Elasticsearch Vector Search & Relevance Deep Dive

Interview Preparation Guide for Elastic Search Team

Overview

This guide provides comprehensive coverage of Elasticsearch vector search, relevance ranking, and neural search for engineers interviewing with Elastic's search relevance team. It covers modern semantic search, embedding-based retrieval, hybrid search strategies, and production optimization techniques.

Key Focus Areas for Elastic Interview:

- **Vector Search**: Dense vectors, k-NN, HNSW, similarity metrics
- **Relevance**: BM25, TF-IDF, scoring algorithms, query-time boosting
- **Neural Search**: Embeddings, ELSER, semantic search, RAG
- **Hybrid Search**: Combining keyword + vector search with RRF
- **Production**: Performance tuning, memory optimization, BBQ quantization

1 Recommended Reading List

1.1 Priority 1: Vector Search Specific

"Vector Search for Practitioners with Elastic"
Authors: Bahaaldine Azarmi, Jeff Vestal, Shay Banon (2023-2024)

Why This Book:

- **Most relevant** - Written specifically for Elasticsearch 8+ vector capabilities
- Foreword by Elastic's founder (Shay Banon)
- Covers production deployment at scale
- Modern techniques: ELSER, RAG, RRF, BBQ quantization

Key Chapters to Focus On:

1. **Chapters 1-3**: NLP fundamentals, embeddings, dense vectors
2. **Chapter 4**: Dense vector storage in Elasticsearch
3. **Chapter 5-6**: k-NN search, HNSW algorithm, performance
4. **Chapter 7**: ELSER (Elastic Learned Sparse EncodeR) - Elastic's secret sauce
5. **Chapter 8**: Neural search and semantic search patterns
6. **Chapter 9**: Hybrid search with RRF (Reciprocal Rank Fusion)
7. **Chapter 10**: RAG (Retrieval-Augmented Generation) with ChatGPT
8. **Chapter 11-12**: Production tuning, node scaling, memory optimization

1.2 Priority 2: Relevance Theory

"Relevant Search: With Applications for Solr and Elasticsearch"
Authors: Doug Turnbull, John Berryman

Why This Book:

- **The bible** of search relevance - explains the "why" behind scoring
- Deep dive into Lucene internals (Elasticsearch's foundation)
- Practical relevance tuning strategies
- Covers both keyword and semantic approaches

Essential Chapters:

1. **Chapter 1-2**: Relevance fundamentals, user intent
2. **Chapter 3**: Lucene scoring (TF-IDF, BM25)
3. **Chapter 4-5**: Query DSL for relevance, boosting strategies
4. **Chapter 8**: Debugging relevance (explain API)
5. **Chapter 10**: Learning to Rank (LTR)

1.3 Priority 3: Elasticsearch Foundations

"Elasticsearch in Action, Second Edition"
Author: Madhusudhan Konda (2023)

Use Case: Refresh on core Elasticsearch concepts if rusty

Focus Areas:

- Cluster architecture, sharding, replicas
- Index mappings, analyzers, tokenizers
- Query DSL mastery
- Aggregations and analytics
- Performance optimization

2 Vector Search Fundamentals

2.1 What is Vector Search?

Traditional Keyword Search (BM25):

```
Query: "best pizza restaurant"
Matches documents with: best, pizza, restaurant
Score based on: term frequency, inverse document frequency
```

Vector Search (Semantic Search):

```
Query: "best pizza restaurant"
Embedding: [0.23, -0.45, 0.67, ..., 0.12] (768-dim vector)
Matches documents with SIMILAR embeddings (even without exact words)
Can match: "top Italian eatery", "great pizzeria"
Score based on: cosine similarity, dot product, or L2 distance
```

Key Insight: Vector search captures *semantic meaning*, not just *keyword overlap*.

2.2 Dense Vectors vs Sparse Vectors

Dense Vectors (Traditional Embeddings):

- **Dimensions**: 384, 768, 1024 (every dimension has a value)
- **Models**: BERT, Sentence Transformers, OpenAI embeddings
- **Storage**: 768 floats × 4 bytes = 3KB per document
- **Use Case**: General semantic search, cross-lingual search
- **Pros**: Captures deep semantic meaning
- **Cons**: High memory usage, slower indexing

Sparse Vectors (ELSER):

- **Dimensions**: 30,000+ (but only 200 non-zero values)
- **Elastic's ELSER**: Learned sparse representations
- **Storage**: Much smaller due to sparsity
- **Use Case**: Domain-specific search, better explainability
- **Pros**: Faster, more interpretable, better for rare terms
- **Cons**: Requires training, less universal than dense vectors

When to Use Which:

- **Dense**: Cross-domain search, multilingual, semantic similarity
- **Sparse (ELSER)**: Domain-specific, explainable results, exact term matching important
- **Hybrid**: Combine both with RRF for best results!

2.3 k-NN Search in Elasticsearch

Exact k-NN (Brute Force):

```
POST /my-index/_search
{
  "query": {
    "script_score": {
      "query": {"match_all": {}},
      "script": {
        "source": "cosineSimilarity(params.query_vector, 'vector_field') + 1.0",
        "params": {"query_vector": [0.2, 0.4, ...]}
      }
    }
  }
}
```

Time Complexity: O(N × D) where N = docs, D = dimensions
Approximate k-NN (HNSW):

```
POST /my-index/_search
{
  "knn": {
    "field": "vector_field",
    "query_vector": [0.2, 0.4, ...],
    "k": 10,
    "num_candidates": 100
  }
}
```

HNSW (Hierarchical Navigable Small World):

- **Time Complexity:** O(log N) - much faster!
- **Tradeoff:** Approximate results (may miss some neighbors)
- **Parameters:**
  - `m`: Max connections per layer (default: 16, higher = better recall, more memory)
  - `ef_construction`: Build-time search depth (default: 100, higher = better graph quality)
  - `num_candidates`: Query-time candidates (trade recall vs latency)
- **Memory:** 8-10 bytes per dimension per document (significant!)

3 Similarity Metrics

3.1 Cosine Similarity

Formula:

$$\text{cosine\_similarity}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

Range: [-1, 1] where 1 = identical direction, 0 = orthogonal, -1 = opposite
Properties:

- Measures angle, not magnitude
- Normalized - good for comparing docs of different lengths
- Most common for text embeddings (BERT, Sentence Transformers)

Elasticsearch: Converts to score: `(1 + cosine) / 2` $\rightarrow$ [0, 1]

3.2 Dot Product

Formula:

$$\text{dot\_product}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{n} a_i \times b_i$$

Range: Unbounded (depends on vector magnitudes)
Properties:

- Faster than cosine (no normalization)
- Magnitude matters - longer vectors score higher
- Good when vector length is meaningful

Use Case: OpenAI embeddings (pre-normalized), product recommendations

3.3 L2 Distance (Euclidean)

Formula:

$$\text{L2\_distance}(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

Range: [0,] where 0 = identical, higher = more different
Properties:

- Measures absolute distance in vector space
- Sensitive to magnitude
- Elasticsearch inverts for scoring: `1 / (1 + L2)`

Use Case: Image embeddings, face recognition

3.4 Which Metric to Choose?

Metric	Best For	Training Model
Cosine	Text embeddings (BERT, ST)	Normalize vectors
Dot Product	Pre-normalized (OpenAI)	Unit vectors
L2	Image, face recognition	Euclidean space

Pro Tip: Match the metric to how your embedding model was trained!

4 Elasticsearch Relevance Scoring

4.1 BM25 (Best Match 25)

The default scoring algorithm in Elasticsearch 7+

Formula:

$$\text{score}(D, Q) = \sum_{t \in Q} \text{IDF}(t) \times \frac{f(t, D) \times (k_1 + 1)}{f(t, D) + k_1 \times (1 - b + b \times \frac{|D|}{\text{avgdl}})}$$

Components:

- **IDF(t)**: Inverse Document Frequency - rare terms score higher
- **f(t, D)**: Term frequency in document
- —**D**—: Document length
- **avgdl**: Average document length in corpus
- **k1**: Term saturation parameter (default: 1.2)
- **b**: Length normalization (default: 0.75)

Key Properties:

- **Term Saturation**: Diminishing returns for repeated terms
- **Length Normalization**: Penalizes very long documents
- **Better than TF-IDF**: Handles term frequency saturation

Tuning Parameters:

```
PUT /my-index
{
  "settings": {
    "index": {
      "similarity": {
        "custom_bm25": {
          "type": "BM25",
          "k1": 1.5,     // Higher = more term freq influence
          "b": 0.5       // Lower = less length normalization
        }
      }
    }
  }
}
```

4.2 Query-Time Boosting

Field Boosting:

```
{
  "query": {
    "multi_match": {
      "query": "search query",
      "fields": ["title^3", "body^1", "tags^2"]
    }
  }
}
```

Function Score:

```
{
  "query": {
    "function_score": {
      "query": {"match": {"title": "pizza"}},
      "functions": [
        {"filter": {"term": {"featured": true}}, "weight": 2},
        {"field_value_factor": {"field": "rating", "modifier": "log1p"}},
        {"gauss": {"location": {"origin": "40.7,-74.0", "scale": "10km"}}}
      ],
      "score_mode": "sum",
      "boost_mode": "multiply"
    }
  }
}
```

Boosting Strategies:

- **Field Boost**: Title ¿ Body ¿ Tags
- **Recency**: Decay function for time-sensitive content
- **Popularity**: Log(views), log(ratings) to prevent dominance
- **Geo-Proximity**: Distance-based boosting
- **Business Rules**: Featured items, paid placements

5 Hybrid Search with RRF

5.1 Why Hybrid Search?

Problem: Keyword search and vector search have different strengths

- **Keyword (BM25)**: Exact matches, rare terms, technical jargon
- **Vector**: Semantic meaning, synonyms, paraphrases

Example:

- Query: "CPU overheating solutions"
- **BM25 misses**: "processor temperature fixes" (no keyword overlap)
- **Vector misses**: "Core i9-13900K thermal throttling" (too specific)
- **Hybrid**: Captures both semantic meaning AND specific terms

5.2 Reciprocal Rank Fusion (RRF)

Algorithm:

$$\text{RRF\_score}(d) = \sum_{r \in R} \frac{1}{k + \text{rank}_r(d)}$$

Where:

- **R**: Set of retrieval methods (e.g., BM25, vector search)
- **rank_r(d)**: Rank of document d in retrieval r
- **k**: Constant (default: 60) to smooth rankings

Example:

```
Document A: BM25 rank=1, Vector rank=5
  RRF_score(A) = 1/(60+1) + 1/(60+5) = 0.0164 + 0.0154 = 0.0318

Document B: BM25 rank=10, Vector rank=2
  RRF_score(B) = 1/(60+10) + 1/(60+2) = 0.0143 + 0.0161 = 0.0304

Winner: Document A (balanced high ranks beats one very high)
```

Elasticsearch RRF Query:

```
POST /my-index/_search
{
  "retriever": {
    "rrf": {
      "retrievers": [
        {
          "standard": {
            "query": {"match": {"title": "pizza restaurant"}}
          }
        },
        {
          "knn": {
            "field": "title_vector",
            "query_vector": [0.2, 0.4, ...],
            "k": 10,
            "num_candidates": 100
          }
        }
      ],
      "rank_constant": 60,
      "rank_window_size": 100
    }
  }
}
```

RRF Advantages:

- **No training required** - parameter-free fusion
- **Robust** - handles different score scales automatically
- **Balanced** - rewards documents that rank well in multiple retrievers
- **Production-ready** - fast, deterministic

6 ELSER: Elastic's Secret Sauce

6.1 What is ELSER?

ELSER = Elastic Learned Sparse EncodeR

Key Idea: Learned sparse vectors that combine:

- Semantic understanding (like dense vectors)
- Interpretability (like keyword search)
- Efficiency (sparse = less memory)

How It Works:

1. Train a model to predict "expansion terms" for a query/document
2. Generate sparse vector: 30,000 dimensions, 200 non-zero
3. Non-zero dimensions = relevant terms with weights
4. Match using efficient inverted index (like BM25)

6.2 ELSER vs Dense Vectors

Aspect	Dense (BERT)	Sparse (ELSER)
Dimensions	768 (all non-zero)	30k (200 non-zero)
Storage	3KB per doc	800 bytes per doc
Indexing	Slow (HNSW build)	Fast (inverted index)
Query Speed	Medium (ANN)	Very Fast (inverted index)
Explainability	Opaque	Transparent (see terms)
Domain Adaptation	Hard (retrain)	Easier (fine-tune)

When to Use ELSER:

- Domain-specific search (legal, medical, e-commerce)
- Need explainable results (why did this match?)
- Memory/cost constrained
- Exact term matching still important
- Elasticsearch 8.8+ available

6.3 ELSER Setup in Elasticsearch

1. Deploy ELSER Model:

```
PUT _ml/trained_models/.elser_model_2
{
  "input": {"field_names": ["text_field"]}
}

POST _ml/trained_models/.elser_model_2/deployment/_start
{
  "number_of_allocations": 2
}
```

2. Create Inference Pipeline:

```
PUT _ingest/pipeline/elser-ingest
{
  "processors": [
    {
      "inference": {
```

```
        "model_id": ".elser_model_2",
        "input_output": [
          {"input_field": "content", "output_field": "content_sparse"}
        ]
      }
    }
  ]
}
```

3. Index with ELSER:

```
PUT /my-elser-index/_doc/1?pipeline=elser-ingest
{
  "content": "Best pizza restaurants in New York"
}
```

4. Query with ELSER:

```
POST /my-elser-index/_search
{
  "query": {
    "text_expansion": {
      "content_sparse": {
        "model_id": ".elser_model_2",
        "model_text": "top Italian eateries NYC"
      }
    }
  }
}
```

7 Production Optimization

7.1 Memory Management for Vector Search

Memory Calculation:

```
Dense Vector Memory = N_docs × Dimensions × 4 bytes × (1 + HNSW_overhead)
HNSW_overhead  2-3x (depending on m parameter)

Example:
10M docs × 768 dims × 4 bytes × 3 = 92 GB
```

Optimization Strategies:
1. Reduce Dimensions (PCA/Quantization):

- 768 dims → 384 dims = 50% memory savings
- Use `element_type: byte` for 8-bit quantization (75% savings)
- Tradeoff: 5-10% recall loss for 4x memory reduction

2. BBQ Quantization (Elasticsearch 8.11+):

```
PUT /my-index
{
  "mappings": {
    "properties": {
      "vector_field": {
        "type": "dense_vector",
        "dims": 768,
        "index": true,
        "similarity": "cosine",
        "index_options": {
          "type": "hnsw",
```

```
          "m": 16,
          "ef_construction": 100
        },
        "element_type": "byte"  // BBQ quantization
      }
    }
  }
}
```

BBQ (Byte-quantized Binary Quantization):

- Quantizes float32 → int8 (1 byte instead of 4)
- 4x memory reduction
- Minimal recall loss (2-3%)
- Faster query time (integer arithmetic)

7.2 Indexing Performance

Bulk Indexing with Vectors:

```
POST _bulk
{"index": {"_index": "my-index", "_id": "1"}}
{"title": "Doc 1", "vector": [0.1, 0.2, ...]}
{"index": {"_index": "my-index", "_id": "2"}}
{"title": "Doc 2", "vector": [0.3, 0.4, ...]}
```

Performance Tips:

1. **Disable Refresh During Bulk**: `index.refresh_interval: -1`
2. **Increase Bulk Size**: 1000-5000 docs per request
3. **Use Multiple Shards**: Parallelize HNSW graph building
4. **Allocate More Heap**: Vector indexing is memory-intensive
5. **SSD Storage**: HNSW graph access is I/O intensive

7.3 Query Performance Tuning

k-NN Parameters:

```
{
  "knn": {
    "field": "vector",
    "query_vector": [...],
    "k": 10,              // Top-k results
    "num_candidates": 100 // Search space (higher = better recall, slower)
  }
}
```

Tuning Guidelines:

- **num_candidates**: Start with 10x k, increase if recall poor
- **Trade-off**: 100 candidates = 10ms, 1000 candidates = 50ms
- **Latency Budget**: Allocate 20-50ms for vector search in p95

Filtering with k-NN:

```
{
  "knn": {
    "field": "vector",
    "query_vector": [...],
    "k": 10,
    "num_candidates": 100,
    "filter": {
      "term": {"category": "restaurants"}
    }
  }
}
```

Warning: Filters are applied *after* k-NN retrieval!

- Bad: Retrieve 100 candidates, 95 filtered out → only 5 results
- Fix: Increase `num_candidates` when using filters
- Rule of Thumb: `num_candidates = k / selectivity`

7.4 Node Sizing for Vector Workloads

Data Node Sizing:

- **CPU**: Vector search is CPU-intensive (similarity calculations)
  - Recommended: 8-16 cores per node
  - Higher clock speed ¿ more cores
- **RAM**: Vectors + HNSW graphs must fit in memory
  - Calculate: (Index size × 3) + 4GB heap
  - Example: 90GB vectors → 270GB RAM + 4GB heap = 274GB
- **Storage**: SSD mandatory for HNSW graph access
  - NVMe preferred for low-latency p99

ML Node Sizing (for ELSER/embeddings):

- **CPU**: Inference is CPU-heavy (or GPU if available)
- **RAM**: Model size × number of allocations
  - ELSER v2: 1GB per allocation
  - Sentence Transformer: 500MB per allocation
- **Throughput**: Scale allocations based on QPS
  - 1 allocation 10-20 inferences/sec
  - 100 QPS → 5-10 allocations

8 Retrieval-Augmented Generation (RAG)

8.1 What is RAG?

Problem: LLMs have outdated knowledge, hallucinate, no private data
Solution: RAG = Retrieve relevant docs + Generate answer with context
RAG Pipeline:

1. **User Query**: "What's our return policy for electronics?"
2. **Retrieve**: Use Elasticsearch (vector/hybrid) to find top-k docs
3. **Augment**: Inject retrieved docs into LLM prompt
4. **Generate**: LLM generates answer grounded in retrieved context

8.2 RAG with Elasticsearch

Architecture:

```
User Query
    ↓
[Embedding Model] → Query Vector
    ↓
[Elasticsearch] → Hybrid Search (BM25 + Vector + ELSER)
    ↓
Top-K Documents
    ↓
[Prompt Template] → Context: {retrieved_docs}\nQuestion: {query}
    ↓
[LLM: GPT-4, Claude] → Generated Answer
    ↓
User
```

Elasticsearch Query for RAG:

```
POST /knowledge-base/_search
{
  "retriever": {
    "rrf": {
      "retrievers": [
        {
          "standard": {
            "query": {"match": {"content": "return policy electronics"}}
          }
        },
        {
          "knn": {
            "field": "content_vector",
            "query_vector": [...],  // Embedded query
            "k": 5,
            "num_candidates": 50
          }
        }
      ]
    }
  },
  "size": 5,
  "_source": ["title", "content", "url"]
}
```

Prompt Construction:

```
results = es.search(index="kb", body=query)

context = "\n\n".join([
    f"[{i+1}] {doc['_source']['title']}\n{doc['_source']['content']}"
    for i, doc in enumerate(results['hits']['hits'])
])

prompt = f"""Use the following context to answer the question.
If the answer is not in the context, say "I don't know."

Context:
{context}

Question: {user_query}

Answer:"""
```

```
answer = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[{"role": "user", "content": prompt}]
)
```

8.3 RAG Optimization Strategies

1. Chunking Strategy:

- **Chunk Size**: 256-512 tokens (balance context vs precision)
- **Overlap**: 50-100 tokens to avoid boundary issues
- **Metadata**: Store parent doc ID, section headers

2. Re-ranking After Retrieval:

```
Elasticsearch (k=50) → Re-ranker Model (top-5) → LLM
```

- Use cross-encoder (BERT-based) for fine-grained relevance
- 10x more compute than retrieval, but 10x better precision

3. Hybrid Retrieval:

- Combine BM25 (exact matches) + Vector (semantic) + ELSER (learned sparse)
- Use RRF to fuse results
- Higher recall → better RAG quality

4. Query Expansion:

- Generate multiple variations of user query
- Retrieve for each variation, merge results
- Example: "return policy" → ["refund policy", "exchange policy", "money back guarantee"]

5. Prompt Engineering:

- **Citation**: Ask LLM to cite source document [1], [2]
- **Confidence**: Ask LLM to indicate confidence level
- **Brevity**: Limit answer length to avoid hallucination

9 Interview Preparation Strategy

9.1 Week 1: Deep Dive (7 days before)

Reading Plan:

1. **Days 1-3**: "Vector Search for Practitioners" chapters 1-7
   - Focus: Dense vectors, k-NN, HNSW, ELSER
   - Take notes on production considerations
2. **Days 4-5**: "Relevant Search" chapters 1-5
   - Focus: BM25, scoring, relevance tuning
   - Practice explain API examples
3. **Days 6-7**: Hands-on lab (see below)

9.2 Hands-On Lab (2-3 days before)

Setup:

```
# Start Elasticsearch 8.x locally
docker run -p 9200:9200 -e "discovery.type=single-node" \
  docker.elastic.co/elasticsearch/elasticsearch:8.11.0
```

Lab Exercises:

1. **Index Documents with Vectors**:
   - Generate embeddings with Sentence Transformers
   - Index 1000+ documents with dense_vector field
   - Test different similarity metrics (cosine, dot_product, L2)
2. **k-NN Queries**:
   - Run exact k-NN (script_score)
   - Run approximate k-NN (HNSW)
   - Compare latency and recall
   - Test different num_candidates values
3. **Hybrid Search with RRF**:
   - Create BM25 query
   - Create k-NN query
   - Combine with RRF retriever
   - Compare results vs individual methods
4. **ELSER (if time permits)**:
   - Deploy ELSER model
   - Create inference pipeline
   - Index documents with ELSER
   - Query with text_expansion
5. **Performance Tuning**:
   - Benchmark query latency
   - Test BBQ quantization (element_type: byte)
   - Measure memory usage
   - Tune HNSW parameters (m, ef_construction)

9.3 Day Before Interview: Review & Mock

Elasticsearch Docs to Review:

- k-NN search: `https://www.elastic.co/guide/en/elasticsearch/reference/current/knn-search.html`
- Dense vector field: `https://www.elastic.co/guide/en/elasticsearch/reference/current/dense-vector.html`
- RRF retriever: `https://www.elastic.co/guide/en/elasticsearch/reference/current/rrf.html`
- ELSER: `https://www.elastic.co/guide/en/machine-learning/current/ml-nlp-elser.html`

Recent Elastic Blogs (2024):

- BBQ quantization for memory optimization

- Semantic reranking improvements
- ELSER v2 updates
- Vector search performance benchmarks

Mock Interview Topics:

1. **System Design**: "Design a semantic search system for 100M documents"
   - Discuss indexing pipeline, embedding generation, k-NN configuration
   - Talk through latency, memory, cost tradeoffs
2. **Relevance Debugging**: "BM25 scores are too low for short docs"
   - Explain BM25 formula, length normalization
   - Suggest tuning b parameter, field boosting
3. **Vector Search Optimization**: "k-NN queries are slow at p99"
   - Discuss HNSW tuning, num_candidates
   - Suggest BBQ quantization, dimension reduction
4. **Hybrid Search**: "When to use keyword vs vector vs hybrid?"
   - Compare use cases, strengths, weaknesses
   - Explain RRF fusion strategy

10 Key Discussion Points for Interview

10.1 Vector Search Tradeoffs

Be Prepared to Discuss:

1. Dimensionality Tradeoffs:

- **Higher dims (768, 1024)**: Better semantic representation, more memory
- **Lower dims (384, 256)**: Less memory, faster, slight quality loss
- **When to reduce**: Cost-constrained, latency-sensitive, large corpus

2. Exact vs Approximate k-NN:

- **Exact (Brute Force)**: 100% recall, O(N) time, too slow for ¿10k docs
- **Approximate (HNSW)**: 90-95% recall, O(log N) time, production-ready
- **When to use exact**: Small corpus, offline batch jobs, quality critical

3. Dense vs Sparse Vectors:

- **Dense**: General-purpose, cross-domain, semantic understanding
- **Sparse (ELSER)**: Domain-specific, explainable, efficient
- **Hybrid**: Combine both with RRF for best results

4. Memory vs Latency vs Cost:

- **High memory**: Store full-precision vectors, fast queries
- **Low memory**: BBQ quantization, slower indexing, slight recall loss
- **Tradeoff**: 4x memory savings vs 2-3% recall drop

10.2 Production Considerations

Scaling Vector Search:

1. **Indexing Throughput**:
   - Bottleneck: HNSW graph construction
   - Solution: Multiple shards, parallel indexing, SSD storage
   - Benchmark: 1000 docs/sec with 768-dim vectors on 8-core node
2. **Query Latency**:
   - Target: p50 ¡ 20ms, p95 ¡ 50ms, p99 ¡ 100ms
   - Tuning: num_candidates, HNSW m/ef_construction, BBQ quantization
   - Monitoring: Track slow queries, cache hit rates
3. **Memory Management**:
   - Calculate: (N_docs × dims × 4 bytes × 3) for HNSW
   - Mitigation: BBQ quantization, dimension reduction, tiered storage
4. **Model Deployment**:
   - ELSER/embedding models on ML nodes
   - Allocations based on QPS: 1 allocation = 10-20 req/sec
   - Failover: Multiple allocations for HA

10.3 Recent Elasticsearch Innovations (2024)

Highlight These in Interview:

1. BBQ Quantization:

- Float32 → Int8 (1 byte) for 4x memory savings
- Minimal recall loss (2-3%)
- Faster query time (integer arithmetic)

2. RRF (Reciprocal Rank Fusion):

- Parameter-free fusion of multiple retrievers
- Combines BM25 + Vector + ELSER seamlessly
- Better than simple score combination

3. ELSER v2:

- Improved sparse representations
- Better domain adaptation
- Lower latency than dense vectors

4. Semantic Reranking:

- Two-stage retrieval: Fast k-NN → Slow cross-encoder
- 10x quality improvement for top-k results
- Use for RAG, question answering

11 Common Interview Questions

11.1 Technical Deep Dives

Q1: How does HNSW work? Why is it faster than brute-force k-NN?
Answer:

- HNSW builds a hierarchical graph with multiple layers
- Top layer: Sparse, long-range connections (navigation)
- Bottom layer: Dense, short-range connections (refinement)
- Search starts at top, greedily navigates to bottom
- Time complexity: O(log N) vs O(N) for brute-force
- Tradeoff: Approximate results, memory overhead (2-3x)
- Parameters: m (connections per layer), ef_construction (build quality)

Q2: Explain BM25. Why is it better than TF-IDF?
Answer:

- BM25 = Best Match 25, improved TF-IDF with saturation
- Key improvement: Term frequency saturation (diminishing returns)
  - TF-IDF: Linear growth with term freq → spam vulnerability
  - BM25: Logarithmic saturation → natural text favored
- Length normalization with parameter b:
  - b=1: Full normalization (penalize long docs)
  - b=0: No normalization (favor long docs)
  - Default b=0.75: Balanced
- k1 parameter: Term saturation rate (default 1.2)

Q3: How would you debug poor search relevance?
Answer:

1. **Use Explain API**:

```
GET /index/_explain/doc_id
{
  "query": {"match": {"field": "query"}}
}
```

   - Shows scoring breakdown: TF, IDF, field norms
   - Identify which component is off

2. **Check Analyzers**:

```
POST /_analyze
{
  "analyzer": "standard",
  "text": "query text"
}
```

   - Verify tokenization, stemming, stopwords

- Mismatch between index and query analysis

3. **Review Mapping**:
   - Check field types, analyzers, similarity settings
   - Verify boosting, multi-field setup
4. **Tune BM25 Parameters**:
   - Adjust k1 (term saturation)
   - Adjust b (length normalization)
5. **Add Field Boosting**:
   - Boost title ¿ body ¿ tags
   - Use function_score for business logic

Q4: When would you use cosine vs dot product vs L2 distance?
Answer:

- **Cosine Similarity**:
  - Use for: Text embeddings (BERT, Sentence Transformers)
  - Why: Normalized, focuses on direction not magnitude
  - Training: Models trained with cosine loss
- **Dot Product**:
  - Use for: Pre-normalized embeddings (OpenAI, Cohere)
  - Why: Faster (no normalization), magnitude meaningful
  - Training: Models output unit vectors
- **L2 Distance**:
  - Use for: Image embeddings, face recognition
  - Why: Euclidean distance in latent space
  - Training: Triplet loss, contrastive loss
- **Key Rule**: Match the metric to how the model was trained!

11.2 System Design Questions

Q5: Design a semantic search system for 100M documents.
Answer Outline:
1. Requirements Clarification:

- QPS: 1000 queries/sec
- Latency: p95 ¡ 100ms
- Index updates: Real-time or batch?
- Budget: Memory/cost constraints?

2. High-Level Architecture:

```
User Query → Embedding API → Elasticsearch Cluster
                                    ↓
                          [Hybrid Search: BM25 + k-NN]
                                    ↓
                               Top-K Results
```

3. Embedding Generation:

- **Model**: Sentence Transformers (all-MiniLM-L6-v2, 384 dims)
- **Deployment**: Dedicated embedding service (5-10 instances)
- **Caching**: Cache query embeddings (Redis, 1hr TTL)

4. Elasticsearch Setup:

- **Cluster**: 10 data nodes, 3 master nodes
- **Shards**: 50 primary shards (2M docs per shard)
- **Replicas**: 1 replica for HA
- **Node Specs**: 32 cores, 256GB RAM, NVMe SSD

5. Memory Calculation:

```
Vector memory = 100M docs × 384 dims × 4 bytes × 3 (HNSW) = 461 GB
BM25 index = 100M docs × 1KB avg = 100 GB
Total per shard = (461 + 100) / 50 = 11 GB
Node capacity = 256GB RAM / 11GB = 23 shards per node
```

6. Query Strategy:

- **Hybrid Search**: RRF fusion of BM25 + k-NN
- **k-NN Params**: k=10, num_candidates=100
- **Optimization**: BBQ quantization for 4x memory savings

7. Indexing Pipeline:

- **Batch Processing**: Kafka → Embedding Service → Elasticsearch
- **Throughput**: 10k docs/sec (100 indexing clients)
- **Refresh Interval**: 30s for near-real-time

8. Monitoring:

- Query latency (p50, p95, p99)
- Indexing lag, backlog size
- Memory usage, GC pauses
- k-NN recall metrics (offline)

Q6: How would you improve search relevance for e-commerce?
Answer:
1. Hybrid Search Foundation:

- **BM25**: Exact product names, SKUs, brands
- **Vector**: Semantic similarity ("running shoes" → "sneakers")
- **RRF**: Fuse both for balanced results

2. Query Understanding:

- **Query Expansion**: Synonyms ("laptop" → "notebook")
- **Query Correction**: Spell check ("iphon" → "iphone")
- **Intent Detection**: Navigate ("nike") vs Browse ("running shoes")

3. Feature Engineering:

- **Popularity**: log(sales), log(views) as boost
- **Recency**: Decay for seasonal products
- **Price**: Match user's price range
- **Ratings**: Boost high-rated products
- **Availability**: Penalize out-of-stock

4. Personalization:

- **User History**: Boost categories user browses
- **Collaborative**: "Users who bought X also bought Y"
- **A/B Testing**: Test personalized vs generic

5. Business Rules:

- **Promoted Products**: Boost sponsored items
- **Diversity**: Don't show 10 variants of same product
- **Merchandising**: Seasonal campaigns, new arrivals

6. Evaluation:

- **Offline**: NDCG, MRR on human-labeled data
- **Online**: CTR, conversion rate, revenue per search
- **A/B Testing**: Compare relevance improvements vs baseline

12 Summary Checklist

12.1 Must-Know Concepts

Vector Search:

☐ Dense vs sparse vectors (ELSER)

☐ k-NN: Exact (brute-force) vs Approximate (HNSW)

☐ Similarity metrics: Cosine, Dot Product, L2

☐ HNSW algorithm, parameters (m, ef_construction)

☐ Memory calculation for vector indices

☐ BBQ quantization for memory optimization

Relevance:

☐ BM25 formula, parameters (k1, b)

☐ TF-IDF vs BM25 differences

☐ Field boosting, function_score

☐ Explain API for debugging

☐ Analyzers, tokenizers, filters

Hybrid Search:

☐ Why hybrid search? (keyword + semantic)

☐ RRF (Reciprocal Rank Fusion) algorithm

☐ Elasticsearch RRF retriever syntax

☐ When to use hybrid vs pure vector

Production:

☐ Node sizing (CPU, RAM, storage)

☐ Indexing performance tuning

☐ Query latency optimization (num_candidates)

☐ Filtering with k-NN (selectivity issues)

☐ Monitoring and alerting

Advanced:

☐ ELSER setup and querying

☐ RAG architecture with Elasticsearch

☐ Semantic reranking strategies

☐ Recent Elastic innovations (2024)

12.2 Hands-On Lab Checklist

☐ Setup Elasticsearch 8.x locally

☐ Generate embeddings with Sentence Transformers

☐ Index documents with dense_vector field

☐ Run k-NN queries (exact and approximate)

☐ Test different similarity metrics

☐ Implement hybrid search with RRF

☐ Benchmark query latency

☐ Test BBQ quantization

☐ (Optional) Deploy and test ELSER

☐ (Optional) Build simple RAG pipeline

12.3 Interview Day Checklist

Morning Of:

☐ Review BM25 formula and parameters

☐ Review HNSW algorithm intuition

☐ Review RRF fusion formula

☐ Skim Elasticsearch vector search docs

☐ Practice explaining concepts out loud

During Interview:

- ☐ Ask clarifying questions (requirements, constraints)
- ☐ Think out loud (show reasoning process)
- ☐ Discuss tradeoffs (memory vs latency vs quality)
- ☐ Mention recent Elastic innovations (BBQ, ELSER, RRF)
- ☐ Draw diagrams (architecture, pipelines)
- ☐ Be honest about what you don't know
- ☐ Show enthusiasm for vector search/relevance work!

13 Final Thoughts

Key Strengths to Highlight:

- Experience with search systems (even if not recent)
- Strong ML fundamentals (embeddings, neural networks)
- Production systems thinking (scaling, monitoring, tradeoffs)
- Curiosity about modern techniques (ELSER, RAG, hybrid search)

Topics to Show Depth:

- Vector search is not magic - understand HNSW, memory, latency
- Hybrid search ¿ pure vector - leverage strengths of both
- Production is hard - discuss real-world constraints
- Evaluation matters - offline metrics, online A/B tests

Questions to Ask Interviewer:

- What are the biggest challenges in Elastic's vector search today?
- How is ELSER adoption vs dense vectors in production?
- What's the roadmap for hybrid search improvements?
- How does the team evaluate relevance quality?
- What ML/NLP techniques is the team exploring next?

Good luck with your interview! You've got this!