Indexing, Text Analysis & Aggregations

Day 3 — 4-Day Elasticsearch Course

Elasticsearch 8.18 | Bulk API · Analyzers · Mappings · Aggregations · Nested/Join

Agenda

Index API & Bulk Operations

Efficient data management

Index Settings

Every index has configurable settings:

```
PUT /my-index
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0,
    "refresh_interval": "5s",
    "analysis": { ... }
  },
  "mappings": {
    "properties": { ... }
  }
}
```

Setting	Default	Note
`number_of_shards`	1	Fixed at creation
`number_of_replicas`	1	Can change dynamically

Bulk API

Index many documents in one request:

```
POST /_bulk
{"index": {"_index": "movies", "_id": "200"}}
{"title": "Movie A", "genres": ["Drama"], "vote_average": 7.5}
{"index": {"_index": "movies", "_id": "201"}}
{"title": "Movie B", "genres": ["Action"], "vote_average": 6.8}
{"delete": {"_index": "movies", "_id": "999"}}
{"update": {"_index": "movies", "_id": "1"}}
{"doc": {"vote_average": 9.0}}
```

Actions: `index`, `create`, `update`, `delete`

Performance tip: Optimal bulk size is 5-15 MB per request (not document count).

Bulk API: Error Handling

The bulk response tells you which operations succeeded/failed:

```
{
  "took": 30,
  "errors": true,
  "items": [
    { "index": { "_id": "200", "status": 201, "result": "created" } },
    { "index": { "_id": "201", "status": 409, "error": { ... } } }
  ]
}
```

- Check `errors: true` to see if any operations failed
- Each item has its own `status` code
- Failed items don't affect successful ones (partial success)

Update Operations

Partial update (merge)

```
POST /movies/_update/1
{
  "doc": {
    "tagline": "Hope can set you free"
  }
}
```

Script update

```
POST /movies/_update/1
{
  "script": {
    "source": "ctx._source.vote_average += params.boost",
    "params": { "boost": 0.5 }
  }
}
```

Update Operations: Upsert

Upsert (insert if not exists)

```
POST /movies/_update/42
{
  "doc": { "vote_average": 8.0 },
  "upsert": { "title": "New Movie", "vote_average": 8.0 }
}
```

- If document `42` exists → merge `doc` fields
- If document `42` does not exist → insert the `upsert` body

Delete Operations

Delete by ID

```
DELETE /movies/_doc/42
```

Delete by query

```
POST /movies/_delete_by_query
{
  "query": {
    "range": {
      "vote_average": { "lt": 5.0 }
    }
  }
}
```

`_delete_by_query` is a heavy operation — it scans and deletes matching documents. Use with caution in production.

Refresh & Flush

```
Document → In-memory buffer → [refresh] → Segment (searchable)
                                                ↓
                                          [flush] → Disk (durable)
```

```
# Force refresh (make recent docs searchable)
POST /movies/_refresh

# Force flush (write to disk)
POST /movies/_flush
```

Operation	Default Interval	Purpose
Refresh	1 second	Make docs searchable
Flush	Automatic	Persist to disk (translog)

Disable refresh during bulk loading: `"refresh_interval": "-1"` then re-enable after.

Reindex API

Copy documents from one index to another:

```
POST /_reindex
{
  "source": {
    "index": "movies"
  },
  "dest": {
    "index": "movies-v2"
  }
}
```

With query filter

```
POST /_reindex
{
  "source": {
    "index": "movies",
    "query": {
      "range": { "vote_average": { "gte": 7.0 } }
```

Aliases

A virtual name that points to one or more indices:

```
# Create an alias
POST /_aliases
{
  "actions": [
    { "add": { "index": "movies-v1", "alias": "movies" } }
  ]
}

# Swap alias to new index (atomic)
POST /_aliases
{
  "actions": [
    { "remove": { "index": "movies-v1", "alias": "movies" } },
    { "add": { "index": "movies-v2", "alias": "movies" } }
  ]
}
```

Use cases:

Index Templates

Automatically apply settings/mappings when new indices match a pattern:

```
PUT /_index_template/movies-template
{
  "index_patterns": ["movies-*"],
  "template": {
    "settings": {
      "number_of_shards": 1,
      "number_of_replicas": 0
    },
    "mappings": {
      "properties": {
        "title": { "type": "text", "analyzer": "english" },
        "genres": { "type": "keyword" },
        "vote_average": { "type": "float" },
        "release_date": { "type": "date" }
      }
    }
  },
  "priority": 100
}
```

Practice 3A

Indexing Operations

`day3-exercises.md` — Part A (5 tasks) | ~20 min

Text Analysis

How Elasticsearch understands text

The Analysis Pipeline

```
"The Quick Brown FOX jumped!"
       ↓
┌──────────────────────┐
│   Character Filter   │  → Strip HTML, map characters
├──────────────────────┤
│      Tokenizer       │  → Split into tokens
├──────────────────────┤
│    Token Filters     │  → Lowercase, stemming, stop words
└──────────────────────┘
       ↓
["quick", "brown", "fox", "jump"]
```

Each step transforms the text. The final tokens go into the **inverted index**.

The _analyze API

Test any analyzer interactively:

```
GET /_analyze
{
  "analyzer": "standard",
  "text": "The Quick Brown FOX jumped over the lazy dog!"
}
```

Response:

```
{
  "tokens": [
    { "token": "the", "position": 0 },
    { "token": "quick", "position": 1 },
    { "token": "brown", "position": 2 },
    { "token": "fox", "position": 3 },
    { "token": "jumped", "position": 4 },
    { "token": "over", "position": 5 },
    { "token": "the", "position": 6 },
```

Built-in Analyzers

Analyzer	Tokenizer	Token Filters	Output for "The Quick FOX!"
`standard`	standard	lowercase	`[the, quick, fox]`
`english`	standard	lowercase, stop, stemmer	`[quick, fox]`
`whitespace`	whitespace	(none)	`[The, Quick, FOX!]`
`keyword`	keyword	(none)	`[The Quick FOX!]`
`simple`	letter	lowercase	`[the, quick, fox]`

Standard Analyzer

```
GET /_analyze
{
  "analyzer": "standard",
  "text": "The runners were running quickly"
}
// → ["the", "runners", "were", "running", "quickly"]
```

- Just **lowercase + split** on whitespace/punctuation
- Keeps all tokens including stop words ("the", "were")

English Analyzer

```
GET /_analyze
{
  "analyzer": "english",
  "text": "The runners were running quickly"
}
// → ["runner", "run", "quick"]
```

- **Removes stop words** ("the", "were")
- **Stems** words → `"running"` and `"runs"` both become `"run"`
- Much better for full-text search in English

Custom Analyzers

Build your own analysis pipeline:

```
PUT /my-index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": {
          "type": "custom",
          "char_filter": ["html_strip"],
          "tokenizer": "standard",
          "filter": ["lowercase", "stop", "snowball"]
        }
      }
    }
  }
}
```

Testing Custom Analyzers

```
GET /my-index/_analyze
{
  "analyzer": "my_custom_analyzer",
  "text": "<p>The Quick Runners are running!</p>"
}
// → ["quick", "runner", "run"]
```

- `html_strip` removes `<p>` tags
- `lowercase` → `the quick runners are running`
- `stop` removes "the", "are"
- `snowball` stems "runners" → "runner", "running" → "run"

Synonym Filter

Map related terms to each other:

```
PUT /movies-synonyms
{
  "settings": {
    "analysis": {
      "filter": {
        "my_synonyms": {
          "type": "synonym",
          "synonyms": [
            "film,movie,picture",
            "scary,horror,frightening",
            "sci-fi => science fiction"
          ]
        }
      },
      ...
    }
  }
}
```

Synonym Filter: Analyzer

```
PUT /movies-synonyms
{
  "settings": {
    "analysis": {
      "filter": { "my_synonyms": { ... } },
      "analyzer": {
        "synonym_analyzer": {
          "tokenizer": "standard",
          "filter": ["lowercase", "my_synonyms"]
        }
      }
    }
  }
}
```

- `"film,movie,picture"` — bidirectional: all terms expand to each other
- `"sci-fi => science fiction"` — one-way mapping

Edge N-gram: Autocomplete

Tokenize prefix substrings for type-ahead search:

```
PUT /autocomplete-index
{
  "settings": {
    "analysis": {
      "filter": {
        "edge_ngram_filter": {
          "type": "edge_ngram",
          "min_gram": 2,
          "max_gram": 15
        }
      },
      "analyzer": {
        "autocomplete_analyzer": {
          "tokenizer": "standard",
          "filter": ["lowercase", "edge_ngram_filter"]
        }
      }
    }
  }
}
```

search_analyzer

Use different analyzers for **indexing** vs **searching**:

```
PUT /autocomplete-index
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "autocomplete_analyzer",
        "search_analyzer": "standard"
      }
    }
  }
}
```

- **Index time:** `"Inception"` → `["in", "inc", "ince", ...]`
- **Search time:** `"inc"` → `["inc"]` (standard, no edge_ngram)
- Result: typing `"inc"` matches the indexed token `"inc"` → autocomplete works!

Mappings & Field Types

Defining your schema

Dynamic vs Explicit Mapping

Dynamic mapping (auto-detected)

```
POST /my-index/_doc/1
{
  "name": "John",        // → text + keyword
  "age": 30,             // → long
  "score": 9.5,          // → float
  "active": true,        // → boolean
  "created": "2024-01-01" // → date
}
```

Explicit mapping (defined upfront)

```
PUT /my-index
{
  "mappings": {
    "properties": {
      "name": { "type": "text" },
      "age": { "type": "integer" },
      "score": { "type": "float" }
```

Common Field Types

Type	Use Case	Example
`text`	Full-text search (analyzed)	Title, description
`keyword`	Exact match, sort, aggregate	Genre, status, email
`integer` / `long`	Whole numbers	Count, ID
`float` / `double`	Decimal numbers	Price, rating
`date`	Dates and timestamps	`"2024-01-15"`
`boolean`	True/false	`active`, `published`
`object`	Nested JSON (flattened)	Address, metadata
`nested`	Independent inner objects	Tags with attributes
`dense_vector`	ML embeddings	384-dim vectors (Day 4)

text vs keyword

```
"properties": {
  "title": {
    "type": "text",            // Analyzed → inverted index
    "analyzer": "english"      // "The Godfather" → ["godfather"]
  },
  "status": {
    "type": "keyword"          // Exact value → "Published"
  }
}
```

	`text`	`keyword`
Analyzed?	Yes (tokenized)	No (exact string)
Search with	`match`, `multi_match`	`term`, `terms`
Sortable?	No	Yes
Aggregatable?	No	Yes

Multi-fields

Index the same data in **multiple ways**:

```
"properties": {
  "title": {
    "type": "text",
    "analyzer": "english",
    "fields": {
      "keyword": {
        "type": "keyword",
        "ignore_above": 256
      },
      "autocomplete": {
        "type": "text",
        "analyzer": "autocomplete_analyzer"
      }
    }
  }
}
```

Access via: `title` (full-text), `title.keyword` (exact), `title.autocomplete` (type-ahead)

Viewing & Updating Mappings

View current mapping

```
GET /movies/_mapping
```

Add a new field (non-breaking)

```
PUT /movies/_mapping
{
  "properties": {
    "tagline": {
      "type": "text"
    }
  }
}
```

You **cannot change** an existing field's type. You must reindex to a new index with the correct mapping.

Practice 3B

Analyzers & Mappings

`day3-exercises.md` — Part B (5 tasks) | ~25 min

Aggregations

Analytics and data exploration

Aggregation Basics

Aggregations let you **summarize data** alongside search results:

```
GET /movies/_search
{
  "size": 0,
  "aggs": {
    "my_agg_name": {
      "agg_type": {
        "field": "field_name"
      }
    }
  }
}
```

- `size: 0` — skip search hits, only return aggregation results
- Three types: **Metric**, **Bucket**, **Pipeline**
- Aggregations can be **nested** (bucket inside bucket)

Metric Aggregations

Calculate a single value from documents:

```
GET /movies/_search
{
  "size": 0,
  "aggs": {
    "avg_rating": {
      "avg": { "field": "vote_average" }
    },
    "max_rating": {
      "max": { "field": "vote_average" }
    },
    "rating_stats": {
      "stats": { "field": "vote_average" }
    },
    "unique_genres": {
      "cardinality": { "field": "genres" }
    }
  }
}
```

Bucket Aggregations: terms

Group documents into buckets:

```
GET /movies/_search
{
  "size": 0,
  "aggs": {
    "genres_breakdown": {
      "terms": {
        "field": "genres",
        "size": 10
      }
    }
  }
}
```

Response:

```
"genres_breakdown": {
  "buckets": [
```

Bucket Aggregations: date_histogram

Group by time intervals:

```
GET /movies/_search
{
  "size": 0,
  "aggs": {
    "movies_by_decade": {
      "date_histogram": {
        "field": "release_date",
        "calendar_interval": "year"
      }
    }
  }
}
```

Fixed intervals

```
"fixed_interval": "30d"    // Every 30 days
"calendar_interval": "month"  // Calendar month
"calendar_interval": "year"   // Calendar year
```

Bucket Aggregations: range & histogram

Numeric range buckets

```
GET /movies/_search
{
  "size": 0,
  "aggs": {
    "rating_ranges": {
      "range": {
        "field": "vote_average",
        "ranges": [
          { "to": 6.0, "key": "Low" },
          { "from": 6.0, "to": 8.0, "key": "Medium" },
          { "from": 8.0, "key": "High" }
        ]
      }
    }
  }
}
```

Histogram (even intervals)

Nested Aggregations

Put metric aggs **inside** bucket aggs:

```
GET /movies/_search
{
  "size": 0,
  "aggs": {
    "genres": {
      "terms": { "field": "genres", "size": 5 },
      "aggs": {
        "avg_rating": {
          "avg": { "field": "vote_average" }
        },
        "best_rated": {
          "max": { "field": "vote_average" }
        }
      }
    }
  }
}
```

Aggregations + Query Scope

Aggregations run on the **filtered result set**:

```
GET /movies/_search
{
  "size": 0,
  "query": {
    "range": {
      "release_date": {
        "gte": "2000-01-01"
      }
    }
  },
  "aggs": {
    "genre_counts": {
      "terms": { "field": "genres", "size": 10 }
    }
  }
}
```

Only movies from 2000+ are included in the aggregation.

Pipeline Aggregations

Compute on the output of **other** aggregations:

```
GET /movies/_search
{
  "size": 0,
  "aggs": {
    "by_year": {
      "date_histogram": {
        "field": "release_date",
        "calendar_interval": "year"
      },
      "aggs": {
        "avg_rating": {
          "avg": { "field": "vote_average" }
        }
      }
    },
    "max_avg_rating_year": {
      "max_bucket": {
        "buckets_path": "by_year>avg_rating"
      }
    }
  }
```

ES|QL vs Aggregations

The same analysis, two syntaxes:

Query DSL Aggregations

```
GET /movies/_search
{
  "size": 0,
  "aggs": {
    "genres": {
      "terms": { "field": "genres", "size": 5 },
      "aggs": {
        "avg_rating": { "avg": { "field": "vote_average" } }
      }
    }
  }
}
```

ES|QL

```
FROM movies
```

Kibana Lens Visualizations

Turn aggregations into charts without code:

1. **Kibana → Visualize → Create visualization → Lens**
2. Drag fields to axes
3. Lens auto-generates the right aggregation type

Try these:

- Bar chart: movie count by genre (`genres` on X-axis)
- Line chart: average rating by year (`release_date` with `date_histogram`)
- Pie chart: rating distribution (`vote_average` with `range`)

> Lens is the recommended way to build dashboards in Kibana.

Practice 3C

Aggregations

`day3-exercises.md` — Part C (6 tasks) | ~25 min

Nested & Join Types

Modeling relationships in Elasticsearch

The Problem with Object Arrays

```
PUT /blog/_doc/1
{
  "title": "ES Guide",
  "comments": [
    { "author": "Alice", "rating": 5 },
    { "author": "Bob", "rating": 2 }
  ]
}
```

Internally, ES **flattens** this:

```
{
  "comments.author": ["Alice", "Bob"],
  "comments.rating": [5, 2]
}
```

Query: "Find posts where Alice gave rating 2" → **false positive!**

The association between author and rating is **lost**.

Solution: Nested Type

```
PUT /blog
{
  "mappings": {
    "properties": {
      "title": { "type": "text" },
      "comments": {
        "type": "nested",
        "properties": {
          "author": { "type": "keyword" },
          "rating": { "type": "integer" }
        }
      }
    }
  }
}
```

Each object in the array is stored as a **separate hidden document** → associations preserved.

Nested Query

```
GET /blog/_search
{
  "query": {
    "nested": {
      "path": "comments",
      "query": {
        "bool": {
          "must": [
            { "term": { "comments.author": "Alice" } },
            { "range": { "comments.rating": { "gte": 4 } } }
          ]
        }
      }
    }
  }
}
```

Now correctly finds posts where **Alice specifically** rated >= 4.

Nested queries require the `nested` wrapper with the `path` parameter.

Join Field Type (Parent-Child)

For truly independent documents with a relationship:

```
PUT /blog-pc
{
  "mappings": {
    "properties": {
      "join_field": {
        "type": "join",
        "relations": {
          "post": "comment"
        }
      },
      "title": { "type": "text" },
      "body": { "type": "text" },
      "author": { "type": "keyword" }
    }
  }
}
```

Parent-Child: Indexing

```
# Parent document
PUT /blog-pc/_doc/1
{
  "title": "ES Guide",
  "body": "Learn Elasticsearch...",
  "join_field": "post"
}

# Child document (must specify routing!)
PUT /blog-pc/_doc/c1?routing=1
{
  "author": "Alice",
  "body": "Great post!",
  "join_field": {
    "name": "comment",
    "parent": "1"
  }
}
```

Children **must** be routed to the same shard as their parent

has_child / has_parent Queries

Find posts that have a comment by Alice

```
GET /blog-pc/_search
{
  "query": {
    "has_child": {
      "type": "comment",
      "query": {
        "term": { "author": "Alice" }
      }
    }
  }
}
```

Find comments whose parent post contains "ES"

```
GET /blog-pc/_search
{
  "query": {
    "has_parent": {
```

When to Use What

Approach	Best For	Trade-offs
Flat / denormalized	Most cases	Data duplication, simple queries
Nested	Small arrays that change together	Extra hidden docs, nested query syntax
Parent-child (join)	Large child sets, independent updates	Slower queries, routing required

Default choice: Denormalize. Use nested/join only when you have a clear need.

Practice 3D

Nested & Join Types

`day3-exercises.md` — Part D (3 tasks) | ~15 min

Summary

Day 3 Recap

Topic	Key Concepts
Index API	Bulk operations, settings, refresh/flush
Reindex & Aliases	Zero-downtime reindexing, alias swaps
Templates	Auto-apply settings to matching indices
Text Analysis	Char filters → Tokenizer → Token filters
Analyzers	standard, english, custom, synonyms, edge_ngram
Mappings	Dynamic vs explicit, text vs keyword, multi-fields
Aggregations	Metric, bucket, nested, pipeline
Nested/Join	Nested objects, parent-child relations

Day 4 Preview — Semantic Search

Tomorrow (3-hour session):

- **Vector Search** — dense_vector, kNN queries, similarity metrics
- **ELSER** — Elastic's built-in ML model, semantic_text field
- **Hybrid Search** — Combining BM25 + kNN with RRF
- **Advanced** — Quantization, chunking, production tips

> **Important:** Start the `elk-ml` stack **before** class tomorrow!

```
docker compose -f docker/elk-ml/docker-compose.yml --env-file .env up -d
```

This stack needs 8GB+ RAM and takes a few minutes to initialize.

Thank You!

Questions?

Day 3 — Indexing, Text Analysis & Aggregations