1 - Overview

> [!caution] This page contained a drawing which was not converted.

Baseado em Apache Lucene (Java)

Relevância da busca é importante para um negócio

Dados estruturados (tem schema):
Resultados de busca são binários - sim ou não.
Filtros são precisos.
Não tem score de relevância.

Dados não-estruturados (e.g. texto):
Sem *schema*
Pode ser semi-estruturado (i.e. texto com alguns metadados)
Elastic é bom nisso - procura termos iguais e parecidos.
Elastic consegue unir busca de estruturado com não-estruturado (Não é um banco de dados relacional)

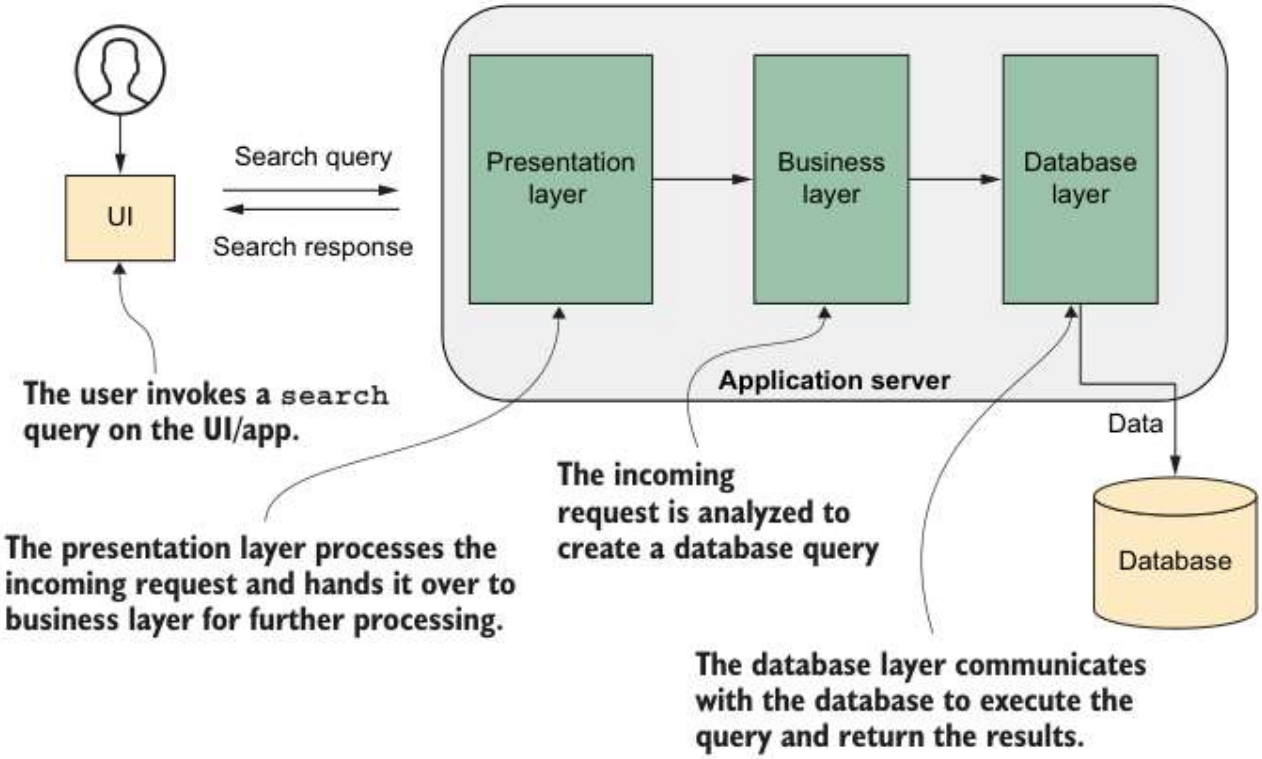


Figure 1.1 Search based on a traditional database

Modelo tradicional com SQL é pouco eficiente para texto não-estruturado.
São melhor adaptados a dados estruturados.
Devagar mesmo quando suporta.
Ainda assim pode ser uma boa solução dependendo do contexto.

- Bancos de dados comuns não conseguem indexar grandes quantidades de dados eficientemente.
- Não costumam ter recursos de busco como fuzzy logic, sinônimos, etc.
- Busca de texto não pode ser por matches exatos apenas.
- Normalização de bancos de dados torna difícil buscar em vários campos diferentes de uma vez.

Um bom buscador de texto deve:

- Ter suporte para dado estruturado e não-estruturado
- Ser flexível com o user input
- Auto-correct
- Escalável, rápido, confiável, sempre-disponível
- Poder buscar geolocalização (?)
- Permitir machine-learning

Fazer buscável = quebrar texto em tokens e mapear cada token ao documento completo.

Solr e **OpenSearch** são alternativas ao **ElasticSearch,** todos vindos do Lucene.
Solr costuma ser usado em datasets grandes e estáticos.

ElasticSearch já vem com features de análise, monitoramento, segurança e machine learning.

Ecossistema Elastic:

Stack Elastic:

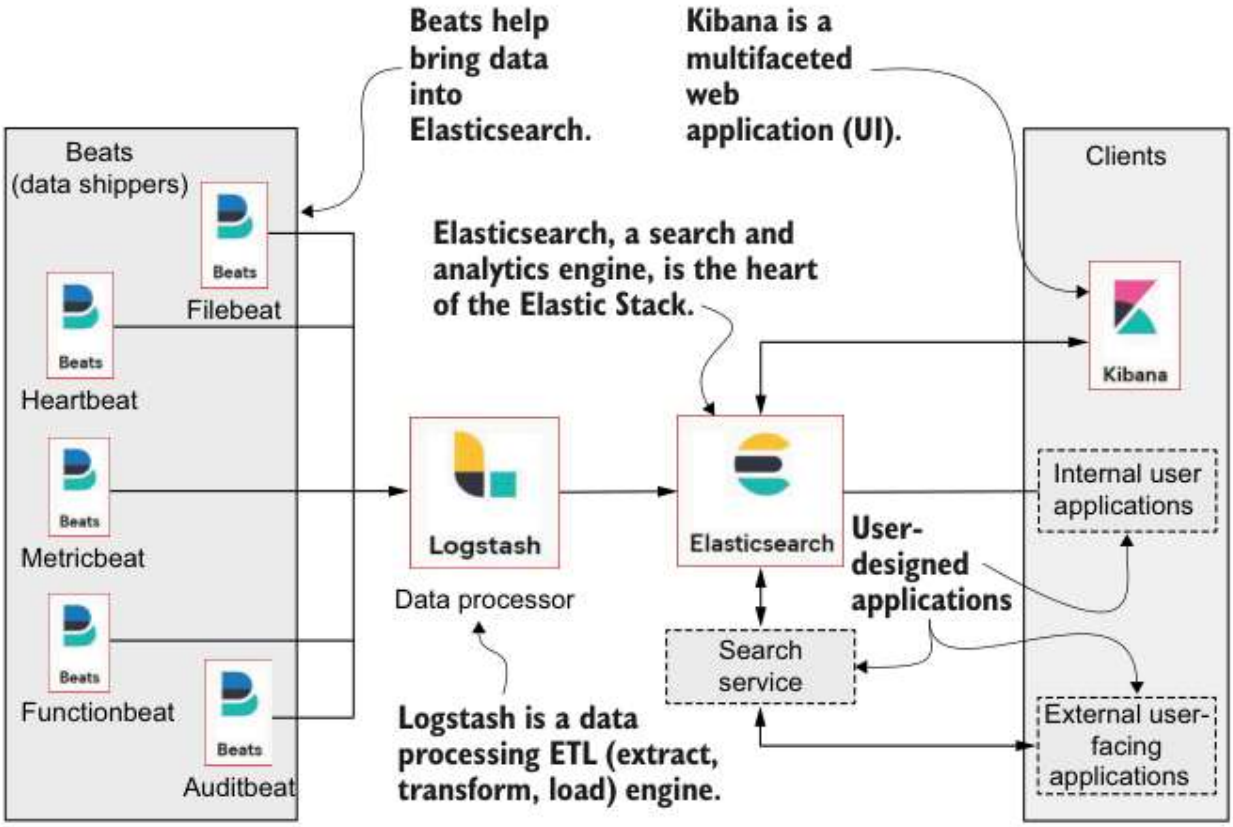


Figure 1.3 The Elastic Stack: Beats, Logstash, Elasticsearch, and Kibana

Beat: carrega dado vindo de algum lugar, consome dados
Logstash: Transforma e transfere dados. Pipeline de ingestão de dados.
Kibana: visualizar e interagir com dados.

Elastic Search é usado em várias aplicações:

- Como busca, três escopos:
  - **App Search** (busca dados internos para uma aplicação)
  - **Enterprise Search** (busca intra organização)
  - **Site Search** (busca em um website)
- Business Analytics
- Security Analytics
- Logging & Monitoramento

É usado por várias empresas grandes como Uber e Netflix.

Elastic Search **NÃO** deve ser usado para:

- Busca em dados relacionais
- Dados transacionais
- Dados geoespaciais em alta escala
- Muitos writes, é bom apenas para muitos reads.
- OLAP análise
- Muitos arquivos binários como imagem e vídeo
- Analytics em tempo real
- Baixa latência
- Séries temporais, grafos, etc.

Não substitui todas as outras soluções de busca.

Gerenciamento do Elastic não é simples

LLMs como ChatGPT têm sido um concorrente de buscadores de texto.
Trazem potencial de tornar as buscas mais intuitivas, personalizadas e preditivas.

Summary

- Search is the new normal and the most sought-after functionality for organizations, enabling competitive advantage.
- Search engines built using relational databases as the backend used to serve our search purposes but can't fulfill the full-fledged search functionality found in modern search engines.
- Modern search engines provide multifaceted, full-text search capabilities and multifold benefits from basic search to advanced search and analytical functions, all with split-second performance. They are also expected to handle terabytes to petabytes of data and scale if needed.
- Elasticsearch is an open source search and analytics engine built over Apache Lucene. It is a highly available server-side application developed in Java.
- Because Elasticsearch was designed as a programming-language-agnostic product, communication with the server takes place over HTTP using rich RESTful APIs. These APIs receive and send data in JSON format.
- The Elastic Stack is a suite of products composed of Beats, Logstash, Elasticsearch, and Kibana. Beats are single-purpose data shippers, Logstash is a data-processing ETL (extract, transform, load) engine, Kibana is the administrative UI tool, and Elasticsearch is the heart and soul of the stack.
- The Elastic Stack enables an organization to position itself in three core areas: search, observability, and security.
- Elasticsearch has become popular over the last few years due to its structured/unstructured search and analytics capabilities; rich set of RESTful APIs; its schema-free nature; and performance, high-availability, and scalability characteristics.
- AI-powered search is here. With the advent of generative AI and ChatGPT, the search space will be explored further, and search will become more intuitive and predictive.

What does the book mean by "Search capabilities on geolocations, when talking about ElasticSearch?

When the book refers to **"Search capabilities on geolocations"** in the context of **Elasticsearch**, it means that Elasticsearch provides built-in support for **geospatial search**, allowing users to store, index, and query data based on geographic locations (latitude/longitude coordinates). This functionality is useful for applications that need location-based searching, filtering, and ranking.

Key Geolocation Features in Elasticsearch:

1. **Geo-Point Data Type**
   - Elasticsearch allows you to store location-based data as a **geo_point** field type, which represents latitude and longitude.
2. **Geo Distance Queries**
   - Users can search for documents within a specified radius from a given point (e.g., "Find all restaurants within 10 miles of my location").
3. **Bounding Box Queries**
   - Searches can be restricted within a rectangular area (bounding box), which is useful for map-based applications.
4. **Geo-Shape Queries**
   - Elasticsearch supports **geo_shapes** (polygons, lines, etc.), allowing complex spatial queries (e.g., "Find all delivery zones that cover a specific address").
5. **Sorting and Ranking by Distance**
   - Results can be **sorted by proximity** to a given point, which is useful for "nearest location" searches.

Example Use Case:

A ride-hailing app like Uber or a food delivery service like DoorDash can use Elasticsearch's **geo capabilities** to:

- Find the **closest drivers** to a passenger.
- Locate restaurants within a delivery range.
- Show users businesses near their current location.

Would you like a specific Elasticsearch query example for geolocation search?

10 - Full-text searches

- Busca de dados não-estruturados

OVERVIEW

- Relevância relacionado a precisão e recall
  - Precisão:
    - Densidade de bons resultados
  - Recall:
    - Completude da busca
  - Trade-off entre as duas medidas

THE MATCH_ALL QUERY

- Busca todos os documentos do index
- Aceita boost ou nenhum argumento

Listing 10.2 Boosting the query with a predefined score

```
GET books/_search
{
  "query": {
    "match_all": {
      "boost": 2
    }
  }
}
```

The match_all query sets a score of 2 for all returned documents.

THE MATCH_NONE QUERY

- Retorna nenhum resultado
- Útil aliado a alguma lógica na aplicação
  - Por exemplo, para retornar nada durante uma manutenção, em vez de dar erro.

THE MATCH QUERY

- O mais comum
  - Podemos procurar por um texto direto no field
  - Ou então passar um key:value com os parâmetros adicionais ao campo.

```
GET books/_search
{
  "query": {
    "match": {
      "FIELD": {
        "query":"<SEARCH TEXT>",
        "<parameter>":"<MY_PARAM>",
      }
    }
  }
}
```

Declares FIELD as an object with additional parameters

The query attribute holds the search text.

The parameter (analyzer, operator, prefix_length, fuzziness, etc.) expects a value to be set.

- Termos procurados são analyzed.
  - A princípio, pelo menos analyzer da indexação.

- Por default, passa por um token filter de lowercase.
- Podemos passar várias palavras
  - Termos são procurados com operador OR
    - Embora mais termos traga score maior.
    - Pode ser alterado para AND
    - Também podemos manter o OR mas estabelecer um número mínimo de palavras para dar match com o parâmetro "*minimum_should_match*"
- Parâmetro fuzziness (distância de levenshtein) também é possível de configurar, valores 0, 1, 2 ou AUTO.

THE MATCH_PHRASE QUERY

- Retorna o match da phrase exata, na mesma sequência
  - Diferente do Match com AND
- Parâmetro slop permite flexibilizar a busca e settar um número de palavras que permite-se pular na busca.
  - Número inteiro, número de palavras que permite-se pular e ainda retornar resultados.

THE MATCH_PHRASE_PREFIX QUERY

- Igual match_phrase mas apenas o prefixo, não o campo inteiro.
- e.g. "Concepts and found" retorna resultados tipo "Concepts and foundations"
- Também suporta slop

THE MULTI_MATCH QUERY

- Procura um string em vários fields ao mesmo tempo.
  - Passamos um array de fields

Listing 10.14 Searching multiple fields using `multi_match`

```
GET books/_search
{
  "_source": false,
  "query": {
    "multi_match": {
      "query": "Java",
      "fields": [
        "title",
        "synopsis",
        "tags"
      ]
    }
  },
  "highlight": {
    "fields": {
      "title": {},
      "tags": {}
    }
  }
}
```

Suppresses the source document from appearing in the results

Specifies the multi_match query

Defines the search criterion as the word "Java"

Searches across multiple fields provided in an array

Highlights the matches returned in the results

- Tem várias formas de scoring:
  - Parâmetro "type"
    - Default é "best_fields", score é o field que tem o melhor score, ignorando os demais.
    - outros: cross_fields, most_fields, phrase, phrase prefix

- Usa query tipo dis_max por trás
  - disjuction max query
  - Separa cada field para uma query separada

Listing 10.16 Disjunction max (dis_max) query in use

```
GET books/_search
{
  "_source": false,
  "query": {
    "dis_max": {
      "queries": [
        {"match": {"title": "Design Patterns"}},
        {"match": {"synopsis": "Design Patterns"}}]
    }
  }
}
```

Specifies the dis_max query type

Defines the set of queries to include in a dis_max query block

Specifies a match query

- Podemos definir tie_breakers
  - Usa score dos outros campos não-best-field para o desempate de relevância
  - float que multiplica score dos não-best-fields
- Podemos dar um boost no score de certos campos
  - e.g. aumentar score para "título" para priorizar sobre outros campos.

THE QUERY_STRING QUERY

- Query com operadores booleanos.
- Feita como a busca direto por URI

Listing 10.20 Creating a query string with operators

```
GET books/_search
{
  "query": {
    "query_string": {
      "query": "author:Bert AND edition:2 AND release_date>=2000-01-01"
    }
  }
}
```

Specifies the query_string query

Provides the search criteria in the query

-
- Útil para criar templates para query do usuário.
- Também podemos definir se os termos são buscados como OR ou AND
- Ou então buscar por frases.
  - Permite slop também

FUZZY QUERIES

- Utilizamos "~" no sufixo do termo da query para definir fuzzy

THE SIMPLE_QUERY_STRING QUERY

- versão simplificada de query_string
- permite só alguns operadores:
  - e.g. "Java + Cay" busca por Java AND Cay
- Não retorna erros mesmo com erro de sintaxe

Summary

- Elasticsearch is big on searching unstructured data using full-text queries. Full-text queries yield relevancy, meaning the documents matched and returned have a positive relevancy score.
- Elasticsearch provides a `_search` API for querying purposes.
- Several kinds of `match` queries work for various use cases when searching full text with relevance. The most common query is the `match` query.
- The `match` query searches on text fields for search criteria and scores documents using the relevance algorithm.
- Match all (`match_all`) queries search across all indexes and do not require a body.
- To search for a phrase, we use a `match_phrase` query or its variant, `match_phrase_prefix`. Both types of queries let us search for specific words in a defined order. Additionally, we can use the `phrase_slop` parameter if the phrase is missing words.
- Searching for user criteria across multiple fields is enabled by using a `multi_match` query.
- A query string (`query_string`) query uses logical operators such as `AND`, `OR`, and `NOT`. However, the `query_string` query is strict on syntax, so we receive an exception if the input syntax is incorrect.
- If we need Elasticsearch to be less strict about the query string syntax, then instead of using a `query_string` query, we can use a `simple_query_string` query. With that query type, all syntactical errors are suppressed by the engine.

11 - Compound Queries

- Fazer queries mais complexas com múltiplos critérios simultâneos não-triviais

COMPOUND QUERIES

- Cinco tipo de compound query:

Compound query	Description
Boolean (`bool`) query	A combination of conditional clauses that wraps individual leaf (term and full-text) queries. Works similarly to the `AND`, `OR`, and `NOT` operators. Example: products= TV AND color = silver NOT rating < 4.5 AND brand = Samsung OR LG
Constant score (`constant_score`) query	Wraps a `filter` query to set constant scores on results. Also helps boost the score. Example: Search for all TVs with user ratings higher than 5 but set a constant score of 5 for each result regardless of the search engine's calculated score.
Function score (`function_score`) query	User-defined functions to assign custom scores to result documents Example: Search for products, and if the product is from LG and is a TV, boost the score by three (via a `script` or `weight` function).
Boosting (`boosting`) query	Boosts the score for positive matches while negating the score for non-matches Example: Fetch all TVs but lower the score of those that are expensive.
Disjunction max (`dis_max`) query	Wraps several queries to search for multiple words in multiple fields (similar to a `multi_match` query) Example: Search for smart TVs in two fields (say, `overview` and `description`) and return the best match.

THE BOOLEAN (BOOL) QUERY

- junta critérios booleanos:
- Quatro termos possíveis

Clause	Description
`must`	An `AND` query where all the documents must match the query criteria Example: Fetch TVs (`product = TV`) that fall within a specific price range
`must_not`	A `NOT` query where none of the documents match the query criteria Example: Fetch TVs (`product = TV`) that fall within a specific price range *but* with an exception, such as not being a certain color
`should`	An `OR` query where one of the documents must match the query criteria Example: Search for fridges that are frost-free *or* energy rated above C grade.
`filter`	A `filter` query where the documents must match the query criteria (similar to the `must` clause), but the `filter` clause does not score the matches Example: Fetch TVs (`product = TV`) that fall within a specific price range (but the score of the documents returned will be zero)

- É possível fazer compound queries (ou queries leaf (simples))dentro deo compound queries:

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": {"FIELD": "TEXT"}},
{ "term": {"FIELD": {"value": "VALUE"}}}
      ],
      "must_not": [
        {"bool": { "must": [{}]}}}
      ]
      "should": [
        { "range": { "FIELD": {"gte": 10,"lte": 20}}},
{ "terms": { "FIELD": [ "VALUE1", "VALUE2" ]}}
      ]
    }
  }
}
```

    - Match query é um caso específico da compound query bool
    - bool { must { match }}
- Vários exemplos
- Podemos usar o should para boost de score, junto com must e must_not
    - A princípio should não filtra nada quando junto de must
    - Podemos definir um threshold mínimo de matches para o should retornar um resultado, neste caso o shuld acaba filtrando sim.
        - *minimum_should_match*
    - Sem nenhum must, should serve de filtro (default de minimum_should_match vai para 1, em vez de 0)
- Filter clause é como o must, mas pula a etapa de scoring, mais rápido.
    - É comum combinar filter com must
        - Não altera o score que o must determinou
- Podemos atribuir nomes para as leaf queries dentro de uma compound query
    - Resultado retorna então quais queries foram usadas para o resultado, e exclui quais não foram

CONSTANT SCORES

- Serve para atribuir um score fixo não-zero aos resultados, sem passar pela função de scoring. Por exemplo para atribuir um score não-zero aos resultados de um filter, que a princípio não gera score.
- Pode servir para compor com os resultados de uma query com must e fazer o ranqueamento melhor.

THE BOOSTING QUERY

- altera os scores dos resultados de uma query
- Tem duas partes:
    - Positiva
        - Retorna os matches da parte positiva
    - Negativa
        - Diminui os scores que atendem um certo critério
- Multiplica score por um coeficiente.
- e.g. diminuir pela metade o score de TVs com preço maior que $2500

THE DISJUNCTION MAX (DIS_MAX) QUERY

- São várias leaf queries em paralelo
- multi_match usa esse por trás.
- Também suporta tiebreaker para usar os não-best fields como desempate

THE FUNCTION_SCORE QUERY

- Cria scoring customizado, como random ou baseado em um script, ignore o scoring default da busca.
- random_score
  - randômico
  - suporta seed (junto de field) para reprodutibilidade
- script_score
  - Podemos criar um score baseado nos valores dos campos, não necessariamente relacionado com a relevância da query em si.

Listing 11.28 Multiplying the field's value by an external parameter

```
GET products/_search
{
  "query": {
    "function_score": {
      "query": {
        "term": {
          "product": "tv"
        }
      },
      "script_score": {
        "script": { #B
          "source":"_score * doc['user_ratings'].value * params['factor']",
          "params": {
            "factor":3
          }
        }
      }
    }
  }
}
```

The script_score function holds the key to generating a score based on its defined script.

The script object

Passes the external parameters to the script

The source is where we define our logic.

- field_value_factor
  - Simplesmente atribui um outro campo como score (e.g. "user rating")
  - Dispensa script
  - Podemos modificar o score por um coeficiente ou elevando ao quadrado, por exemplo
- Podemos combinar diversos scores diferenets para compor um score final.
  - Por default esses scores se multiplicam
    - Podemos alterar com parâmetro "score_mode"
  - Igualmente para boost, boost default é de multiplicação

- "boost_mode" pode ser alterado para min, max, replace, avg ou soma.

Summary

- Compound queries combine leaf queries to create advanced queries that satisfy multiple search criteria.
- The `bool` query is the most popular compound query, consisting of four clauses: `must`, `must_not`, `should`, and `filter`.
- The queries in `must_not` and `filter` clauses do not contribute to the overall relevance score. On the other hand, queries in `must` and `should` clauses always improve the scoring.
- The `constant_score` query wraps a `filter` query and produces a constant score set by the user.
- The `boosting` query increases the score of a `positive` clause while suppressing the score on the queries that aren't a match (the `negative` clause).
- The `dis_max` query, used by the `multi_match` query, wraps queries and executes them individually.
- The `function_score` query sets a custom score based on a user-defined function, such as a field's value or weight or a random value.

12 - Advanced Search

INTRODUCING LOCATION SEARCH

- Busca de localização
- geoquery
    - Largamente suportado no ES
    - Data types:
        - geo_point
        - geo_shape
    - Queries suportadas
        - bounding_box
        - geo_distance
        - geo_shape
- bounding_box query
    - Busca em uma dada área (e.g. um retângulo)
    - definido com dois pontos de latitude e longitude
- geo_distance query
    - busca pontos em torno de um círculo dado por um ponto e um raio.
- geo_shape query

Busca pontos dentro de um polígono dado por um conjunto de pontos
Cada aresta é definida um par de pontos

GEOSPATIAL DATA TYPES

- geo_point
    - representa um ponto
    - Tem uma latitude e uma longitude

Listing 12.2 Indexing a bus stop with a location defined as a string

```
POST bus_stops/_doc
{
  "name":"London Bridge Station",
  "location":"51.07, 0.08"
}
```

Inputs the location as a string with latitude and longitude values

    - Existem várias formas de definir um ponto para o Elastic
        - "location" : "POINT (51.49 0.14)"
        - "location" : {"lon":-0.12, "lat":51.50}
        - "location" : [51.54, 0.23]
        - "location" : "gcpvh2bg7sff" (geohash)
    - Suporta formato WKT
    - Dependendo do formato, longitude ou latitude podem ter que vir antes no array.
- geo_shape
    - Representa uma forma geométrica
        - pontos, multipontos, linhas, polígonos, multipolígono.

GEOSPATIAL QUERIES

- e.g. buscar o restaurante mais próximo de casa

- geo_bounding_box query:
  - Busca documentos contidos em uma dado retângulo definido em um georetângulo
- geo_distance query:
  - Busca endereços dentro de uma certa distância de um dado ponto
- geo_shape query:
  - Acha formas geométricas dentro de outra forma geométrica (e.g. fazendas dentro de um dado território)

THE GEO_BOUDING_BOX QUERY

- Busca endereços dentro de uma forma (círculo, retângulo, polígono, etc)

Listing 12.6 Matching restaurant locations in a georectangle

```
GET restaurants/_search
{
  "query": {
    "geo_bounding_box": {
      "location": {
        "top_left": {
          "lat": 52,
          "lon": 0.2
        },
        "bottom_right": {
          "lat": 49,
          "lon": 0.1
        }
      }
    }
  }
}
```

Constructs a georectangle

Sets the document's geo_point field

Defines the top_left point, formed by a latitude/longitude pair

Defines the bottom_right point, formed by a latitude/longitude pair

- Também podemos buscar por data types geoshape, não apenas pontos.
  - Basta buscar num index que contém geoshape e não geopoints, o resto é igual.

Listing 12.8 Geoquery with a location represented as WKT

```
GET restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match_all": {}
        }
      ],
      "filter": [
        {
          "geo_bounding_box": {
            "location": {
              "wkt":"BBOX(0.08, 0.04, 52.00, 49.00)"
            }
          }
        }
      ]
    }
  }
}
```

Sets the coordinates in WKT format

- Não há diferença exceto de estilo em buscar por array ou WKT ou JSON.

THE GEO_DISTANCE QUERY

- Usado para buscar lista de endereços em torno de um ponto

- Figure 12.5 Returning schools with a `geo_distance` query

Listing 12.9 Searching for restaurants within a given radius

```
GET restaurants/_search
{
  "query": {
    "geo_distance": {
      "distance": "175 km",
      "location": {
        "lat": 50.00,
        "lon": 0.10
      }
    }
  }
}
```

Declares the geo_distance query

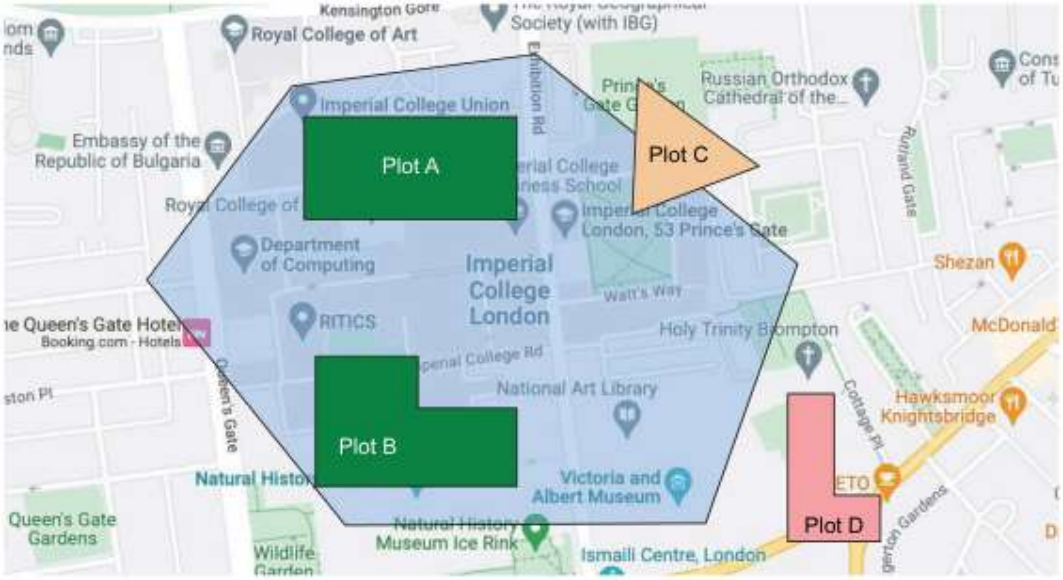
Sets the vicinity of an area to search (distance from a central point)

Sets the central location, defined as a point on the map

  - Distância pode ser em km ou milhas.

THE GEO_SHAPE QUERY

- Usado para buscar formas dentro de formas, não apenas pontos.



- **Figure 12.6 Plots of farmland in an area of London**
- Podemos retornar os documentos que, em relação à forma envelope da query:
  - Têm **qualquer ponto** em comum (insersects) -- default
  - Estão inteiramente dentro da forma da query (within)
  - Contêm a forma da query (contains)
  - Estão fora (disjoint)

THE SHAPE QUERY

- Criar formas 2D
  - Desenhos técnicos
- Podemos buscá-las com queries:
  - Por exemplo, todas as formas dentro de um dado envelope:

Listing 12.13 Searching for all the shapes in a given envelope

```
GET myshapes/_search
{
  "query": {
    "shape":{
      "myshape": {
        "shape": {
          "type":"envelope",
          "coordinates":[[10,16],[14,10]]
        }
      }
    }
  }
}
```

Specifies a shape query

Field on which the query is run

Shape we want to construct

The envelope constructed by the given coordinates

THE SPAN QUERY

- Busca de conjuntos de palavras em uma dada ordem com um dado intervalo de palavras entre elas.
  - **Regular query:** "Does this document contain 'apple' and 'pie'?" (Doesn't care if it's "apple pie", "pie made of apple", or "apple juice and blueberry pie").
  - **Span query:** "Does this document contain 'apple' immediately followed by 'pie'?" or "Does 'apple' appear within 3 words of 'pie'?"

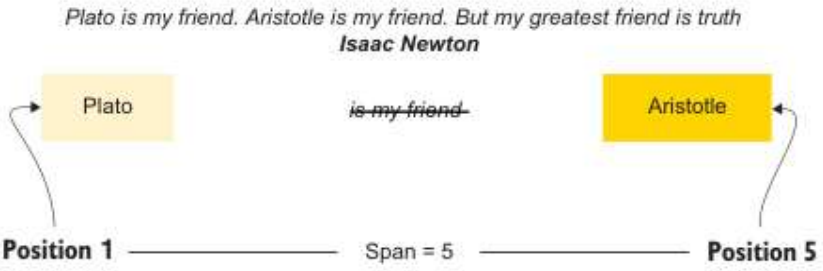


Figure 12.8 Finding a quote using a positional query

- Bom para buscar jargões técnicos específicos em textos complexos.
- Algumas variações:
  - span_first query:
    - Busca se uma palavra aparece nas n primeiras palavras de um documento
  - span_near query:
    - Busca se uma palavra aparece nas n primeiras palavras após (ou antes) uma outra palavra

Listing 12.16 Searching for documents with terms near each other

```
GET quotes/_search
{
  "query": {
    "span_near": {
      "clauses": [
        {
          "span_term": {
            "quote": "plato"
          }
        },
        {
          "span_term": {
            "quote":  "aristotle"
          }
        }
      ],
      "slop": 3,
      "in_order": true
    }
  }
}
```

The span_near query definition with a couple of clauses

The clauses consisting of two independent span_terms to search for individual words

The slop attribute gives the acceptable number of positions between the words.

The in_order attribute sets the order of the attributes.

  - span_within query:
    - Busca termos cercados por outros dois termos
  - span_or query:
    - Combina vários span_queries com o operador OR, retorna caso dê match em qqr um deles
  - Há outros span queries ainda.

SPECIALIZED QUERIES

- ES tem várias outras queries bem nichadas
- distance_feature query:
  - Dá um boost no score para um certo critério
    - e.g. se está mais próximo de uma localização ou uma data

Listing 12.21 Boosting scores of universities closer to London Bridge

```
GET universities/_search
{
  "query": {
    "distance_feature": {
      "field": "location",
      "origin": [-0.0860, 51.5048],
      "pivot": "10 km"
    }
  }
}
```

The distance_feature query declaration

The location to search for

Focal point where the distance is measured from an origin

Distance from the focal point

- pinned query:

- Fixa documentos no topo da lista independente do resultado da query

Listing 12.24 Modifying search results by adding sponsored results

```
GET iphones/_search
{
  "query": {
    "pinned":{
      "ids":["1","3"],
      "organic":{
        "match":{
          "name":"iPhone 12"
        }
      }
    }
  }
}
```

Specifies the pinned query

List of document IDs scored higher than the rest of the results

Carries out the query search

A match query that searches for the iPhone 12

- more_like_this query:

- Busca itens similares a um dado critério

Listing 12.26 Searching for More Like This documents

```
GET profiles/_search
{
  "query": {
    "more_like_this": {
      "fields": ["name", "profile"],
      "like": "Sotherby",
      "min_term_freq": 1,
      "max_query_terms": 12,
      "min_doc_freq":1
    }
  }
}
```

Defines the more_like_this query

Searches the given input in fields

Defines the query criterion

Sets the term frequency (defaults to 2)

Sets the number of terms to be selected

- percolate query:

- Procura por queries feitas para um dado documento
- Útil para buscar por queries que no passado retornaram nulo mas hoje contêm algum resultado (e.g. um produto que apareceu em estoque)
  - Inverso do usual

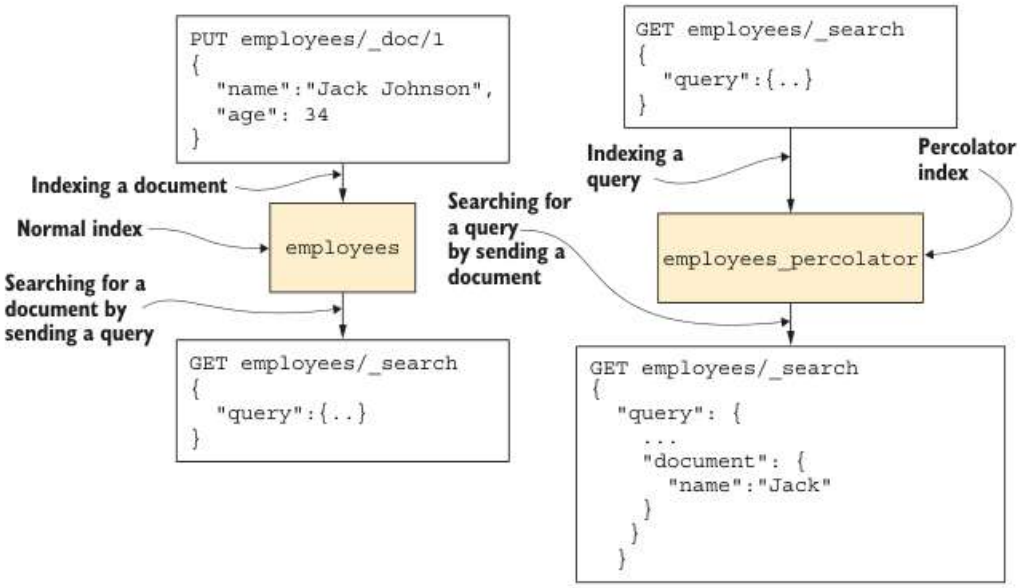


Figure 12.14 Normal vs. `percolate` queries

- indexamos queries em um dado index percolator e depois buscamos as mesmas queries

Listing 12.32 Searching for queries in the `percolator` index

```
GET tech_books_percolator/_search
{
  "query": {
    "percolate": {
      "field": "query",
      "document": {
        "name":"Python in Action",
        "tags":["Python","Software Programming"]
      }
    }
  }
}
```

Specifies the percolate query

Sets the field's name to query

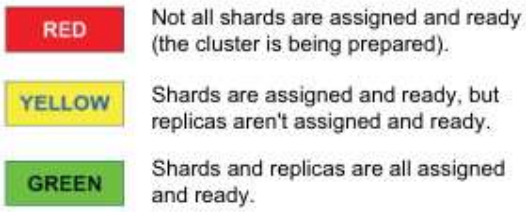
Specifies the document consisting of the original book indexed into tech_books

Summary

- Elasticsearch supports the `geo_point` and `geo_shape` data types to work with geodata.
- Geospatial queries fetch locations and addresses using coordinates formed from longitude and latitude values.
- A `geo_bounding_box` query fetches the addresses in a georectangle, which is constructed using a pair of longitude and latitude values as the top-left and bottom-right coordinates.
- A `geo_distance` query finds locations inside a circular area with a center provided as a pivot and the radius as the distance from the pivot.
- A `geo_shape` query fetches all suitable locations inside a given envelope formed by coordinates.
- A `geo_shape` query searches 2D shapes in a rectangular coordinate system (Cartesian plane).
- Span queries are advanced queries that work with lower-level positions of individual tokens or words. Elasticsearch supports several `span` queries: `span_first`, `span_within`, `span_near`, and more.
- A `distance_feature` query is a specialized query where the proximity of documents to a given focal point increases the relevance score and, thus, gives the documents higher priority.
- A `pinned` query lets us bundle additional (even unmatched) documents with the original result set, potentially creating sponsored search results.
- A `more_like_this` query finds related or similar-looking results.
- A `percolate` query lets us notify users about their negative searches at a future date.

14 - Administration

SCALING THE CLUSTER

- Possível escalar um cluster de um node (em dev) até para um grande número de nodes
- ES é resistente a falhas em nodes
- Cada node roda uma instância do ES
  - Recomendado ser um servidor exclusivo para o ES
- Réplicas só fazem sentidos em nodes diferentes
- Cluster health:
  - 

  - Explicação de erros:

Listing 14.2 Asking the cluster for a shard failure explanation

```
GET _cluster/allocation/explain
{
  "index": "chats",
  "shard": 0,
  "primary": false
}
```

```
{
  "index" : "chats",
  "shard" : 0,
  "primary" : false,
  "current_state" : "unassigned",
  "allocate_explanation" : "cannot allocate because
  ➥ allocation is not permitted to any of the nodes",
  "node_allocation_decisions" : [{
    ...
    "deciders" : [{
      "decider" : "same_shard",
       "explanation" : "a copy of this shard is already allocated to
       ➥ this node ..]"}
      ]
...
}
```

- Cada node tem um papel, que pode ser atribuido manualmente
  - master, ingest, data, ml, transform

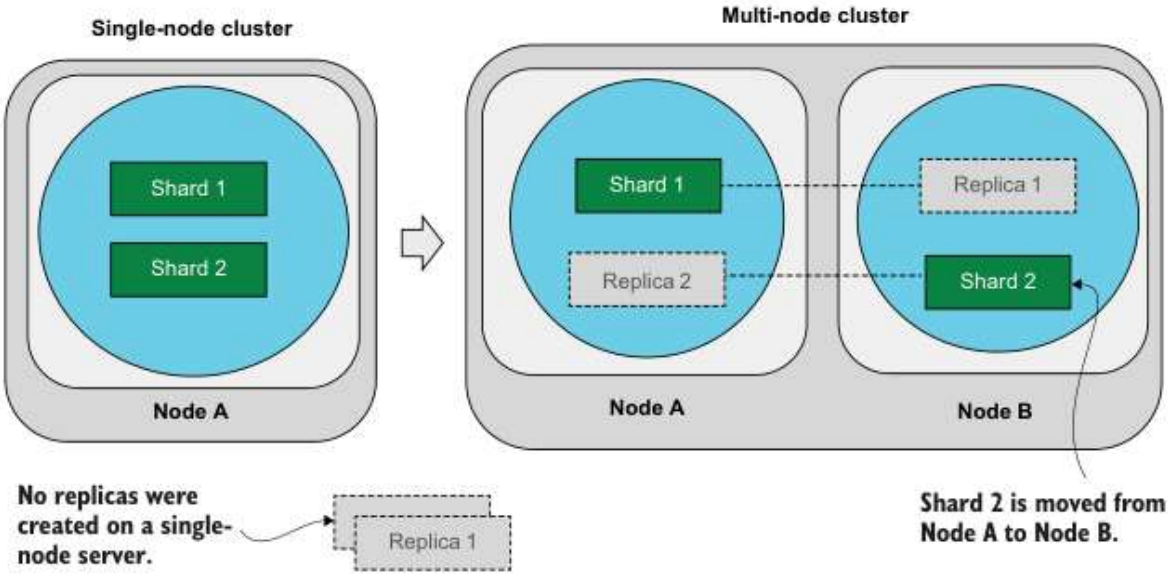


Figure 14.5 A newly joined node gets new shards moved from the single-node cluster.

- Shards são realocados automaticamente quando um node é integrado ao cluster
- Aumentar réplicas pode aumentar velocidade de leitura
  - Mas para escrita não, pois é feita nos shards
- Número de réplicas do índice pode ser alterado dinamicamente no _settings
  - Requer mais memória e cpu da máquina, tanto quanto o shard primário.
  - Número de shards não pode ser aumentado dinamicamente, requer index inoperante.

NODE COMMUNICATION

- Comunicação Node-Client é feita por HTTP(S), via API REST, na porta default 9200.
- Comunicação intranode ocorre na porta 9300 por default.
- Mudar esses default pode ser complexo continuamente num cluster grande.

SHARD SIZING

- Shards primários e réplicas demandam mesmos recursos de máquina.
  - E.g. Index com 10 shards de 30GB cada com 2 réplicas cada demandam 900GB de espaço, no mínimo.
  - Importante ainda ter memória adicional para realizar as operações.
- Se tivermos múltiplos índices igual, multiplicamos novamente os requisitos pelo número de índices.

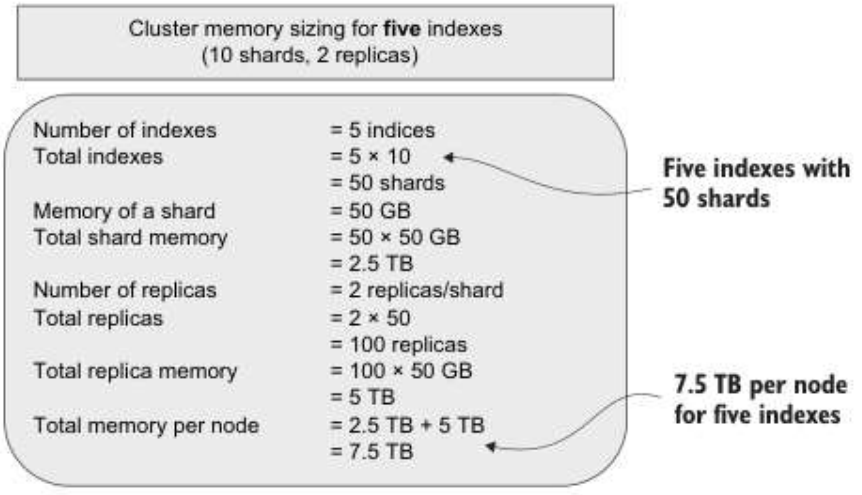


Figure 14.8 Exponential memory usage for five indexes on a node

- Scaling pode ser vertical (máquinas melhores) ou horizontal (mais máquinas)
  - Horizontal costuma ser melhor, mais fácil.

SNAPSHOTS

- Importante ter backups
  - Cloud e/ou local
  - Idealmente automatizado
- Snapshot pode ser de um index, vários ou do cluster todo
- Snapshots são incrementais
  - Além de pela API, Kibana tem uma funcionalidade para fazer snapshot também
  - Snapshots podem ter metadados que definimos na requisição
- Snapshots são úteis para fazer mudanças que não podem ser feitas dinamicamente nos index
  - Criar index novo com settings certo, fazer snapshot do index antigo e restaurá-lo sob o index recém-criado
- Com _slm (snapshot lifecycle management) podemos automatizar snapshots com algumas regras
  - e.g. tempo entre snapshots, tempo de retenção etc
  - Kibana também tem ferramentas para facilitar este SLM
  - SLM pode ser executado manulamente se quisermos, mesmo fora das regras que definimos.
- Nas versões Enterprise 7.12 em diante, snapshots são buscáveis como um index normal.
  - Pode servir como réplica para busca, poupando recursos.

ADVANCED CONFIGURATIONS

- *convention over configuration*
  - Geralmente preferível usar defaults, que funcionam bem sem ajustes
- Existem três arquivos de configuração
  - elasticsearch.yml
    - informações mais básicas e fundamentais
    - e.g. mudar portas, paths etc.
  - log4j2.properties
    - Settings de level de logging do node individual
      - Posso precisar resetar node para config valer
    - Mesma alteração pode ser feita para todo o cluster via API, mas também só se cluster estiver desligado.
      - e.g. alterar level de logs para DEBUG para resolver algum problema
        - sugerível alterar de volta ao INFO após o termino, por exemplo tendo feito o settings como 'transient' e não 'persistent', para que setting não perdure após reset do cluster
  - jvm.options
    - Ajuste de heap memory do node, assunto complexo e detalhado.
    - Sempre Criar outros arquivos auxiliares, nunca editar o arquivo original, pois pode corromper Elasticsearch
    - Nunca deixar -Xms (inicial) e Xmx (máximo) exceder 50% da memória RAM total do node.

CLUSTER MASTERS

- Node master é responsável por todas operações
  - node master na verdade é *master-eligible*, i.e. todo node master pode ser master caso o node de fato master crashe.
- Master é escolhido por eleição de todos os masters na inicialização ou quando o master morre.
  - Há alguns settings de como essa eleição ocorre (e.g. tempo)
- Só master pode alterar o cluster state, e ele que verifica se está em ordem.
  - Comunica aos outros nodes quando há alteração.
- quorum é um subconjunto dos master-eligible nodes que se comunicam com o master node para atualizações.
  - Número de master-eligible nodes mínimo é metade de todos master-eligible nodes + 1
  - Número mínimo absolute recomendado é 3

        - Torna mais difícil que surjam dois masters ao mesmo tempo caso haja problemas de conetividade entre nodes.
- Nodes podem ter múltiplos papéis ao mesmo tempo, mas a custo de desempenho.
    - Master nodes é uma boa ideia ser apenas o master node e nada mais.
    -

Summary

- Elasticsearch horizontally scales when adding new nodes to the cluster. The new nodes join the cluster as long as they are associated with the same cluster name and network settings.
- One way to improve read throughput and, thus, performance is to add replicas to the cluster. The replicas take the read hits and serve data quickly.
- Nodes communicate with each other on a transport port, set to 9300 by default. This can be modified by adjusting the `transport.port` property in the elasticsearch.yml file.
- HTTP clients communicate with Elasticsearch on an http.port (set to 9200 by default) using RESTFul endpoints.
- A node can consist of multiple indexes, and each index can consist of multiple shards. The ideal size of a shard should be not more than 50 GB of memory.
- Shards and replicas occupy space, so our organizational strategy should define the appropriate sizes based on current requirements and future use.
- Although adding replicas improves the client's read/query performance, it comes with a cost. We must be sure to carry out appropriate trials and watch for spikes before assigning a standard size for each shard.
- Elasticsearch allows backing up and restoring data using its snapshot and restore feature. Snapshots let us make backups of the cluster to a repository.
- A repository can be a local filesystem or a cloud-based object store such as AWS S3.
- A snapshot can consist of indexes, data streams (as well as the cluster state, such as persistent or transient), indexing templates, index lifecycle management (ILM) policies, and more.
- A snapshot lifecycle management (SLM) endpoint declared as `_slm` creates a policy that defines the snapshot along with its schedule and other attributes.
- We can initiate restoring data from a snapshot manually by invoking the `_restore` API.
- We can use Kibana's rich interface to define snapshots and policies and restore them. The policies are available under the Stack Management navigation menu.
- Elasticsearch exposes various settings via its elasticsearch.yml, jvm.options, and log4j2.properties files.
- We can tweak many attributes, such as changing the cluster name, moving the log and data path, adding network settings, and so on, by editing the elasticsearch.yml file.
- The config/jvm.options file defines JVM-related data for the node, but we must never edit this file.
- To customize the settings for JVM options, we need to create a new file ending with *.options (like custom_settings.options) and drop it in a folder called jvm.options.d, which should be present in the config folder for archived installations (tar or zip).
- We can set the required heap memory using the `-Xms` and `-Xmx` flags, where `-Xms` sets the maximum heap and `-Xmx` sets the minimum heap size. As a rule of thumb, never set heap size over 50% of available RAM.

- The master node in a cluster is a critical node that looks after managing and maintaining cluster-related operations and updating the distributed cluster state.
- The master node consults a quorum of nodes to commit the cluster state or elect a new master.
- A quorum is a minimum number of master-eligible nodes carefully selected by the cluster to alleviate node failures when making decisions.
- We should provide a minimum of three master-eligible nodes when forming a cluster, to avoid a split-brain cluster.

13 - Aggregations

OVERVIEW

- Metric aggregations
  - métricas tipo soma ou média
- Bucket aggregations
  - Juntar dados em intervalos
- Pipeline aggregations
  - Linka aggregations anteriores para aggregations complexos
- Usa o mesmo _search endpoint das queries comuns, mas com aggs no body
- Podemos combinar query com aggregations, gera as aggregations sobre o resultado da query.
- Podemos fazer várias aggregations ao mesmo tempo e aninhadas
- Setamos size=0 na query para não retornar os documentos em si, só as agregações.

METRIC AGGREGATIONS

- Somas, médias, mínimo, etc.
- value_count: contagem simples sobre uma variável booleana
- Aggregations não são feitas para text fields:
  - Causa erro
  - Pode ser contornado alterando o parâmetro fielddata no mapping
  - Não recomendado, desempenho ruim
- stats retorna várias estatísticas de uma vez (ou extended_stats para mais ainda)
- cardinality retorna número de valores únicos de um campo.

BUCKET AGGREGATIONS

- Agrupa documentos segundo alguma critério
- e.g. para criar histogramas
  - Definimos os bins manualmente (ranges) na query ou por intervalo (interval)
    - Podemos agregar IPs com o parâmetro "ip_range"
  - Também pode ser por intervalo de data (date_histogram)
    - Pode ser calendar-aware (calendar_interval) ou não (fixed_interval)
- Podemos ter buckets dentro de buckets (sub-aggregation):

Listing 13.13 Average metric on books categorized per year

```
GET books/_search
{
  "size":0,
  "aggs": {
    "release_date_histogram": {
      "date_histogram": {
        "field": "release_date",
        "calendar_interval": "1y"
      },
      "aggs": {
        "avg_rating_per_bucket": {
          "avg": {
            "field": "amazon_rating"
          }
        }
      }
    }
  }
}
```

The bucketing histogram that categorizes books by year

Names the sub-aggregation

Applies a single-value metric across individual buckets

- Podemos agregar itens pela presença de um dado termo em algum campo (e.g. nome de autor)
  - Ou ainda um conjunto de condições, não apenas um único termo. (multi_terms)

PARENT AND SIBLING AGGREGATIONS

- Há dois tipos de agregations: parent e sibling
- Parent aggregations:
  - aggregation sobre uma outra agregação
    - e.g. média por dia
- Sibling aggregations:
  - agregações em paralelo, todas feitas em cima do conjunto original.
    - e.g. média e histograma

PIPELINE AGGREGATIONS

- Concatenar várias agregações
- Também tem os dois tipos sibling e parent
- Parent:

```
GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "sales_by_coffee": {
      "date_histogram": {...},
      "aggs": {
        "cappuccino_sales": {
          "sum": {...}
        },
        "total_cappuccinos": {
          "cumulative_sum": {
            "buckets_path": "cappuccino_sales"
          }
        }
      }
    }
  }
}
```

The `cumulative_sum` refers to the parent aggregation (defined by `cappuccino_sales`) by setting `buckets_path` to `cappuccino_sales`

Figure 13.8 Parent pipeline aggregation `buckets_path` setting

- Sibling:

```
GET coffee_sales/_search
{...}

GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "sales_by_coffee": {
      "date_histogram": {...},
      "aggs": {
        "cappuccino_sales": {...}
      }
    },
    "highest_cappuccino_sales_bucket":{
      "max_bucket": {
        "buckets_path": "sales_by_coffee>cappuccino_sales"
      }
    }
  }
}
```

Sibling aggregations

The `max_bucket` (a sibling aggregation) refers to the constituents of sibling aggregations (defined by `sales_by_coffee` and `cappuccino_sales`) by setting `buckets_path` to `sales_by_coffee>cappuccino_sales`.

The ">" operator is the aggregation separator.

The `buckets_path` for a sibling pipeline aggregation

Figure 13.9 Sibling pipeline aggregation `buckets_path` setting

- Existem várias pipeline aggregations pré definidas
  - Algumas são sibling aggregations (e.g. sum_bucket), outras são parent aggregations (e.g. bucket_sort)
- cumulative_sum parent pipeline aggregation:
  - Retorna somas acumuladas

Listing 13.22 Cumulative sales (sum) of cappuccinos sold daily

```
GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "sales_by_coffee": {
      "date_histogram": {
        "field": "date",
        "calendar_interval": "1d"
      },
      "aggs": {
        "cappuccino_sales": {
          "sum": {
            "field": "sales.cappuccino"
          }
        },
        "total_cappuccinos": {
          "cumulative_sum": {
            "buckets_path": "cappuccino_sales"
          }
        }
      }
    }
  }
}
```

Parent aggregation that calculates the cumulative total of cappuccino sales

- max_bucket e min_bucket sibling pipeline agreggation:
  - Retorna o bucket máximo/mínimo de um conjunto de buckets
  - e.g. retorna bucket (dia) que vendeu mais cappuccinos

Listing 13.23 Pipeline aggregation to find to sales of cappuccinos

```
GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "sales_by_coffee": {
      "date_histogram": {
        "field": "date",
        "calendar_interval": "1d"
      },
      "aggs": {
        "cappuccino_sales": {
          "sum": {
            "field": "sales.cappuccino"
          }
        }
      }
    },
    "highest_cappuccino_sales_bucket":{
      "max_bucket": {
        "buckets_path": "sales_by_coffee>cappuccino_sales"
      }
    }
  }
}
```

Summary

- Whereas a search finds answers in the amassed data based on a search criterion, an aggregation compiles patterns, insights, and information for data collected by organizations.
- Elasticsearch allows us to perform nested and sibling aggregations on data.
- Elasticsearch classifies aggregations into three types: metrics, buckets, and pipelines.
- Metric aggregations fetch single-value metrics such as `avg`, `min` and `max`, `sum`, and so on.
- Bucket aggregations classify data into various buckets based on a bucketing strategy. With a bucketing strategy, we can ask Elasticsearch to split data into buckets as needed.
- We can either let Elasticsearch create predefined buckets based on the interval we provide or create custom ranges:
  - If the interval is 10 for an age group, for example, Elasticsearch splits data into steps of 10.
  - If we want to create a range like 10 to 30 or 30 to 100, where the interval differs, we can create a custom range.
- Pipeline aggregations work on the output from other metric and bucket aggregations to create new aggregations or new buckets.

15 - Performance and Troubleshooting

SEARCH AND SPEED PROBLEMS

- Pode ter múltiplas causas
    - Hardware:
        - Alocar SSD e Memória abundantes ajudam a evitar gargalos no Lucene subjacente
        - Memória Heap armazena objetos novos
            - Mais Heap evita que se encha e requer menos garbage collection, dá tempo para dados serem movidos à outra memória.
            - Recomendado é metade da memória do node vai à aplicação como heap.
        - Mais shards pequenos aumenta necessidade de comunicação entre os nodes
            - Pode gerar gargalo de rede.
        - Mais réplicas melhora desempenho de read, mas consome memória e pode gargalar e ser difícil de gerir.
        - Tamanho de shard deve ser equilibrado
    - Document Modeling:
        - ES é NoSQL
            - Dados devem ser denormalizados (i.e. sem joins)
            - Melhor não haver hierarquia entre dados, cada documento é autossuficiente.
        - Também é bom evitar buscar em muitos campos ao mesmo tempo, operação cara.
            - Melhor combinar vários campos em um só e buscar nesse, se for o caso.
                - Podemos usar o copy_to no mapping do index para isso.
        - Usar Keyword type no lugar ao junto de text type para busca de termos exatos pode melhorar desempenho
            - Poupa etapa de análise do texto

INDEX SPEED PROBLEMS

- Ids aleatórios gerados pelo elastic poupam trabalho de checar unicidade do id.
    - Mas sujeito a duplicação
- Request bulk é bem melhor que um documento por vez
    - Mas tem um limite de tamanho até ser prejudicial.
    - Aumentar refresh rate pode melhorar isso, usa melhor o disco.
- Ajuste da refresh rate
    - Taxa com que elastic verifica se existe algum documento novo e o indexa.
    - Refresh pode ser desabilitado por completo para inserir vários documentos de uma vez só.
    - _refresh pela API para refresh manual.
    - Ajuste do thread_pool para o write ou para o search também pode influir.

UNSTABLE CLUSTER

- Estado RED é uma emergência.
    - Múltiplas causas (Rede, hardware, arquivos corrompidos etc)
- Se o ping do master não for bem sucedido, nova eleição do master é feita imediatamente.
- Pode haver shards não alocados de um index existente.
    - Geralmente durante a etapa de rebalanceamento e/ou após falha de algum node
    - Read e Write são bloqueados até que o shard seja alocado corretamente.
    - API de _explain explica por que um shard não pode ser alocado
- Limites de uso de disco
    - Por default, num node o low-disk watermark acontece com 85% do disco utilizado.
        - ES não aloca novos shards a este node, neste caso
        - Volta a poder alocar quando disco volta para abaixo deste nível
    - High-disk watermark
        - Idem para 90% do disco utilizado.

        - Shards neste node são realocados para outros nodes
    - Flood-stage-disk watermark
        - 95% do disco utilizado.
        - Todos os index nesse node se tornam não-writable, viram read-only
            - Volta ao normal quando disco é liberado

CIRCUIT BREAKERS

- Alguma função que é ativada quando ocorre algum problema na cadeia de chamadas.
    - e.g. disparos de funções que evitam falha de node quando há um aumento no uso de memória.
- Erro de Out of Memory antes que a operação seja realizada.

Table 15.1 Types of circuit breakers and their memory settings

Circuit breaker	Description and minimum memory limit	Property
Parent	The total memory that can be used across all the other circuit breakers. Defaults to 70% of JVM heap memory if real-time memory is taken into account (`indices.breaker.total.use_real_memory=true`); otherwise, it is 95% of JVM heap memory.	`indices.breaker.total.limit`
Inflight requests	The sum total of memory of all inflight requests, which is not to be exceeded by a threshold. The default is 100% of JVM heap memory, although it derives the actual percentage from the parent circuit breaker.	`network.breaker.inflight_requests.limit`
Request	Helps prevent exceeding the heap memory to serve a single request. The default is 60% of the JVM heap.	`indices.breaker.request.limit`
Field data	Helps prevent exceeding memory when loading fields into the `fields` cache. The default limit is 40% of heap memory.	`indices.breaker.fielddata.limit`
Accounting requests	Avoids accumulating memory after the request is served. The default is 100%, which means it inherits the threshold from the parent.	`indices.breaker.accounting.limit`
Script compilation	While all other circuit breakers look after memory, this circuit breaker limits the number of inline compilations in a fixed time period. The default is 150/5 min (150 script compilations in a period of 5 minutes).	`script.max_compilations_rate`

-

FINAL WORDS

- Assunto de troubleshooting é muito extenso
    - Este capítulo é apenas um panorama raso.

- Importante ter prática e ler documentação

Summary

- Elasticsearch is a complex search engine, and maintaining and managing healthy clusters requires expertise.
- Slower search speeds are a common problem that clients complain about when searching for data with Elasticsearch. Provisioning modern hardware with appropriate allocation of memory and computing resources helps alleviate speed problems. Options like choosing keyword data types and tweaking the document data model also help increase search speed.
- Indexing speeds are a concern as well, especially when the system is ingestion heavy. If allowed, measures like using system-generated IDs for documents can speed up the indexing process. Loading data using bulk APIs (rather than single-document APIs) is a surefire way to improve ingestion performance.
- We can turn off or increase the refresh rate during indexing and then switch it on when indexing completes. Doing so makes the documents unavailable instantly for searches but improves indexing performance by reducing I/O hits.
- Due to the distributed nature of Elasticsearch, things can go wrong in many areas, from the cluster to the filesystem, node communication, memory, and so on. Keeping a healthy cluster is paramount, and administrators must strive to maintain a GREEN cluster state. Religiously following the traffic light management-based cluster health system helps with smooth running of the cluster.
- Occasionally, shards are left unassigned, and troubleshooting to find the exact cause helps us reattach nodes if necessary or take the appropriate action.
- Elasticsearch provides thresholds on the available disk memory, including low-, high-, and flood-stage-disk watermarks. The low- and high-disk watermarks allow Elasticsearch to manage shard reallocation and alert the administrator about upcoming cluster problems. The flood-stage watermark is a dire warning about an out-of-disk memory problem on the node. To keep the cluster alive when this happens, Elasticsearch makes all shards read-only and does not allow any indexing operations on the shards of that node.
- Elasticsearch is a memory-hungry application, and appropriate controls should be implemented so that errors are instantly propagated to the client. It uses circuit-breaker patterns to create circuit breakers so the client doesn't need to wait longer than necessary. Each circuit breaker triggers Elasticsearch if the memory for a certain action grabs more memory than expected. A child circuit breaker inherits the memory thresholds from a parent circuit breaker.

2 - Getting Started

> [!caution] This page contained a drawing which was not converted.

```
GET books/_doc/1
```

Interação com REST direto ou Kibana (mais amigável)

“ch02_getting_started.txt” could not be found.

“docker-compose.yml” could not be found.

- To retrieve multiple documents, if the document identifiers are available, we can use an `ids` query.
- Elasticsearch exposes a wide set of search APIs, including basic and advanced queries.
- Full-text queries search through unstructured data to find relevant documents.
- Term queries search through structured data like numbers and dates to find documents that match.
- Compound queries allow us to compile leaf queries and create a more advanced set of queries. The `bool` query, one of the compound queries, provides a mechanism to create an advanced query with multiple clauses (for example, `must`, `must_not`, `should`, and `filter`). These clauses help us create sophisticated queries.
- Whereas a search looks for matching documents based on given criteria, analytics lets us aggregate data by providing statistical functions.
- Metric aggregations fetch common aggregations like `max`, `min`, `sum`, and `avg`.
- Bucketing aggregations segregate documents into various groups (buckets) based on certain criteria.

Summary

- Elasticsearch provides a set of document APIs for indexing data, and Kibana’s Dev Tools console helps with writing indexing queries for persistence.
- To retrieve a document using a single document API, we issue a `GET` command on the index with an ID (`GET <index_name>/_doc/<ID>`).

" * " para definir intervalo iniciando/finalizando em infinito/-infinito

Bucket_aggregation gera estatísticas para grupos (e.g histograma, i.e. contagem por bin)
Para definir bucket por bucket em vez de apenas um intervalo fixo, é possível realizar "*range bucketing*".

Operação retorna todos os documentos analisados, mas pode ser suprimido com size=0.

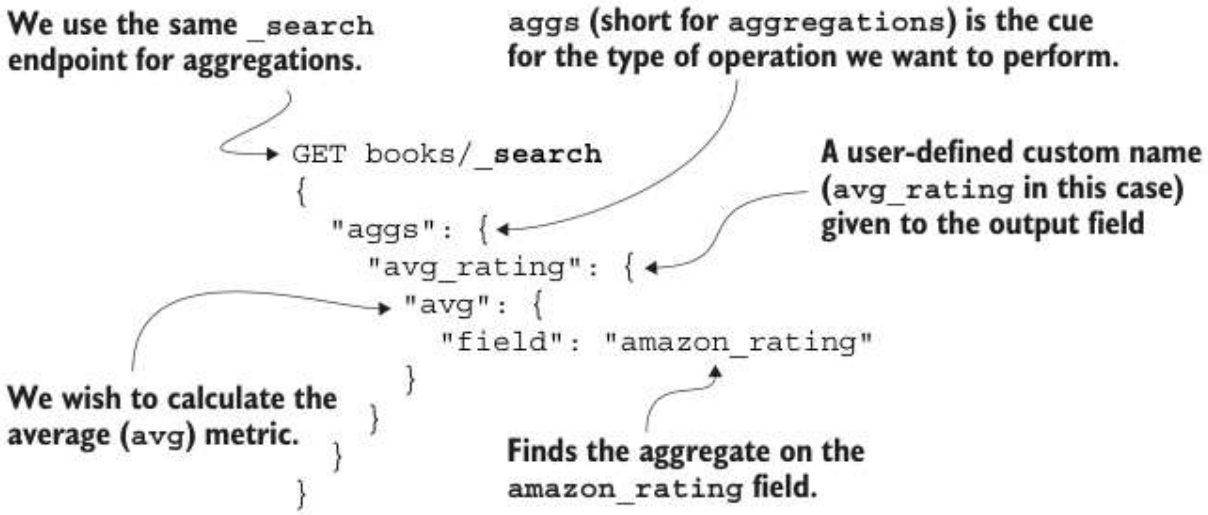


Figure 2.16 Query DSL syntax for finding an average rating aggregation

Objeto *"aggs"*

- *max*
- *min*
- *sum*
- *avg*
- *all_stats*
- *extended_stats, etc...*

Agregações:
Estatísticas, recortes, resultados de outras agregações.

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [{"match": {"author": "Joshua"}}],
      "must_not":[{"range":{"amazon_rating":{"lt":4.7}}}],
      "should": [{"match": {"tags": "Software"}}],
      "filter":[
        {"range":{"release_date":{"gte": "2015-01-01"}}},
        {"term": {"edition": 3}}
      ]}
  }
}
```


MUST e **MUST_NOT** são obrigatórios para o match ser retornado.
Desses dois apenas **MUST** altera o score.
SHOULD apenas melhora o score, ajusta relevância sem alterar o conjunto retornado.
FILTER remove os não-matchs, não altera scores (é o que difere do MUST).

Listing 2.18 Format of a `bool` query with the expected clauses

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [{ }],
      "must_not": [{ }],
      "should": [{ }],
      "filter": [{ }]
    }
  }
}
```

A bool query is a combination of conditional boolean clauses.

The criteria must match the documents.

The criteria must not match (no score contribution).

The query should match.

The query must match (no score contribution).

- Boolean (`bool`)
- Constant score (`constant_score`)
- Function score (`function_score`)
- Boosting (`boosting`)
- Disjunction max (`dis_max`)

Compostas:
Queries com várias requisições de match simultaneamente (cada uma é uma "leaf query").
Vários tipos, uma delas e a mais comum é a *bool:*

Range:
Igual term query mas permite ranges de valores, greater than, lower than, etc.

Listing 2.16 Fetching third edition books

```
GET books/_search
{
  "_source": ["title","edition"],
  "query": {
    "term": {
      "edition": {
        "value": 3
      }
    }
  }
}
```

We return just two fields in the response document.

Declares the query as a term-level query

Provides the field and value as search criteria

Termo:
Busca em dados estruturados.
Data types são inferidos pela primeira indexação. Há campos de não-texto.
Resultados são binários, há match ou não, sem fuzziness ou operações textuais, logo não há score de relevância (1.0 ou 0).

"match_phrase" procura por frase inteira, exatamente. "slop" pode tornar busca mais flexível, número inteiro de palavras faltantes / desordenadas.

Parâmetro **"highlight"** pode destacar match encontrado com tags .
"fuzziness" é um número inteiro que permite matches a esta distância de Levenshtein (e.g. trocas de caracteres)

```
GET books/_search
{
  "query": {
    "multi_match": {
      "query": "Java",
      "fields": ["title^3","synopsis"]
    }
  }
}
```

Searches through multiple fields

A caret followed by the boost number indicates the field being boosted.

"multi_match" para procurar em vários campos ao mesmo tempo.
Posso usar "boost" para priorizar matches em um campo em vez do outro. Sintaxe "^3" ou "^2"

Texto:

"_bulk" serve para indexar vários documentos de uma vez.
Sobreescreve.

Se tiver mais de uma palavra, presume-se "OR"
Posso definir como "AND" manualmente na query para match exato.

"prefix" no lugar do "match" funciona também, pega pelo início.

```
"author":"JoShUa"
"author":"joshua" (or "JOSHUA")
"author":"Bloch"
```

All these queries will succeed, but searching for a shortened name will fail:

```
"author":"josh"
```

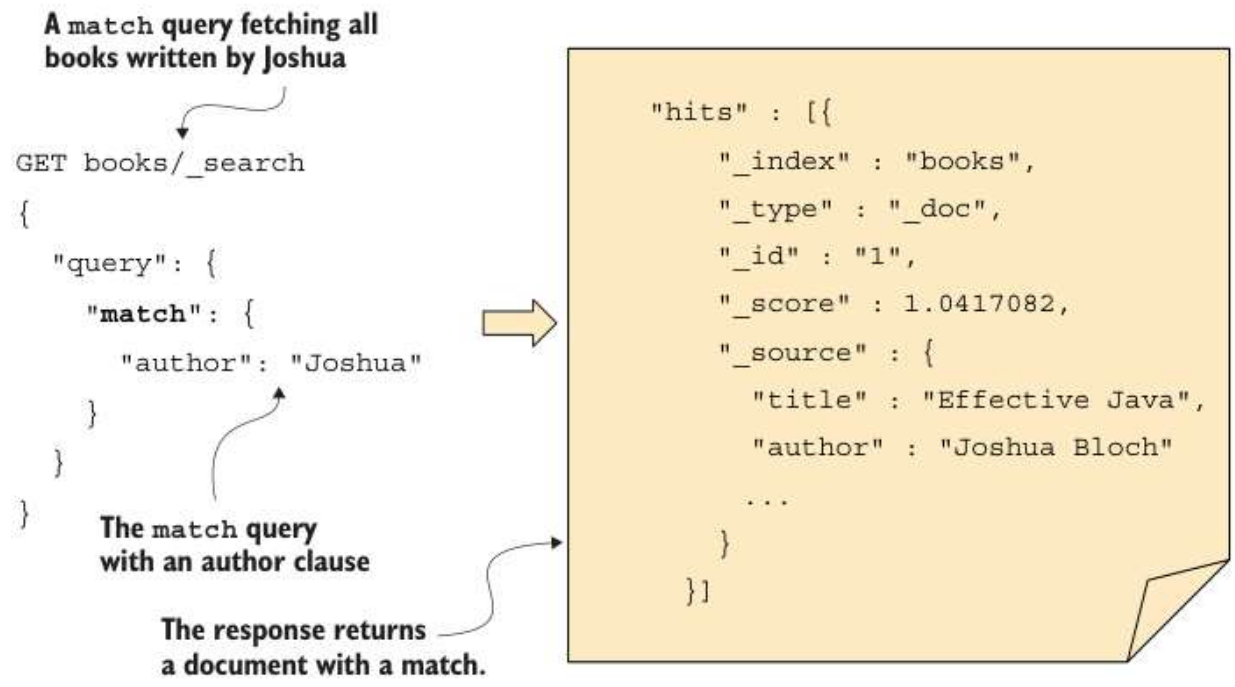


Figure 2.13 Fetching books authored by Joshua

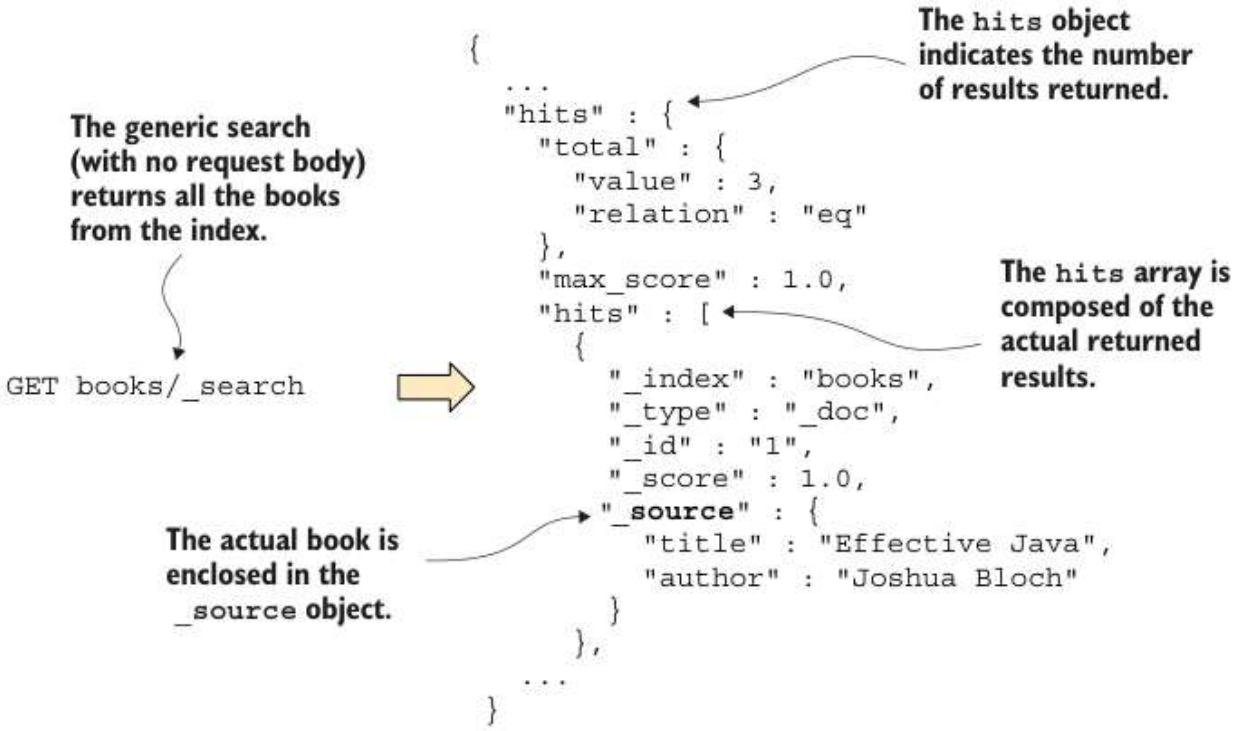


Figure 2.12 Retrieving all documents using the search API

(é igual a:)

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

GET books/_search (todos)

```
GET books/_search
{
"query": {
"ids": {
"values": [1,2,3,4]
}
}
}
```

Também pode ser um, vários ou todos OBJ

Get:

Contagem:
GET books/_count
GET books1,books2/_count (múltiplos índices de uma vez)
GET _count (todos)

Não há definição de schema a priori. Schema é derivado a partir da primeira entrada.

Indexar um documento é equivalente a inserir uma entrada num banco de dados relacional.

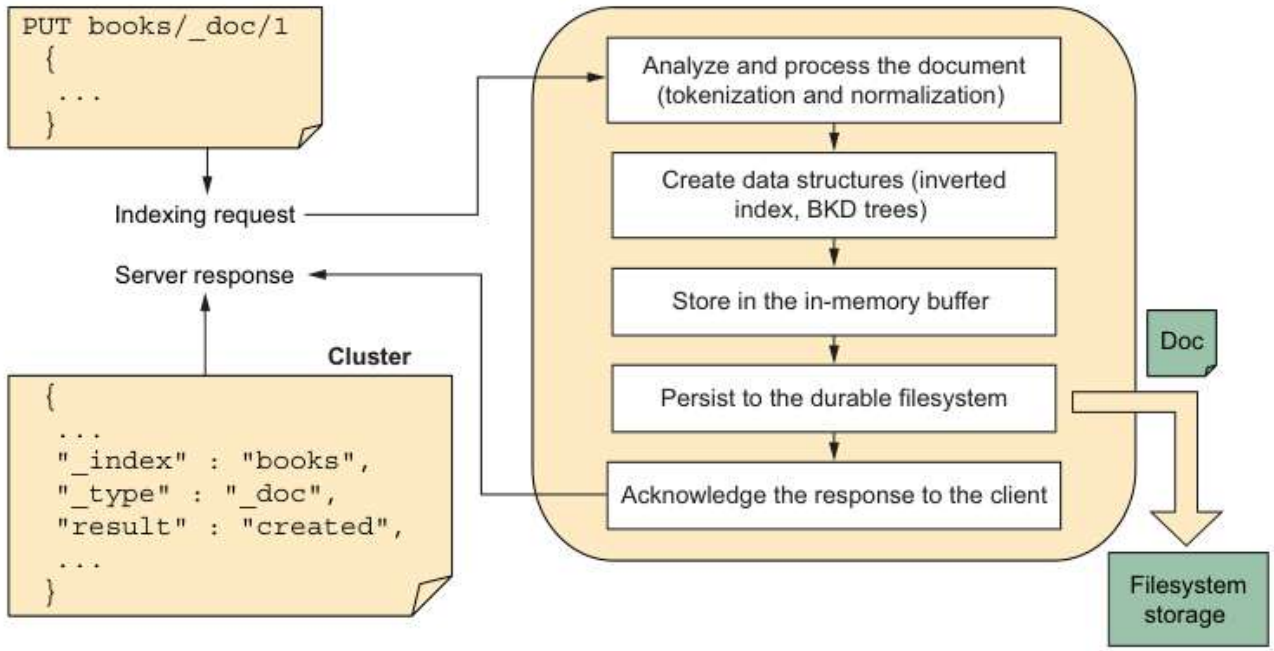


Figure 2.8 Elasticsearch's request and response flow at a high level

Kibana permite pular o boilerplate da requisição, usar só o corpo.
Devtools no console permite testasr comandos.
HTTP code da resposta diz se indexação funcionou certo (200 = OK)

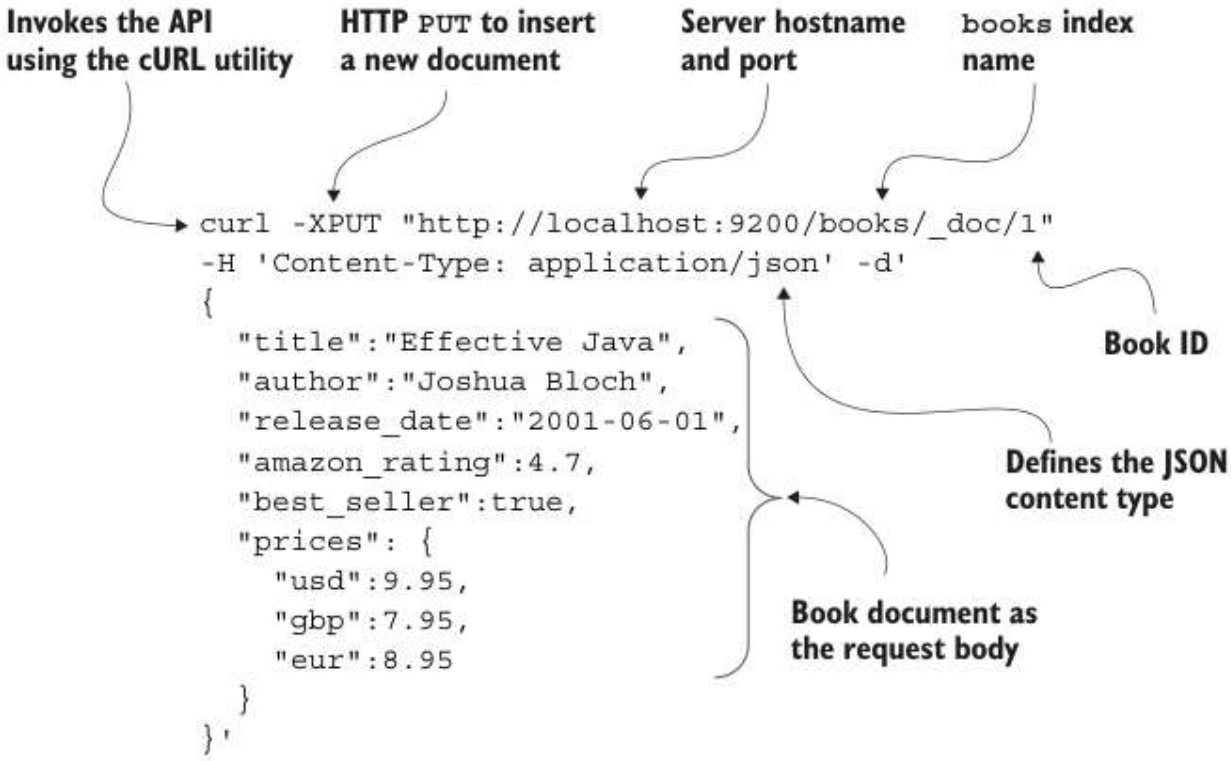


Figure 2.3 Elasticsearch URL invocation endpoint using cURL

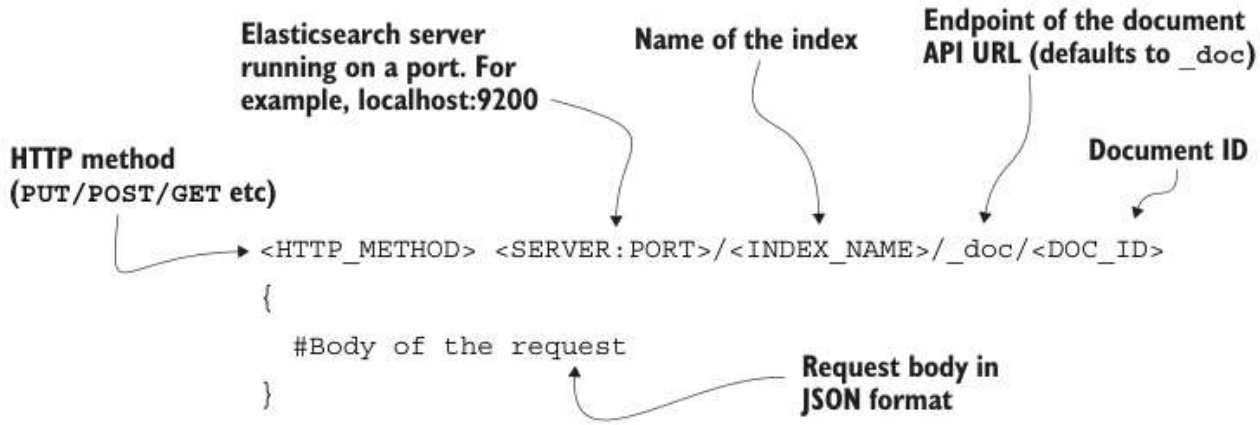


Figure 2.2 Elasticsearch URL invocation endpoint using an HTTP method

Versões anteriores incluíam campo de "type" depois do index_name, mas não mais desde 7.x. Agora só um único tipo por index.

Formato cURL:

Exemplo: livros

```
# Create an index
PUT books/_doc/1
{
  "title":"Effective Java",
  "author":"Joshua Bloch",
  "release_date":"2001-06-01",
  "amazon_rating":4.7,
  "best_seller":true,
  "prices": {
    "usd":9.95,
    "gbp":7.95,
    "eur":8.95
  }
}
```

3 - Architecture

> [!caution] This page contained a drawing which was not converted.

Baseado em Apache Lucene, que é um buscador/indexador de texto mas sem muitos features.
ElasticSearch envelopa isso de um jeito melhor com mais recursos.

DATA IN

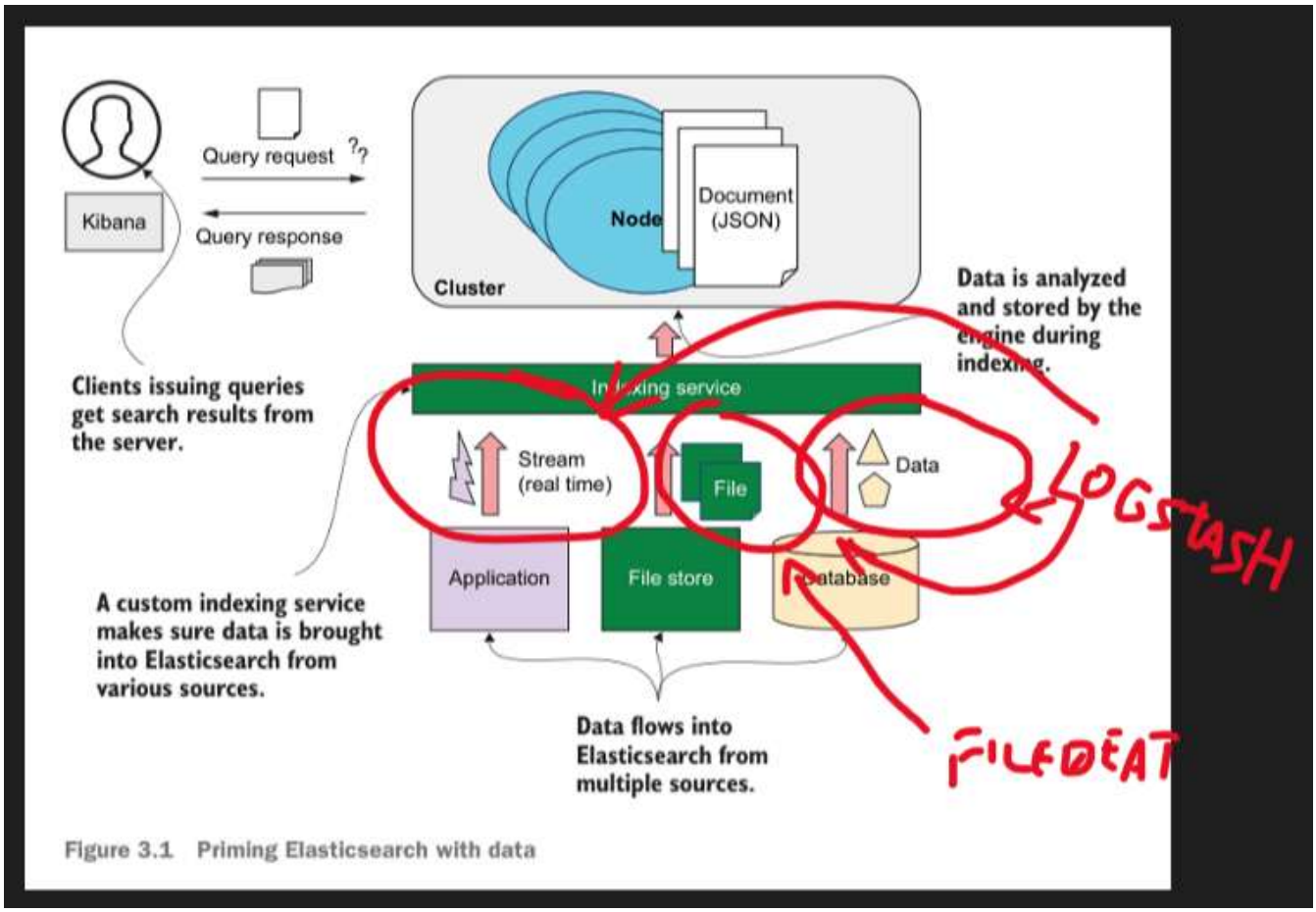


Figure 3.1 Priming Elasticsearch with data

Dado é separado por node e por tipo, no armazenamento.
Index é um bucket, uma coleção lógica. Como uma tabela num banco de dados relacional.
Um *shard* é uma instância do Apache Lucene.

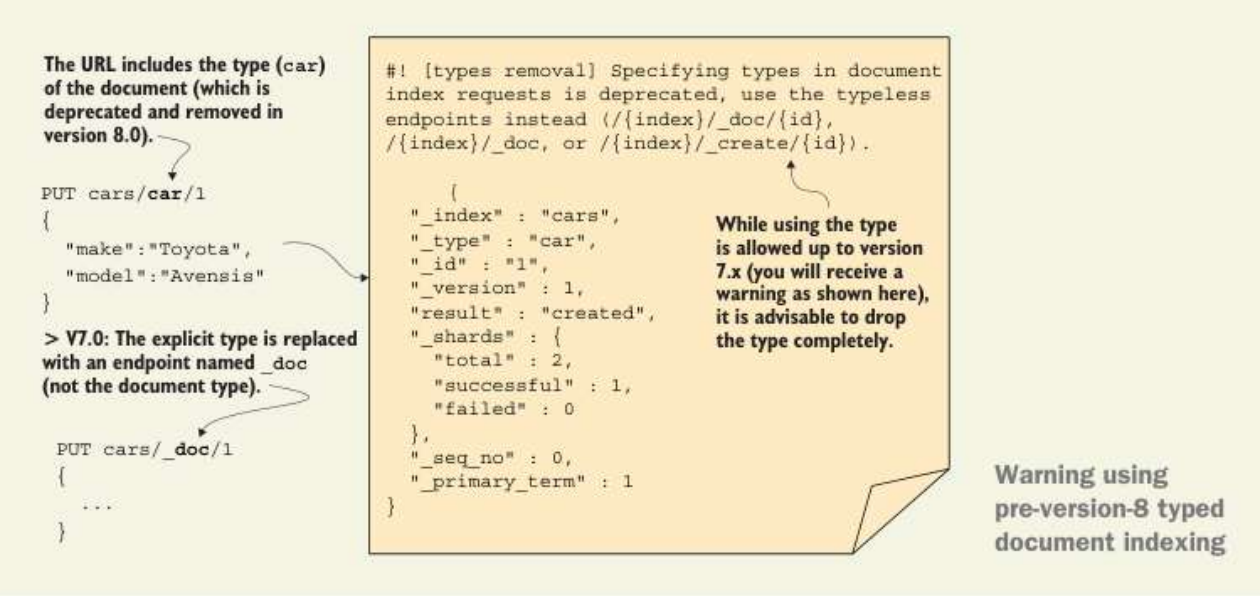
- Mapping atribui cada campo do JSON na ingestão a um data type apropriado (e.g. numeric).
- Texto também passa por isso, mas com uma etapa adicional de análise do texto
- Isso tem duas etapas: **tokenização** (é desmontado em tokens) e **normalização**.
- Há várias formas de tokenização, por exemplo, por espaço em branco.
- Tokens são persistidos sem transformações.
- Na normalização, tokens são enriquecidos, e.g. stemming, sinônimos.
- Texto analisado é persistido em um node.

DATA OUT

- Busca e analytics dos dados.
- Se a query for de texto, texto da query é analisado (i.e. tokenizado e normalizado) de forma igual com o processo na etapa de *data in*.
- Tokens resultantes disso é que são procurados no índice.

- **Document** é a unidade de dado armazenada em um index. Formato JSON.

- Esse JSON é lido e cada valor é armazenado por tipo e index específico.
- Permite campos aninhados.
- O documento inteiro é armazenado em um único ID, sem joins.
- Há APIs que operam sobre vários documentos e outras sobre um único documento.
- "*Document type*" foi deprecado na versão 8.x - um único índice podia ter vários tipos diferentes de documentos.
- Foi substituído pelo document type padrão "_doc"



Index

Equivale a um bucket, uma coleção de dados.

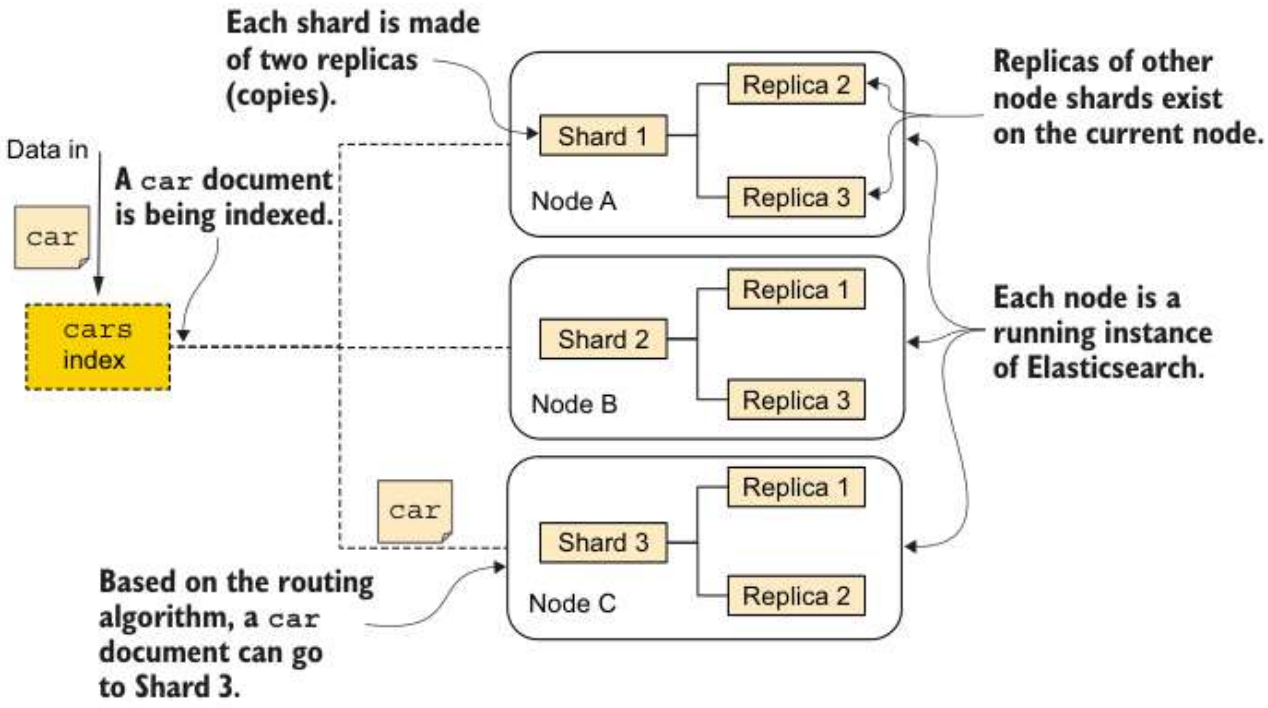


Figure 3.5 An index designed with three shards and two replicas per shard

- Shard é uma instância do Apache Lucene rodando.
- Default a partir da versão 7.x é 1 shard e 1 réplica.
- Shards podem ser *primário* ou *réplicas* - recomendável nunca ter zero réplicas.
- Um index pode existir num node só ou estar distribuído.
- Quantidade de documentos por index é uma variável importante de otimização.
- Alias é um nome alternativo para um índice ou conjunto de índices.
- Número de réplicas pode ser alterado dinamicamente, numa instância operante, mas número de shards não.

- Adicionar nós pode aliviar excesso de dados, ao distribuir mais os shards.
- E.g. para coletar dados separados em vários dias (série temporal), cada coleção-dia um em um índice, podemos fazer uma query num único índice-mãe, criando um alias de agrega todos os índices desejados.

Data stream. -

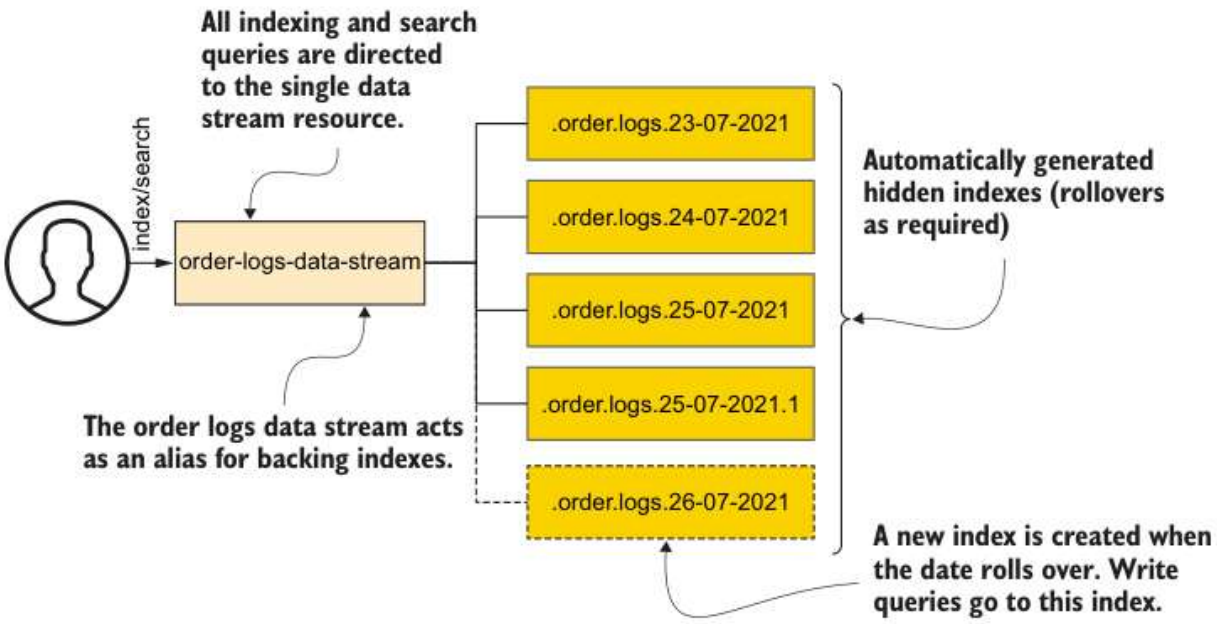


Figure 3.7 A data stream consists of automatically generated hidden indexes.

Shards e Replicas:

Mecanismo sofisticado de salvamento quando indexamos um documento.

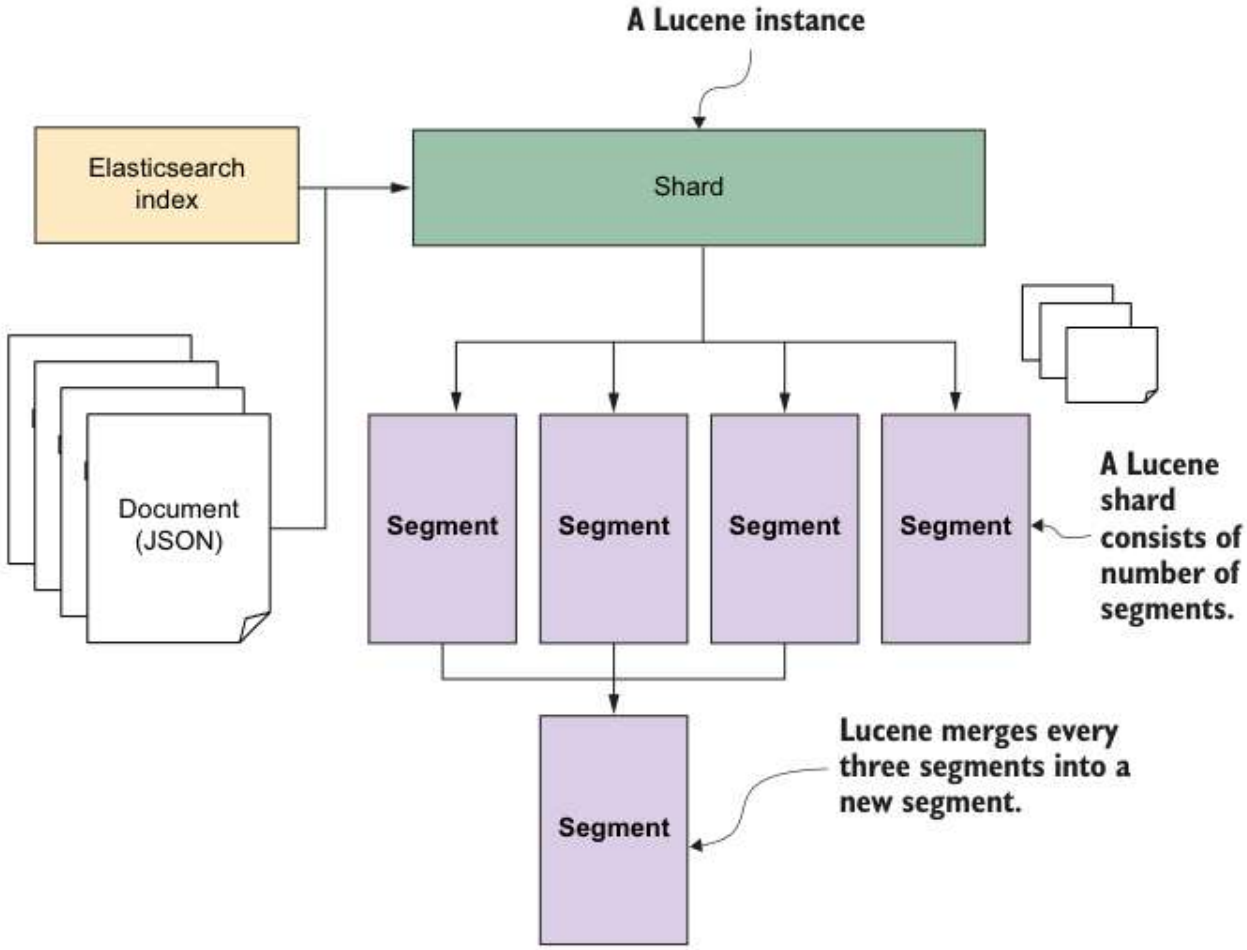


Figure 3.8 Lucene's mechanism for indexing documents

- Réplicas permitem distribuir as requisições de leitura

- Réplicas existem em nodes diferentes, por definição de redundância.
- Um node é uma instância do ElasticSearch
- Um cluster é uma coleção de Nodes
- Cluster tem 3 estados, obtidos por "GET _cluster/health" ou "GET localhost:9200/_cat/health" :
    - Red (algum shard está indisponível)
    - Yellow (Todos os shards estão disponíveis, mas algumas réplicas não)
    - Green (Todos os shards e réplicas em ordem)

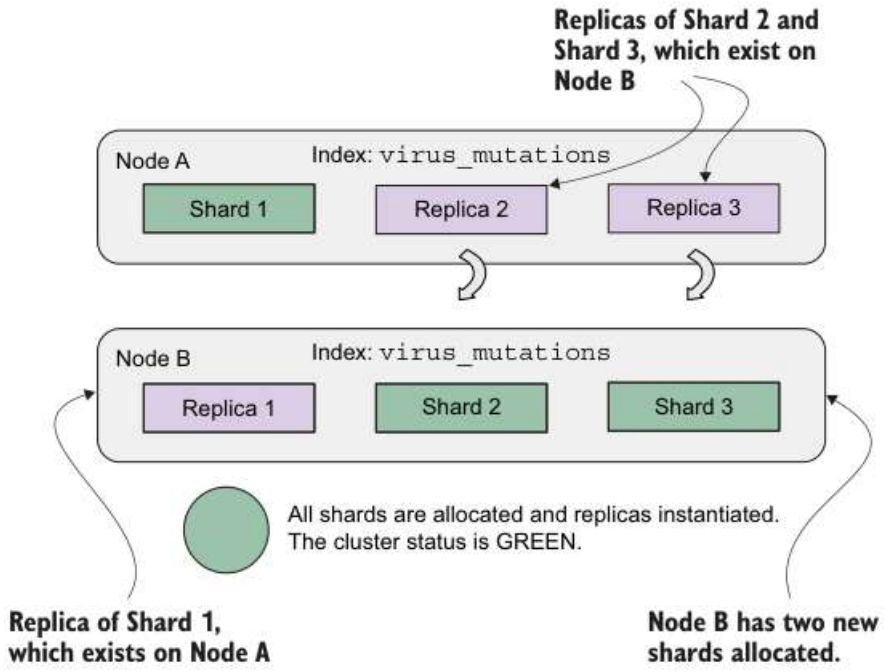


Figure 3.12 Happy days! All shards and replicas are allocated.

- Réplica pode ser promovida a shard primário se algum node crashar
- Uma questão importante é tamanho de shard, não existe receita de bolo.
- Por padrão, não se recomenda passar de 50GB, embora haja exemplos com mais do que isso.
- Se passar, deve ser distribuído em vários shards.
- Em relação à memória, recomenda-se hospedar até 20 shards por GB de heap memory.
- Default de memory de uma instância ES é 1GB, que é customizável.
- Número de shards não é alterável em um instância operante, corromperia os documentos.
- Solução para manipular shards em operação é **reindexação**

Nodes e Clusters:

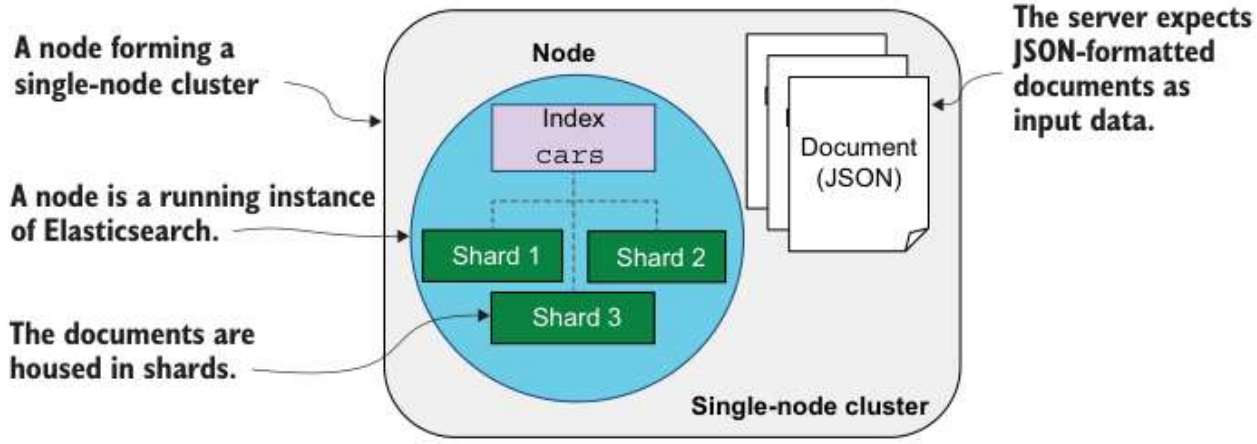


Figure 3.14 A single-node Elasticsearch cluster

- Não se recomenda ter vários nodes na mesma máquina, pois não haveria réplicas, portanto sem backup, YELLOW.
- Arquivo *elasticsearch.yml* define as configurações do cluster, como o nome do cluster onde os nodes seguintes serão agregados.
- Mais nodes <--> mais backups <--> mais desempenho.
- Mais nodes não altera o número de shards, que é fixo, apenas réplicas (até haver reindexação)
- Também é possível criar um **multicluster**. Recomendável separar em clusters diferentes para organizar e assegurar dados.
- A cada node é atribuído um conjunto de responsabilidades (e.g. gerenciar comunicação internodes, indexar dados, machine learning etc) - Um node pode mudar de responsabilidades se algum outro crashar
- Data node pode ter variantes como *data_hot*, *data_cold* etc
- Role de cada node pode ser definido no .yml

Inverted Index:

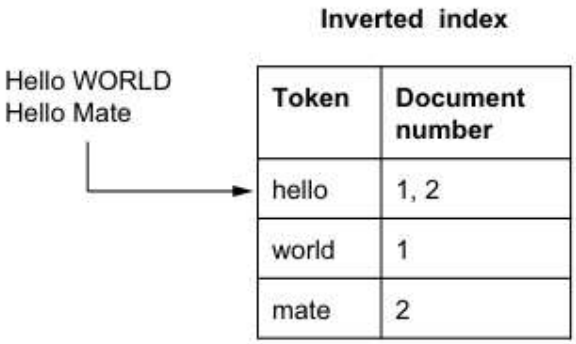


Figure 3.17 An inverted index data structure

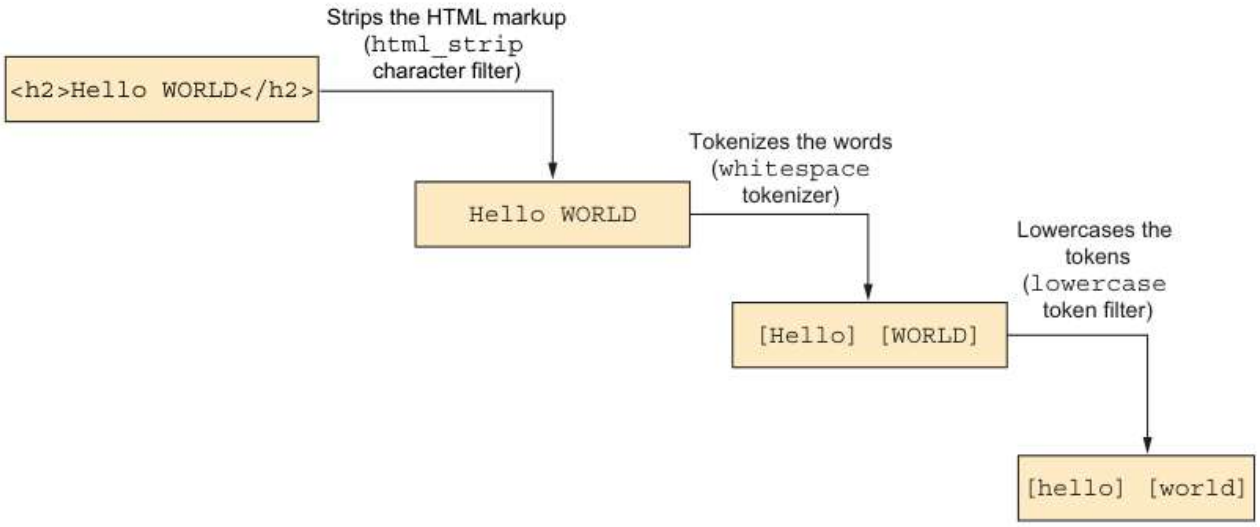


Figure 3.18 Text analysis procedure where Elasticsearch processes text

- Inverted Index também tem um campo "Frequency" que diz quantas vezes um token aparece, afeta scoring.

Relevância:

- ES usa BM25 para atribuir score de relevância por default, que é uma versão melhorada do TF/IDF.

- Podemos customizar uma função de *similarity* pra cada campo

Table 3.5 Elasticsearch's similarity algorithms

Similarity algorithm	Type	Description
Okapi BM25 (default)	`BM25`	An enhanced TF/IDF algorithm that considers field length in addition to term and document frequencies
Divergence from Randomness (DFR)	`DFR`	Uses the DFR framework developed by its authors, Amati and Rijsbergen, which aims to improve search relevance by measuring the divergence between the actual term and an expected random distribution. Terms that occur more often in relevant documents than in a random distribution are assigned higher weights when ranking search results.
Divergence from Independence (DFI)	`DFI`	A specific model of the DFR family that measures the divergence of the actual term frequency distribution from an independent distribution. DFI aims to assign higher scores to documents by comparing the observed term frequencies with those expected in a random, uncorrelated term frequency.
LM Dirichlet	`LMDirichlet`	Calculates the relevance of documents based on the probability of generating the query terms from the document's language model
LM Jelinek-Mercer	`LMJelinek-Mercer`	Provides improved search result relevance compared to models that do not account for data sparsity
Manua		Creates a manual script
Boolean similarity	`boolean`	Does not consider ranking factors unless the query criteria are satisfied

- No Okapi BM25, frequência no documento do termo procurado aumenta a pontuação. Também premia palavras menos comuns no index.
- Normaliza score para o tamanho do documento, evita viés em favor de documentos longos.
- Podemos settar stopwords para serem ignoradas na busca.

Routing Algorithm:

- Atribui um documento a dado shard, pela seguinte função: -

```
shard_number = hash(id) % number_of_shards
```

- Por isso não podemos mudar número de shards, bagunçaria esta função, documentos seriam inacháveis.

- Escalagem vertical (aumenta a máquina) requer desligar o cluster, se não tivermos modo de evitar o downtime

- Escalagem horizontal (colocar mais máquinas) apenas agrega mais nodes, mais prático e recomendável.

Summary

- Elasticsearch expects data to be brought in to be indexed. Data sources can include simple files, databases, live streams, Twitter, and so on.
- In the indexing process, data undergoes a rigorous analysis phase, during which advanced data structures like inverted indexes are created.
- Data is retrieved or searched via the search APIs (along with the document APIs for single document retrieval).
- Incoming data must be wrapped up in a JSON document. Because the JSON document is the fundamental data-holding entity, it is persisted to shards and replicas.
- Shards and replicas are Apache Lucene instances whose responsibility is to persist, retrieve, and distribute documents.
- When we start up the Elasticsearch application, it boots up as a one-node, single-cluster application. Adding nodes expands the cluster, making it a multi-node cluster.
- For faster information retrieval and data persistence, Elasticsearch provides advanced data structures like inverted indexes for structural data (such as textual information) and BKD trees for nonstructural data (such as dates and numbers).
- Relevancy is a positive floating-point score attached to retrieved document results. It defines how well the document matches the search criteria.

- Elasticsearch uses the Okapi Best Match (BM) 25 relevancy or similarity algorithm, an enhanced version of the term frequency/inverse document frequency similarity algorithm.
- You can scale Elasticsearch up or out, based on your requirements and available resources. Scaling up beefs up existing machines (adding additional memory, CPU, RAM, etc.), while scaling out spins up more virtual machines (VMs), allowing them to join the cluster and share the load.

Table 3.2 Node roles and responsibilities

Role	Description
Master node	Its primary responsibility is cluster management.
Data node	Responsible for document persistence and retrieval.
Ingest node	Responsible for the transformation of data via pipeline ingestion before indexing.
Machine learning node	Handles machine learning jobs and requests.
Transform node	Handles transformation requests.
Coordination node	This is the default role. It takes care of incoming client requests.

4 - Mapping

- Schema é inferido pelo ElasticSearch
    - Mas é melhor declarar explicitamente antes
- Mapping é o que cria a definição do schema
    - Um mapping por index
        - (Um field pode ser multi-tipo)

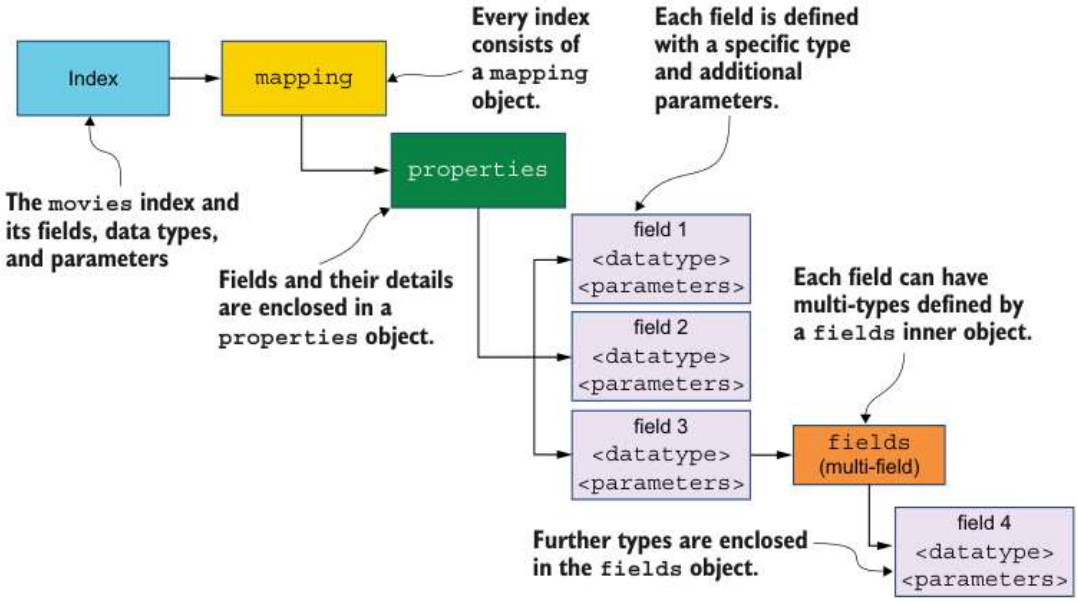


Figure 4.1 Anatomy of a mapping schema

- Etapas quando damos PUT num documento novo em index novo:
    - Um novo índex é criado com tudo default
    - Um schema é criado para esse index baseado nos data types inferidos
        - Mapping dinâmico
    - Documento é indexado e guardado
    - Documentos seguintes pulam as primeiras etapas

- Podemos dar GET no mapping em si

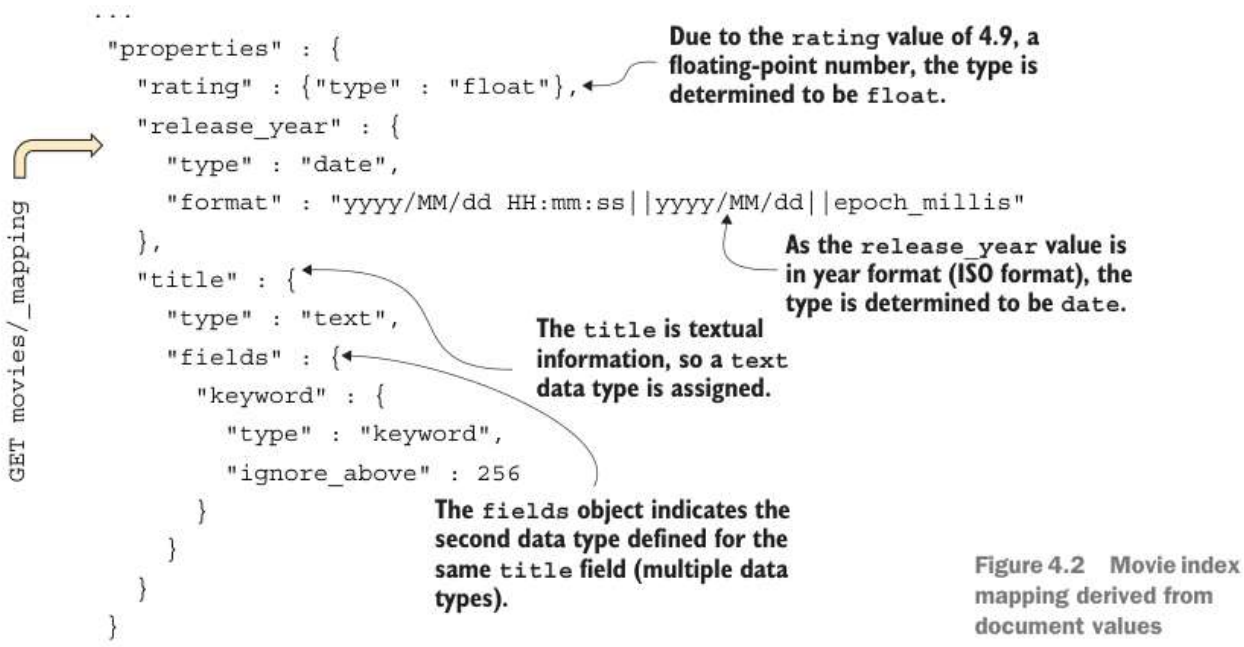
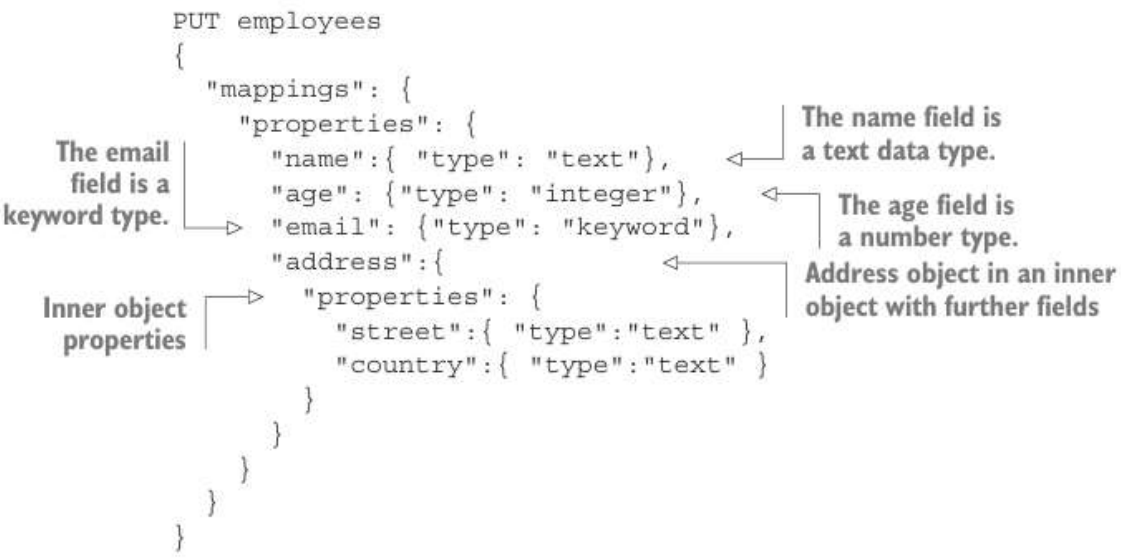


Figure 4.2 Movie index mapping derived from document values

- Fields multi-tipos
- Tipo data type **keyword** não é fatiado em tokens, é tratado como um token só.
- Mapping dinâmico pode errar, melhor só usar em testes.
    - E.g. números envoltos em aspas são tratados como strings
        - Não vai ser ordenável
- Texto pode ser ordenável se configurarmos *fielddata* como true
    - Mas *field data* é computacionalmente caro
    - Melhor usar keywords no lugar
    - Todo field pode ter um segundo tipo keyword
        - Daí ordenamos os **field.keywords**
- Única data padrão reconhecível automaticamente é YYYY-MM-DD - **Mapping explítico**
    - Geralmente recomendado
- Podemos atualizar um schema também.
- Podemos manipular schema:
    - API de criar index direto quando index ainda não existe

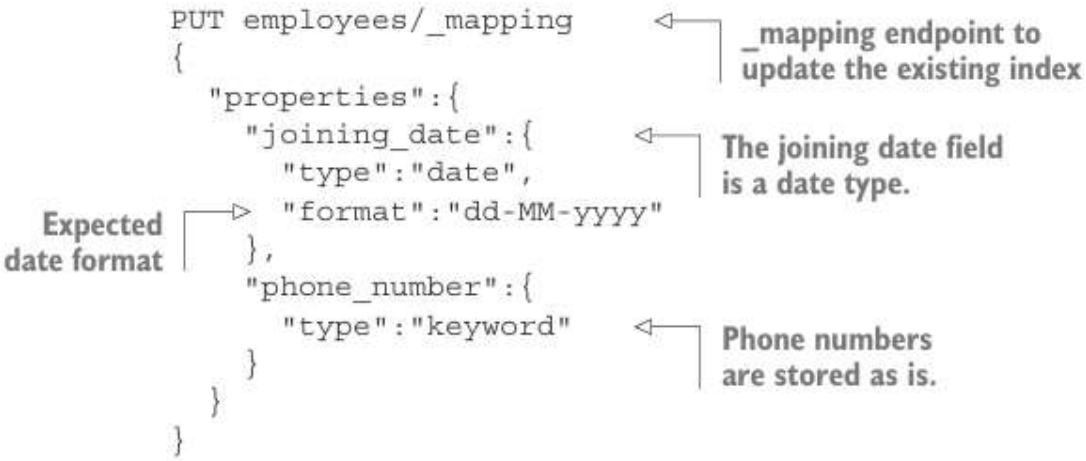
- API de mapping em si para atualizar um index que já existe.

Listing 4.4 Creating the schema up front



- No exemplo acima, *address* é um data type **object**, mas não precisa ser explicitado quando contém subcampos.

Listing 4.5 Updating the existing index with additional fields



Aqui **properties** está no nível root, no exemplo anterior era um campo de *mappings*

- Previne erros de busca
- i.e. criar um novo index com as especificações desejadas
- Mover os dados do index antigo para o novo, até pode deletar o antigo.
- Tem um API que já faz isso por conta própria
- Mantém o mesmo nome que pode apontar para index diferentes.
- Permite inferir transformação do dado inputado um pouquinho para encaixar no tipo aceitável.
    - E.g. número envolto de aspas não dão erro se o data type é numerico.
    - *Type Coercion*

- Types simples:
    - Boolean, text, long, date etc
- Types complexos:
    - Objects, nested, join etc
- Types especializados:
    - Geo_point, ip, date_range, ip_range etc.

Há vários tipos de text types, types bem específicos.

- Texto é decomposto nas suas partes, sem:
    - Pontuação
    - maiúsculas
    - terminações de palavras (stemming)
        - Ficam armazenados numa lista palavra por palavra.
- API _analyze mostra processo de análise.
- Token_count data type
    - Conta número de tokens
    - Field pode ser tanto texto quanto token_count

Listing 4.12 Adding `token_count` as an additional data type to a `text` field

```
PUT tech_books2
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "fields": {
          "word_count": {
            "type": "token_count",
            "analyzer": "standard"
          }
        }
      }
```

The title field is defined as text data type.

The title field is declared to have multiple data types.

word_count is the additional field.

The type of word_count

It is mandatory to provide the analyzer.

Listing 4.13 Searching for books with a four-word title

```
GET tech_books/_search
{
  "query": {
    "term": {
      "title.word_count": {
        "value": 4
      }
    }
  }
}
```

We are using a term query.

The name of the field

- Inclui tipos *keyword, contant_keyword, wildcard*
- Keyword é usado para resultado exatos, nunca parciais (e.g. um email)

    - Campo não é analisado pelo analyzer
    - Não é capaz de ser buscado por range

- *Constant_keyword*

    - Keyword já pré-definido no mapping
    - Pode ser omitido ao indexar documentos nesse index

- *Wildcard*

    - Permite realizar buscas com regexs e caracters coringas.

- **Date Data Type**

    - Dado estruturado, pode ser ordenado, filtrado e agregado.
    - ISO 8601 por padrão
        - Pode ser customizado para permitir múltiplos formatos
    - Permite busca por range - **Numeric Data Types**
    - **Integers** e **Floats**
        - Vários tipos de cada
            - E.g. long, float, unsigned_long, byte

- **Boolean Data Type**

- **Range Data Type**

    - Limite inferior e superior para um campo.
    - Há vários tipos de range
        - Integer_range, float_range etc.

Listing 4.19 Searching for courses between two dates

```
GET trainings/_search
{
  "query": {
    "range": {
      "training_dates": {
        "gt": "2021-08-10",
        "lt": "2021-08-12"
      }
    }
  }
}
```

We use a range query to search.

Searches for courses between these two dates

```
PUT trainings/_doc/1
{
  "name":"Functional Programming in Java",
  "training_dates":{
    "gte":"2021-08-07",
    "lte":"2021-08-10"
  }
}
```

ElasticSearch não permite modificar campos existentes, apenas acrescentar mais campos.
Uma alternativa a isso é Reindexação:
Para evitar problemas com a mudança do nome do index, podemos usar *aliases*.
ElasticSearch faz certo type coercion

Existem muitos data types, e sempre há novos.

Table 4.1 Common data types with examples

Type	Description	Examples
`text`	Represents textual information (such as string values); unstructured text	Movie title, blog post, book review, log message
`integer, long, short, byte`	Represents a number	Number of infectious cases, flights canceled, products sold, book's rank
`float, double`	Represents a floating-point number	Student's grade point average, moving average of sales, reviewer average ratings, temperature
`boolean`	Represents a binary choice: true or false	Is the movie a blockbuster? Are COVID cases on the rise? Has the student passed the exam?
`keyword`	Represents structured text: text that must not be broken down or analyzed	Error codes, email addresses, phone numbers, Social Security numbers
`object`	Represents a JSON object	(In JSON format) employee details, tweets, movie objects
`nested`	Represents an array of objects	Employee address, email routing data, a movie's technical crew

Text Types

Keyword data type

Listing 4.16 A `wildcard` query with a wildcard value

```
GET errors/_search
{
  "query": {
    "wildcard": {
      "description": {
        "value": "*obj*"
      }
    }
  }
}
```

Uses a wildcard query

Searches using wildcards

- **IP Data type**
  - IPv4 ou IPv6

DATA TYPES AVANÇADOS

- **Geo_point data type**
  - Latitude e longitude
  - Permite buscar por *geo_bounding_box*

Table 4.3 Location information-related formats

Format	Explanation	Example
Array	Geopoint represented as an array. Note the order of the geopoint inputs: it takes `lon` and `lat`, not the other way around (unlike the string format—see the next row).	`"address":[0.12,51.5]`
String	Geopoint as string data with `lat` and `lon` inputs.	`"address":"51.5,0.12"`
Geohash	An encoded string formed by hashing the longitude and latitude coordinates. The alphanumeric string points to a place on earth.	`u10j4`
Point	A precise location on a map. Known as well-known text (WKT), a standard mechanism to represent geometrical data.	`POINT(51.5,-0.12)`

- **Object Data Type**
  - É um JSON, tem hierarquia de campos
  - E.g. campo *attachment* com dois subcampos text

Listing 4.26 Searching for emails with an attachment file

```
GET emails/_search
{
  "query": {
    "match": {
      "attachments.filename": "file1.txt"
    }
  }
}
```

- Têm uma limitação de que objects internos são armazenados como array, não individualmente.
  - E.g. Se eu buscar pelo campo A por um valor e o campo B por outro, vai me retornar resultados mesmo que não seja o mesmo objeto que tenha esses values, vai procurar no array inteiro.
    - Se eu tiver no meu documento um hambúrguer com picles e um hotdog com pimenta, retornará o documento se eu pesquisar por hambúrguer com pimenta.
      - Pode gerar erros, contraintuitivo.
- **Nested Data Type**
  - Resolve exatamente o problema descrito anteriormente
  - Forma especializada de **object**
  - Deve ser declarado no mapping
- Não existe tipo Array no Elastic Search
  - Mas dá pra armazenar quantos valores forem num dado campo.
  - `"name": "John Doe"` to `"name": ["John Smith", "John Doe"]`
  - Array não pode ter tipos diferentes
- **Flattened Data Type**

- Previne que todos os subcampos dentro do flattened sejam analisados
  - Todos os subcampos viram keywords.
  - Subcampos não precisam ser declarados no mapping, quaisquer subcampos que apareçam
  - Podemos buscar por qualquer valor de qualquer subcampo ao mesmo tempo sem saber as chaves

Listing 4.33 Indexing a consultation document with doctor notes

```
PUT consultations/_doc/1
{
  "patient_name":"John Doe",
  "doctor_notes":{
    "temperature":103,
    "symptoms":["chills","fever","headache"],
    "history":"none",
    "medication":["Antibiotics","Paracetamol"]
  }
}
```

The flattened field can hold any number of subfields.

All of these fields are indexed as keywords.

Listing 4.34 Searching the `flattened` data type field

```
GET consultations/_search
{
  "query": {
    "match": {
      "doctor_notes": "Paracetamol"
    }
  }
}
```

Searching for a patient's medication

- **Join Data Type**
  - Cria relação pais-filhos
  - Parece com uso em Banco de Dados Relacionais

Listing 4.36 Mapping of the `doctors` schema definition

```
PUT doctors
{
  "mappings": {
    "properties": {
      "relationship":{
        "type": "join",
        "relations":{
          "doctor":"patient"
        }
      }
    }
  }
}
```

Declares a property as the join type

Names of the relations

The query has two important points to note:

- We declare a `relationship` property of type `join`.
- We declare a `relations` attribute and give the names of the relations (in this case, only one `doctor:patient` relation).

Listing 4.37 Indexing a doctor document

```
PUT doctors/_doc/1
{
  "name":"Dr. Mary Montgomery",
  "relationship":{
    "name":"doctor"
  }
}
```

The relationship attribute must be one of the relations.

Listing 4.38 Creating two patients for our doctor

```
PUT doctors/_doc/2?routing=mary
{
  "name":"John Doe",
  "relationship":{
    "name":"patient",
    "parent":1
  }
}

PUT doctors/_doc/3?routing=mary
{
  "name":"Mrs. Doe",
  "relationship":{
    "name":"patient",
    "parent":1
  }
}
```

Documents must have the routing flag set.

We define the type of relationship in this object.

The document is a patient.

The patient's parent (doctor) is the document with ID 1.

- Pais e filhos ficam armazenados no mesmo shard

- Desempenho desse tipo de busca não é muito bom
  - Costuma ser melhor evitar e deixar para banco de dados relacionais.
- **Search_as_you_type Data Type**
  - Analyzer cria sub tokens e multi-tokens para tentar buscar com o que o usuário está digitando

Table 4.5 Subfields created automatically by the engine

Fields	Explanation	Examples
`title`	The `title` field is indexed with either a chosen analyzer or the default analyzer if one is not chosen.	If a standard analyzer is used, the title is tokenized and normalized based on the standard analyzer's rules.
`title._2gram`	The `title` field's analyzer is customized with a shingle-token filter. The shingle size is set to 2 on this filter.	Generates two tokens for the given text. For example, the 2-grams for the title `"Elasticsearch in Action"` are `["elasticsearch", "in"], ["in","action"].`
`title._3gram`	The `title` field's analyzer is customized with a shingle-token filter. The shingle size is set to 3 on this filter.	Generates three tokens for the given text. For example, the 3-grams for the title `"Elasticsearch for Java developers"` are `["elasticsearch","for","java"], ["for","java","developers"]`
`title._index prefix`	The analyzer of `title._3gram` is applied along with an edge n-gram token filter	Generates edge n-grams for the field `title._3grams`. For example, the `_index_prefix` generates the following edge n-grams for the world `"Elastic": [e, el, ela, elas, elast, elasti, elastic]`

Listing 4.42 Searching in `search_as_you_type` and its subfields

```
GET tech_books4/_search
{
  "query": {
    "multi_match": {
      "query": "in",
      "type": "bool_prefix",
      "fields": ["title","title._2gram","title._3gram"]
    }
  }
}
```

  - **Multiple Types**
  - Campos podem ser multi-tipos no Elastic Search, se declararmos isso no mapping
  - Pode ser *Text*, *keyword* e *completion* ao mesmo tempo

Listing 4.43 Schema definition with a multi-typed field

```
PUT emails_multi_type
{
  "mappings": {
    "properties": {
      "subject":{
        "type": "text",
        "fields": {
          "kw":{ "type":"keyword" },
          "comp":{ "type":"completion" }
        }
      }
    }
  }
}
```

The text type

The subject is also a keyword.

The subject is a completion type, too.

Summary

- Every document consists of fields with values, and each field has a data type. Elasticsearch provides a rich set of data types to represent these values.
- Elasticsearch consults a set of rules when indexing and searching data. These rules, called mapping rules, let Elasticsearch know how to deal with the varied data shapes.
- Mapping rules are formed by either dynamic or explicit mapping processes.

- Mapping is a mechanism for creating field schema definitions up front. Elasticsearch consults the schema definitions while indexing documents so the data is analyzed and stored for faster retrieval.
- Elasticsearch also has a default mapping feature: we can let Elasticsearch derive the mapping rather than providing it explicitly ourselves. Elasticsearch determines the schema based on the first time it sees a field.
- Although dynamic mapping is handy, especially in development, if we know more about the data model, it is best to create the mapping beforehand.
- Elasticsearch provides a wide range of data types for text, Booleans, numerical values, dates, and so on, stretching to complex fields like joins, completion, geopoints, nested, and others.

5 - Working with Documents

- Operações single document e multi document tipo CRUD

INDEXAÇÃO

- Indexação pode ser feita com ou sem Ids
  - HTTP PUT para doc com ID provido pelo cliente na indexação
  - HTTP POST para doc ter ID autogerado pelo sistema

Listing 5.1 Indexing a new document into the `movies` index

```
PUT movies/_doc/1
{
  "title":"The Godfather",
  "synopsis":"The aging patriarch of an organized crime dynasty transfers
  ➥ control of his clandestine empire to his reluctant son"
}
```

Body of the request

Document indexing URL

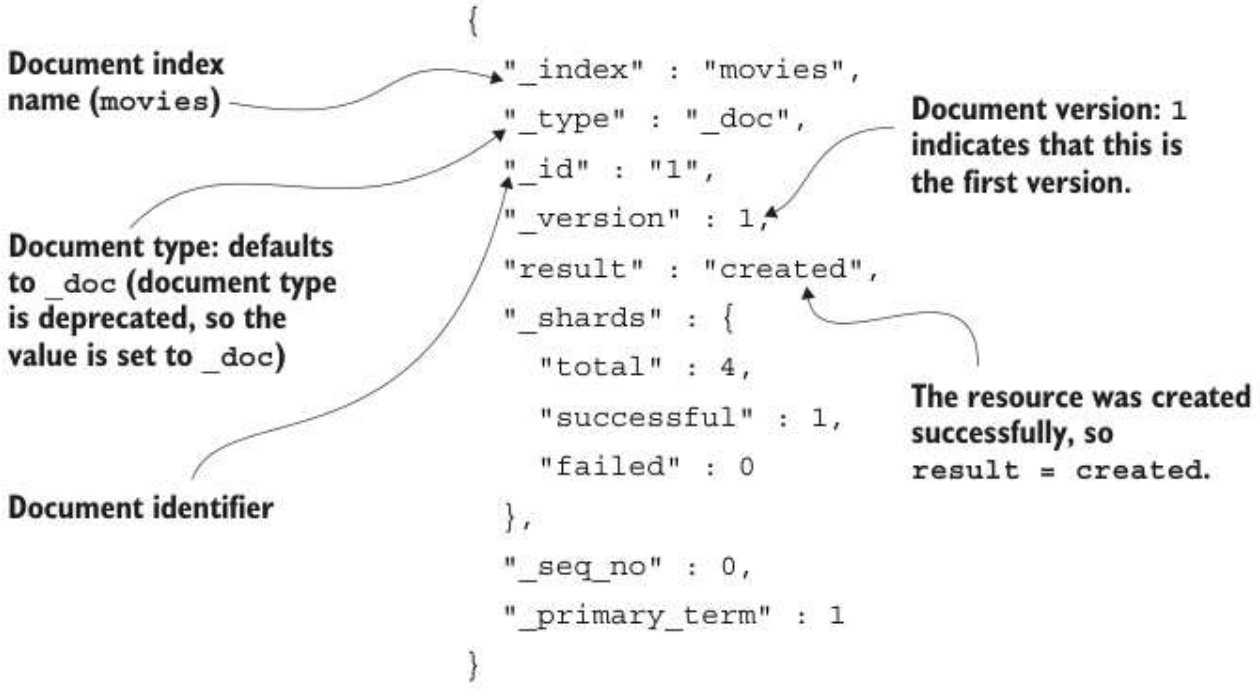


Figure 5.2 Response from the server when a document is indexed successfully

Listing 5.2 Indexing a document with no ID using `POST`

```
POST myindex/_doc
{
  "title":"Elasticsearch in Action"
}
```

The URL has no ID attached to it.

The request body is a JSON document.

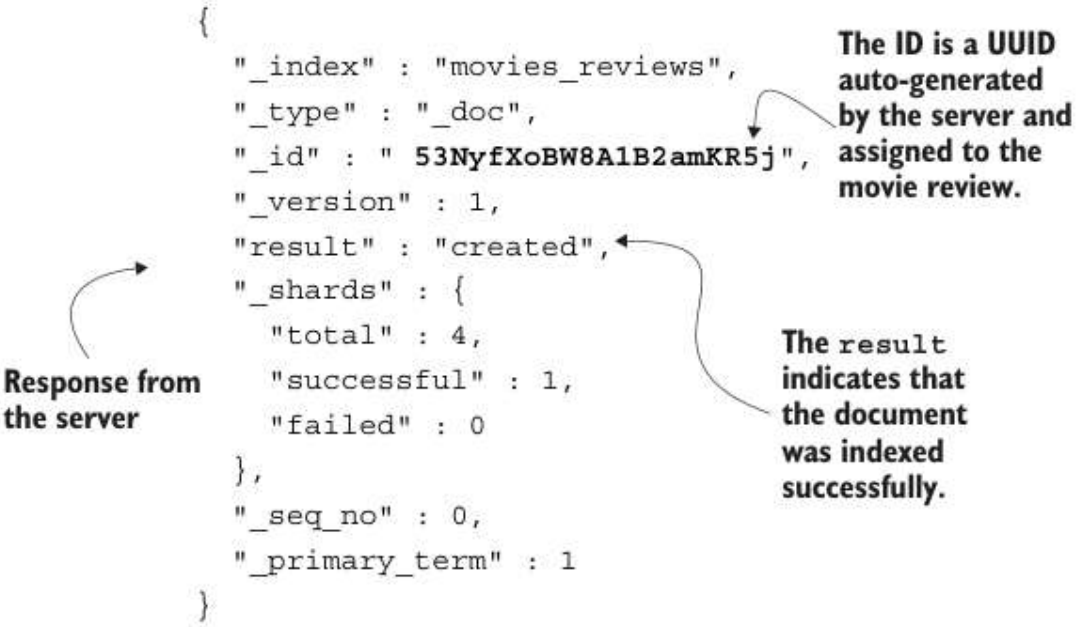


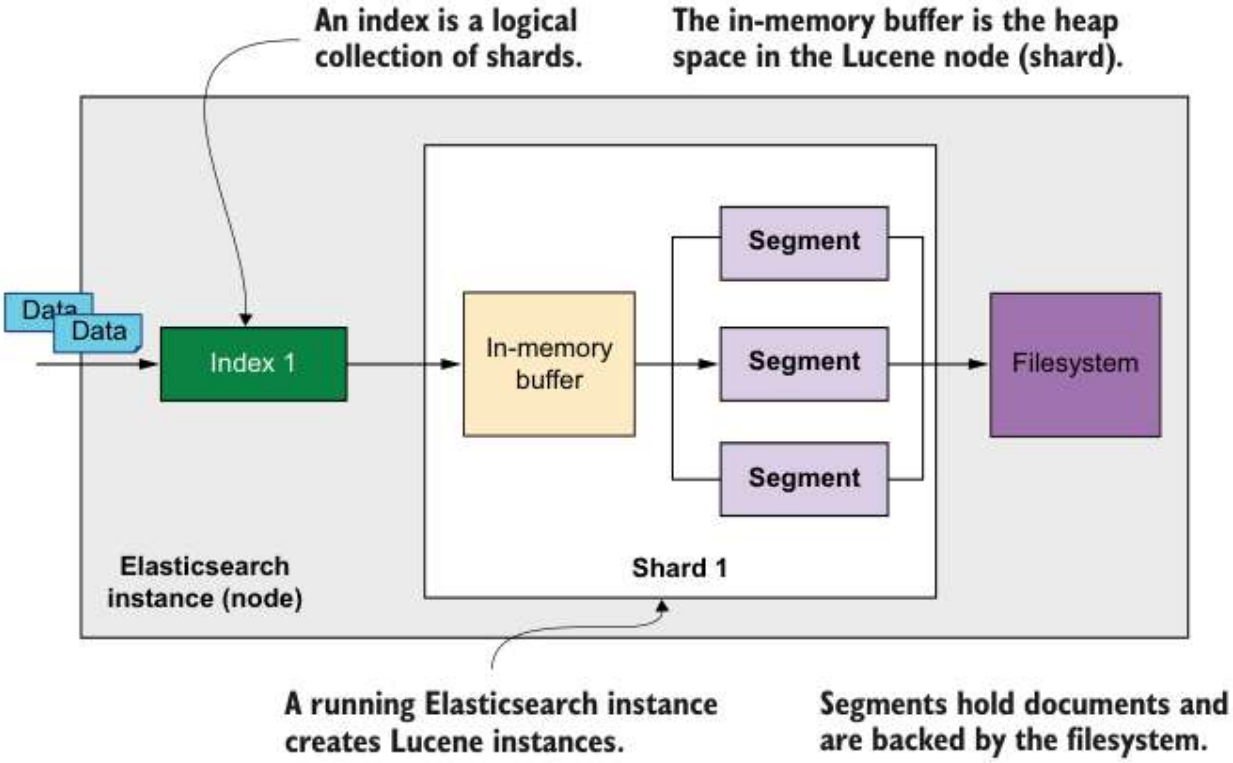
Figure 5.3 The server creates and assigns a randomly auto-generated ID to the document.

- Por default, operação de indexação ocorre igualmente caso o documento já exista na base ou não, sobreescreve.
    - API *_create* evita isso, não sobreescreve.
        - Dá erro caso id já exista na base

Listing 5.5 Indexing a new movie using the `_create` endpoint

```
PUT movies/_create/100
{
  "title":"Mission: Impossible",
  "director":"Brian De Palma"
}
```

- Criação dinâmica de index não pré-declarado pode ser desativada nas configurações do cluster
    - dá erro caso não exista mapping de index.



- Figure 5.4 Mechanics of indexing documents

- Documento antes vai pra heap do shard, e a cada segundo tem um refresh para criar os segmentos com os documentos e respectivos índices invertidos.

    - Depois vai para o disco físico
    - Nos segmentos documentos já são buscáveis
    - Cada três segmentos são mergeados em um único maior, recursivamente.
      - Até aos poucos salvar tudo
      - Caso algo seja deletado, o segmento é apenas marcado como deletado, e depois aos poucos deletado de fato.
      - It's like writing notes on sticky pads, grouping three pads into a notebook, then three notebooks

into a binder—so searching through them stays organized and quick!

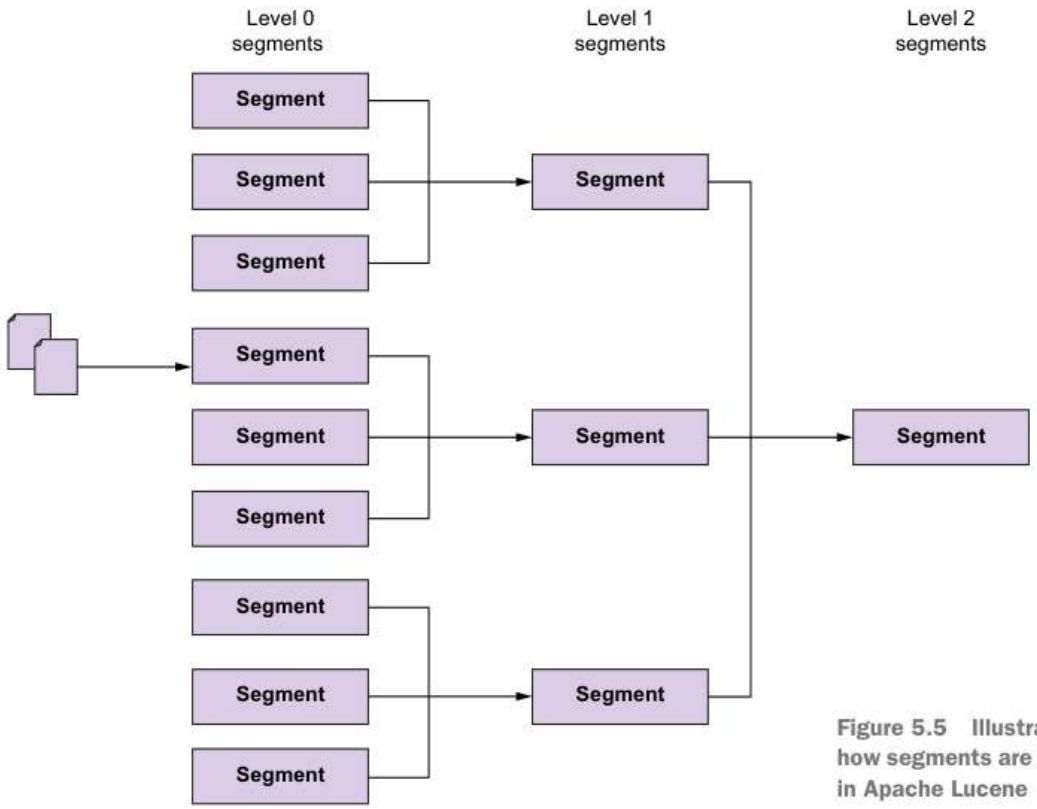


Figure 5.5 Illustration of how segments are merged in Apache Lucene

- Falha no servidor no intervalo entre refreshes pode causar perda de dados.
    - Mas refreshes mais frequentes são mais computacionalmente caros
        - I/O é lento.
- Elastic Search é *eventually consistent:*
    - Documentos são escritos no data store em algum momento.
- Intervalo de refresh é configurável por index - default é 1 segundo.
    - Refresh também pode ser feito manualmente com API *_refresh*.
    - Podemos pausar refresh completamente para por exemplo esperar migrar vários documentos de fora e torná-los buscáveis todos de uma vez, quando reativar refresh.
    - Tempo de refresh também pode ser alterado na própria operação de indexar um documento
        - "PUT movies/_doc/1?refresh=true"
            - TRUE - docs imediatamente buscáveis
            - FALSE(default)
            - WAIT_FOR - indexação só ocorre no próximo ciclo de refresh

BUSCA

- Busca pode ser de um documento ou vários de uma vez. -

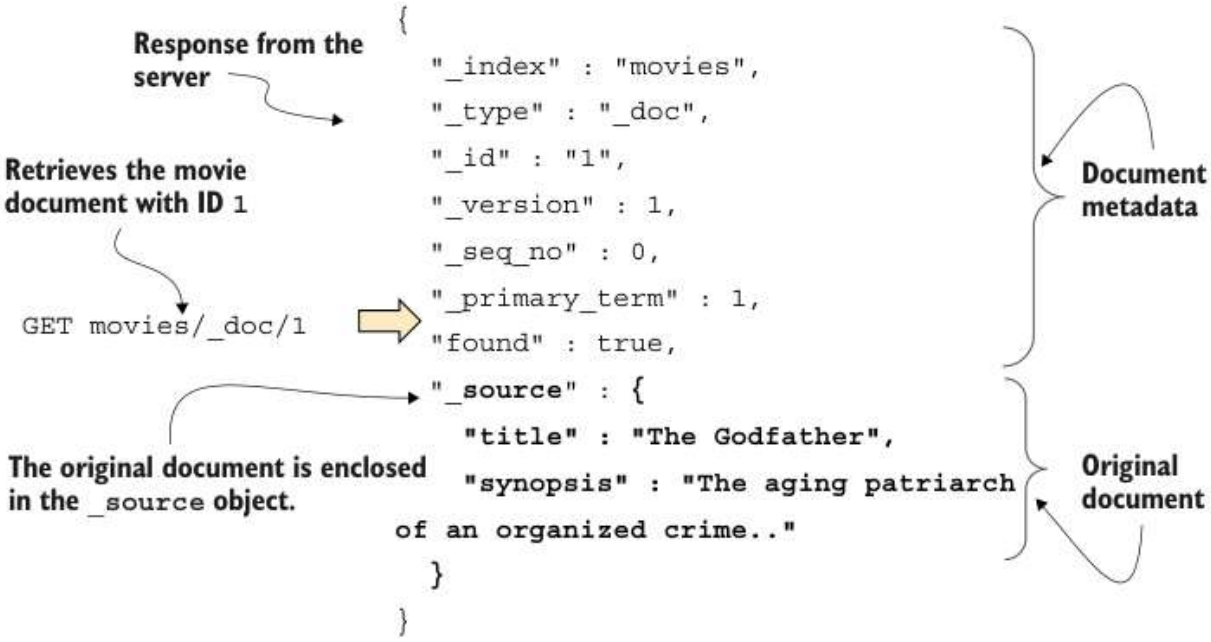


Figure 5.6 Retrieving a document using the GET API call

- Podemos apenas checar a existência de um documento:

  - `HEAD movies/_doc/1`

- Para buscar múltiplos documentos, podemos pegar:
  - De um único índice
  - De vários índices

Listing 5.9 Fetching multiple documents at once

```
GET movies/_mget
{
  "ids" : ["1", "12", "19", "34"]
}
```

Listing 5.10 Fetching documents from three different indexes

```
GET _mget                          <-- _mget call with no index mentioned in the URL
{
  "docs":[
   {
     "_index":"classic_movies",    <-- The first index is provided here.
     "_id":11
   },

   {
     "_index":"international_movies",   <-- Index 2
     "_id":22
   },
   {
     "_index":"top100_movies",     <-- Index 3
     "_id":33
   }
  ]
}
```

- Neste último, não tem como passar um array de Ids
- Também podemos buscar vários documentos usando a api *_search*

Listing 5.11 Using an `ids` query to fetch multiple documents

```
GET classic_movies/_search
{
  "query": {
    "ids": {
      "values": [1,2,3,4]
    }
  }
}
```

MANIPULATING RESPONSES

- Podemos querer omitir algumas das informações na resposta
  - Segurança
  - Eficiência
- Por exemplo, buscamos direto o atributo **source_ em vez de** doc_.
  - Não vem metadados
- Para fazer o oposto, suprimir o documento e pegar só o metadado:
  - `GET movies/_doc/1?_source=false`
- Para customizar ainda mais, podemos incluir ou excluir campos específicos com os atributos **source*includes*** **e** source*excludes* na chamada

- `GET movies/_doc/3?_source_includes=title,rating,genre`
- `GET movies/_source/3?_source_excludes=actors,synopsis`
- Também podemos incluir e excluir ao mesmo tempo:
  - `GET movies/_source/13?_source_includes=rating*&_source_excludes=rating_amazon`

UPDATING DOCUMENTS

- API _update atualiza um único documento
- API *_update_by_query* permite atualizar vários ao mesmo tempo
- Por trás, esta operação é uma substituição, consiste das três etapas:
  - Busca
  - Modificação
  - Reindexação
- *_update* pode tanto modificar o que já existe quanto adicionar campos novos
- Podemos usar scripts para facilitar e granularizar a atualização
  - E.g. um script que apenas adiciona um item a um array, sem precisar reescrever o array inteiro.

Listing 5.23 Adding an actor to the `actors` list via a script

```
POST movies/_update/1
{
  "script": {
   "source": "ctx._source.actors.add('Diane Keaton')"    <-- Additional actor
  }
}
```

- Exemplos de script, que podem coexistir na mesma requisição:
  - Adicionar um ator a um array
  - Remove um ator de um array
  - Adicionar um novo campo a um documento
  - Remover um campo de um documento
    - Remover um campo inexistente não retorna erro
  - Adicionar múltiplos campos
- Atualização pode ser condicional:

Listing 5.28 Conditionally updating the document using an if/else block

```
POST movies/_update/1
{
  "script": {
    "source": """
    if(ctx._source.boxoffice_gross_in_millions > 125)
      {ctx._source.blockbuster = true}
     else
      {ctx._source.blockbuster = false}
    """
  }
}
```

- Um script tem três partes:

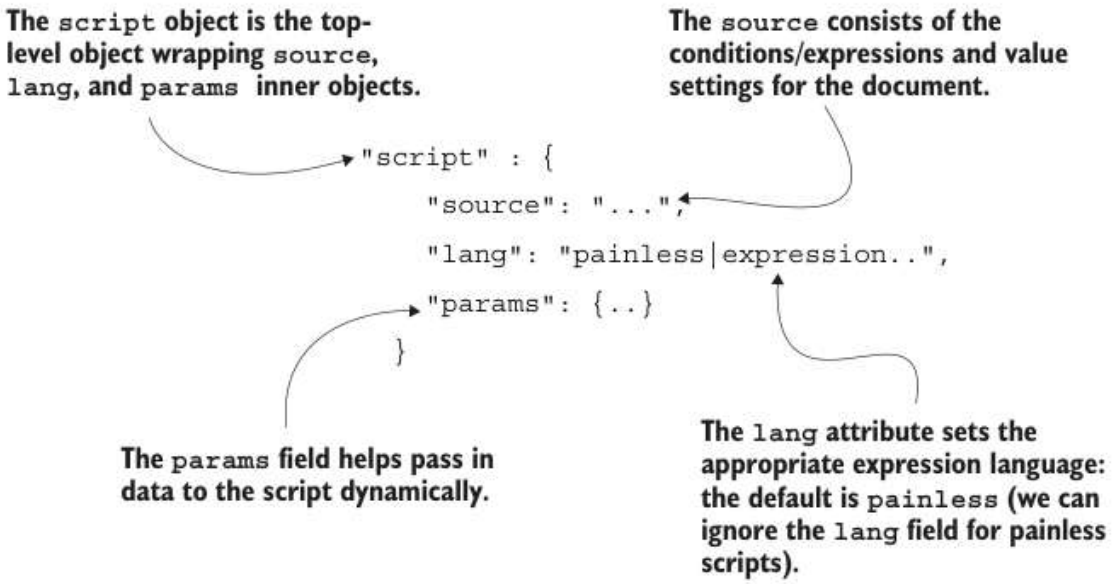


Figure 5.13 The anatomy of a script

- Source
  - Contém a lógica
- Params
- Language
  - Linguagem do próprio script
    - Painelss(default), expression, moustache, ou java - Podemos passar variáveis ao script sem deixá-la hardcoded

Listing 5.29 Dynamically passing a parameter to the script

```
POST movies/_update/1
{
  "script": {
    "source": """
 if(ctx._source.boxoffice_gross_in_millions >
     params.gross_earnings_threshold)
   {ctx._source.blockbuster = true}
 else
   {ctx._source.blockbuster = false}
 """,
    "params": {
      "gross_earnings_threshold":150
    }
  }
}
```

Business logic goes here.

Checks the value against params

Provides the parameter values

- Scripts são compilados quando rodam pela primeira vez
- Computacionalmente caro

- Com a parametrização, evitamos compilar várias vezes para cada valor hard-coded - Para substituir um documento, bastar usar um PUT no ID do documento que eu quero substituir, como se ele nem existisse antes. É sobrescrito.

```
PUT movies/_doc/1
{
  "title":"Avatar"
}
```

- - Para modificar ou inserir um documento, usamos a operação *Upsert*.

- Insere se não existe, atualiza (com um script, por exemplo) se existe.

Listing 5.31 Upsert block example

```
POST movies/_update/5
{
  "script": {
    "source": "ctx._source.gross_earnings = '357.1m'"
  },
  "upsert": {
    "title":"Top Gun",
    "gross_earnings":"357.5m"
  }
}
```

- Script só será executado se documento já existir, no exemplo acima.

    - Primeiro entraria como 357.7m, e na execução seguinte iria para 357.1m

- Para fazer upsert num campo só, usamos flag *doc_as_upsert*:

Listing 5.33 Updating the document if the field doesn't exist

```
POST movies/_update/11
{
  "doc": {
    "runtime_in_minutes":110
  },
  "doc_as_upsert":true
}
```

- Com a flag, se não houver doc com ID 11, não há erro. - Também podemos atualizar os documentos a partir de uma query, sem saber os Ids previamente:

- API *_update_by_query*

Listing 5.34 Updating a document using a `query` method

```
POST movies/_update_by_query                    <-- Searches for documents and updates the set
{
  "query": {                                    <-- Search query: searches for all movies with Al Pacino
    "match": {
      "actors": "Al Pacino"
    }
  },
  "script": {                                   <-- Applies the following script logic to the matching documents
    "source": """
    ctx._source.actors.add('Oscar Winner Al Pacino');
    ctx._source.actors.remove(ctx._source.actors.indexOf('Al Pacino'))
    """,
    "lang": "painless"
  }
}
```

- Essa função tem várias etapas por trás dos panos:
  - Busca os documentos
  - Recheca se a versão do documento é a mesma na busca acima e quando entra a operação de update.
    - Se não for, faz retry
  - Quando há erro, só o documento gerador de erro não é atualizado, os outros são.

DELETING DOCUMENTS

- Podemos fazer deleção por ID (um por vez) ou por query
- **DELETAR É IRREVERSÍVEL!**
  - `DELETE movies/_doc/1`
  - Dá erro se documento não existir
- Por query, usamos a API *_delete_by_query*

Listing 5.36 Deleting all movies based on a criterion

```
POST movies/_delete_by_query
{
  "query":{
    "match":{
      "director":"James Cameron"
    }
  }
}
```

  - Tão flexível quanto uma busca em si
    - Por ranges de algum campo
    - Com vários must e must not
    - Etc.
- Para deletar todos os documentos:

Listing 5.39 Deleting all documents at once

```
POST movies/_delete_by_query
{
  "query": {
    "match_all": {}
  }
}
```

- Funciona igualmente para vários index:
  - **POST <index_1>,<index_2>,<index_3>/_delete_by_query**
- Para listar todos os índex:
  - **GET _cat/indices**

WORKING WITH DOCUMENTS IN BULK

- API _bulk usado para indexar documentos em escala.
  - Podemos manipular e deletar com bulk também
  - Poupa bandwith do que fazer documento a documento

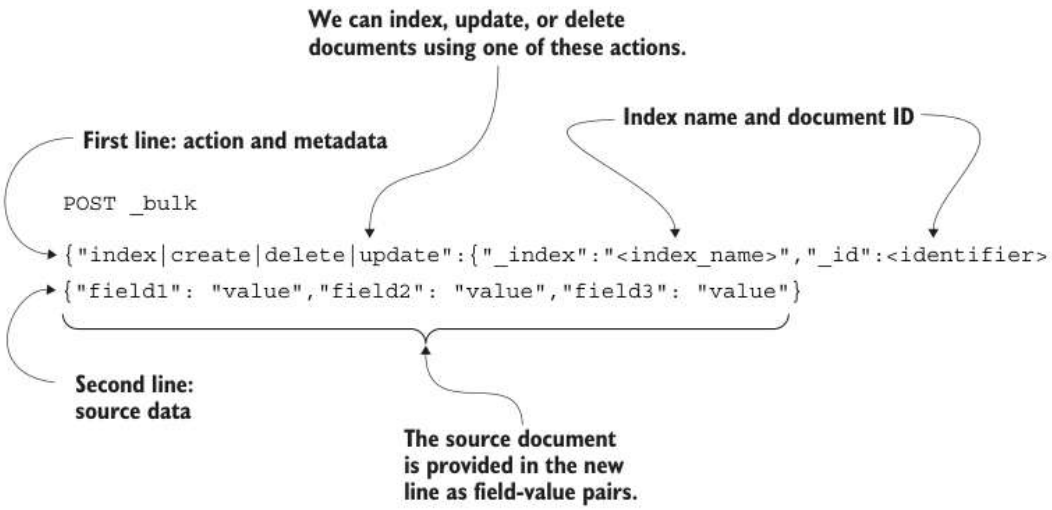


Figure 5.15 The _bulk API's generic format

  - Cada documento é definido numa linha.
    - NDJSON.
  - Podemos pré-definir o ID ou deixar o sistema autogerar.
  - Index pode estar incluído no URL em vez do corpo da requisição.
- Não precisam ser documentos do mesmo tipo
- Podemos usar create em vez de index para não substituir nada.
- Update e delete também funcionam analogamente
- Para requisição curl, passamos o json com os documentos como binário usando @

REINDEXING DOCUMENTS

- API *_reindex*
- Mover documentos para outro índex
  - Por exemplo, para alterar mapping schema

Listing 5.50 Migrating data between indexes using the `_reindex` API

```
POST _reindex
{
  "source": {"index": "movies"},
  "dest": {"index": "movies_new"}
}
```

-
- Unido a aliases, é uma ferramenta importante para fazer migrações com zero downtime

Summary

- Elasticsearch provides a set of APIs that work on documents. We can use these APIs to execute CRUD actions (create, read, update, and delete) on individual documents.
- Our documents are held in the shard's in-memory buffer and pushed into a segment during the refresh process. Lucene employs a strategy of creating a new segment with the documents during the refresh. It then cumulatively merges three segments to form a new segment, and the process repeats.

- Documents with identifiers (IDs) use the HTTP `PUT` action when indexing (for example, `PUT <index>/_doc/<ID>`), whereas documents with no IDs invoke the `POST` method.
- Elasticsearch generates random unique identifiers (UUIDs) and assigns them to documents during the indexing process.
- To avoid overriding a document, we can issue the following commands:
  - `_create`—This API throws an error if the document already exists.
  - `_mget`—This API lets us retrieve multiple documents at once, given their IDs.
  - `_bulk`—This API performs document operations such as indexing, deleting, and updating multiple documents in one invocation call.
- We can tweak the source and the metadata retrieved as a result of query invocations and then customize the returned document source to include and/or exclude fields by setting the `_source_includes` and `_source_excludes` properties, respectively.
- The `_update` API lets us modify an existing document by updating and adding fields. The expected updates are wrapped in a `doc` object and passed as the request body.
- Multiple documents can be modified by constructing an `_update_by_query` query.
- Scripted updates allow us to modify documents based on conditions. If the conditional clause mentioned in the request body is evaluated to `true`, the script is put into action.
- Documents can be deleted using HTTP `DELETE` for a single document or by running a `_delete_by_query` method on multiple documents.
- Migrating data between indexes is performed by the reindexing API. The `_reindex` API call expects to be given the source and destination indexes to transfer data.

6 - Indexing Operations

- Configurar index a baixo nível é importante
- Tem três configurações básicas:
  - Settings
    - E.g. número de shards e réplicas
  - Mapping
    - Schema
  - Alias
    - Renomear para melhorar querying e reindexação
- Index devem ser montados com **templates**, não manualmente.
- Políticas de rollover:
  - Alterações conforme o tempo e o tamanho crescem
  - Aposentar dados
- Várias operações com índices:
  - Creating, closing, shrinking, cloning, freezing, deleting, etc

CREATING INDEXES

- Index pode ser criado implicitamente (automaticamente)
  - Basta indexar o documento onde ele deve ficar
  - Podemos alterar depois mas nem tudo é alterável num index existente
  - Para desabilitar auto create index:

Listing 6.2 Disabling automatic creation of indexes

```
PUT _cluster/settings
{
  "persistent": {
    "action.auto_create_index":false
  }
}
```

Updates the settings across the whole cluster

The changes can be persistent or transient.

Shuts down auto-creation

    - Desativar criação automática pode causar problemas por exemplo com o Kibana, que cria index ocultos para administração
    - Index oculto tem um ponto antes (e.g. .user_profiles, .admin etc)
    - Para desativar auto_create seletivamente, por padrão de nome:
    - `action.auto_create_index: [".admin*, cars*, "*books*", "-x*","-y*","+z*"]`
    - Premite index automaticos com admin e cars e books no nome, mas não permite começando com x ou y.
    - Index fora de qualquer padrão também é desabilitado
- **Mapping**:
  - CAP 04
- **Settings**

Listing 6.3 Creating an index with custom settings

```
PUT cars_index_with_sample_settings
{
  "settings": {
    "number_of_replicas": 3,
    "codec": "best_compression"
  }
}
```

Creates the index

Applies the settings

- Dynamic setting:
  - Pode ser definito em index operante
- Static setting:
  - Só pode ser definido em index não em operação, não depois
  - Para alterar, precisamos fechar o index e reaplicar os settings, ou recriar o index
  - Dá exception se tentar trocar
- `GET cars_index_with_sample_settings/_settings` - **Alias**
- Nomes alternativos
- Pode apontar para um ou vários index

Listing 6.4 Creating an index with an alias

```
PUT cars_index_with_sample_alias
{
  "aliases": {
    "alias_for_cars_index_with_sample_alias": {}
  }
}
```

Creates the index

Declares the aliases object to configure the alias

The alias itself

- **Criar Index Explicitamente**
- Ideal para produção
- "PUT cars" cria um index cars com configurações default

Listing 6.5 Creating an index with custom settings

```
PUT cars_with_custom_settings
{
  "settings":{
    "number_of_shards":3,
    "number_of_replicas":5,
    "codec": "best_compression",
    "max_script_fields":128,
    "refresh_interval": "60s"
  }
}
```

Creates an index with custom settings

The settings object encloses the required properties.

Sets the number of shards to 3

Sets the number of replicas to 5

Changes the compression from its default value

Increases the maximum number of script fields from its default of 32

Changes the refresh interval from its default of 1 second

- Para reconfigurar os settings de um index:
  1. Fechar o index atual (desabilita read/write)
  2. Criar o index novo com os novos settings
  3. Migrar os dados do index velho para o novo (reindexing)
  4. Reapontar o alias para o novo index (se houver alias)
-

Listing 6.8 Fetching the settings of multiple indexes at once

```
GET cars1,cars2,cars3/_settings
GET cars*/_settings
```

Fetches multiple index settings

Fetches the settings for the index identified with a wildcard (*)

-

Listing 6.9 Fetching a single attribute

```
GET cars_with_custom_settings/_settings/index.number_of_shards
```

- Para criar index com **Alias**:
  - Podemos usar API _alias ou passar no corpo da requisição

Listing 6.12 Creating an alias using an `aliases` object

```
PUT cars_for_aliases
{
  "aliases": {
    "my_new_cars_alias": {}
  }
}
```

Creates an index with an alias

Points the alias to the index

-

Listing 6.13 Creating an alias using an `_alias` endpoint

```
PUT cars_for_aliases/_alias/my_cars_alias
```

-

  - Podemos criar vários índices juntos apontando a o mesmo alias
  - Wildcard pode ser usado aqui também, em vez da vírgula

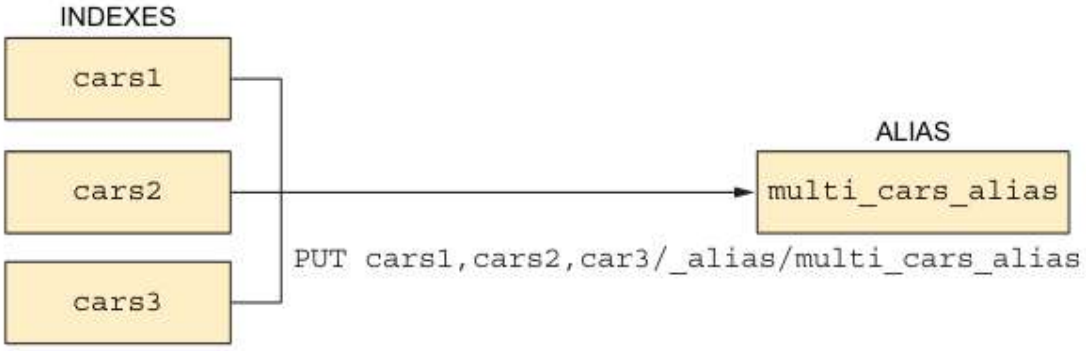


Figure 6.3 Creating an alias pointing to multiple indexes

  - Neste caso com vários, pelo menos um deles deve ter o atributo *is_write_index*

```
PUT cars3
{
  "aliases": {
    "cars_alias": {
    "is_write_index":true
    }
  }
}
```

- **Migração sem Downtime**
  - Por exemplo, atualizar um index vintage_cars
    1. Criar um aliás *vintage_car_alias* apontando para *vintage_cars*
    2. Criar um novo index *vintage_cars_new* com os novos settings desejados
    3. Copiar (reindexar) dados do index antigo para o index novo
    4. Recriar o alias existente para apontar para o novo index
    5. Fazer as queries serem executadas para o alias, não o index direto
    6. Apagar o index antigo e o index novo estiver em operação
- Podemos configurar vários aliases com a API *_aliases*

Listing 6.19 Performing multiple aliasing operations

```
POST _aliases
{
  "actions": [
    {
      "remove": {
        "index": "vintage_cars",
        "alias": "vintage_cars_alias"
      }
    },
    {
      "add": {
        "index": "vintage_cars_new",
        "alias": "vintage_cars_alias"
      }
    }
  ]
}
```

Uses the _aliases API to execute multiple actions

Lists the individual actions

Removes the alias that points to an old index

Adds an alias that points to a new index

  - "index" pode ser um array (daí a key vira "indices")

READING INDEXES

- Estamos lidando com index públicos

Listing 6.24 Getting individual configurations for an index

```
GET cars/_settings
GET cars/_mapping
GET cars/_alias
```

Gets the settings from the _settings endpoint

Gets the mappings from the _mapping endpoint

Gets the aliases from the _alias endpoint

- Com comando "HEAD cars" nos diz se index existe
- Há também os hidden index
  - Prefixo por ".", e.g. ".admin"

- Hoje podemos criar hidden index livremente, mas no futuro deve se tornar apenas para index de sistema
- *GET _all* ou GET * inclui hidden index

DELETING INDEXES

- Comando "DELETE cars, movie, order"
- Também podemos usar wildcard para deletar
  - Mas para deleter também os hidden indexes, precisamos usar o _all
  - Podemos desabilitar ou não delete por wildcard/_all:
    - Default é true

```
PUT _cluster/settings
{
 "transient": {
   "action.destructive_requires_name":false
 }
}
```

- Também podemos deletar apenas alias:

Listing 6.26 Deleting an alias explicitly

```
DELETE cars/_alias/cars_alias
```

Deletes cars_alias

- Não tem volta

CLOSING AND OPENING INDEXES

- **Closing:**
  - Index fechado fica sem ler/escrever dados

Listing 6.27 Closing the `cars` index indefinitely

```
POST cars/_close
```

  - Também podemos usar wildcard ou _all
    - (também requer propriedade de destructive_requires_name = True)
  - Fechar index limpa memória e disponibiliza recursos, busca requer menos recursos
  - Podemos desabilitar fechamento completamente:
    - Default é true

Listing 6.30 Disabling the closing indexes feature

```
PUT _cluster/settings
{
  "persistent": {
    "cluster.indices.close.enable":false
  }
}
```

- **Opening:**
  - Podemos simplesmente abrir um index fechado para restaurá-lo:

Listing 6.31 Putting the index back into operation

```
POST cars/_open
```

- Análogo ao close

INDEX TEMPLATE

- Pré-settar configurações (settings, mappings, alias) de um index seguindo um default definido
- Por exemplo, cada ambiente ter um template com um número de réplicas
- Se o index tiver um nome que bate com um certo padrão de nome (glob) definido, é aplicado o template ao index.
- Existem dois tipos de template:
  - Component
  - Composable
    - Composto de zero ou mais index templates components
    - Template também pode existir em avulso.

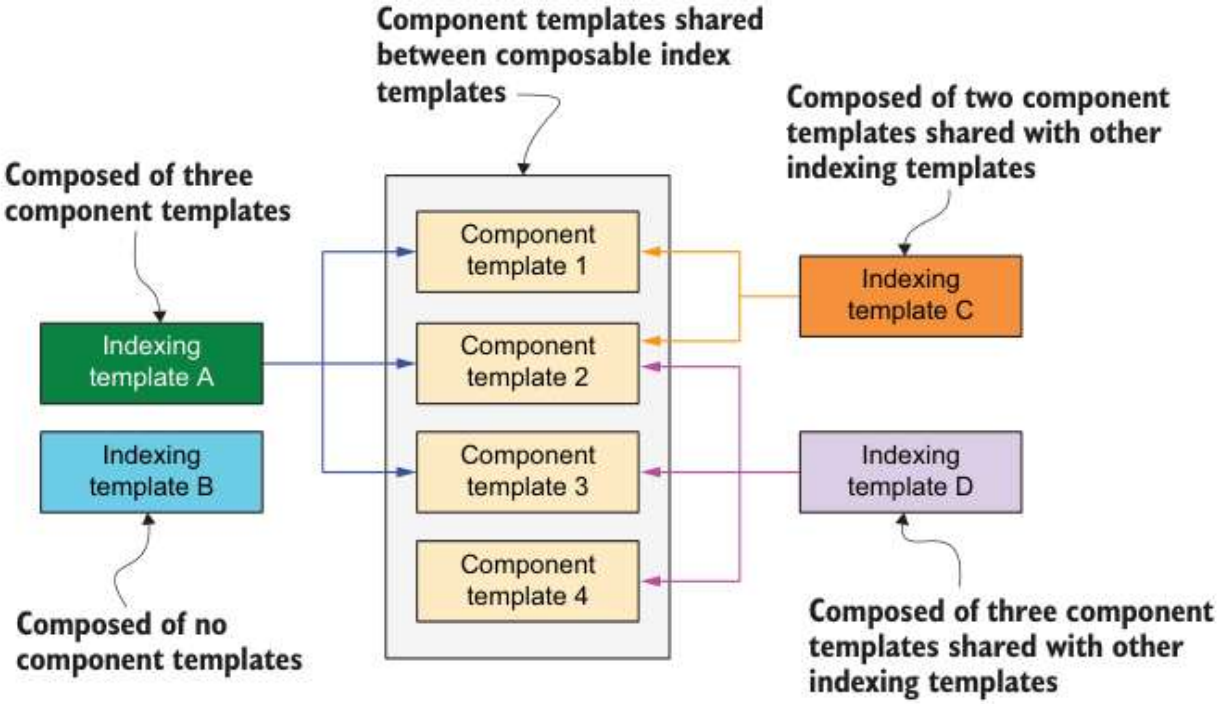


Figure 6.6 Composable (index) templates are composed of component templates.

- Assim vamos montando templates de configurações a partir de pequenos padrões menores.
  - Configurações mandadas explicitamente por comando têm prioridade sobre templates.
  - Legacy template tem prioridade menor que composable template
  - Podemos definir prioridade de template explicitamente com um número inteiro:
    - Default priority = 1

```
POST _index_template/cars_template_mar21
{
  "index_patterns": ["*cars*"],
  "priority": 20,          <-- Lower-priority index template
  "template": { ... }
}
POST _index_template/cars_template_feb21
{
  "index_patterns": ["*cars*"],

    "priority": 30,        <-- Matching template, but with a higher priority
    "template": { ... }
  }
```

    - Feb21 neste exemplo tem prioridade sobre mar21
    - Aplica todas as configurações de todas os templates, mas se houver alguma igual em ambos os templates, prevalece a do template de maior prioridade
  - Se um index der match em mais de um template, usa-se a prioridade

- API _index_template para criar template:

Listing 6.32 Creating an index template

```
PUT _index_template/cars_template
{
  "index_patterns": ["*cars*"],
  "priority": 1,
  "template": {
    "mappings": {
      "properties":{
        "created_at":{
          "type":"date"
        },
        "created_by":{
          "type":"text"
        }
      }
    }
  }
}
```

  -
  - Qualquer index com "cars" no nome receberá este template

- Component template é um bloco de configurações reutilizável
  - *_component_template* endpoint

Listing 6.33 Developing a component template

```
POST _component_template/dev_settings_component_template
{
  "template":{
    "settings":{
      "number_of_shards":3,
      "number_of_replicas":3
    }
  }
}
```

-
- Fará parte de um template, não é um template em si
  - Por exemplo, não tem um index pattern associado
- Para utilizar o componente em um template:

Listing 6.35 Index template composed of component templates

```
POST _index_template/composed_cars_template
{
  "index_patterns": ["*cars*"],
  "priority": 200,
  "composed_of": ["dev_settings_component_template",
                  "dev_mapping_component_template"]
}
```

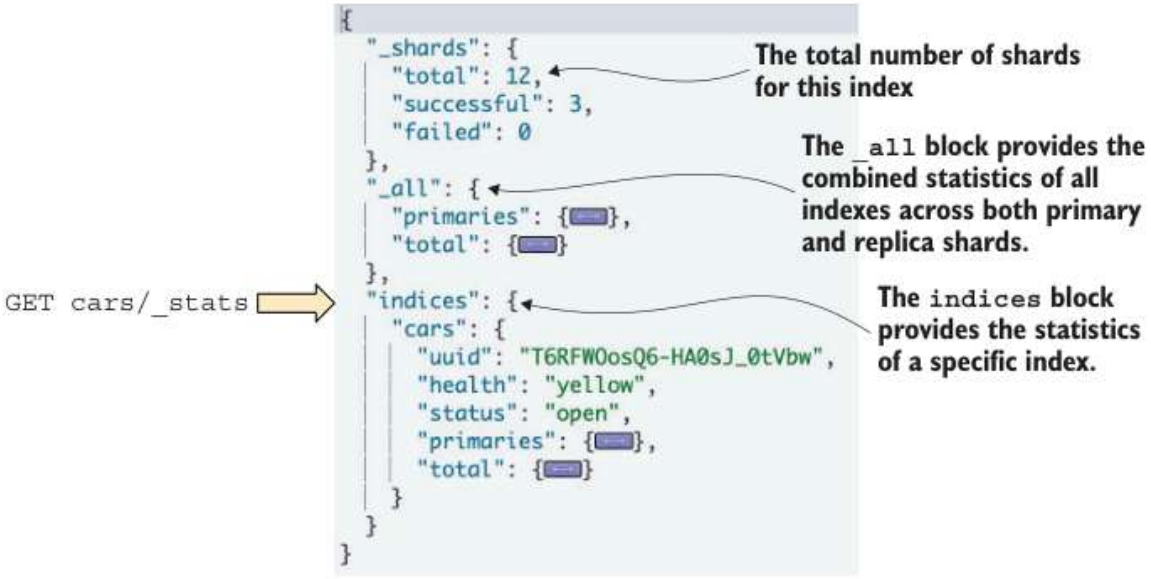
-

MONITORING AND MANAGING INDEXES

- Estatísticas dos indexes
  - Número total de documentos, documentos deletados, etc
  - *_stats* API

Listing 6.36 Fetching the statistics of an index

```
GET cars/_stats
```



- **Figure 6.7 Statistics for a `cars` index**

Table 6.1 Index statistics fetched by a call to the `_stats` endpoint (the list is shortened for brevity)

Statistic	Description
`docs`	Number of documents in the index and number of documents deleted
`store`	Size of the index (in bytes)
`get`	Number of GET operations on the index
`search`	Search operations including query, scroll, and suggest times.
`refresh`	Number of refresh operations

- Também dá pra listar vários indexes pra ter statisticas de vários index, ou então o wildcard
- Podemos querer só alguns tipos de estatísticas, como segments:

Listing 6.40 Segment statistics

```
GET cars/_stats/segments
```

This command returns the following data:

```
"segments" : {
        "count" : 1,
        "memory_in_bytes" : 1564,
        "terms_memory_in_bytes" : 736,
        "stored_fields_memory_in_bytes" : 488,
        "term_vectors_memory_in_bytes" : 0,
        "norms_memory_in_bytes" : 64,
        "points_memory_in_bytes" : 0,
        "doc_values_memory_in_bytes" : 276,
        "index_writer_memory_in_bytes" : 0,
        "version_map_memory_in_bytes" : 0,
        "fixed_bit_set_memory_in_bytes" : 0,
        "max_unsafe_auto_id_timestamp" : -1,
        "file_sizes" : { }
      }
```

- Também podemos chamar o endpoint *_segments* diretamente

ADVANCED OPERATIONS

- Repartir um index, rolling periódico etc - **Splitting**
  - Bom quando um index tem dados demais e não queremos sobrecarregá-lo
  - Dividir um index para um index com mais shards
  - É simplesmente um novo index com settings diferentes e os dados copiados
  - API *_split*
    - Primeiro passo é bloquear as operações de write:

Listing 6.41 Making sure the index is read-only

```
PUT all_cars/_settings                    <- Uses the _settings API
{
  "settings":{
    "index.blocks.write":"true"           <- Closes the index for writing operations
  }
}
```

    - Daí então fazemos o split:

Listing 6.42 Splitting the `all_cars` index

```
POST all_cars/_split/all_cars_new         <- _split expects a target index.
{
  "settings": {
    "index.number_of_shards": 12          <- Sets the number of shards on the new index
  }
}
```

    - Processo síncrono (só termina quando resposta é enviada)
      - Novo index terá todos os dados do index antigo

        - **Condições:**
            - **Index target não pode já existir**
            - **Número de shards do target tem que ser um múltiplo do valor original**
            - **Target não pode ter menos shards do que o original**
            - **Node do target tem que ter espaço o suficiente**
        - Splitting só muda o número de shards, o resto, como os settings, fica igual - **Shrinking**
    - Operação reversa do splitting: diminuir o número de shards.
    - Otimizar recursos
        - Primeiro fazemos como no split de bloquear write:
            - Também restringimos os shards existentes a um único node

Listing 6.43 Prerequisites before shrinking the index

```
PUT all_cars/_settings
{
  "settings": {
    "index.blocks.write": true,
    "index.routing.allocation.require._name": "node1"
  }
}
```

        - Daí sim fazemos o shrink:

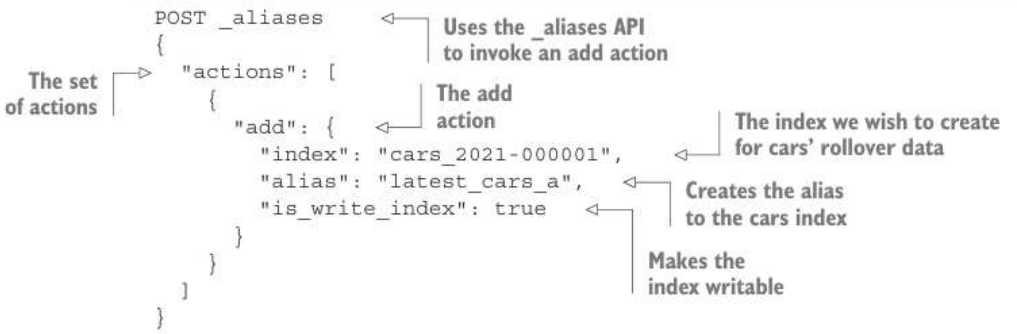
Listing 6.44 Shrinking the index

```
PUT all_cars/_shrink/all_cars_new
{
  "settings":{
    "index.blocks.write":null,
    "index.routing.allocation.require._name":null,
    "index.number_of_shards":1,
    "index.number_of_replicas":5
  }
}
```

Shrinks the source index
Removes the read-only instruction
Sets the node name to null
Reduces the number of shards

        - Número de shards deve ser um fator do número de shards original
    - **Condições:**
        - **Source index tem que ser desligado (tornar read-only)**
            - É uma boa ideia também desabilitar réplicas (igual a 0)
        - **Target index não pode já existir**
        - **Todos os shards do source index devem estar no mesmo node**
        - **Número de shards do target deve ser um fator do número original**
        - **O node do target index deve ter memória o suficiente - Rolling Over de um alias**
    - Rolamento automático do Index atual a um index novo
    - Dados do index antigo não são copiados para o novo.
    - Antigo se torna read-only e todos os documentos novos vão para o index novo
    - Uso comum é em séries temporais (i.e. dado relativo a um período específico)
    - Roll over são basicamente dois passos:
        1. *Alias* é criado apontando ao index original (que tem que ser writeable)
        2. ES chama a API *rollover que cria um novo index e aponta o _alias* para ele
            - Sempre tem que ter ao menos um index writeable
            - ES automaticamente incremente o index com uma unidade (e.g. *my-index-000005* depois de *my-index-04*)
    - Execução do Roll over:
        - Para fazer isso, antes de tudo criamos um alias para o index original, que já existe:

Listing 6.45 Creating an alias for an existing index



-
  - Deve ser writeable
  - Se o alias apontar para vários index, ao menos um deve ser writeable
- Em seguida, utiliamos o endpoint _rollover **sobre o alias, não sobre o index**:

Listing 6.46 Rolling over the index

```
POST latest_cars_a/_rollover
```

-
  - Index novo então foi criado e o antigo virou read-only
- Roll over faz três coisas:
  - Torna index antigo read-only
  - Cria o novo index com o nome incrementado, o resto permanece igual
  - Reaponta o alias para o novo index
- Resta saber como automatizar isso:
  - Como fazer roll over quando o tamanho de shard ultrapassa um determinado limite?
  - Como fazer um index novo todo dia pra logs diários?

INDEX LIFECYCLE MANAGEMENT (ILM)

- Queremos gerir os index e agir sobre eles para otimizar uso de recursos (para mais ou para menos) conforme alterações no cluster vão acontecendo ao longo do tempo
- Podemos definir um conjunto de regras para fazer isso
  - E.g. regras para roll over.
    - Index atingiu um dado tamanho
    - Número de documentos atingiu um certo número
    - Virou o dia
- A vida de um index tem 5 fases:
  1. **Hot**
     - Index operante e usado com frequência para read e write
  2. **Warm**
     - Index é read-only e pode ser consultado
  3. **Cold**
     - Index é read-only. Queries devem ser bem raras e devem ser devagar.
  4. **Frozen**
     - Index é read-only. Queries são BEM raras e respostas BEM lentas.
  5. **Delete**
     - Dados são apagados e memória é liberada
     - É comum tirar um snapshot do index antes de apagar, caso dados sejam necessários no futuro

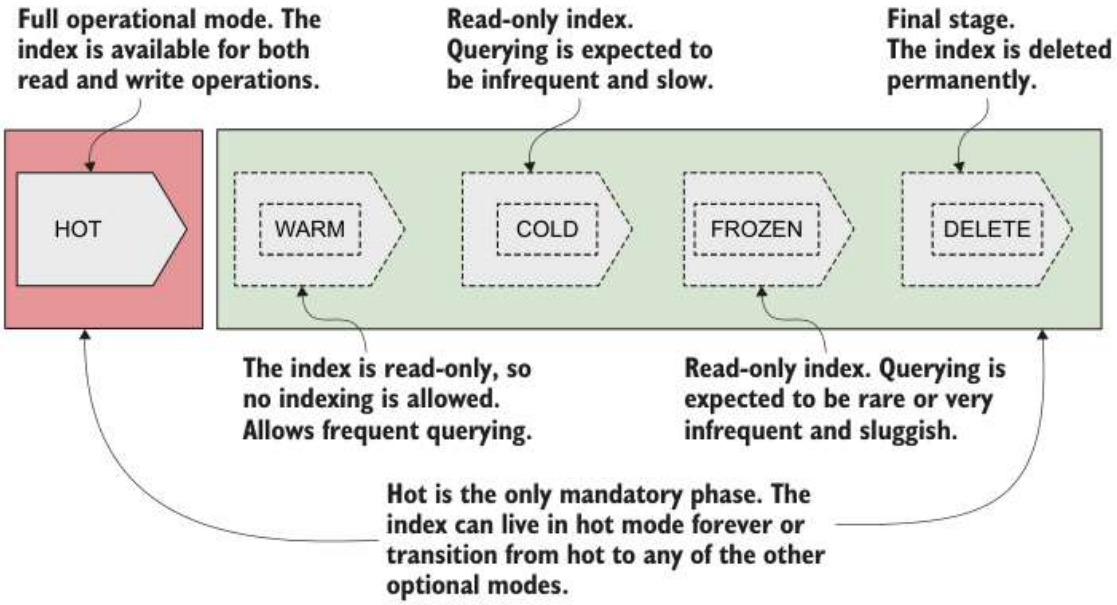


Figure 6.11 Lifecycle of an index

6. - Até então aprendemos como agir manualmente sobre os index, mas não como fazer isso com regras estabelecidas.

- Para isso, usamos API *_ilm*, em duas etapas:

1. Definimos uma política:

Listing 6.48 Creating a policy with hot and delete phases

```
PUT _ilm/policy/hot_delete_policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "1d",
        "actions": {
          "set_priority": {
            "priority": 250
          }
        }
      },
      "delete": {
        "actions": {
          "delete" : { }
        }
      }
    }
  }
}
```

The ILM API
Defines the policy and its phases
Defines the first phase (hot)
Sets the minimum age before other actions are carried out
Sets the priority
Defines the actions that must be carried out
Defines the delete phase

- No exemplo definimos duas fases só: hot e delete.
  - Index deve ser hot por 1d
    - Após isso, ele executa as actions (set priority, no caso) e passa para a fase delete
  - Então index se torna delete
    - Delete não tem min_age, então ocorre instantaneamente
    - Index é deletado

2. Associamo-la com um index:

Listing 6.49 Creating an index with an associated index lifecycle

```
PUT hot_delete_policy_index
{
  "settings": {                          <-- Uses the settings object to set the property
    "index.lifecycle.name":"hot_delete_policy" # Name of the policy
  }
}
```

-

> **Is It Just a Label?**
>
> No, **warm** and **frozen** phases are not just labels. They are actionable phases within an ILM policy, and Elasticsearch performs specific actions associated with each phase to manage resource usage effectively. However, the exact changes depend on the ILM policy configuration for the index. If no actions are defined for a phase in the ILM policy, then the phase change might act as a label only.

- Lifecycle com Rollover:
  - Podemos definir um rollover como action de uma política:

Listing 6.50 Simple policy definition for the hot phase

```
PUT _ilm/policy/hot_simple_policy
{
  "policy": {
    "phases": {
      "hot": {                           <-- Declares a hot phase
        "min_age": "0ms",                <-- The index enters this phase instantly.
        "actions": {
          "rollover": {                  <-- The index rolls over if any of the conditions are met.
            "max_age": "1d",
            "max_docs": 10000,
            "max_size": "10gb"
          }
        }
      }
    }
  }
```

  -
  - Neste exemplo, política é aplicada imediatamente, já que min_age é 0ms
  - Podemos também associar uma lifecycle policy com um template:
    - Necessário para a ILM ser aplicada ao rollover

Listing 6.51 Attaching a lifecycle policy to a template

```
PUT _index_template/mysql_logs_template
{
  "index_patterns": ["mysql-*"],         <-- Index pattern for all MySQL indexes
  "template":{
    "settings":{
      "index.lifecycle.name":"hot_simple_policy",   <-- Attaches the policy

      "index.lifecycle.rollover_alias":"mysql-logs-alias"   <-- Attaches an alias
    }
  }
}
```

    -

        - Precisamos determinar a policy e o rollover_alias nesta ação
    - O último passo é criar um index que dê match no index_pattern definido no template:

Listing 6.52 Setting the index as writable for the alias

```
PUT mysql-index-000001          <-- Creates an index with the appropriate format
{
  "aliases": {
    "mysql-logs-alias": {       <-- Enables the alias
      "is_write_index":true     <-- The backing index must be writable.
    }
  }
}
```

    - Logo:
        - Index terá a policy imediatamente
            - Será hot imediatamente
        - Permanece Hot até alguma condição estabelecida ocorrer
            - Quando houver alguma das condições, rollover ocorre
                - E sucessivamente para os novos index criados - Condições das polcies são verificadas a cada 10minutos, por default
    - Pode ser alterado no settings do endpoint _cluster
    - Podemos resetar ciclo de varredura manualmente
    - Policies são uma background task, e esperam o próximo ciclo de der algum ruim, então pode não ocorrer exatamente no momento esperado.

Summary

- Elasticsearch exposes index APIs to create, read, delete, and update indexes.
- Every index has three sets of configurations: aliases, settings, and mappings.
- Indexes can be created implicitly or explicitly:
  - Implicit creation kicks in when an index doesn't exist and a document is indexed for the first time. Default configurations (such as one replica and one shard) are applied on an index that's created implicitly.
  - Explicit creation occurs when we instantiate indexes with a custom set of configurations using the index API.
- An index template lets us create indexes with predetermined configuration settings that, based on a matching name, are applied during index creation.
- An index can be resized using a shrinking or splitting mechanism. Shrinking reduces the number of shards, while splitting adds more primary shards.
- An index can be conditionally rolled over as required.
- Index lifecycle management (ILM) helps transition indexes between these lifecycle phases: hot, warm, cold, frozen, and delete. In the hot phase, the index is fully operational and open for searching and indexing; but in the warm and cold phases, the index is read-only.

7 - Text Analysis

- Quebrar texto em tokens e comparar com tokens existentes
- Ideal é ser mais inteligente do que só fazer um match exato. - Query de dado não-estruturado é mais complicado do que retornar sim ou não pra um documento.
- Tanto query quanto documentos indexados são analisados pelos mesmos analyzers.
- Ideal é que engine entenda abreviações, erros de ortografia, sinônimos, emojis, etc.
    - ElasticSearch faz isso

ANALYZER MODULES

- Analyzer tem dois processos:
    - Tokenização
        - Fatiar o texto em tokens, usando alguma regra
            - Por exemplo, separar tokens por espaço em branco
                - Mas pode ter outras separações
    - Normalização
        - Transformação dos tokens
            - Por exemplo, stemming i.e. transformar texto em seu radical
            - Ignorar stopwords - palavras irrelevantes muito frequentes
- Input é texto e o output é uma lista de tokens normalizados.
- Analyzer têm **três** componentes:
    - Character filters
        - Aplicado aos caracteres
        - Remove caracteres indesejados, e.g. tags HTML
        - Transforma caracteres estrangeiros em latinos.
        - São opcionais
    - Tokenizer
        - Deve existir e só pode haver apenas um único tokenizer
    - Token Filter
        - Transformações, como ignorar maiúsculas, stemming, sinônimos, etc.
        - São opcionais.

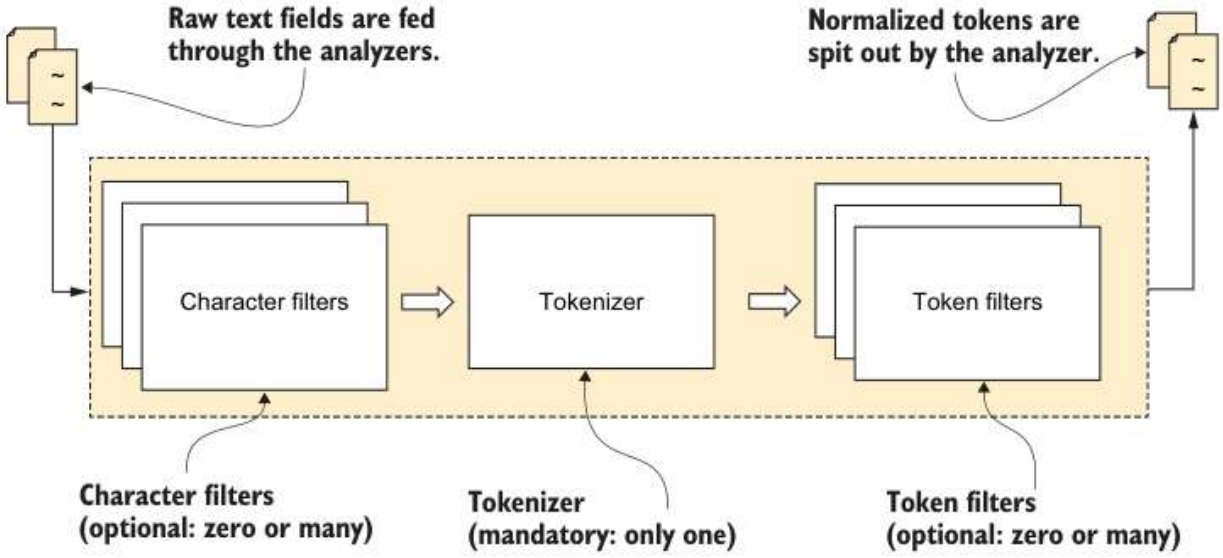


Figure 7.1 Anatomy of an analyzer module

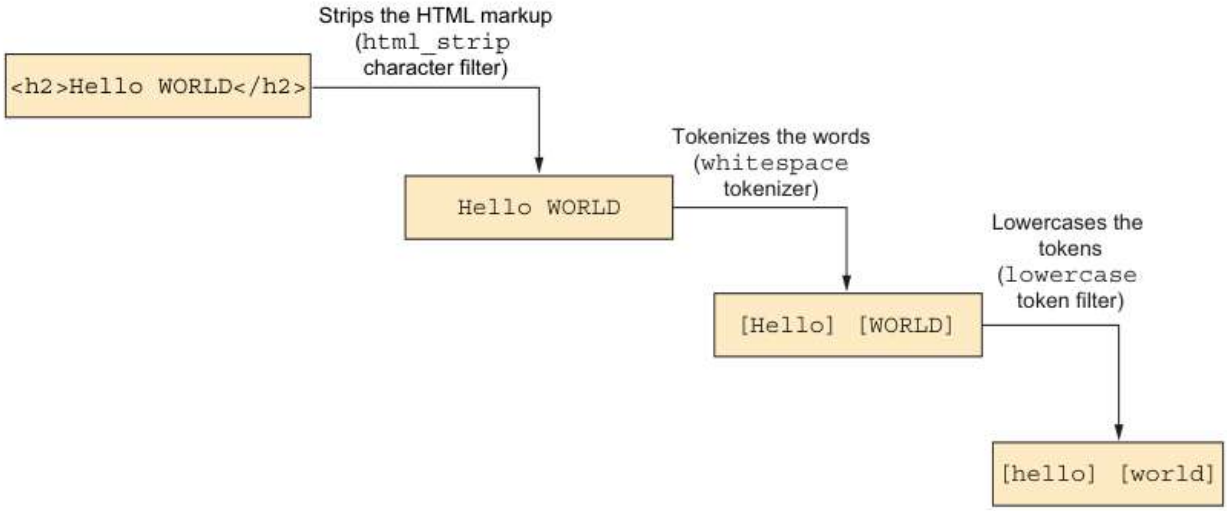


Figure 7.2 An example of text analysis in action

- Endpoint *_analyze* serve para testar analyzer.
  - Podemos passar analyzers não-default também, especificando "analyzer"
  - E também o token filter
  - Não muda nada nos índex ou no cluster, só transforma um texto ad hoc mesmo.

Listing 7.3 Creating a custom analyzer

```
GET _analyze
{
  "tokenizer": "path_hierarchy",
  "filter": ["uppercase"],
  "text": "/Volumes/FILES/Dev"
}
```

- Cada token gerado tem um tipo e uma posição:

- **The `standard` analyzer is used by default as it was not specified in the query.**
- **The input text is broken into a set of tokens. There are three tokens in this case: "james", "bond", and "007".**
- **Elasticsearch deduces the type associated with each token: `ALPHANUM` for alphanumerics and `NUM` for numbers.**
- **`start_offset` and `end_offset` indicate the start and end character offset of the word.**
- **All tokens are lowercased.**

Figure 7.3 Tokens produced by invoking the `_analyze` endpoint

-

BUILT-IN ANALYZERS

- ES já vem com vários analyzers que podem ser usados, para diversas situações.
- Também poderíamos fazer analyzers customizados.

Table 7.3 Built-in analyzers

Analyzer	Description
`standard`	The default analyzer that tokenizes input text based on grammar, punctuation, and whitespace. The output tokens are lowercased.
`simple`	Splits input text on any non-letters, such as whitespace, dashes, and numbers. Unlike the `standard` analyzer, the `simple` analyzer also lowercases the output tokens.
`stop`	A `simple` analyzer with English stop words enabled by default
`whitespace`	Tokenizes input text based on whitespace delimiters
`keyword`	Doesn't mutate the input text. The field's value is stored as is.
`language`	Helps work with human languages, as the name suggests. Elasticsearch provides dozens of analyzers for different language such as English, Spanish, French, Russian, Hindi, and so on.
`pattern`	Splits tokens based on a regular expression (regex). By default, all non-word characters help to split the sentence into tokens.
`fingerprint`	Sorts the tokens and removes duplicates to produce a single concatenated token

-

  - **Standard**
- É o mais comum:
- Tokeniza baseado em espaço, gramática e pontuação.
- Remove pontuações.

- Inclui um token filter de stopwords mas por default está desativado

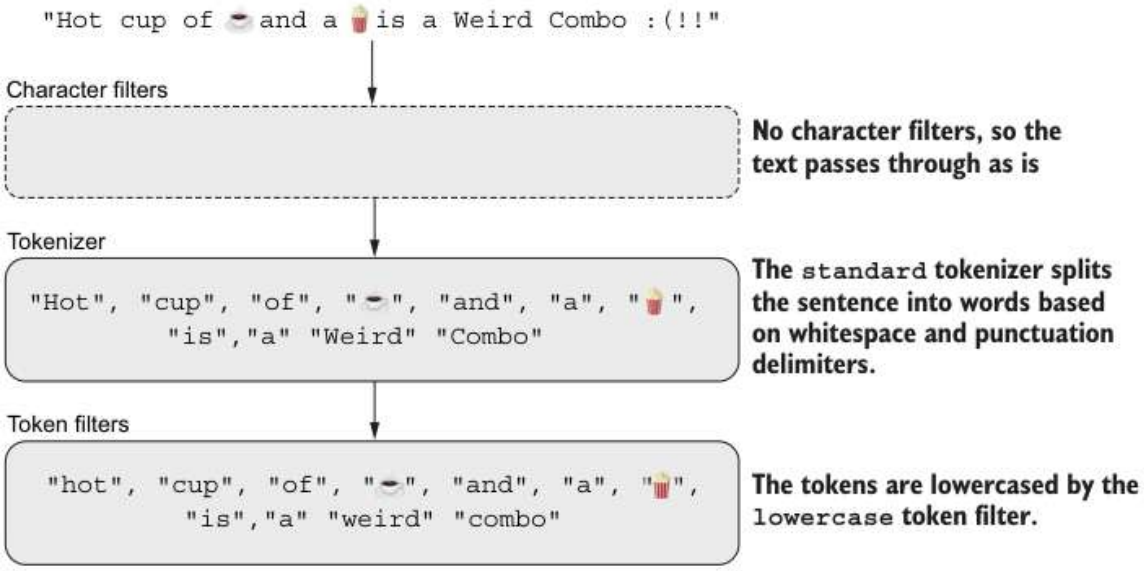


Figure 7.4 The standard (default) analyzer in action

-

```
{
  "tokens" : [
    {
      "token" : "hot",          <-- The lowercase token filter lowercases the words.
      "start_offset" : 0,
      "end_offset" : 3,
      "type" : "<ALPHANUM>",
      "position" : 0
    },
    { "token" : "cup", ... },
    { "token" : "of", ... },    <-- Stop words are not removed because the stop filter is disabled by default.
    { "token" : "☕", ... },    <-- The coffee cup is indexed as is—the width is one character.
    { "token" : "and", ... },
    { "token" : "a", ... },
    { "token" : """🍿""", ... },  <-- The popcorn emoji is indexed as is—the width is two characters.
    { "token" : "is", ... },
    { "token" : "a", ... },
    { "token" : "weird", ... },
    { "token" : "combo", ... }  <-- The smiley and exclamation marks are removed by the standard tokenizer.
  ]
}
```

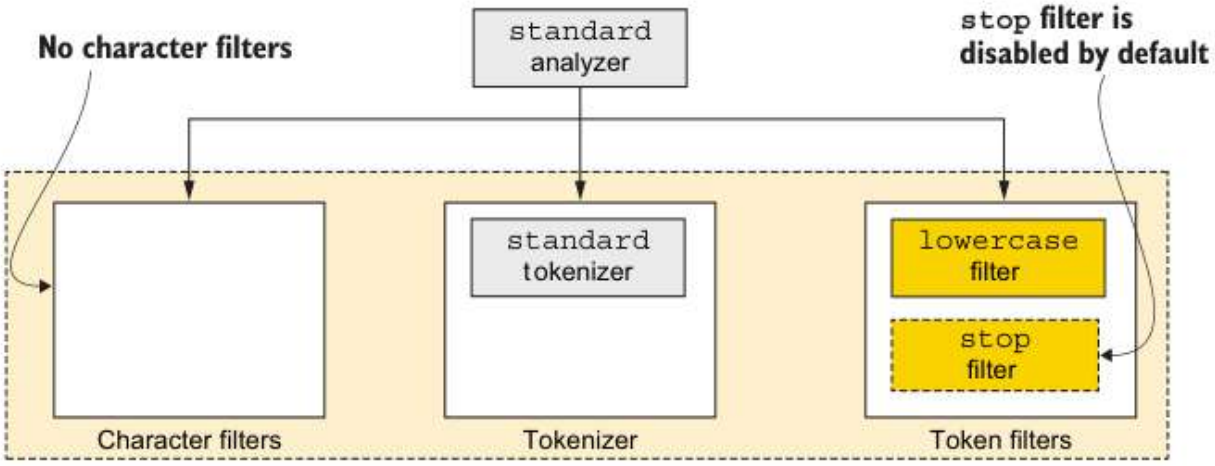


Figure 7.6 Anatomy of a `standard` analyzer

- Para alterar o analyzer, alteramos nos *settings* do index.
  - A partir de então, todo texto passa pelo analyzer modificado.

Listing 7.5 The `standard` analyzer with a stop words filter enabled

```
PUT my_index_with_stopwords
{
  "settings": {
    "analysis": {
      "analyzer": {
        "standard_with_stopwords":{
          "type":"standard",
          "stopwords":"_english_"
        }
      }
    }
  }
}
```

Sets the analyzer for the index

Names the analyzer

The standard analyzer type

Enables the English stop words filter

  - Podemos definir stopwords customizados a partir de um arquivo externo txt (um termo por linha).
  - Podemos settar o token length máximo também (Default é 7 caracteres). - **Simple**
  - Apenas um lowercase tokenizer, sem filters.
    - Separa baseado em não-letras

Listing 7.12 Analyzing text using a `simple` analyzer

```
POST _analyze
{
  "text": ["Lukša's K8s in Action"],
  "analyzer": "simple"
}
```

This results in the following:

```
["lukša","s","k","s","in","action"]
```

    - **Whitespace**

- Apenas um whitespace tokenizer.
  - Tokeniza baseado em espaço em branco

- **Keyword**
  - Apenas um "noop" (no-operation) tokenizer.
    - Tokenizer não faz nada, guarda o texto exatamente como ele vem, sem cortes.
  - Sem filtros.
  - Tokens resultantes são tipo "keyword"
  - Um token por texto.
  - Requer match exato para retornar resultado.

- **Fingerprint**
  - Remove palavras duplicadas, ordena palavras e gera um único token
  - Útil para detecção de itens duplicados.

- **Pattern**
  - Tokeniza baseado em patterns (Regex padrão Java)

Listing 7.17 Testing the `pattern` analyzer

```
POST index_with_dash_pattern_analyzer/_analyze
{
  "text": "1234-5678-9000-0000",
  "analyzer": "pattern_analyzer"
}
```

  - The output of this command contains four tokens: `["1234","5678","9000","0000"]`.
  - Inclui token filters lowercase e stopwords.

- **Language**
  - Analyzers específicos para cada idioma.
  - Já inclui as stopwords do idioma
  - Stemming também é específico de idioma.
    - Às vezes stemming age em excesso, é preciso regular isso com *stem_exclusions*.
      - Por exemplo, stemming de authorization e authority recaem em author.
        - Logo, busca por "authorization" não retornaria resultados relevantes. (indistinguível de author)

CUSTOM ANALYZER

- Podemos compor um analyzer não-default com os componentes que desejamos.

- Por exemplo, adicionar um filtro de tags HTML (e outras possibilidades) -

Listing 7.24 Custom analyzer with filters and a tokenizer

```
PUT index_with_custom_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "custom_analyzer":{
          "type":"custom",
          "char_filter":["html_strip"],
          "tokenizer":"standard",
          "filter":["uppercase"]
        }
      }
    }
  }
}
```

Specifies the custom analyzer

The type must be custom to let Elasticsearch know about our custom analyzer.

Array of character filters

Declares a single tokenizer (standard, in this case)

Token filter to uppercase incoming tokens

- Podemos definir char filters em si customizados também.
    - E.g. transformar & em "and".
- Podemos especificar analyzers diferentes para indexing e para searching.
- Analyzer pode ser especificado em vários níveis:
- Para Indexing, dois níveis:
    - **Index**
        - Analyzer se aplica a todos os text fields do index

Listing 7.28 Creating an index with a default analyzer

```
PUT authors_with_default_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "default":{
          "type":"keyword"
        }
      }
    }
  }
}
```

Setting this property sets the index's default analyzer.

        -
    - **Field**
        - Analyzer diferente para cada text field

Listing 7.27 Setting field-level analyzers during index creation

```
PUT authors_with_field_level_analyzers
{
  "mappings": {
    "properties": {
      "name":{
        "type": "text"
      },
      "about":{
        "type": "text",
        "analyzer": "english"
      },
      "description":{
        "type": "text",
        "fields": {
          "my":{
            "type": "text",
            "analyzer": "fingerprint"
          }
        }
      }
    }
  }
}
```

Uses the standard analyzer

Explicitly sets the english analyzer

Fingerprint analyzer on a multi-field

- Para Searching:
  - Em **Query**:
    - Definimos analyzer no momento da query, no corpo da requisição.
    - Pode ser diferente do analyzer usado durante o indexing.
  - Em **Field**:
    - Definimos nos mappings

Listing 7.31 Setting a search analyzer at the field level

```
PUT authors_index_with_both_analyzers_field_level
{
  "mappings": {
    "properties": {
      "author_name":{
        "type": "text",
        "analyzer": "stop",
        "search_analyzer": "simple"
      }
    }
  }
}
```

  - Podemos definir um search_analyzer padrão nos settings do index.
  - Análogo ao analyzer da indexação.

- Engine usa um analyzer apenas, e seleciona baseado nas seguintes prioridades, prioridade descrescente:

1. Analyzer da Query
2. Search_analyzer dos fields, nos mapping
3. Analyzer no Index
4. Indexing Analyzer no field ou index

CHARACTER FILTERS

- Limpa texto antes da tokenização
  - Remove HTML, pontuação.
  - Substitui caracteres
- Três tipos:
  - **HTML strip**
    - Remove tags HTML
    - Pode ser customizado para só remover alguns tipos de tags
  - **Mapping**
    - Transforma caracteres em algum texto.
      - E.g. UK --> United Kingdom
    - Também customizável
    - Mapping pode ser importado a partir de um arquivo txt.
  - **Pattern Replace**
    - Substitui por regex

SPECIFYING ANALYZERS

TOKENIZERS

- Separa o texto em tokens segundo algum critério
- Existem vários tipos de tokenizers que já vêm com o ElasticSearch
- Alguns deles:
  - **Standard:**
    - Separa baseado em separação entre palavras (e.g. espaço) e pontuação.
    - Única customização é max_token_length (default é 255)
  - **Tokenizers Ngram e edge_ngram**:
    - N-grama são sequencias se caracteres a partir de uma palavra:
      - Bigramas de *Coffee* --> co, of, ff, fe
      - Trigrama de *Coffee* --> coo, oof, off, ffe, fee.
    - Edge n-gramas são n-gramas ancorados no início da palavra:
      - Edge n-gramas de *Coffee*:
        - C, co, coo, coff, coffe, coffee.
    - **N-gram tokenizer:**
      - Transforma texto em n-gramas, n pode ser um range de valores (min e max)
        - Devolve todos os 1-grama e todos os 2-grama por default.
    - **Edge-gram tokenizer:**
      - Transforma texto em edge-grams.
      - Também podemos definir tamanho mínimo e máximo dos edge-gram.
  - **Outros:**

Table 7.4 Out-of-the-box tokenizers

Tokenizer	Description
`pattern`	Splits a field into tokens on a regex match. The default pattern is to split words when a non-word character is encountered.
`uax_url_email`	Parses fields and preserves URLs and emails. The URLs and emails are returned as they are, without any tokenization.
`whitespace`	Splits text into tokens when whitespace is encountered
`keyword`	Doesn't touch the tokens. This tokenizer returns the text as is.
`lowercase`	Splits text into tokens when a non-letter is encountered and lowercases the tokens
`path_hierarchy`	Splits hierarchical text such as filesystem folders into tokens based on path separators

TOKEN FILTERS

- Transformação dos tokens após o tokenizador
  - E.g. fazer mínusculas, reverter, stemming, sinônimos, etc.
  - ES já inclui uns 50 filters possíveis
  - Stemmer:
    - Reduz palavras para sua raíz
  - Shingle:
    - N-gramas de palavras, não de caracteres.
      - E.g. James Bond --> ["James", "James Bond"]
    - Default é n-gramas de 1 e 2 palavras.
      - Isso pode ser customizado.
  - Synonym:
    - Transforma palavras em seus sinônimos.
    - Requer uma lista de sinônimos.
      - Pode ser passado por um arquivo txt.

Summary

- Elasticsearch analyzes `text` fields via the text analysis process. Text analysis is carried out using either built-in or custom analyzers. Non-text fields are not analyzed.
- Text analysis consists of two phases: tokenization and normalization. Tokenization splits the input field into individual words or tokens, and normalization enhances the word (for example, changing it to a synonym, stemming it, or removing it).
- Elasticsearch employs a software module called an analyzer to carry out text analysis. An analyzer is a package made of character filters, token filters, and a tokenizer.
- Elasticsearch uses a `standard` analyzer by default if no analyzer is given explicitly during indexing and searching. A `standard` analyzer employs no character filter, a `standard` tokenizer, and two token filters (`lowercase` and `stop`), although the `stop` filter is off by default.
- Every analyzer must have one (and only one) tokenizer, but it can have zero or more character or token filters.
- Character filters help strip unwanted characters from input fields. Tokenizers act on the fields processed by character filters (or on raw fields, since character filters are optional). Token filters work on the tokens emitted by the tokenizer.
- Elasticsearch provides several out-of-the-box analyzers. We can mix and match existing tokenizers with character or token filters to make custom analyzers that suit our requirements.

8 - Introducing Search

OVERVIEW

- Há busca estruturada e não-estruturada
  - Estruturada -- match exato, sem score de relevância
  - Não-estruturada -- por relevância, aproximação.
- Precision: densidade de documentos relevantes
- Recall: quanto de todos que eu gostaria de retornar foram retornados.

COMO FUNCIONA?

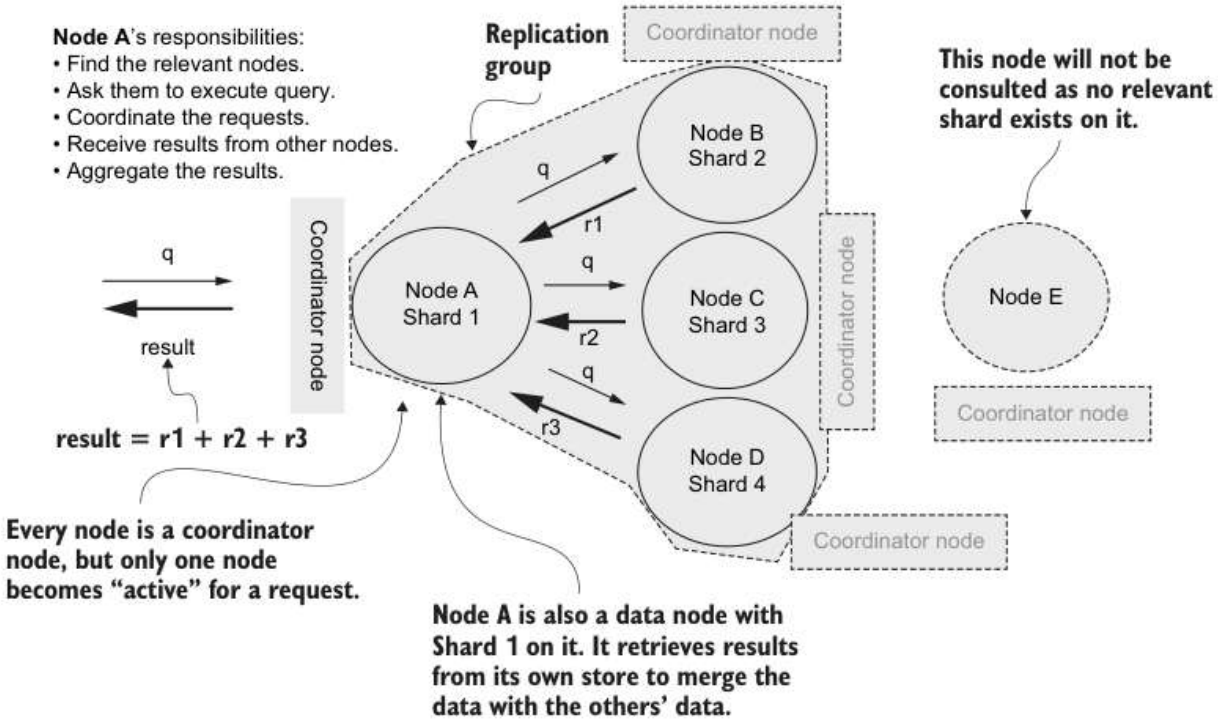


Figure 8.1 A typical search request, and the mechanics of how a search works

-

MOVIE SAMPLE DATA

- Exemplo de index de filmes

SEARCH FUNDAMENTALS

- _search endpoint
  - GET com parametros
  - POST com payload
    - Query DSL (Domain Specific Language)
    - Melhor pra buscas mais complexas
- Toda query ocorre em um de dois contextos:
  - *Query context*:
    - Gera score de relevância
  - Filter context:
    - Sem score, só retorna ou não documentos.
    - Poupa recursos da máquina, não requer passar por scoring, usa caching

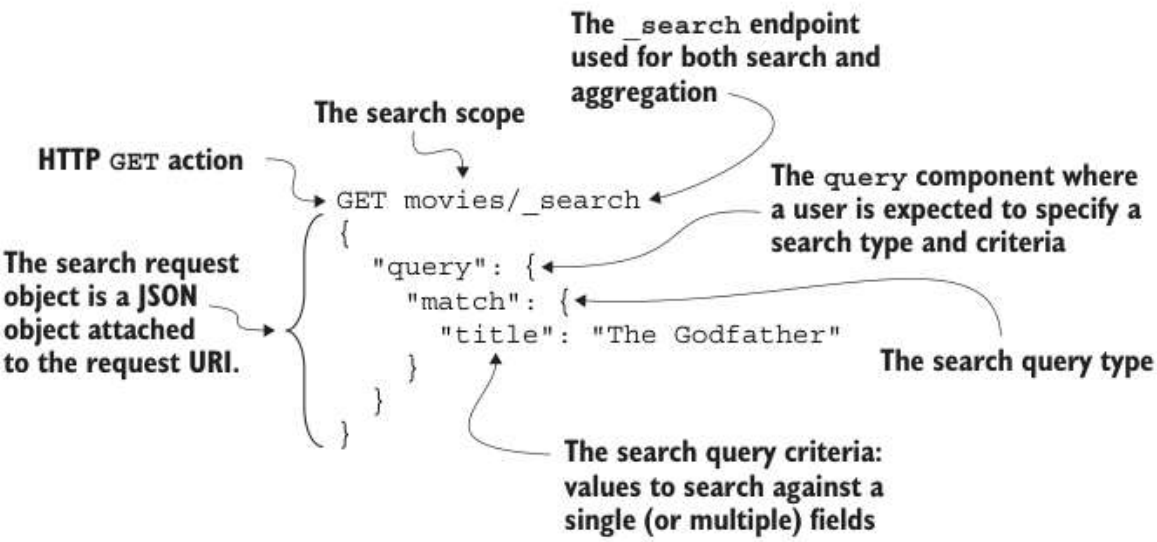


Figure 8.2 Components of a search request

-
- GET e POST são equivalentes na prática, mas se costuma usar POST quando há body.
- Se não fornecermos um index na busca, pressupõe-se que são todos.

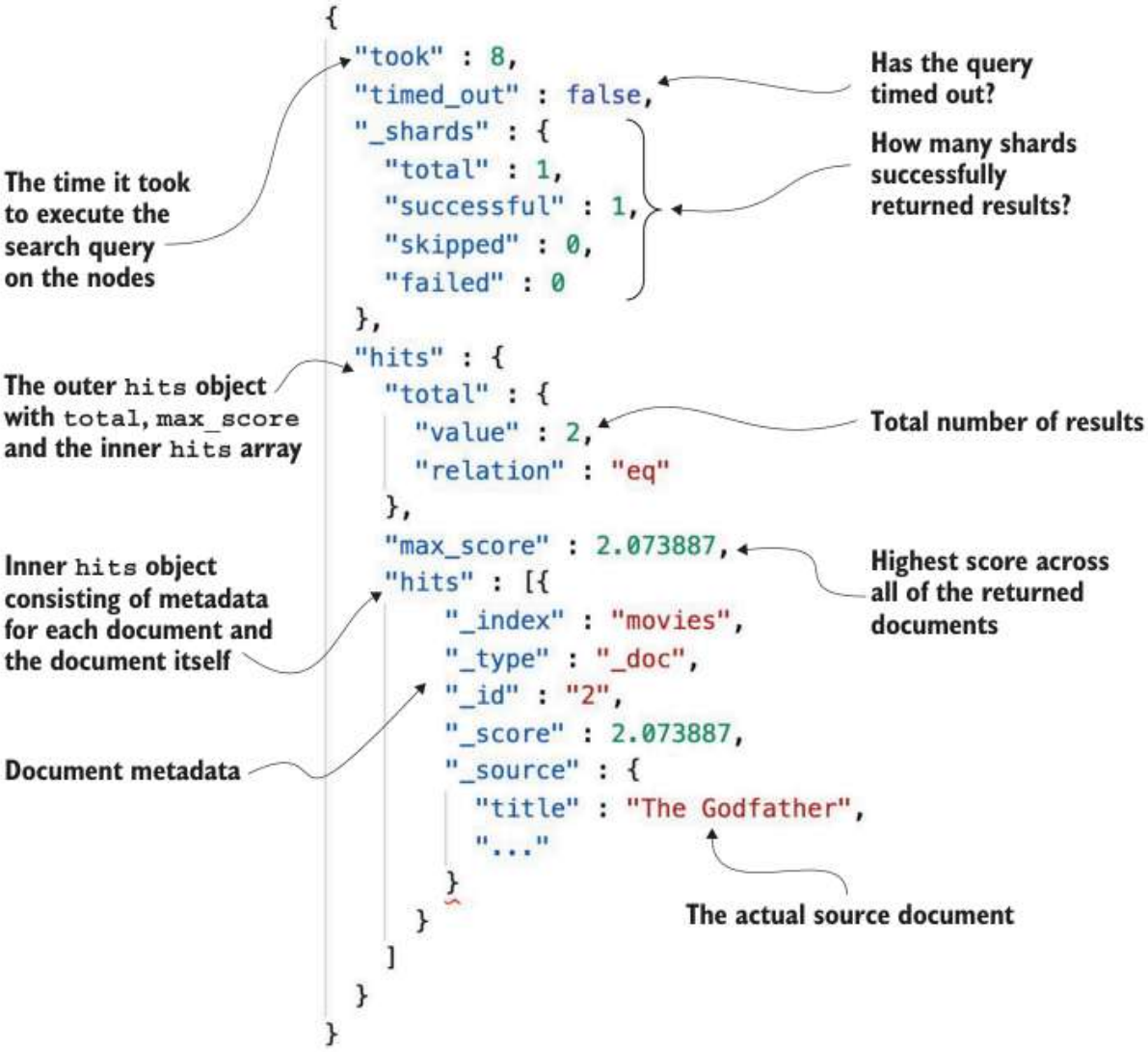


Figure 8.3 Constituents of a search response

-

URI REQUEST SEARCHES

- Busca por URI:
  - Busca com params, menos comum e menos flexível.
  - *GET movies/_search?q=title:Godfather Knight*
  - Passando o parâmetro explain=true o score é explicado.
  - Permite AND operator (e outros booleanos)
  - Busca por múltiplos campos ao mesmo tempo.
  - Podemos editar o *default_operator* para ser AND e não OR, para os vários critérios.
  - Busca por desigualdades (>= < etc)
- Podemos passar "query_string" na query por body que funciona igual ao ?q= pelo URI (não recomendado)

QUERY DSL

- Baseado em JSON
- Bastante flexível
- Define um "query" object:
  - Dentro dele é aplicado a lógica da busca.

Listing 8.12 Query DSL sample query

```
GET movies/_search
{
  "query": {
    "multi_match": {
      "query": "Lord",
      "fields": ["synopsis","title"]
    }
  }
}
```

- Dá pra usar com CURL normalmente sendo o body de um POST/GET
- Para buscas de agregados (analytics), usamos o objeto "aggs" no lugar do "query":

Listing 8.14 Aggregation query written in Query DSL format

```
GET movies/_search
{
  "size": 0,
  "aggs": {
    "average_movie_rating": {
      "avg": {
        "field": "rating"
      }
    }
  }
}
```

- Leaf Query:
  - Busca simples que busca um dado objetivo.
  - Um predicado só
- Query Composta:
  - Complexo, vários critérios ao mesmo tempo
  - Combina várias leaf queries

SEARCH FEATURES

- Podemos manipular a forma de retornar os resultados
- Paginação:
  - 10 resultados por default
  - Parâmetro size
    - Número de resultados retornados
  - Importante calibrar para não sobrecarregar client ou servidor
  - Máximo é 10_000 por default
    - Pode ser alterado nos settings do index
    - Aumentar é pouco recomendável
  - Parâmetro from
    - Só retorna a partir da n-ésima página, ignora as anteriores
  - Se houver muitos resultados, é melhor usar o parâmetro *search_after* e não *from* ou *size* por questões de desempenho.
- Highlighting:
  - Destaca fields especificados
  - Por default, envolve em tags :

- Pode ser editado

Listing 8.19 Highlighting results when matched

```
GET movies/_search
{
  "_source": false,          <- Suppresses the source to be returned
  "query": {
    "term": {
      "title": {
        "value": "godfather"
      }
    }
  },"highlight": {           <- Includes a highlight object along with the fields to be highlighted
    "fields": {
      "title": {}            <- The field we want to highlight
    }
  }
}
```

  - - Explaining:
  - Flag explain=true
    - Declarado no mesmo escopo que o objeto query
  - Explica como foi calculado o score de relevância para os resultados.
    - Boost *IDF (inverse document frequency)* TF (term frequency)
  - Também podemos usar a api *_explain*
- Sorting:
  - Declarado no mesmo escopo que o objeto query
  - Permite editar critério de ordenamento:
    - Default é score decrescente
  - Se _score não for o critério de ordenamento, não é computado.
    - Mas podemos gerar score mesmo assim com flag "*track_scores*"
  - Podemos ordenar por múltiplos campos
    - Campos posteriores servem de desempate.
- Manipulating Results:
  - Podemos omitir o conteúdo dos documentos encontrados:
    - Parâmetro _source=False
  - Podemos devolver só um subconjunto dos campos dos documentos, em vez do documento inteiro.
    - Usamos o campo "fields" para isso e um array com os nomes dos campos.
      - Permite uso de wildcard
  - Podemos manipular o resultado com um script antes de retorná-lo
    - Usamos o campo "script_fields", que inclui os fields e os scripts respectivos.
  - Podemos definir _source como uma lista de fields:
    - Incluir ou excluir fields
      - Aceita wildcard
  - Podemos priorizar os resultados de alguns índices mais do que outros:
    - Parâmetro "indices_boost"
      - Multiplica-se um coeficiente para o score dos resultados daquele index.

Summary

- Searching can be categorized as structured and unstructured search types.
- Structured data works with non-text fields like numeric and date fields or fields that are not analyzed during indexing time and produce binary results (either they exist or they don't).
- Unstructured data deals with text fields expected to have a relevancy score. The engine scores the results based on how well the resultant documents match the criteria.
- We use term-level search for structured queries and full-text search for unstructured data.
- Every search request is processed by a coordinator node. Coordinator nodes are responsible for asking other nodes to execute the query, return partial data, aggregate it, and respond to the client with a final result.
- Elasticsearch exposes a `_search` endpoint for queries and aggregations. We can invoke the `_search` endpoint by either using a URI request with parameters or building a full request using a special syntax called Query DSL.
- Query DSL is the preferred choice for creating search queries. We can construct a plethora of queries, including advanced queries, using Query DSL.
- Query DSL allows us to create leaf and compound queries. Leaf queries are simple search queries with a single criterion. Compound queries are used for advanced queries built with conditional clauses.
- Crosscutting features are available for most types of queries: pagination, highlighting, explanation of the scoring, manipulation of the results, and so on.

9 - Term-level search

- Buscas de match exato
- Para dados estruturados.

OVERVIEW OF TERM-LEVEL SEARCH

- E.g.procura datas e números
- Essas buscas não envolvem o analyzer:
- Computacionalmente mais barato
- A não ser que usemos um normalizer.
- Engine fará a busca pelo termo exato no índice invertido, sem modificações.
- Ideal para busca tipo keyword
    - Busca de termos exatos.

THE TERM QUERY

- Term não é analisado, é procurado direto no índice invertido.

Listing 9.1 Fetching movies with a given rating

```
GET movies/_search
{
  "query": {
    "term": {
      "certificate": "R"
    }
  }
}
```

The term query declaration

- Maiúsculo ou minúsculo é importante, pode alterar resultado (pois não passa por analyzer)
- Pouco recomendado para buscar em text fields, exceto se o campo já for previsto para isso.
- Sintaxe não-simplificada:

Listing 9.3 Full syntax of a query

```
GET movies/_search
{
  "query": {
    "term": {
      "certificate": {
        "value": "R",
        "boost": 2
      }
    }
  }
}
```

The certificate field has values enclosed in an object.

The certificate's search criteria

In addition to the value, we can provide parameters like boost.

THE TERMS QUERY

- Busca múltiplos critérios de uma vez (operador OR)

Listing 9.4 Searching for multiple search criteria in a field

```
GET movies/_search
{
  "query": {
    "terms": {
      "certificate": ["PG-13","R"]
    }
  }
}
```

The terms query expects an array of search criteria.

Multiple search criteria against a single field

-
- Há um setting (pode ser alterado dinamicamente, sem parar o index) para alterar o número máximo de termos permitidos na busca
    - default é 2^16.
- Exemplo de busca a partir de um resultado de um documento no lugar de um termo pré-definido:

Listing 9.8 A `terms` lookup search

```
GET classic_movies/_search
{
  "query": {
    "terms": {
      "director": {
        "index":"classic_movies",
        "id":"3",
        "path":"director"
      }
    }
  }
}
```

A terms query (with a twist!)

The field that we are interested in searching against

The index field denotes the name of the index where the document resides.

Name of the field containing the terms for the query

Search field in the current document

-

THE IDS QUERY

- Busca direto por Ids dos documentos.

Listing 9.9 Fetching multiple documents using an `ids` query

```
GET movies/_search
{
  "query": {
    "ids": {
      "values": [10,4,6,8]
    }
  }
}
```

Name of the query

Provides the document IDs as an array

-
- Podemos fazer a mesma busca fazendo uma terms query que busca por dados "*_id*"

THE EXISTS QUERY

- Verifica se um campo existe antes de fazer a busca de fato
    - Pode economizar tráfego.

- Só retorna documentos em que um dado campo existe.

Listing 9.11 Running an `exists` query to check whether a field exists

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

Defines the query type as exists

Provides the field that we want to check in the document

-
- Também pode ser útil para buscar documentos que **não** contêm um dado campo, usando *must_not*

THE RANGE QUERY

- Procura valores dentro de um range

Listing 9.13 Fetching movies with a specified range of ratings

```
GET movies/_search
{
  "query": {
    "range": {
      "rating": {
        "gte": 9.0,
        "lte": 9.5
      }
    }
  }
}
```

-
- Podemos fazer operações com datas:
  -

```
GET movies/_search
{
  "query": {
    "range": {
      "release_date": {
        "gte": "now-1y"
      }
    }
  }
}
```

Fetches all the movies from last year: the current date (represented as now) minus one year

THE WILDCARD QUERY

- Busca por padrões

Listing 9.15 Wildcard search in action

```
GET movies/_search
{
  "query": {
    "wildcard": {
      "title": {
        "value": "god*"
      }
    }
  }
}
```

The wildcard query type

Searches the value field with a wildcard operator

-
- Queries aqui também não são analisadas. Lowercase é importante.
- Para buscar wildcard que subsituem apenas um único caractere, usamos "?" no lugar de "*".:

Table 9.2 Types of wildcards

Character	Description
* (asterisk)	Searches for zero or more characters
? (question mark)	Searches for a single character

-
- Este tipo de query é computacionalmente cara
  - Este tipo pode ser desabilitado nos settings no cluster.:

```
PUT _cluster/settings
{
  "transient": {
    "search.allow_expensive_queries": "false"
  }
}
```

-

THE PREFIX QUERY

- Como wildcard mas apenas para prefixos

Listing 9.17 Using a `prefix` query

```
GET movies/_search
{
  "query": {
    "prefix": {
      "actors.original": {
        "value": "Mar"
      }
    }
  }
}
```

Specifies the prefix type query

Queries for words that start with "Mar"

-

- Versão abreviada:

Listing 9.18 Shortening `prefix` query usage

```
GET movies/_search
{
  "query": {
    "prefix": {
      "actors.original":  "Leo"
    }
  }
}
```

- Também é uma operação cara.
  - Para mitigar isso, podemos definir "index_prefixes" quando criamos um index.
    - Escolhemos qual campo vai ser o alvo dos index_prefixes
    - ES então indexa prefixos daquele campo de tamanho de 2 a 5 (por default)
      - E.g. title="Gladiador" preindexa os prefixos ["Gl","Gla","Glad","Gladi"]
      - Acelera a busca de prefixos posteriores
      - Tamanho dos prefixo preindexados pode ser customizado no mappings
        - Límite absoluto mínimo é maior que 0 e máximo menor que 20.

FUZZY QUERIES

- Buscas não-exatas de termos.
- Perdoa erros de ortografia e digitação.
- Baseado na distância de Levenshtein
  - A.k.a "Edit Distance"
- Em queries fuzzy, podemos definir a *fuzziness* (edit distance) desejada, para saber quão flexível a busca será feita.

Listing 9.22 A `fuzzy` query in action

```
GET movies/_search
{
  "query": {
    "fuzzy": {
      "genre": {
        "value": "rama",
        "fuzziness": 1
      }
    }
  },
  "highlight": {
    "fields": {
      "genre": {}
    }
  }
}
```

- Podemos também deixar o fuzziness no "auto":

- É o default quando não definimos nenhum *fuzziness*
- ES define o grau de fuzziness dependendo do tamanho do termo procurado.
  - 0 a 2 - fuzziness=0
  - 3 a 5 - fuzziness=1
  - >5 - fuzziness=2

Summary

- Term-level searching is carried out on structured data such as numbers, keywords, Booleans, and dates.
- A term-level search produces a binary result: the search leads to a result or none. There is no *likely match* scenario.
- Term-level queries are not analyzed, meaning that when applied to `text` fields undergoing text analysis during indexing, they can yield incorrect or no results.
- There are many term-level search queries: `term`, `terms`, `prefix`, `range`, `fuzzy`, and others.
- A `term` query searches for a single term against a field, while a `terms` (plural) query search for multiple values against a single field.
- A `range` query helps search data within a set of bounds: for example, searching for crimes that happened in London in the last month.
- A `wildcard` query uses `*` and `?` operators to fetch results.
- Fuzziness employs Levenshtein's edit distance to fetch similar-looking words. Elasticsearch employs `fuzzy` queries to support the user's spelling inconsistencies.
- A `prefix` query retrieves results given a prefix (no need to specify a wildcard operator). As prefix operations are expensive to execute on a live index, we can ask Elasticsearch to pre-create them at index time to avoid finding them during a live query phase.

READ-ME

Elasticsearch IN ACTION

SECOND EDITION

Madhusudhan Konda

Foreword by Shay Banon

MANNING

brief contents