

# ElasticSearch Hands-on Lab

Great! Since you've already covered the theoretical content, the 4-hour **ElasticSearch hands-on lab tutorial** should focus purely on practical exercises, realistic use cases, and skill-building workflows using **Elastic Cloud (online) with Gmail login**.

## ✓ Session Objective

Enable students to confidently perform CRUD operations, analyze search queries, work with mappings, use filters and aggregations, and visualize results on Elastic Cloud.

## 🕒 4-Hour Practical Lab Plan (Divided into 4 Parts)

Each part is about **1 hour**, containing 2–4 exercises, a short guided walkthrough, and a challenge.

### ♦ Part 1: Getting Started with ElasticSearch Cloud & Indexing Documents

**Goal:** Practice document indexing, CRUD operations, and data formats.

**Segments:**

#### 1. Login & Cloud Setup (5–10 mins)

- Log into [Elastic Cloud](#) using Gmail.
- Create a new deployment (or use existing one).
- Access Kibana → Dev Tools → Console.

#### 2. Basic Indexing & Retrieval

- Create an index `students`
- Add documents using `POST /students/_doc`
- Retrieve by ID with `GET /students/_doc/{id}`
- Use `GET /students/_search` with `match_all`

#### 3. Update & Delete Operations

- Update fields via `_update`
- Delete by ID

#### 4. Challenge:

- Create your own `books` index with 3 custom documents (title, author, rating).
- Retrieve them using `GET` and `match` queries.

### ♦ Part 2: Search Queries & Filters

**Goal:** Use basic and full-text search queries with filtering and Boolean logic.

**Segments:**

1. **Match vs Term Queries**

- Compare `match`, `term`, and `match_phrase`
- Use `.keyword` fields where needed

2. **Boolean Logic**

- Combine conditions using `must`, `should`, `must_not`
- Search `books` where author is X and rating > 4

3. **Filters vs Queries**

- Use `bool` → `filter` for performance
- Add range filter on numeric field (e.g., price)

4. **Challenge:**

- Create a `movies` index
- Search for movies with genre = comedy and rating ≥ 7
- Use both filter and query context

---

♦ **Part 3: Mappings, Data Types & Analyzers**

**Goal:** Control data indexing behavior and understand field types.

**Segments:**

1. **Create Custom Mappings**

- Define an index `products` with `text`, `keyword`, `date`, and `float` fields
- Example: `name`, `category`, `release_date`, `price`

2. **Dynamic vs Explicit Mapping**

- Show behavior with dynamic mapping
- Update a field mapping with `PUT` template

3. **Analyzers & Text Fields**

- Use `_analyze` API to check tokenization of text
- Compare `text` vs `keyword` fields

4. **Challenge:**

- Recreate `students` index with mapping: `name (text)`, `id (keyword)`, `grade (float)`
- Insert data and query by keyword vs text field

---

♦ **Part 4: Aggregations & Real-Time Use Case**

**Goal:** Use aggregations and simulate a mini-dashboard use case.

## Segments:

### 1. Basic Aggregations

- Use **terms**, **avg**, **min**, **max**, **range**, and **date\_histogram**
- Example: Avg price per product category

### 2. Nested Aggregations

- Avg salary by job title and department
- Count students by grade range

### 3. Real-Time Challenge (30–40 mins)

- Dataset: Create a **sales** index
- Fields: **product**, **region**, **date**, **amount**
- Tasks:
  - Aggregate total sales per region
  - Time series: sales per month using **date\_histogram**
  - Top 3 products by revenue
  - Filter sales for a specific region in a time window

---

## Optional Tools & Resources

- **Elastic Playground** (online trial): <https://elastic.co/playground>
- **REST API JSON tools**: [Postman](#) or Kibana Console
- **Small CSV-to-JSON converters** for data generation: [convertcsv.com](https://convertcsv.com)
- **Sample datasets**: <https://github.com/elastic/examples>

---

## Optional Extras (if time allows or as homework)

- Import a CSV file using **Elastic Ingest Pipelines** or **File Upload** in Kibana.
- Visualize aggregation results using **Kibana Lens** (basic drag-and-drop).
- Use **Watchers** (if license supports) to alert on specific conditions (e.g., sales drop).

---

Would you like me to start generating **Segment 1** with step-by-step examples and commands?

Perfect. Let's begin with the **first segment** of the 4-hour hands-on Elasticsearch lab:

## ◆ Part 1 – Segment 1: Logging into Elastic Cloud and Setting Up Dev Tools

 **Estimated Time: 5–10 minutes**

This segment ensures all students have access to the Elastic environment and are ready to run commands.

### Objective

- Access Elasticsearch & Kibana using Gmail
- Navigate to Dev Tools → Console to interact with the cluster

### Step-by-Step Walkthrough

#### Step 1: Open Elastic Cloud

1. Go to <https://cloud.elastic.co>
2. Click **“Log in with Google”**
3. Select your Gmail account
4. If this is the first time, you may be prompted to:
  - Accept permissions
  - Set up a default organization


#### Step 2: Create or Use a Deployment

If no deployment exists:

1. Click **“Create Deployment”**
2. Choose **“Start with Elasticsearch and Kibana”**
3. Use default settings (or choose a name like **es-lab**)
4. Wait 2–5 minutes for it to be ready

Once it's ready:

- Click **“Open Kibana”** to access the Kibana dashboard

 If you already have a deployment:

- Click on your deployment name
- Then click **“Open Kibana”**

## Step 3: Open Dev Tools

Once inside Kibana:

1. Look at the left menu (🕒 navigation panel)
2. Click **"Dev Tools"** (you may need to expand the menu if collapsed)
3. You should see a **Console** window with two panes:
  - Left: request editor
  - Right: response output

You are now ready to execute Elasticsearch REST queries!

### Hands-On Checkpoint

To make sure everything is working, run the following query in Dev Tools Console:

```
json
```

```
GET /
```

This returns basic info about your cluster. You should see:

- Cluster name
- Cluster UUID
- Version number (e.g., **"number": "8.12.0"**)

✅ If you see a response like that, everything is correctly configured.

### Troubleshooting Tips

Issue	Solution
"Create Deployment" disabled	Use a free trial or check your account limit
Kibana not loading	Try incognito or different browser
Dev Tools missing	Make sure you're in Kibana, not Elastic Cloud home dashboard

✅ **You are now ready to start creating indexes and documents.**

Would you like me to continue with **Segment 2: Basic Indexing & Retrieval?**

Great! Let's move on to:

## ◆ Part 1 – Segment 2: Basic Indexing & Retrieval

 **Estimated Time: 20–25 minutes**

This segment teaches how to create an index, insert documents, and retrieve data using Elasticsearch JSON queries.

### Objective

- Create a custom index (**students**)
- Insert documents with structured data
- Retrieve documents by ID and with a simple search

### Step-by-Step Lab

#### ◆ Step 1: Create an Index and Insert a Document


In the Dev Tools console, paste and run:

```
json

POST /students/_doc/
{
  "name": "Amina Saleh",
  "age": 21,
  "grade": 89.5,
  "department": "Computer Science"
}
```

#### ◆ Explanation:

- **POST /students/\_doc/**: creates a new document under index **students**
- **\_doc** is the default document type (still required for backward compatibility)
- No ID is given → Elastic will auto-generate one

 The response will include a **\_id** field (e.g., "**id**": "**t0J6hIABBA...**")

#### ◆ Step 2: Retrieve Document by ID

Use the returned ID from the previous response to retrieve the document:

```
json

GET /students/_doc/t0J6hIABBA7o8_YMz4zv
```

(Replace the ID with your actual one)

✓ Output will contain `_source` with your original document fields.

### ♦ Step 3: Add More Documents

Run the following to add two more students:

```
json

POST /students/_doc/
{
  "name": "Omar Youssef",
  "age": 23,
  "grade": 76.3,
  "department": "Information Systems"
}

POST /students/_doc/
{
  "name": "Layla Hasan",
  "age": 22,
  "grade": 91.2,
  "department": "Computer Science"
}
```

You should now have 3 documents in the **students** index.

### ♦ Step 4: Retrieve All Documents in the Index

Use `match_all` to get everything:

```
json

GET /students/_search
{
  "query": {
    "match_all": {}
  }
}
```

♦ You'll get all documents in the `hits.hits` array.

✓ Tip: Use `"size": 10` to limit how many documents are shown:

```
json

GET /students/_search
{
  "size": 10,
  "query": {
    "match_all": {}
  }
}
```

### ♦ Step 5: Retrieve Only Specific Fields

You can filter which fields are returned:

```
json

GET /students/_search
{
  "_source": ["name", "grade"],
  "query": {
    "match_all": {}
  }
}
```

This hides **age**, **department**, etc.

---

## Hands-On Practice

Try doing the following:

- Add a fourth student of your choice
  - Retrieve only their **name** and **department**
  - Search using **match\_all**
- 

Would you like to proceed to **Segment 3: Update and Delete Operations**?



Excellent. Let's proceed to:

## ◆ Part 1 – Segment 3: Update and Delete Operations

 **Estimated Time: 15–20 minutes**

Learn how to modify existing documents and remove them from your index.

### Objective

- Update fields within an existing document
- Delete a document by ID
- Understand partial vs full updates

### Step-by-Step Lab

#### ◆ Step 1: Update a Document (Partial Update)


Let's say you want to update **Layla Hasan's grade** from **91.2** to **94.0**.

You need the document's `_id`. If you don't have it, search for it first:

```
json
GET /students/_search
{
  "query": {
    "match": {
      "name": "Layla Hasan"
    }
  }
}
```

Once you get her document ID (e.g., `"ZpNshYABBA8pHdpM--hg"`), update it:

```
json
POST /students/_update/ZpNshYABBA8pHdpM--hg
{
  "doc": {
    "grade": 94.0
  }
}
```

 This only modifies the **grade** field. All other fields remain unchanged.

#### ◆ Step 2: Add a New Field to a Document

You can also **add** a new field using `_update`.

For example, add `"status": "honors"` to Layla:

```
json

POST /students/_update/ZpNshYABBA8pHdpM--hg
{
  "doc": {
    "status": "honors"
  }
}
```

Now check the document again:

```
json

GET /students/_doc/ZpNshYABBA8pHdpM--hg
```

You should now see the new **status** field.

### ◆ Step 3: Full Overwrite of a Document (PUT)

⚠ This will **replace the entire document**, not just update fields.

```
json

PUT /students/_doc/ZpNshYABBA8pHdpM--hg
{
  "name": "Layla Hasan",
  "age": 22,
  "grade": 94.0,
  "department": "Computer Science",
  "status": "honors"
}
```

Be careful: if you forget any field, it will be lost.

### ◆ Step 4: Delete a Document by ID

To delete Omar's document, first find his ID:

```
json

GET /students/_search
{
  "query": {
    "match": {
      "name": "Omar Youssef"
    }
  }
}
```

Then delete:

```
json

DELETE /students/_doc/{your_id_here}
```

For example:

```
json
```

```
DELETE /students/_doc/XvZJhYABBA6QYMKAZrz_
```

You should receive a confirmation response:

```
json
```

```
{  
  "result": "deleted"  
}
```

## Hands-On Practice

Try the following:

- Add a new field called **passed** with a boolean value to Amina's record
- Overwrite one student's document with fewer fields (see what happens)
- Delete one document and verify it's gone with **GET**

✓ You now understand document lifecycle management (create, update, delete).

Would you like to continue with **Segment 4: Challenge – Create Your Own books Index and Query It?**

Perfect — this next segment will wrap up Part 1 with a creative mini-project.

## ◆ Part 1 – Segment 4: Challenge – Create Your Own books Index and Query It

 **Estimated Time: 20–25 minutes**

You'll now apply CRUD concepts to your own dataset. This reinforces learning and prepares you for future segments on search and filters.

### Objective

- Create a custom index **books**
- Add multiple documents
- Retrieve and search data using **GET** and **match** queries

### Scenario

You are building a book catalog with details like:

- Title
- Author
- Genre
- Rating
- Year of publication

### Step-by-Step Challenge Tasks

#### ◆ Step 1: Create and Insert 3–5 Books

Insert at least 3 documents. Here are some examples:

```
json

POST /books/_doc/
{
  "title": "Clean Code",
  "author": "Robert C. Martin",
  "genre": "Programming",
  "rating": 4.8,
  "year": 2008
}

POST /books/_doc/
{
  "title": "1984",
  "author": "George Orwell",
  "genre": "Dystopian",
```

```

    "rating": 4.6,
    "year": 1949
  }

POST /books/_doc/
{
  "title": "The Pragmatic Programmer",
  "author": "Andy Hunt",
  "genre": "Programming",
  "rating": 4.5,
  "year": 1999
}

```

✓ You can add more books of your choice with different authors and genres.

## ◆ Step 2: Retrieve All Books

Use a match-all query:

```

json

GET /books/_search
{
  "query": {
    "match_all": {}
  }
}

```

## ◆ Step 3: Search by Author (Full-Text Query)

Example: Find books by "Robert C. Martin"

```

json

GET /books/_search
{
  "query": {
    "match": {
      "author": "Robert C. Martin"
    }
  }
}

```

Note: `match` uses full-text search. It is case-insensitive and allows tokenization.

## ◆ Step 4: Search by Genre Using `term` (Exact Match)

Try this:

```

json

GET /books/_search
{
  "query": {
    "term": {
      "genre.keyword": {
        "value": "Programming"
      }
    }
  }
}

```

```
}
}
```

Why `.keyword`? Because `term` requires exact matching. By default, `genre` is a `text` field analyzed into tokens — `genre.keyword` is its exact-match subfield.

### ◆ Step 5: Search by Rating Threshold (Preview of range filter)

Find books with rating > 4.6:

```
json

GET /books/_search
{
  "query": {
    "range": {
      "rating": {
        "gt": 4.6
      }
    }
  }
}
```

### ◆ Step 6: Retrieve Books Published After 2000

```
json

GET /books/_search
{
  "query": {
    "range": {
      "year": {
        "gt": 2000
      }
    }
  }
}
```

## Optional Hands-On Enhancements

- Add `pages` or `ISBN` field to one book using `_update`
- Try a `match_phrase` search on "The Pragmatic Programmer"
- Delete one book and reinsert it with updated info

✓ After completing this challenge, you now know how to:

- Structure an index
- Populate it with meaningful documents
- Query using both full-text and exact matches

Would you like to proceed to **Part 2 – Segment 1: Match vs Term Queries?**

Excellent — you're now ready to begin **Part 2**, where we dive into full-text search and exact-match logic.

## ◆ Part 2 – Segment 1: Match vs Term Queries

 **Estimated Time: 20–25 minutes**

This segment introduces the difference between **full-text (match)** and **exact-match (term)** queries — a crucial skill in Elasticsearch.

### Objective

- Understand how **match** works with text fields
- Compare it with **term** queries on keyword fields
- Use both in real examples with the **books** index

### Theory Brief (5 mins)

Query Type	Field Type	Case Sensitive?	Tokenizes Input?	Use Case
<b>match</b>	<b>text</b>	✗ No	✓ Yes	Full-text search (title, content)
<b>term</b>	<b>keyword</b>	✓ Yes	✗ No	Filters, IDs, categories, tags

### ✓ Step-by-Step Lab with Examples

You should have your **books** index populated from the previous segment. We'll now experiment on it.

#### ◆ Step 1: Try a **match** Query on Title

```
json
GET /books/_search
{
  "query": {
    "match": {
      "title": "pragmatic"
    }
  }
}
```

✓ Elastic will:

- Tokenize **title** into terms like `["the", "pragmatic", "programmer"]`
- Tokenize **"pragmatic"** → `["pragmatic"]`
- Return documents where there's a match in **title**

#### ◆ Step 2: Try a **term** Query on Title (Fail Case)

```
json

GET /books/_search
{
  "query": {
    "term": {
      "title": {
        "value": "pragmatic"
      }
    }
  }
}
```

💥 No results! Why?

- **title** is a **text** field, and **term** does not tokenize input
- It's looking for an exact match: "**pragmatic**" as a whole field — which doesn't exist

### ♦ Step 3: Use term on .keyword Subfield

```
json

GET /books/_search
{
  "query": {
    "term": {
      "genre.keyword": {
        "value": "Programming"
      }
    }
  }
}
```

✅ This works because:

- **genre.keyword** is a **keyword** field
- It holds the raw, untokenized string "**Programming**"

### ♦ Step 4: Match with Case-Insensitive Query

```
json

GET /books/_search
{
  "query": {
    "match": {
      "author": "george orwell"
    }
  }
}
```

Even if the document says "**George Orwell**", this works because:

- **author** is a **text** field
- **match** is **case-insensitive** and supports **analyzers**



## ◆ Step 5: Match Phrase (Optional Precision)

```
json
GET /books/_search
{
  "query": {
    "match_phrase": {
      "title": "the pragmatic programmer"
    }
  }
}
```

This ensures the words appear **in order** and **together** in the field.

## Practice Exercise

Try the following queries:

1. **match** search on a title or author name (any case)
2. **term** query that fails due to being run on a **text** field
3. **term** query on **genre.keyword** that succeeds
4. Add a new field **language**: "**English**" to one document and filter by exact match

✅ You've now understood how different field types affect search behavior.

Would you like to continue to **Segment 2: Boolean Logic with must, should, must\_not?**

Great — let's dive into:

## ◆ Part 2 – Segment 2: Boolean Logic with `must`, `should`, `must_not`

 **Estimated Time: 25–30 minutes**

This segment introduces compound queries using the `bool` query — Elastic's way of combining multiple search conditions with logic.

### Objective

- Combine multiple criteria in a single query
- Use `must`, `should`, and `must_not` blocks
- Understand query scoring when using optional (`should`) clauses

### Step-by-Step Lab

We'll continue using your `books` index from earlier.

#### ◆ Step 1: Use `must` to Combine Conditions (AND)

Let's find:

All books authored by "Andy Hunt" AND in the "Programming" genre

```
json

GET /books/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "author": "Andy Hunt" } },
        { "term": { "genre.keyword": "Programming" } }
      ]
    }
  }
}
```

 Explanation:

- `must`: All conditions **must be true**
- Mixed `match` and `term` are allowed

#### ◆ Step 2: Use `should` (Optional Match – OR)

Let's find:

Books authored by **either** "Robert C. Martin" **or** "Andy Hunt"

```
json

GET /books/_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "author": "Robert C. Martin" } },
        { "match": { "author": "Andy Hunt" } }
      ]
    }
  }
}
```

✓ This returns documents that match **either** condition.

💡 Note: By default, at least one **should** must match, unless you're combining with **must**.

### ♦ Step 3: Require Both **must** and **should** Together

Let's find:

Books in the **Programming** genre (must), and ideally authored by Robert C. Martin (should)

```
json

GET /books/_search
{
  "query": {
    "bool": {
      "must": [
        { "term": { "genre.keyword": "Programming" } }
      ],
      "should": [
        { "match": { "author": "Robert C. Martin" } }
      ]
    }
  }
}
```

This prioritizes documents that match the **should** clause but still returns those that match only the **must**.

### ♦ Step 4: Use **must\_not** (NOT Operator)

Let's find:

Books NOT in the "Dystopian" genre

```
json

GET /books/_search
{
  "query": {
    "bool": {
      "must_not": [
        { "term": { "genre.keyword": "Dystopian" } }
      ]
    }
  }
}
```

```
}
}
```

✓ The **must\_not** clause filters out matching documents entirely.

## ◆ Step 5: Combine All Three

Let's find:

Books in **Programming** genre, **NOT** authored by "Andy Hunt", and **preferably** published after 2010

```
json

GET /books/_search
{
  "query": {
    "bool": {
      "must": [
        { "term": { "genre.keyword": "Programming" } }
      ],
      "must_not": [
        { "match": { "author": "Andy Hunt" } }
      ],
      "should": [
        {
          "range": {
            "year": {
              "gt": 2010
            }
          }
        }
      ]
    }
  }
}
```

## Practice Exercise

Try the following:

1. Search for books **by Andy Hunt or George Orwell**
2. Find books **in genre Programming** and **rating > 4.6**
3. Filter books that are **not authored** by Robert C. Martin
4. Combine **must** and **should** to influence score priority

✓ You've now learned how to create powerful multi-condition queries using Boolean logic.

Would you like to continue to **Segment 3: Filters vs Queries – Performance and Use Cases?**

Great — let's move to:

## ◆ Part 2 – Segment 3: Filters vs Queries – Performance and Use Cases

 **Estimated Time: 20–25 minutes**

This segment explains the critical difference between **query** context and **filter** context — essential for both relevance and performance tuning in Elasticsearch.

### Objective

- Understand what filters do and how they differ from queries
- Use **bool** → **filter** for constant scoring
- Learn when to prefer **filter** for performance

### Concept: What's the Difference?

Feature	query	filter
Relevance Score	✅ Yes, used to compute score	❌ No score; binary match
Use Case	Full-text search, ranked results	Filtering, faceting, aggregations
Performance	Slower	Faster (results can be cached)
Example	Search "clean code" in title	Filter books from 2000 to 2010

 Filters are **not scored** — they just decide inclusion/exclusion.

### ✅ Step-by-Step Lab

Let's compare the two in practice using your **books** index.

#### ◆ Step 1: Use a Range in a Query (Scored)

```
json
GET /books/_search
{
  "query": {
    "range": {
      "year": {
        "gte": 2000,
        "lte": 2010
      }
    }
  }
}
```

- Results are scored based on how well they match the range

- "\_score" values will be present in the response

## ◆ Step 2: Move the Same Condition into a Filter

```
json
GET /books/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "range": {
            "year": {
              "gte": 2000,
              "lte": 2010
            }
          }
        }
      ]
    }
  }
}
```

✓ What changes?

- **No \_score** is computed → better performance
- Ideal for dashboards, drilldowns, and analytics

## ◆ Step 3: Combine Query + Filter

Let's search for:

Books matching **"programming"** in the title (scored)

And filtered by publication year between 1990 and 2020 (not scored)

```
json
GET /books/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "title": "programming"
          }
        }
      ],
      "filter": [
        {
          "range": {
            "year": {
              "gte": 1990,
              "lte": 2020
            }
          }
        }
      ]
    }
  }
}
```

```
}
}
```

### ✓ Explanation:

- **must** → scoring, full-text matching
- **filter** → narrow the data without impacting score

## ◆ Step 4: Use term in Filter Block

Let's find:

All **Programming** books **after 2000**

```
json

GET /books/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "term": {
            "genre.keyword": "Programming"
          }
        },
        {
          "range": {
            "year": {
              "gt": 2000
            }
          }
        }
      ]
    }
  }
}
```

This is a pure filter-based query — no scoring involved.

## Practice Exercise

1. Use **bool** → **filter** to get all books rated above 4.5
2. Combine **match** on **title** with a **filter** on **genre.keyword**
3. Compare performance of a query with and without filters (check **\_score**)

✓ You now understand the difference between filtering and querying — a powerful optimization technique.

Would you like to proceed to **Segment 4: Challenge – Build a movies Index with Boolean + Filter Queries**?

Awesome — let's wrap up **Part 2** with a practical and creative challenge.

## ◆ Part 2 – Segment 4: Challenge – Build a `movies` Index Using Boolean + Filter Queries

 **Estimated Time: 25–30 minutes**

This is an applied challenge where you'll use everything from Part 2: `match`, `term`, `bool`, `must`, `should`, `must_not`, and `filter`.

### Challenge Objectives

- Create a new `movies` index
- Insert documents with relevant fields
- Query based on genre, rating, and release year
- Combine filter + match + logic conditions

### Suggested Document Structure

Each movie should include at least:

- `title` (text)
- `genre` (text or keyword)
- `rating` (float)
- `year` (integer)
- `director` (text)

### Step-by-Step Instructions

#### ◆ Step 1: Create the `movies` Index and Add 4–5 Documents

Examples:

```
json

POST /movies/_doc/
{
  "title": "The Matrix",
  "genre": "Science Fiction",
  "rating": 8.7,
  "year": 1999,
  "director": "Lana Wachowski"
}

POST /movies/_doc/
{
  "title": "The Godfather",
```



```

"genre": "Crime",
"rating": 9.2,
"year": 1972,
"director": "Francis Ford Coppola"
}

POST /movies/_doc/
{
  "title": "Inception",
  "genre": "Science Fiction",
  "rating": 8.8,
  "year": 2010,
  "director": "Christopher Nolan"
}

POST /movies/_doc/
{
  "title": "The Dark Knight",
  "genre": "Action",
  "rating": 9.0,
  "year": 2008,
  "director": "Christopher Nolan"
}

POST /movies/_doc/
{
  "title": "Pulp Fiction",
  "genre": "Crime",
  "rating": 8.9,
  "year": 1994,
  "director": "Quentin Tarantino"
}

```

## ◆ Step 2: Run a `match_all` to Validate Your Documents

```


json

GET /movies/_search
{
  "query": {
    "match_all": {}
  }
}

```

✓ Confirm your documents are properly stored.

## ◆ Step 3: Use `bool` + `must` + `filter`

 Find movies directed by Nolan in the **Action** genre and released after 2005

```

json

GET /movies/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "director": "Nolan" } }
      ],
      "filter": [
        { "term": { "genre.keyword": "Action" } },
        { "range": { "year": { "gt": 2005 } } }
      ]
    }
  }
}

```

```
}
}
```

#### ◆ Step 4: Use `must_not` to Exclude Genres

Find all movies **not** in the **Crime** genre

```
json

GET /movies/_search
{
  "query": {
    "bool": {
      "must_not": [
        { "term": { "genre.keyword": "Crime" } }
      ]
    }
  }
}
```

#### ◆ Step 5: Use `should` to Prioritize

Find movies **after 1990**, with preference for those rated above 9.0

```
json

GET /movies/_search
{
  "query": {
    "bool": {
      "filter": [
        { "range": { "year": { "gte": 1990 } } }
      ],
      "should": [
        { "range": { "rating": { "gt": 9.0 } } }
      ]
    }
  }
}
```

✓ Results will still include movies from after 1990, but those above 9.0 rating get higher `_score`.

### Final Tasks

1. Create a movie with multiple genres like `"genre": "Action, Drama"` → try a `match_phrase`
2. Add a `language` field and filter for English movies
3. Query all movies directed by **Nolan OR Tarantino** using `should`

✓ That concludes Part 2 with a hands-on integration challenge.

Would you like to proceed to **Part 3 – Segment 1: Custom Mappings and Field Types**?

Great! Let's begin **Part 3**, where we go deeper into how Elasticsearch handles data structure with mappings and types.

## ◆ Part 3 – Segment 1: Custom Mappings and Field Types

 **Estimated Time: 25–30 minutes**

This segment teaches you how to control index structure explicitly using **custom mappings**. You'll learn how to define fields like **text**, **keyword**, **date**, **float**, and **integer**.

### Objective

- Define custom mappings before inserting data
- Understand data types in Elasticsearch
- Prevent issues caused by dynamic typing
- Practice creating and using a mapped **products** index

### What Are Mappings?

Mappings define:

- Field names and types (**text**, **keyword**, **date**, **float**, etc.)
- Indexing behavior (analyzed vs not)
- Search and sort capabilities

ElasticSearch uses **dynamic mapping** by default — it guesses field types. You can override this by **manually defining mappings**.

### Step-by-Step Lab

#### ◆ Step 1: Create **products** Index with Explicit Mappings

Use **PUT** to define the schema before adding data:

```
json

PUT /products
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "category": {
        "type": "keyword"
      },
      "release_date": {
        "type": "date"
      },
      "price": {
        "type": "float"
      },
      "stock": {
```

```

    "type": "integer"
  }
}
}

```

✓ This schema says:

- **name:** searchable full-text
- **category:** filterable exact value
- **release\_date:** supports date queries
- **price:** decimal support
- **stock:** whole number only

## ◆ Step 2: Insert Some Products

json

POST /products/\_doc/

```

{
  "name": "Apple iPhone 15 Pro",
  "category": "Smartphones",
  "release_date": "2023-09-20",
  "price": 1199.99,
  "stock": 35
}

```

POST /products/\_doc/

```

{
  "name": "Samsung Galaxy S24",
  "category": "Smartphones",
  "release_date": "2024-01-10",
  "price": 1099.50,
  "stock": 48
}

```

POST /products/\_doc/

```

{
  "name": "Sony WH-1000XM5",
  "category": "Headphones",
  "release_date": "2022-05-25",
  "price": 349.99,
  "stock": 15
}

```

✓ All values will now follow the expected type.

## ◆ Step 3: Check Mappings After Insertion

You can inspect how your index is structured:

json

GET /products/\_mapping

✓ This will return all your custom fields and types.

#### ◆ Step 4: Search by Category with `term`

```
json

GET /products/_search
{
  "query": {
    "term": {
      "category": "Smartphones"
    }
  }
}
```

Works because `category` is a **keyword** — exact-match field.

#### ◆ Step 5: Match by Name with `text`

```
json

GET /products/_search
{
  "query": {
    "match": {
      "name": "iPhone"
    }
  }
}
```

Works because `name` is a **text** field with analysis.

#### ◆ Step 6: Try Sorting by Price

```
json

GET /products/_search
{
  "sort": [
    {
      "price": {
        "order": "desc"
      }
    }
  ],
  "query": {
    "match_all": {}
  }
}
```

✓ Only numeric and keyword/date fields can be sorted.

#### Practice Exercise

1. Add a new product with a missing field (e.g., no `stock`)
2. Try filtering all products released **after 2023-01-01**
3. Change the field type of `stock` to `long` (requires index deletion and recreation)

✓ You've now built a strongly-typed index — much safer and cleaner than relying on auto-mapping.

Would you like to proceed to **Segment 2: Dynamic vs Explicit Mapping and Mapping Updates**?

Excellent — now let's explore more nuanced behavior in:

## ◆ Part 3 – Segment 2: Dynamic vs Explicit Mapping and Mapping Updates

 **Estimated Time: 25–30 minutes**

This segment dives into how Elasticsearch treats unknown fields automatically (dynamic mapping), and how you can update mappings explicitly when needed.

### Objective

- Observe how Elasticsearch auto-generates mappings
- Know when and why to override dynamic mapping
- Learn safe strategies to update mapping

### What is Dynamic Mapping?

ElasticSearch will **automatically guess** field types if you don't define them:

```
json

POST /auto_example/_doc/
{
  "title": "Elastic in Action",
  "price": 19.99,
  "available": true,
  "release": "2024-06-01"
}
```

If no mapping exists, it will generate one like:

```
json

"title" → text
"price" → float
"available" → boolean
"release" → date
```

### Step-by-Step Lab

#### ◆ Step 1: Try a Dynamic Mapping Example

```
json

POST /autobooks/_doc/
{
  "title": "Search-Powered AI",
  "author": "Jane Smith",
  "rating": 4.6,
```



```

"published": "2021-11-22"
}

```

Now check the mapping Elastic created:

```

json

GET /autobooks/_mapping

```

✓ You'll see Elastic chose:

- **text** for **title** and **author**
- **float** for **rating**
- **date** for **published**

## ◆ Step 2: Disable Dynamic Mapping (Optional)

You can restrict Elastic from guessing field types:

```

json

PUT /strict_books
{
  "mappings": {
    "dynamic": "strict",
    "properties": {
      "title": { "type": "text" },
      "rating": { "type": "float" }
    }
  }
}

```

Now try inserting a document with an **undeclared field**:

```

json

POST /strict_books/_doc/
{
  "title": "Intro to Elastic",
  "rating": 4.9,
  "author": "Unknown"
}

```

☀ You'll get an error:

```

json

"reason": "mapping set to strict, dynamic introduction of [author] is not allowed"

```

✓ Useful for enforcing strict schemas in production.

## ◆ Step 3: Update Mapping to Add a New Field

**Elastic does not allow changing the type** of an existing field. But you **can** add new fields safely:

```
json

PUT /products/_mapping
{
  "properties": {
    "discount": { "type": "float" }
  }
}
```

Now update a document to include the new field:

```
json

POST /products/_update/{some_id_here}
{
  "doc": {
    "discount": 10.0
  }
}
```

✓ You added a new field without recreating the index.

## ◆ Step 4: Delete and Re-Crete an Index to Change a Field Type

Elastic **doesn't allow type changes** (e.g., from `float` to `keyword`). To do that:

1. Delete the index:

```
json

DELETE /autobooks
```

2. Re-create with corrected mapping:

```
json

PUT /autobooks
{
  "mappings": {
    "properties": {
      "title": { "type": "text" },
      "rating": { "type": "keyword" }
    }
  }
}
```

3. Reinsert data with new structure.

## Practice Exercise

1. Create an index with `"dynamic": "false"` — try inserting extra fields
2. Add a new optional field (**publisher**, **format**, or **discount**) to an existing index
3. Try to change a field's type — observe what fails
4. Rebuild an index with the corrected type

✓ This segment gives you critical control over your schema and helps prevent long-term technical debt.

Would you like to proceed to **Segment 3: Analyzers and \_analyze API**?

Excellent — now we enter one of Elasticsearch's most powerful and unique features:

## ◆ Part 3 – Segment 3: Analyzers and `_analyze` API

 **Estimated Time: 25–30 minutes**

This segment will teach how Elasticsearch processes text under the hood — including **tokenization**, **lowercasing**, **stemming**, and more — using **analyzers**.

### Objective

- Understand how text is broken down into searchable tokens
- Use the `_analyze` API to visualize tokenization
- Learn about standard vs custom analyzers
- Apply analyzers in index mappings

### Key Concepts

When you run a `match` query, Elasticsearch:

1. Passes the query value through an **analyzer**
2. Breaks the text into tokens (words, stems, lowercase, etc.)
3. Matches those tokens to the index

There are three types of analyzers:

Type	Purpose	Example Use
<b>standard</b>	Default for most <b>text</b> fields	General-purpose full text
<b>keyword</b>	Treats whole text as one token	Email, IDs, tags
<b>custom</b>	You define your own tokenizer & filters	Advanced scenarios

### Step-by-Step Lab

#### ◆ Step 1: Use `_analyze` with Standard Analyzer

Try this:

```
json
GET /_analyze
{
  "analyzer": "standard",
  "text": "The Quick Brown Fox jumps over the lazy Dog."
}
```

## ✓ Output:

```
json

{
  "tokens": [
    { "token": "the" },
    { "token": "quick" },
    { "token": "brown" },
    ...
  ]
}
```

It lowercases and splits words by spaces and punctuation.

## ◆ Step 2: Use keyword Analyzer

```
json

GET /_analyze
{
  "analyzer": "keyword",
  "text": "The Quick Brown Fox"
}
```

## ✓ Output:

```
json

{ "token": "The Quick Brown Fox" }
```

Treats the entire string as one term. Ideal for exact IDs or codes.

## ◆ Step 3: Visualize Your Field's Analyzer

Let's inspect how Elastic would index a product **name**:

```
json

GET /products/_analyze
{
  "field": "name",
  "text": "Apple iPhone Pro 15 Max"
}
```

This uses the field's configured analyzer (likely **standard** unless overridden).

## ◆ Step 4: Define a Custom Analyzer in Mapping

Let's build a new index with a custom analyzer that:

- Lowercases text
- Removes stop words

```
json

PUT /books_custom
{
  "settings": {
    "analysis": {
      "analyzer": {
        "custom_english": {
          "type": "standard",
          "stopwords": "_english_"
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "custom_english"
      }
    }
  }
}
```

Now try inserting and searching:

```
json

POST /books_custom/_doc/
{
  "title": "The Art of Computer Programming"
}
```

Then analyze it:

```
json

GET /books_custom/_analyze
{
  "field": "title",
  "text": "The Art of Computer Programming"
}
```

✔ You should see "the" and "of" filtered out as stopwords.

## ◆ Step 5: Edge Case – Why Match Sometimes Fails

Try this without a match:

```
json

GET /products/_search
{
  "query": {
    "term": {
      "name": "iPhone"
    }
  }
}
```

💥 It fails because:

- **term** expects **exact value**
- **name** is a **text** field, analyzed into tokens like "apple", "iphone"

✓ This is why understanding analyzers is **critical** to avoid confusing mismatches.

---

## Practice Exercise

1. Analyze "e-Commerce Application" with **standard** vs **whitespace** analyzers
  2. Create an index with a **keyword** field for SKU or **product\_code**
  3. Define a custom analyzer that removes punctuation
  4. Try a **match\_phrase** query using a custom-analyzed field
- 

✓ You've now unlocked the power of Elastic's full-text indexing engine.

Would you like to proceed to **Segment 4: Challenge – Design a Custom Analyzer for a Resume Index?**

Excellent — now let's apply what you've learned to a real-world use case.

## ◆ Part 3 – Segment 4: Challenge – Design a Custom Analyzer for a Resume Index

 **Estimated Time: 25–30 minutes**

This is a creative implementation task where you will define a full-text analyzer optimized for resume/CV search, allowing recruiters to find candidates by skills, titles, and experiences more accurately.

### Challenge Goals

- Create a new index **resumes** with custom analyzers
- Add documents with candidate resume data
- Use advanced full-text search with **match**, **match\_phrase**, and **filters**
- Design a tokenizer for resume fields like **skills**, **summary**, **experience**

### Suggested Fields per Resume Document

```
json

{
  "name": "Ahmed Farg",
  "title": "Data Scientist",
  "skills": "Python, Elasticsearch, NLP, Pandas, NumPy",
  "summary": "Experienced in machine learning, data analytics, and building end-to-end data pipelines.",
  "years_experience": 6,
  "location": "Cairo, Egypt"
}
```



## ✓ Step-by-Step Lab

### ◆ Step 1: Create resumes Index with Custom Analyzer

We will:

- Use a **custom analyzer** that splits by comma/space
- Lowercases all text
- Removes stopwords

```
json

PUT /resumes
{
  "settings": {
    "analysis": {
      "analyzer": {
        "resume_text_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": ["lowercase", "stop"]
        },
        "comma_whitespace": {
          "type": "pattern",
          "pattern": "[,\\s]+",
          "lowercase": true
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "name": { "type": "text", "analyzer": "resume_text_analyzer" },
      "title": { "type": "text", "analyzer": "resume_text_analyzer" },
      "skills": { "type": "text", "analyzer": "comma_whitespace" },
      "summary": { "type": "text", "analyzer": "resume_text_analyzer" },
      "years_experience": { "type": "integer" },
      "location": { "type": "keyword" }
    }
  }
}
```

### ◆ Step 2: Insert Sample Resume Documents

```
json

POST /resumes/_doc/
{
  "name": "Ahmed Farg",
  "title": "Data Scientist",
  "skills": "Python, Elasticsearch, NLP, Pandas, NumPy",
  "summary": "Experienced in machine learning, data analytics, and building end-to-end data pipelines.",
  "years_experience": 6,
  "location": "Cairo, Egypt"
}

POST /resumes/_doc/
{
  "name": "Mona Salah",
  "title": "AI Engineer",
```

```

"skills": "TensorFlow, Keras, Python, Deep Learning, NLP",
"summary": "Focused on building deep learning models and deploying NLP pipelines.",
"years_experience": 4,
"location": "Alexandria, Egypt"
}

```

### ◆ Step 3: Search Candidates by Skill

```

json

GET /resumes/_search
{
  "query": {
    "match": {
      "skills": "NLP"
    }
  }
}

```

✓ This works because your `skills` field uses a **comma + whitespace** analyzer.

### ◆ Step 4: Search by Job Title Phrase

```

json

GET /resumes/_search
{
  "query": {
    "match_phrase": {
      "title": "data scientist"
    }
  }
}

```

This ensures candidates with the exact job title are prioritized.

### ◆ Step 5: Search by Experience + Location Filter

```

json

GET /resumes/_search
{
  "query": {
    "bool": {
      "must": [
        { "range": { "years_experience": { "gte": 5 } } }
      ],
      "filter": [
        { "term": { "location": "Cairo, Egypt" } }
      ]
    }
  }
}

```

## Advanced Search Exercises

1. Search for "machine learning" in the `summary` field (analyzed)
  2. Search for resumes with both "Python" and "ElasticSearch" in `skills`
  3. Filter candidates with `years_experience >= 5` and title match "Engineer"
  4. Create a resume with `skills: "python elasticsearch, pandas"` — test analyzers
- 

✅ You've now applied advanced analyzer control to a realistic use case — and created a highly searchable, performance-optimized resume database.

Would you like to proceed to **Part 4 – Segment 1: Aggregations and Metrics?**

Fantastic — welcome to **Part 4**, where we dive into **Aggregations**: the heart of Elasticsearch analytics.

## ◆ Part 4 – Segment 1: Aggregations and Metrics

 **Estimated Time: 25–30 minutes**

This segment introduces Elasticsearch's **aggregation framework**, used for analytics, dashboards, and real-time reporting — similar to SQL's `GROUP BY` and `AVG()`.

### Objective

- Use **terms**, **avg**, **min**, **max**, and **stats** aggregations
- Group and summarize numerical and categorical data
- Understand nested aggregations (e.g., group by category → avg price)

### Aggregation Basics

There are two types:

Type	Example
<b>Bucket</b>	<b>terms</b> , <b>range</b> , <b>date_histogram</b> — groups docs
<b>Metric</b>	<b>avg</b> , <b>min</b> , <b>max</b> , <b>sum</b> , <b>stats</b> — compute values

Buckets group the data. Metrics calculate values within each group.

### ✓ Step-by-Step Lab – Use Your products Index

You should still have:

- name
- category (keyword)
- price (float)
- stock (integer)

#### ◆ Step 1: Terms Aggregation – Group by Category

```
json

GET /products/_search
{
  "size": 0,
  "aggs": {
    "by_category": {
      "terms": {
        "field": "category"
      }
    }
  }
}
```

```
}
}
```

✓ You'll get output like:

```
json

{"buckets": [
  { "key": "Smartphones", "doc_count": 2 },
  { "key": "Headphones", "doc_count": 1 }
]}
```

## ◆ Step 2: Avg Price Aggregation

```
json

GET /products/_search
{
  "size": 0,
  "aggs": {
    "average_price": {
      "avg": {
        "field": "price"
      }
    }
  }
}
```

Get the average price across all products

## ◆ Step 3: Combine Terms + Metric (Nested Aggregation)

🎯 For each category, get the average price

```
json

GET /products/_search
{
  "size": 0,
  "aggs": {
    "by_category": {
      "terms": {
        "field": "category"
      },
      "aggs": {
        "avg_price": {
          "avg": {
            "field": "price"
          }
        }
      }
    }
  }
}
```

✓ Output structure:

```
json

"buckets": [
  {
    "key": "Smartphones",
    "doc_count": 2,
    "avg_price": {
      "value": 1149.75
    }
  }
]
```

#### ◆ Step 4: Min/Max/Stats on Stock

```
json

GET /products/_search
{
  "size": 0,
  "aggs": {
    "stock_stats": {
      "stats": {
        "field": "stock"
      }
    }
  }
}
```

✓ You get:

- min
- max
- avg
- sum
- count

#### ◆ Step 5: Range Aggregation on Price


```
json

GET /products/_search
{
  "size": 0,
  "aggs": {
    "price_ranges": {
      "range": {
        "field": "price",
        "ranges": [
          { "to": 500 },
          { "from": 500, "to": 1000 },
          { "from": 1000 }
        ]
      }
    }
  }
}
```

✓ Groups products into custom price bands.

## Practice Exercise

1. Count how many products are in each category
2. Get total stock (**sum**) grouped by category
3. Find the highest-priced product in **Smartphones** category (hint: filter + max)
4. Use **stats** on price instead of just **avg**

 You now understand core aggregations for analytics.

Would you like to continue to **Segment 2: Aggregations on Resume Dataset (skills, experience, location)**?

Perfect — let's now apply aggregation skills to a more human-centric dataset.

## ◆ Part 4 – Segment 2: Aggregations on Resume Dataset (Skills, Experience, Location)

 **Estimated Time: 25–30 minutes**

This segment shows how to **analyze people data** with aggregations — such as how many candidates know Python, or which city has the most experienced candidates.

### Objective

- Use **terms**, **avg**, **max**, and **stats** aggregations on the **resumes** index
- Perform insights like top skills, location-based averages, and experience distribution

### Step-by-Step Lab – Use the **resumes** Index


Your documents should include:

- **skills** (text, comma-delimited)
- **location** (keyword)
- **years\_experience** (integer)
- **title** (text)

#### ◆ Step 1: Aggregate by Location (Terms)

```
json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "by_location": {
      "terms": {
        "field": "location"
      }
    }
  }
}
```

 Result shows how many resumes are from each city.

#### ◆ Step 2: Average Experience by Location

```
json

GET /resumes/_search
{
```



```

"size": 0,
"aggs": {
  "by_location": {
    "terms": {
      "field": "location"
    },
    "aggs": {
      "avg_experience": {
        "avg": {
          "field": "years_experience"
        }
      }
    }
  }
}
}
}

```

✓ This helps recruiters prioritize locations with more senior talent.

### ◆ Step 3: Top Job Titles (Terms)

```

json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "top_titles": {
      "terms": {
        "field": "title.keyword",
        "size": 5
      }
    }
  }
}

```

Note: If you don't have `.keyword` for title, you must re-map that field or use a custom analyzer.

### ◆ Step 4: Group Experience by Skill (Optional)

If `skills` were stored as an array of keywords:

```

json

"skills": ["Python", "NLP", "Pandas"]

```

Then you could run:

```

json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "by_skill": {
      "terms": {
        "field": "skills.keyword"
      },
      "aggs": {
        "avg_experience": {
          "avg": {
            "field": "years_experience"
          }
        }
      }
    }
  }
}

```

```
}
}
}
}
```

But if **skills** is stored as a single string (comma-separated), you'll need a custom analyzer + reindexing for this to work cleanly.

## ◆ Step 5: Bucket Range Aggregation on Experience

```
json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "experience_ranges": {
      "range": {
        "field": "years_experience",
        "ranges": [
          { "to": 3 },
          { "from": 3, "to": 6 },
          { "from": 6 }
        ]
      }
    }
  }
}
```

✓ Result:

- Junior: <3 years
- Mid: 3–6 years
- Senior: >6 years

## Practice Challenge

1. Find the **top 3 most common locations** for resumes
2. Group resumes by title and show **average years of experience**
3. Add a **skills.keyword** field and group by it
4. Use **range** aggregation to create **custom seniority levels**

✓ You've now practiced bucketed and metric aggregations in a **resume dataset**, preparing you for dashboards and talent analytics.

Would you like to proceed to **Segment 3: Filtering Aggregations and Nested Filters**?

Excellent — now let's go one level deeper by learning to **filter aggregations**, which is critical when you need **analytics within subsets of data**.

## ◆ Part 4 – Segment 3: Filtering Aggregations and Nested Filters

 **Estimated Time: 25–30 minutes**

This segment shows how to:

- Apply **filters to aggregations** (e.g., average salary for Python devs only)
- Use **filter**, **filters**, and **bucket\_selector** aggregations
- Combine aggregations with **query + filter logic**

### Objective

- Limit aggregations to a subset of documents
- Perform grouped metrics **with filtering conditions**
- Use **filter** and **filters** aggregations for dashboards

### Step-by-Step Lab – Resume Use Case

#### ◆ Step 1: Filter Aggregation – Avg Experience of Python Devs Only

```
json
GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "python_only": {
      "filter": {
        "match": {
          "skills": "Python"
        }
      },
      "aggs": {
        "avg_experience": {
          "avg": {
            "field": "years_experience"
          }
        }
      }
    }
  }
}
```

 Only resumes with "Python" in **skills** are used.

#### ◆ Step 2: Multiple Named Filters (Like CASE in SQL)

This uses the **filters** aggregation — **multiple filters at once**:

```
json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "experience_groups": {
      "filters": {
        "filters": {
          "junior": { "range": { "years_experience": { "lt": 3 } } },
          "mid": { "range": { "years_experience": { "gte": 3, "lt": 6 } } },
          "senior": { "range": { "years_experience": { "gte": 6 } } }
        }
      }
    }
  }
}
```

✓ This gives buckets like:

```
json

{
  "buckets": {
    "junior": { "doc_count": 3 },
    "mid": { "doc_count": 5 },
    "senior": { "doc_count": 2 }
  }
}
```

### ◆ Step 3: Combine Query + Aggregation Filter

Let's find:

Resumes with **title: Data Scientist**, then group them by location.

```
json

GET /resumes/_search
{
  "size": 0,
  "query": {
    "match": {
      "title": "Data Scientist"
    }
  },
  "aggs": {
    "locations": {
      "terms": {
        "field": "location"
      }
    }
  }
}
```

✓ This ensures **both search results and aggregations** are filtered.

### ◆ Step 4: Add bucket\_selector to Filter Results

Let's group by location but **keep only those with avg experience > 5**

```
json
```

```
GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "by_location": {
      "terms": {
        "field": "location"
      },
      "aggs": {
        "avg_exp": {
          "avg": {
            "field": "years_experience"
          }
        }
      },
      "only_senior_locations": {
        "bucket_selector": {
          "buckets_path": {
            "avgExp": "avg_exp"
          },
          "script": "params.avgExp > 5"
        }
      }
    }
  }
}
```

✓ Only locations with **average experience > 5** are shown in results.

## Practice Exercises

1. Create a filter aggregation for resumes mentioning "NLP" in **skills**
2. Build 3 filters: "Data Scientist", "ML Engineer", "AI Engineer" — compare count
3. Use **bucket\_selector** to exclude cities with fewer than 2 resumes
4. Combine filters with nested **avg** aggregations

✓ You now understand how to **filter inside aggregations**, just like **WHERE + GROUP BY** in SQL — essential for dashboards and segmented reports.

Would you like to proceed to **Segment 4: Challenge – Resume Dashboard Aggregation Blueprint?**

Perfect — let's finish strong with a practical and comprehensive challenge.

## ◆ Part 4 – Segment 4: Challenge – Resume Dashboard Aggregation Blueprint

 **Estimated Time: 30–35 minutes**

In this challenge, you'll simulate a full-stack analytics backend for a **resume search dashboard**. You'll create queries for filters, pie charts, bar graphs, and KPIs — all using **ElasticSearch aggregations** only.

### Challenge Goals

- Build a dashboard backend using aggregation queries
- Return values for filters, charts, and summaries
- Combine terms, ranges, metrics, and filters

### Assume Your Resume Documents Include:

```
json
{
  "name": "Ahmed Farg",
  "title": "AI Engineer",
  "skills": ["Python", "NLP", "TensorFlow"],
  "location": "Cairo",
  "years_experience": 5,
  "education_level": "Master's"
}
```

Fields:

- **skills** → keyword (array)
- **title** → keyword
- **location** → keyword
- **years\_experience** → integer
- **education\_level** → keyword

### Dashboard Layout

Section	Description
KPI 1	Total resumes
KPI 2	Average experience
Filter 1	Top 5 job titles
Filter 2	Education levels

Section	Description
Chart 1	Pie chart of resumes by location
Chart 2	Bar chart: avg experience per title
Chart 3	Range chart: experience buckets

## ✓ Build Each Panel Using Aggregations

### ◆ KPI 1: Total Resumes

```
json

GET /resumes/_count
```

✓ Fastest way to get total number of resumes.

### ◆ KPI 2: Average Experience

```
json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "avg_experience": {
      "avg": {
        "field": "years_experience"
      }
    }
  }
}
```

### ◆ Filter 1: Top 5 Job Titles (Dropdown)

```
json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "top_titles": {
      "terms": {
        "field": "title",
        "size": 5
      }
    }
  }
}
```

### ◆ Filter 2: All Education Levels

```
json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "education_levels": {
      "terms": {
        "field": "education_level.keyword",
        "size": 10
      }
    }
  }
}
```

### ◆ Chart 1: Pie Chart – Location Distribution

```
json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "by_location": {
      "terms": {
        "field": "location",
        "size": 10
      }
    }
  }
}
```

### ◆ Chart 2: Bar Chart – Average Experience per Title

```
json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "by_title": {
      "terms": {
        "field": "title",
        "size": 10
      },
      "aggs": {
        "avg_exp": {
          "avg": {
            "field": "years_experience"
          }
        }
      }
    }
  }
}
```

### ◆ Chart 3: Range Chart – Experience Buckets



```
json

GET /resumes/_search
{
  "size": 0,
  "aggs": {
    "experience_ranges": {
      "range": {
        "field": "years_experience",
        "ranges": [
          { "to": 3, "key": "Junior" },
          { "from": 3, "to": 6, "key": "Mid" },
          { "from": 6, "key": "Senior" }
        ]
      }
    }
  }
}
```

✅ Ideal for stacked bar chart or vertical columns.

## Bonus: Filter + Aggregation Combo

Show average experience by **title**, but only for resumes in **Cairo**:

```
json

GET /resumes/_search
{
  "size": 0,
  "query": {
    "term": {
      "location": "Cairo"
    }
  },
  "aggs": {
    "titles": {
      "terms": {
        "field": "title"
      },
      "aggs": {
        "avg_experience": {
          "avg": {
            "field": "years_experience"
          }
        }
      }
    }
  }
}
```

## Dashboard Notes:

- Each section is **query-driven**, real-time, and composable
- You can add **date\_histogram** for trends (if resumes had **created\_at**)
- Combine with UI tools like **Kibana**, **React**, or **Grafana** for visuals

✅ You've now completed a **full Elasticsearch lab tutorial with 4 parts**, covering real-world use cases, filters, analyzers, metrics, and dashboards.

Would you like a **final summary / checklist** for all parts or start a new lab theme (e.g., search optimization, relevance tuning, vector search, or Kibana integration)?

Here's a complete **Final Summary / Checklist** of your 4-hour **ElasticSearch Hands-On Lab** — covering all the practical segments delivered in detail:

---

## ✓ **Part 1: Getting Started and Indexing Basics**

**Goal:** Get familiar with ElasticSearch documents, indexing, and basic queries.

### ✓ **Covered Topics:**

- Creating indexes and documents with **PUT**, **POST**
- Viewing documents (**GET /index/\_search, \_doc/{id}**)
- Updating documents with **\_update**
- Deleting documents and indexes
- Searching with **match**, **match\_all**, and **term**
- Filtering with **range**, **bool**, and **must**

### **Hands-On Exercises:**

- Index and retrieve product documents
  - Practice match vs term queries
  - Update fields like **price** and **stock**
  - Use range filters to find expensive items
- 

## ✓ **Part 2: Advanced Querying and Filters**

**Goal:** Learn advanced query combinations and conditional logic.

### ✓ **Covered Topics:**

- **bool**, **must**, **should**, **must\_not**
- Nested filtering and combining match/range
- Field selection (**\_source**)
- Sorting and pagination (**from**, **size**, **sort**)
- Filtering vs querying (scoring difference)

### **Hands-On Exercises:**

- Write complex resume filters with **must** and **should**
  - Sort products by price or release date
  - Paginate search results for browsing resumes
-

## **Part 3: Mapping, Analyzers, and Custom Fields**

**Goal:** Structure your data precisely for accurate search and performance.

### **Covered Topics:**

- Explicit **mappings** with field types
- Text vs keyword fields
- **dynamic** vs **strict** mappings
- **\_analyze** API and built-in analyzers
- Custom analyzers with stopwords, lowercase, filters
- Resume indexing with analyzers for **skills**, **title**, and **summary**

### **Hands-On Exercises:**

- Create **products** and **resumes** indexes with custom mappings
- Analyze input using **\_analyze** for clarity
- Match and match\_phrase behavior testing
- Resume challenge with full-text skill search

## **Part 4: Aggregations and Analytics**

**Goal:** Use Elasticsearch as an analytics engine (like SQL **GROUP BY**).

### **Covered Topics:**

- Bucket aggregations: **terms**, **range**, **filter**, **filters**
- Metric aggregations: **avg**, **min**, **max**, **sum**, **stats**
- Nested aggregations: group + metric (e.g., group by category → avg price)
- Filtering inside aggregations with **filter**, **bucket\_selector**
- Resume analytics dashboard use case

### **Hands-On Exercises:**

- Aggregate products by category, price, and stock
- Build filters and charts on resume data (skills, experience, titles)
- Apply **bucket\_selector** to exclude groups dynamically
- Build dashboard panels: KPI, pie, bar, range







## **Outcome**

You've:

- Built multiple indexes (**products**, **resumes**, **books\_custom**)
- Mastered core query types and filtering logic
- Controlled document structure using mapping and analyzers
- Designed analytics pipelines and dashboards with aggregations
- Prepared Elasticsearch for real-world search and analytics backends

## Optional Next Steps (Suggestions)

Would you like to proceed with:

Option	Description
 <b>Search Optimization</b>	Relevance scoring, boosting, custom ranking
 <b>Kibana Dashboards</b>	Visualize queries using Kibana
 <b>Elastic + ML</b>	Anomaly detection, model inference via Elastic
 <b>Indexing Pipelines</b>	Ingest pipelines, enrich processors, log ingestion
 <b>Vector Search &amp; Embeddings</b>	Build semantic search with OpenAI/BERT embeddings
 <b>Security</b>	User roles, API keys, secure access

Let me know which you'd like to dive into next!