practice
- Elastic Documentation
- ES|QL Reference
- Vector Search Guide
- ELSER Documentation
- Elasticsearch Labs Blog

Thank You!

Congratulations on completing the course!

4-Day Elasticsearch Course | Elasticsearch 8.18