Day 3 Exercises: Indexing, Text Analysis & Aggregations

Stack: `elk-single` or `elastic` | **Duration:** ~85 minutes total | **Kibana Dev Tools:** `http://localhost:5601`

Prerequisites

- Movies dataset loaded (small: 100 docs)
- Kibana sample data loaded (eCommerce orders)

Part A: Indexing Operations (5 tasks, ~20 min)

Task 1: Load Full Dataset (Basic)

Load the full movies dataset (5000 documents):

```
python data/load_data.py --dataset movies --size full
```

Verify in Dev Tools:

```
GET movies/_count

GET movies/_search
{
  "size": 0,
  "aggs": {
    "genre_counts": {
      "terms": { "field": "genres", "size": 20 }
    }
  }
}
```

Task 1: Questions

Questions:

1. How many documents are in the index now?
2. What are the top 5 genres by document count?

Task 2: Bulk Operations (Basic)

Use the Bulk API to perform these operations in a **single request**:

1. Index a new movie (ID: 9001) -- title: "Bulk Test Movie", genres: ["Action"], vote_average: 7.5
2. Index another movie (ID: 9002) -- title: "Bulk Test Movie 2", genres: ["Drama"], vote_average: 8.0
3. Delete movie ID 9001

Verify that 9001 is gone and 9002 exists.

Task 2: Bulk Operations (continued)

Hint: Use `POST /_bulk` with action lines:

```
POST /_bulk
{"index": {"_index": "movies", "_id": "9001"}}
{"title": "Bulk Test Movie", ...}
{"index": {"_index": "movies", "_id": "9002"}}
{"title": "Bulk Test Movie 2", ...}
{"delete": {"_index": "movies", "_id": "9001"}}
```

Task 3: Script Update (Intermediate)

Write a script update that adds 0.5 to the `vote_average` of movie ID 1 (The Shawshank Redemption), but only if the current value is below 9.0.

Then verify the new value.

> **Hint:** Use `POST /movies/_update/1` with a `script` that uses a conditional: `if (ctx._source.vote_average < 9.0) { ctx._source.vote_average += params.boost }`. Pass `boost` as a param.

Task 4: Reindex with Filter + Alias (Intermediate)

1. Create a new index called `top-movies` by reindexing only movies with `vote_average >= 8.0` from the `movies` index
2. Create an alias called `best-films` pointing to `top-movies`
3. Verify you can search via the alias: `GET /best-films/_count`

> **Hint:** Use `POST /_reindex` with a `source.query` containing a `range` filter. Then use `POST /_aliases` with an `add` action. The reindexed documents will keep the same mappings.

Task 5: Index Template (Intermediate)

Create an index template that:

1. Matches any index starting with `movies-`
2. Sets 1 shard, 0 replicas
3. Defines explicit mappings for: title (text/english), genres (keyword), vote_average (float), release_date (date)

Test it by creating `movies-test` and indexing a document -- verify the mapping was applied automatically.

> **Hint:** Use `PUT /_index_template/movies-template` with `index_patterns: ["movies-*"]`. After creating the template, `PUT /movies-test/_doc/1 { "title": "Test" }` then `GET /movies-test/_mapping` to verify.

Part B: Analyzers & Mappings (5 tasks, ~25 min)

Task 6: Compare Analyzers (Basic)

Use the `_analyze` API to analyze the text `"The runners were running quickly in 2024!"` with each of these analyzers:

1. `standard`
2. `english`
3. `whitespace`
4. `keyword`

Questions:

1. Which analyzer produces the fewest tokens? Why?
2. Which analyzer preserves the original case?
3. How does the english analyzer handle "runners" vs "running"?

Task 6: Compare Analyzers (continued)

> **Hint:** Use `GET /_analyze` with `"analyzer": "standard"` (and each other analyzer) on the text.
> The english analyzer applies stemming so "runners" and "running" both become "run".
> The keyword analyzer keeps the entire string as one token.

Task 7: Explicit Mapping (Basic)

Create an index called `movies-explicit` with this exact mapping:

Field	Type	Analyzer
title	text	english
title.keyword	keyword	-
overview	text	english
genres	keyword	-
vote_average	float	-
release_date	date	-

Index a test movie and verify the mapping with `GET /movies-explicit/_mapping`.

Task 7: Explicit Mapping (continued)

> **Hint:** Use multi-fields for `title`: the main field is `text` with `english` analyzer, and `title.keyword` is a sub-field of type `keyword`. Use `"fields": { "keyword": { "type": "keyword" } }` inside the title property.

Task 8: Custom Analyzer (Intermediate)

Create an index with a custom analyzer called `movie_analyzer` that:

1. Strips HTML tags (char_filter: `html_strip`)
2. Uses the `standard` tokenizer
3. Applies: `lowercase`, `english_stop` (stop words), and `english_stemmer` (stemming)

Test it with: `"<p>The Amazing Spider-Man was RUNNING through NYC!</p>"`

> **Hint:** Define custom token filters: `"english_stop": {"type": "stop", "stopwords": "_english_"}` and `"english_stemmer": {"type": "stemmer", "language": "english"}`. Reference them in your analyzer's filter array.

Task 9: Autocomplete with Edge N-gram (Intermediate)

Build an autocomplete index for movie titles:

1. Create a custom analyzer with `edge_ngram` filter (min: 2, max: 10)
2. Map `title` field with this analyzer for **indexing** and `standard` for **searching**

Task 9: Autocomplete (continued)

3. Index 5 movies (Inception, Interstellar, Indiana Jones, Iron Man, Into the Wild)
4. Search for `"int"` -- it should match Interstellar and Into the Wild

> **Hint:** Set `"analyzer": "autocomplete_analyzer"` and `"search_analyzer": "standard"` on the title field. The `search_analyzer` prevents the search query from being edge-n-grammed (which would produce too many matches).

Task 10: Synonym Analyzer (Bonus)

Create an analyzer with synonyms for movie genres:

- `"film,movie,picture,flick"`
- `"scary,horror,frightening,spooky"`
- `"funny,comedy,humorous,hilarious"`

Create an index using this analyzer for an `overview` field. Index a doc with overview "A scary flick about ghosts" and search for "horror movie" -- does it match?

> **Hint:** Define a `synonym` token filter with `"synonyms"` array. Place the synonym filter AFTER lowercase in the filter chain. The bidirectional synonym mapping means "horror" and "scary" are interchangeable.

Part C: Aggregations (6 tasks, ~25 min)

Use the `movies` index (5000 docs) for tasks 11-15, and `kibana_sample_data_ecommerce` for task 16.

Task 11: Stats Aggregation (Basic)

Get comprehensive statistics for `vote_average` across all movies:

- Count, min, max, average, sum
- Also get the approximate number of **unique genres** (cardinality)

> **Hint:** Use `stats` aggregation for vote_average and `cardinality` aggregation for genres. Set `size: 0` to skip hits.

Task 12: Genre Terms Aggregation (Basic)

Get the top 15 genres by document count. For each genre, also show the average vote_average.

Which genre has the highest average rating?

> **Hint:** Use a `terms` aggregation on `genres` with `size: 15`, and nest an `avg` aggregation inside it.

Task 13: Date Histogram + Nested Avg (Intermediate)

Create a date_histogram aggregation that shows:

- Number of movies per **year**
- Average rating per year

Only include years with at least 10 movies (`min_doc_count: 10`).

> **Hint:** Use `date_histogram` with `calendar_interval: "year"` on `release_date` , nest an `avg` agg on `vote_average` , and add `"min_doc_count": 10` .

Task 14: Range Buckets (Intermediate)

Create rating buckets for movies:

- "Poor" (0-5.0)
- "Average" (5.0-6.5)
- "Good" (6.5-8.0)
- "Excellent" (8.0-10.0)

For each bucket, show the count and the top 3 genres (nested terms aggregation).

> **Hint:** Use a `range` aggregation on `vote_average` with named ranges (use the `key` parameter). Nest a `terms` agg on `genres` with `size: 3` inside each range bucket.

Task 15: Pipeline Aggregation (Intermediate)

Find the **year** with the highest average movie rating:

1. Create a `date_histogram` by year
2. Nest an `avg` aggregation for vote_average
3. Use `max_bucket` pipeline aggregation to find the year with the highest avg

> **Hint:** The pipeline agg goes at the same level as the date_histogram (sibling), not nested inside it. Use `"buckets_path": "by_year>avg_rating"` to reference the nested avg inside the date histogram.

Task 16: eCommerce Revenue Analysis (Bonus)

Using the `kibana_sample_data_ecommerce` index:

1. Calculate total revenue (`taxful_total_price`) by `day_of_week_i` (day of week)
2. For each day, also show: order count, average order value, and top 3 product categories
3. Which day generates the most revenue?

> **Hint:** Use `terms` aggregation on `day_of_week_i`, nest `sum`, `avg` aggs on `taxful_total_price`, and a `terms` agg on `category.keyword`. The `day_of_week_i` field is an integer (0=Monday).

Part D: Nested & Join Types (3 tasks, ~15 min)

Task 17: Object vs Nested (Basic)

Create an index called `movies-nested` with a `cast` field as **nested** type:

```
{
  "cast": [
    { "name": "Tim Robbins", "role": "Andy Dufresne", "billing": 1 },
    { "name": "Morgan Freeman", "role": "Red", "billing": 2 }
  ]
}
```

Index 2-3 movies with cast information.

Task 17: Object vs Nested (continued)

Then write a nested query to find movies where Morgan Freeman had billing position 1. It should return **no results** (he was billing 2). Verify the query correctly excludes false positives.

> **Hint:** Map `cast` as `type: "nested"` with properties: name (keyword), role (text), billing (integer). Use a `nested` query with `path: "comments"` and a `bool` combining `term` on `cast.name` and `term` on `cast.billing`.

Task 18: Nested Aggregation (Intermediate)

Using the `movies-nested` index from Task 17:

Run a nested aggregation to find which actors appear most frequently across movies. Use `nested` agg then `terms` on `cast.name`.

Hint:

```
"aggs": {
  "cast_nested": {
    "nested": { "path": "cast" },
    "aggs": {
      "top_actors": {
        "terms": { "field": "cast.name", "size": 10 }
      }
    }
  }
}
```

Task 19: Parent-Child Blog Model (Bonus)

Create a parent-child model for a blog:

1. Create index `blog` with a join field: `post` (parent) -> `comment` (child)
2. Index 2 posts and 3-4 comments (remember routing!)
3. Write a `has_child` query: find posts that have a comment by "Alice"
4. Write a `has_parent` query: find comments on posts containing "Elasticsearch"

> **Hint:** The join field maps as `"type": "join", "relations": {"post": "comment"}`. Children need `?routing=<parent_id>` and `"join_field": {"name": "comment", "parent": "<parent_id>"}`. Use `has_child` with `type: "comment"` and `has_parent` with `parent_type: "post"`.

Cleanup

```
DELETE /top-movies
DELETE /movies-explicit
DELETE /movies-test
DELETE /movies-synonyms
DELETE /autocomplete-index
DELETE /movies-nested
DELETE /blog
DELETE /blog-pc
```

> Keep the `movies` index -- you'll need it for reference. Start `elk-ml` tonight for Day 4!