Query DSL & ES|QL

Day 2 — 4-Day Elasticsearch Course

Elasticsearch 8.18 | Full-text · Term-level · Bool · ES|QL

Agenda

Query DSL Overview

The JSON-based query language

The Search Request Envelope

Every search follows the same structure:

```
GET /<index>/_search
{
  "query": { ... },           // What to search for
  "_source": [...],           // Which fields to return
  "size": 10,                 // Number of results (default: 10)
  "from": 0,                  // Offset for pagination
  "sort": [...],              // Custom sort order
  "highlight": { ... }        // Highlight matching terms
}
```

Only `query` is required — everything else is optional.

Query Context vs Filter Context

	Query Context	**Filter Context**
Purpose	"How well does this match?"	"Does this match? Yes/No"
Scoring	Calculates `_score`	No scoring
Performance	Slower (scoring overhead)	Faster + cacheable
Use for	Full-text search	Exact filters (dates, status, ranges)

Query Context vs Filter Context: Example

```
{
  "query": {
    "bool": {
      "must": { ... },    // ← Query context (scored)
      "filter": { ... }   // ← Filter context (not scored, cached)
    }
  }
}
```

Rule of thumb: Use `filter` for anything that doesn't need relevance scoring.

Search Response Anatomy

```
{
  "took": 5,                                  // Time in milliseconds
  "timed_out": false,
  "hits": {
    "total": { "value": 42 },,                // Total matching documents
    "max_score": 8.23,                        // Highest relevance score
    "hits": [                                  // The actual results
      {
        "_index": "movies",
        "_id": "1",
        "_score": 8.23,                       // This document's score
        "_source": { ... }                    // The document fields
      }
    ]
  }
}
```

Full-text Queries

match · multi_match · match_phrase

match Query

The **most common** search query — analyzes the input and finds matching documents.

```
GET /movies/_search
{
  "query": {
    "match": {
      "overview": "space exploration"
    }
  }
}
```

What happens:

1. `"space exploration"` → analyzer → `["space", "exploration"]`
2. Finds documents containing **space** OR **exploration** (default: `or`)
3. Results scored by BM25

match: operator

Require ALL terms

```
GET /movies/_search
{
  "query": {
    "match": {
      "overview": {
        "query": "space exploration",
        "operator": "and"
      }
    }
  }
}
```

- Default operator is `or` — any term matches
- `"operator": "and"` — all terms must be present

match: minimum_should_match

Require at least N terms

```
GET /movies/_search
{
  "query": {
    "match": {
      "overview": {
        "query": "epic space adventure exploration",
        "minimum_should_match": "75%"
      }
    }
  }
}
```

- `"75%"` of 4 terms = at least 3 must match
- Useful for longer queries where exact match is too strict

multi_match Query

Search across **multiple fields** at once:

```
GET /movies/_search
{
  "query": {
    "multi_match": {
      "query": "dark knight",
      "fields": ["title^3", "overview"]
    }
  }
}
```

- `title^3` — boost title matches by 3x
- Matches in `title` are weighted more heavily than `overview`
- Great for search boxes where users don't specify a field

multi_match: Types

Type	Behavior
`best_fields` (default)	Score from the single best matching field
`most_fields`	Sum scores from all matching fields
`cross_fields`	Treat all fields as one combined field
`phrase`	Run match_phrase on each field

multi_match: cross_fields

```
GET /movies/_search
{
  "query": {
    "multi_match": {
      "query": "Christopher Nolan batman",
      "fields": ["title^2", "overview"],
      "type": "cross_fields"
    }
  }
}
```

`cross_fields` is ideal when the search terms may span different fields (e.g., name in title, description in overview).

match_phrase Query

Finds documents where terms appear **in the exact order**, adjacent to each other:

```
GET /movies/_search
{
  "query": {
    "match_phrase": {
      "overview": "organized crime"
    }
  }
}
```

- `"organized crime"` matches `"...organized crime dynasty..."`
- Does NOT match `"...crime was organized..."`

match_phrase: slop

Allow terms to be **N positions apart**:

```
GET /movies/_search
{
  "query": {
    "match_phrase": {
      "overview": {
        "query": "imprisoned redemption",
        "slop": 5
      }
    }
  }
}
```

- `slop: 0` → terms must be adjacent (default)
- `slop: 5` → terms can be up to 5 positions apart
- Higher slop = more flexible but less precise

Term-level Queries

Exact matching without analysis

term vs match

	`match`	`term`
Analyzes query?	Yes	No
Use for	`text` fields	`keyword`, `integer`, `date` fields
Example	"The Godfather" → searches for `the`, `godfather`	"Drama" → searches for exact `Drama`

```
// CORRECT: term on keyword field
GET /movies/_search
{ "query": { "term": { "genres": "Drama" } } }

// WRONG: term on text field (analyzed text won't match)
GET /movies/_search
{ "query": { "term": { "title": "The Godfather" } } }
```

terms Query

Match any of several values (like SQL `IN`):

```
GET /movies/_search
{
  "query": {
    "terms": {
      "genres": ["Action", "Thriller", "Horror"]
    }
  }
}
```

Returns documents where `genres` contains **at least one** of the listed values.

range Query

Numeric ranges, date ranges, and more:

```
GET /movies/_search
{
  "query": {
    "range": {
      "vote_average": {
        "gte": 8.0,
        "lt": 9.0
      }
    }
  }
}
```

Operators: `gt`, `gte`, `lt`, `lte`

range: Date Ranges

```
GET /movies/_search
{
  "query": {
    "range": {
      "release_date": {
        "gte": "2000-01-01",
        "lt": "2010-01-01"
      }
    }
  }
}
```

- Dates support ISO 8601 format and date math: `"now-1d"`, `"2024-01||/M"`
- Range queries are commonly used in `filter` context (no scoring needed)

exists Query

Check if a field has a value

```
GET /movies/_search
{
  "query": {
    "exists": {
      "field": "tagline"
    }
  }
}
```

- Returns documents where the field is **not null** and **not empty**
- Useful for filtering incomplete data

prefix Query

Prefix search on keyword fields

```
GET /movies/_search
{
  "query": {
    "prefix": {
      "title.keyword": {
        "value": "The"
      }
    }
  }
}
```

For autocomplete, prefer edge n-gram analyzers (Day 3) over prefix queries — they're faster at scale.

Compound Queries

bool — the Swiss army knife

bool Query Structure

Combine multiple clauses with different logic:

```
GET /movies/_search
{
  "query": {
    "bool": {
      "must": [...],          // AND — scored
      "filter": [...],        // AND — not scored (fast, cached)
      "should": [...],        // OR — scored (boosts matching docs)
      "must_not": [...]       // NOT — excludes, not scored
    }
  }
}
```

bool: Clause Summary

Clause	Logic	Scored?	Cached?
`must`	AND	Yes	No
`filter`	AND	No	Yes
`should`	OR	Yes	No
`must_not`	NOT	No	Yes

> `filter` and `must_not` are faster because they skip scoring and can be cached.

bool: Practical Example

"Find highly-rated sci-fi movies about space, excluding horror"

```
GET /movies/_search
{
  "query": {
    "bool": {
      "must": {
        "match": { "overview": "space adventure" }
      },
      "filter": [
        { "term": { "genres": "Science Fiction" } },
        { "range": { "vote_average": { "gte": 7.5 } } }
      ],
      "must_not": {
        "term": { "genres": "Horror" }
      }
    }
  }
}
```

bool: How Clauses Work Together

Clause	Role in the example
`must`	Contributes to `_score` (relevance for "space adventure")
`filter`	Exact criteria, fast, no scoring overhead
`must_not`	Hard exclusion (removes Horror)

Rule of thumb: Put full-text search in `must`, everything else in `filter`.

Nested bool Queries

"Drama OR Crime movies with rating 8+ that mention 'family' or 'power'"

```
GET /movies/_search
{
  "query": {
    "bool": {
      "must": {
        "match": { "overview": "family power" }
      },
      "filter": {
        "range": { "vote_average": { "gte": 8.0 } }
      },
      "should": [
        { "term": { "genres": "Drama" } },
        { "term": { "genres": "Crime" } }
      ],
      "minimum_should_match": 1
    }
  }
}
```

should as Boost

When `must` or `filter` is present, `should` becomes a **boost** — matching docs rank higher but non-matching docs aren't excluded:

```
GET /movies/_search
{
  "query": {
    "bool": {
      "must": {
        "match": { "overview": "war" }
      },
      "should": [
        { "match": { "title": "war" } },
        { "range": { "vote_average": { "gte": 8.5 } } }
      ]
    }
  }
}
```

Documents matching `should` clauses get a **score boost**.

Pagination, Sorting & Highlighting

Controlling search output

Pagination: from + size

```
GET /movies/_search
{
  "query": { "match_all": {} },
  "from": 0,
  "size": 10
}
```

Page	from	size
1	0	10
2	10	10
3	20	10

Limitation: `from + size` cannot exceed **10,000**. For deep pagination, use `search_after` or the Scroll API.

Sorting

Sort by field

```
GET /movies/_search
{
  "query": { "match_all": {} },
  "sort": [
    { "vote_average": "desc" },
    { "release_date": "asc" }
  ]
}
```

Sort by relevance (default)

```
"sort": ["_score"]
```

When you sort by a field other than `_score`, relevance scoring is **disabled** (faster).

_source Filtering

Control which fields are returned:

```
GET /movies/_search
{
  "query": { "match": { "overview": "adventure" } },
  "_source": ["title", "vote_average", "genres"]
}
```

Exclude fields

```
"_source": {
  "excludes": ["overview"]
}
```

Returning fewer fields = smaller response = faster network transfer.

Highlighting

Show which parts of the text matched:

```
GET /movies/_search
{
  "query": {
    "match": { "overview": "imprisoned redemption" }
  },
  "highlight": {
    "fields": {
      "overview": {}
    }
  }
}
```

Highlighting: Response

```
"highlight": {
  "overview": [
    "Two <em>imprisoned</em> men bond over years, finding <em>redemption</em>..."
  ]
}
```

- Matching terms are wrapped in `<em>` tags by default
- Customize tags: `"pre_tags": ["<b>"], "post_tags": ["</b>"]`

Practice 2A

Query DSL

`day2-exercises.md` — Part A (7 tasks) | ~25 min

ES|QL

Elasticsearch Query Language

What is ES|QL?

A **pipe-based query language** for Elasticsearch (introduced in 8.11):

```
FROM movies
| WHERE vote_average >= 8.0
| SORT vote_average DESC
| LIMIT 10
| KEEP title, vote_average, genres
```

Key differences from Query DSL:

- SQL-like syntax with **pipe** (`|`) chaining
- Built-in aggregations without nesting
- Returns **columnar** data (not JSON documents)
- Runs its own execution engine (not Lucene queries)

ES|QL: Core Commands

Command	Purpose	Example
`FROM`	Source index	`FROM movies`
`WHERE`	Filter rows	`WHERE vote_average > 8`
`EVAL`	Compute new columns	`EVAL decade = release_date / 10 * 10`
`STATS...BY`	Aggregate + group	`STATS avg(vote_average) BY genres`
`SORT`	Order results	`SORT vote_average DESC`
`LIMIT`	Limit rows	`LIMIT 20`
`KEEP`	Select columns	`KEEP title, genres`
`DROP`	Remove columns	`DROP overview`
`RENAME`	Rename columns	`RENAME vote_average AS rating`

ES|QL: Filtering

```
FROM movies
| WHERE vote_average >= 8.0 AND genres == "Drama"
| SORT vote_average DESC
| LIMIT 5
| KEEP title, vote_average
```

Operators

Operator	Example
`==`, `!=`	`genres == "Drama"`
`>`, `>=`, `<`, `<=`	`vote_average >= 8.0`
`AND`, `OR`, `NOT`	`genres == "Drama" AND vote_average > 7`
`LIKE`	`title LIKE "The *"`
`IN`	`genres IN ("Action", "Drama")`

ES|QL: EVAL (Computed Columns)

Create new columns from expressions:

```
FROM movies
| EVAL rating_category = CASE(
    vote_average >= 8.0, "Excellent",
    vote_average >= 6.0, "Good",
    "Average"
  )
| STATS count = COUNT(*) BY rating_category
```

ES|QL: EVAL with Dates

```
FROM movies
| EVAL year = DATE_EXTRACT("year", release_date)
| EVAL decade = FLOOR(year / 10) * 10
| STATS count = COUNT(*), avg_rating = AVG(vote_average) BY decade
| SORT decade
```

- `DATE_EXTRACT` pulls parts from date fields (year, month, day)
- `FLOOR` rounds down for grouping into decades

ES|QL: STATS...BY (Aggregations)

```
FROM movies
| STATS
    count = COUNT(*),
    avg_rating = AVG(vote_average),
    max_rating = MAX(vote_average),
    min_rating = MIN(vote_average)
  BY genres
| SORT count DESC
| LIMIT 10
```

Available functions

`COUNT`, `AVG`, `SUM`, `MIN`, `MAX`, `MEDIAN`, `PERCENTILE`, `COUNT_DISTINCT`, `VALUES`

ES|QL: Multi-value Fields

Genres is an array — use `MV_EXPAND` to unnest:

```
FROM movies
| MV_EXPAND genres
| STATS count = COUNT(*), avg_rating = AVG(vote_average) BY genres
| SORT count DESC
```

Without `MV_EXPAND`, multi-value fields are treated as a single entity.

ES|QL in Kibana: Dev Tools

```
POST /_query
{
  "query": """
    FROM movies
    | WHERE vote_average >= 8.0
    | SORT vote_average DESC
    | LIMIT 10
    | KEEP title, vote_average, genres
  """
}
```

Wrap ES|QL in triple-quotes inside the JSON body.

ES|QL in Kibana: Discover

1. Open **Discover**
2. Click the language dropdown (top left)
3. Switch from **KQL** to **ES|QL**
4. Type your ES|QL query directly

- Discover renders ES|QL results as a **table** (columnar)
- Great for quick data exploration without writing JSON

ES|QL vs Query DSL

Aspect	Query DSL	ES|QL
Syntax	JSON	Pipe-based text
Scoring	BM25 relevance	No scoring
Aggregations	Nested JSON	`STATS...BY`
Pagination	`from`/`size`	`LIMIT`
Use case	Search with relevance	Analytics & exploration
Maturity	Production-ready	GA since 8.14

> Use **Query DSL** for search features, **ES|QL** for data exploration and analytics.

Practice 2B

ES|QL

`day2-exercises.md` — Part B (6 tasks) | ~20 min

Summary

Day 2 Recap

Topic	Key Queries
Full-text	`match`, `multi_match` (types, boosting), `match_phrase` (slop)
Term-level	`term`, `terms`, `range`, `exists`, `prefix`
Compound	`bool`: must / filter / should / must_not
Output	`from` / `size`, `sort`, `_source`, `highlight`
ES|QL	`FROM` | `WHERE` | `EVAL` | `STATS...BY` | `SORT` | `LIMIT`

Day 3 Preview

Tomorrow (3-hour session) we'll cover:

- **Index API** — bulk operations, reindex, aliases, templates
- **Text Analysis** — analyzers, tokenizers, custom analyzers, autocomplete
- **Mappings** — dynamic vs explicit, field types, multi-fields
- **Aggregations** — metric, bucket, nested, pipeline aggs
- **Nested & Join** — nested objects, parent-child relationships

> Keep `elk-single` running!

Thank You!

Questions?

Day 2 — Query DSL & ES|QL